CSE140: Components and Design Techniques for Digital Systems

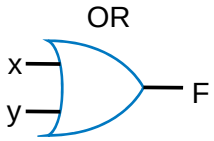
Boolean algebra & logic circuits

Tajana Simunic Rosing

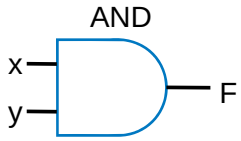
Boolean algebra

- ***Boolean Algebra***
  - B = {0, 1}
  - Variables represent 0 or 1 only
  - Operators return 0 or 1 only
  - Basic operators
    - • is logical AND: *a AND b* returns 1 only when both a=1 and b=1
    - + is logical OR: *a OR b* returns 1 if either (or both) a=1 or b=1
    - ’ is logical NOT: *NOT a* returns the opposite of a (1 if a=0, 0 if a=1)
  - All algebraic axioms hold

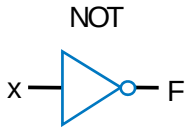
a	b	OR
0	0	0
0	1	1
1	0	1
1	1	1



a	b	AND
0	0	0
0	1	0
1	0	0
1	1	1



a	NOT
0	1
1	0



Axioms and theorems of Boolean algebra

- identity
  - 1. X + 0 = X
  - 1D. X • 1 = X
- null
  - 2. X + 1 = 1
  - 2D. X • 0 = 0
- idempotency:
  - 3. X + X = X
  - 3D. X • X = X
- involution:
  - 4. (X')' = X
- complementarity:
  - 5. X + X' = 1
  - 5D. X • X' = 0
- commutativity:
  - 6. X + Y = Y + X
  - 6D. X • Y = Y • X
- associativity:
  - 7. (X + Y) + Z = X + (Y + Z)
  - 7D. (X • Y) • Z = X • (Y • Z)

Axioms and theorems of Boolean algebra (cont'd)

- distributivity:
  8. $X \bullet (Y + Z) = (X \bullet Y) + (X \bullet Z)$ 8D. $X + (Y \bullet Z) = (X + Y) \bullet (X + Z)$
- uniting:
  9. $X \bullet Y + X \bullet Y' = X$ 9D. $(X + Y) \bullet (X + Y') = X$
- absorption:
  10. $X + X \bullet Y = X$ 10D. $X \bullet (X + Y) = X$
  11. $(X + Y') \bullet Y = X \bullet Y$ 11D. $(X \bullet Y') + Y = X + Y$
- factoring:
  12. $(X + Y) \bullet (X' + Z) = X \bullet Z + X' \bullet Y$ 12D. $X \bullet Y + X' \bullet Z = (X + Z) \bullet (X' + Y)$
- concensus:
  13. $(X \bullet Y) + (Y \bullet Z) + (X' \bullet Z) = X \bullet Y + X' \bullet Z$ 13D. $(X + Y) \bullet (Y + Z) \bullet (X' + Z) = (X + Y) \bullet (X' + Z)$

Axioms and theorems of Boolean algebra (cont'd)

- de Morgan's:

  14. $(X + Y + ...)' = X' \bullet Y' \bullet ...$ 14D. $(X \bullet Y \bullet ...)' = X' + Y' + ...$

- generalized de Morgan's:

  15. $f'(X_1,X_2,...,X_n,0,1,+,\bullet) = f(X_1',X_2',...,X_n',1,0,\bullet,+)$

- establishes relationship between • and +

Axioms and theorems of Boolean algebra (cont'd)

- Duality
  - a dual of a Boolean expression is derived by replacing • by +, + by •, 0 by 1, and 1 by 0, and leaving variables unchanged
  - any theorem that can be proven is thus also proven for its dual!
  - a meta-theorem (a theorem about theorems)
- duality:

  16. $X + Y + \ldots \Leftrightarrow X \bullet Y \bullet \ldots$
- generalized duality:

  17. $f(X_1,X_2,\ldots,X_n,0,1,+,\bullet) \Leftrightarrow f(X_1,X_2,\ldots,X_n,1,0,\bullet,+)$

- Different than deMorgan's Law
  - this is a statement about theorems
  - this is not a way to manipulate (re-write) expressions

Proving theorems

- Using the axioms of Boolean algebra:
  - e.g., prove the theorem: $X \cdot Y + X \cdot Y' = X$

distributivity (8)	$X \cdot Y + X \cdot Y'$	=	$X \cdot (Y + Y')$
complementarity (5)	$X \cdot (Y + Y')$	=	$X \cdot (1)$
identity (1D)	$X \cdot (1)$	=	X ✓

  - e.g., prove the theorem: $X + X \cdot Y = X$

identity (1D)	$X + X \cdot Y$	=	$X \cdot 1 + X \cdot Y$
distributivity (8)	$X \cdot 1 + X \cdot Y$	=	$X \cdot (1 + Y)$
identity (2)	$X \cdot (1 + Y)$	=	$X \cdot (1)$
identity (1D)	$X \cdot (1)$	=	X ✓

Proving theorems example

- Prove the following using the laws of Boolean algebra:
  - (X • Y) + (Y • Z) + (X' • Z) = X • Y + X' • Z

	(X • Y) + (Y • Z) + (X' • Z)
identity	(X • Y) + (1) • (Y • Z) + (X' • Z)
complementarity	(X • Y) + (X' + X) • (Y • Z) + (X' • Z)
distributivity	(X • Y) + (X' • Y • Z) + (X • Y • Z) + (X' • Z)
commutativity	(X • Y) + (X • Y • Z) + (X' • Y • Z) + (X' • Z)
factoring	(X • Y) • (1 + Z) + (X' • Z) • (1 + Y)
null	(X • Y) • (1) + (X' • Z) • (1)
identity	(X • Y) + (X' • Z) ✓

Proving theorems (perfect induction)

- Using perfect induction (complete truth table):
  - e.g., de Morgan's:

$(X + Y)' = X' \bullet Y'$

NOR is equivalent to AND with inputs complemented

X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1		
0	1	1	0		
1	0	0	1		
1	1	0	0		

$(X \bullet Y)' = X' + Y'$

NAND is equivalent to OR with inputs complemented

X	Y	X'	Y'	(X • Y)'	X' + Y'
0	0	1	1		
0	1	1	0		
1	0	0	1		
1	1	0	0		

Example that Applies Boolean Algebra Properties

- Want automatic door opener circuit (e.g., for grocery store)
  - Output: f=1 opens door
  - Inputs:
    - p=1: person detected
    - h=1: switch forcing hold open
    - c=1: key forcing closed
  - Want open door when
    - h=1 and c=0, or
    - h=0 and p=1 and c=0
- Found inexpensive chip that computes:
    - f = c'hp + c'hp' + c'h'p
  - Can we use it?

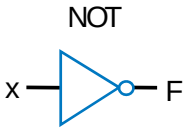
CSE140: Components and Design Techniques for Digital Systems

Combinational circuit building blocks: Building gates from transistors

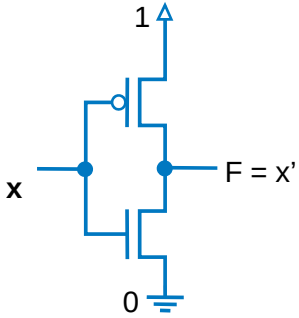
Tajana Simunic Rosing

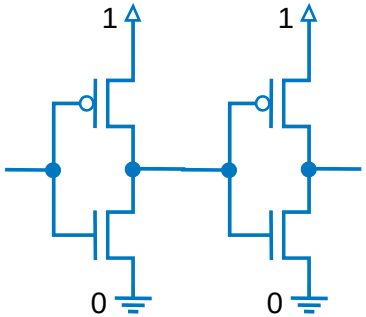
Inverter delay

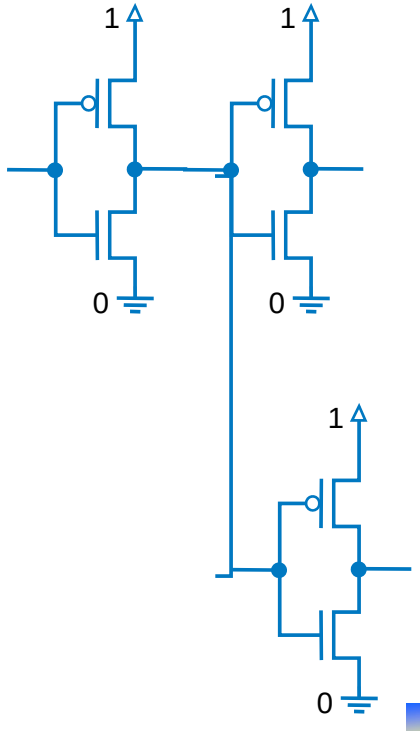
- Delay: estimate using R_n & C_g; $R_P=2R_n$



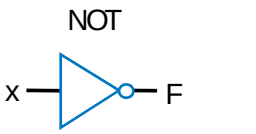
x	F
0	1
1	0



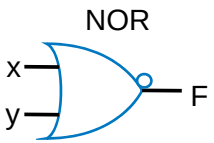




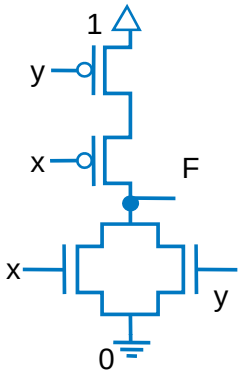
Basic CMOS gates

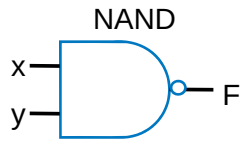


x	F
0	1
1	0

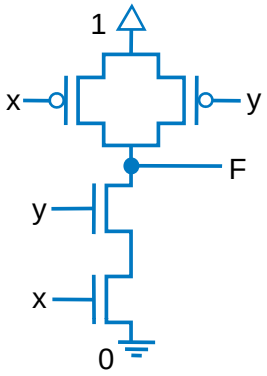


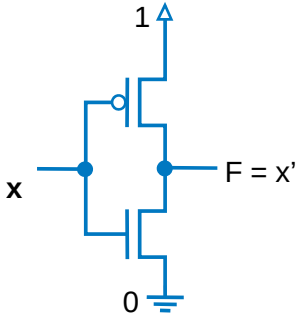
x	y	F
0	0	1
0	1	0
1	0	0
1	1	0





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

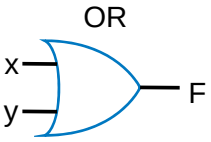




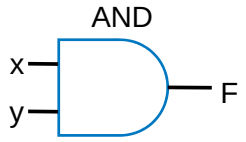
- Implement logic operators using transistors
  - Call those implementations ***logic gates***

AND/OR gates

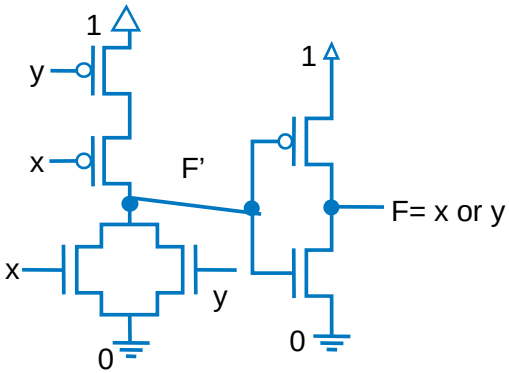
- OR/AND are harder to make than NOR/NAND

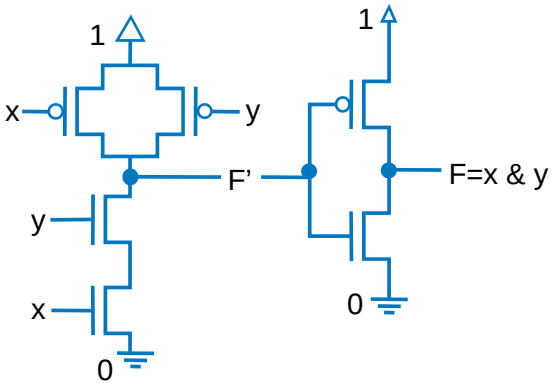


x	y	F
0	0	0
0	1	1
1	0	1
1	1	1



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



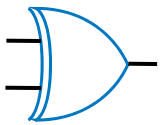


Completeness of NAND

- Any logic function can be implemented *using just NAND gates.* Likewise for *NOR*. Why?
  - Need AND, OR and NOT

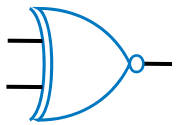
XOR/XNOR Gates

XOR



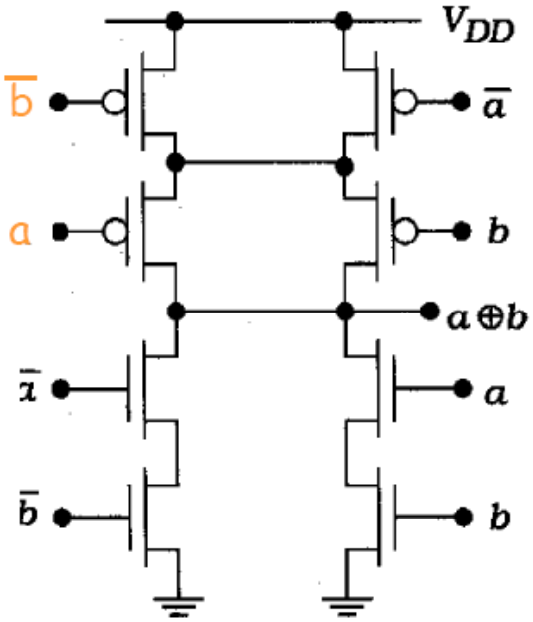
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

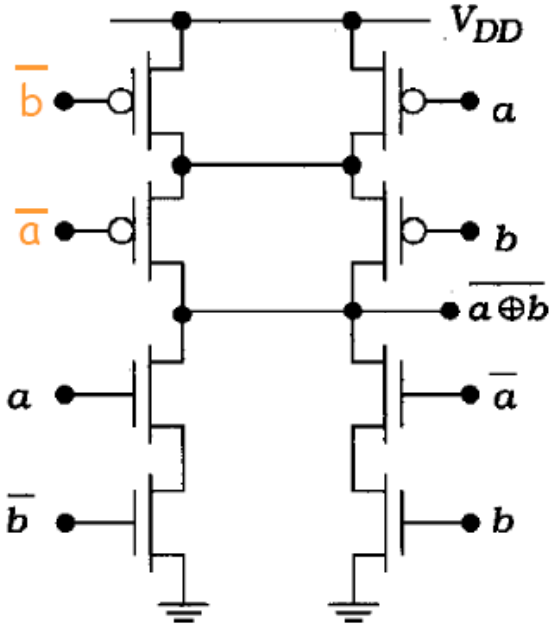


x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

- XOR = x'y + xy' = (xy + x'y')' (Exclusive OR)
- XNOR = (x'y + xy')' = xy + x'y' (Exclusive OR)



(a) Exclusive-OR



(b) Exclusive-NOR

Number of Possible Boolean Functions

- # of possible functions of 2 variables:
  - 2^2 rows in truth table, 2 choices for each
  - $2^{(2^2)} = 2^4 = 16$ possible functions
- N variables
  - 2^N rows
  - $2^{(2^N)}$ possible functions
- Cost

a	b	F	
0	0	0 or 1	2 choices
0	1	0 or 1	2 choices
1	0	0 or 1	2 choices
1	1	0 or 1	2 choices

$2^4 = 16$
possible functions

a	b	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
0	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**
1	**0**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**
1	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
		0	a AND b		a		b	a XOR b	a OR b	a NOR b	a XNOR b	b'		a'		a NAND b	1

- a' (F12) and b' (F10): require 2 transistors for an inverter (not gate)
- a nor b (F4) and a nand b (F14): require 4 transistors
- a or b (F7) and a and b (F1): require 6 transistors
- a = b (F9) and a $\oplus$ b (F6): require 12 transistors

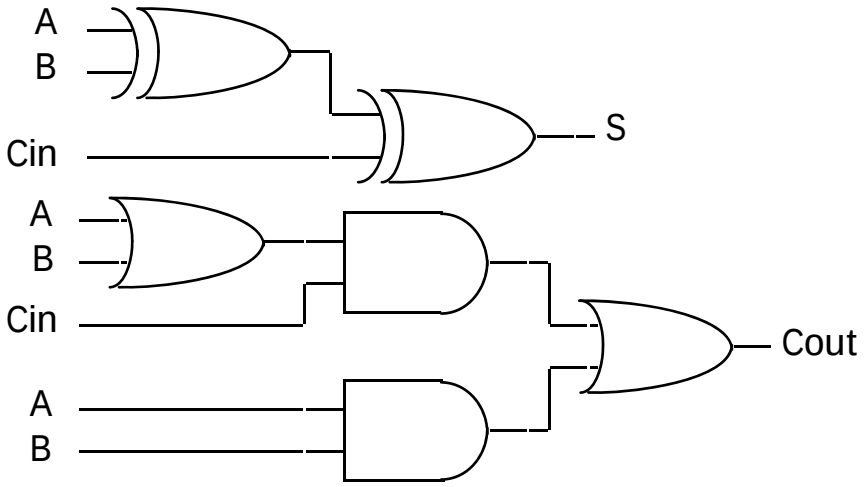
Circuits for binary addition

- Half adder (add 2 1-bit numbers)
  - Sum =
  - Cout =

Ai	Bi	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

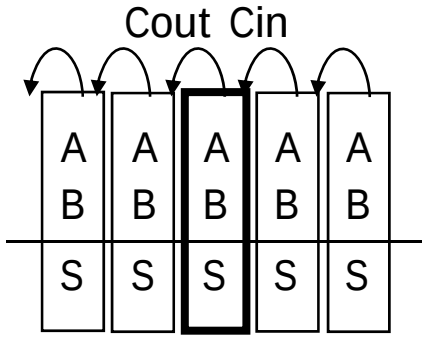
- Full adder (carry-in to cascade for multi-bit adders)
  - Sum =
  - Cout =

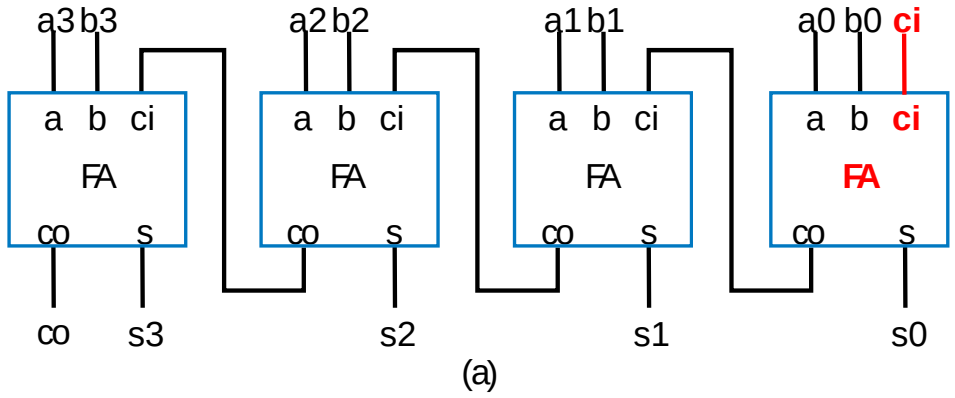
Ai	Bi	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Example: 4-bit binary adder

- Inputs: A, B, Carry-in
- Outputs: Sum, Carry-out

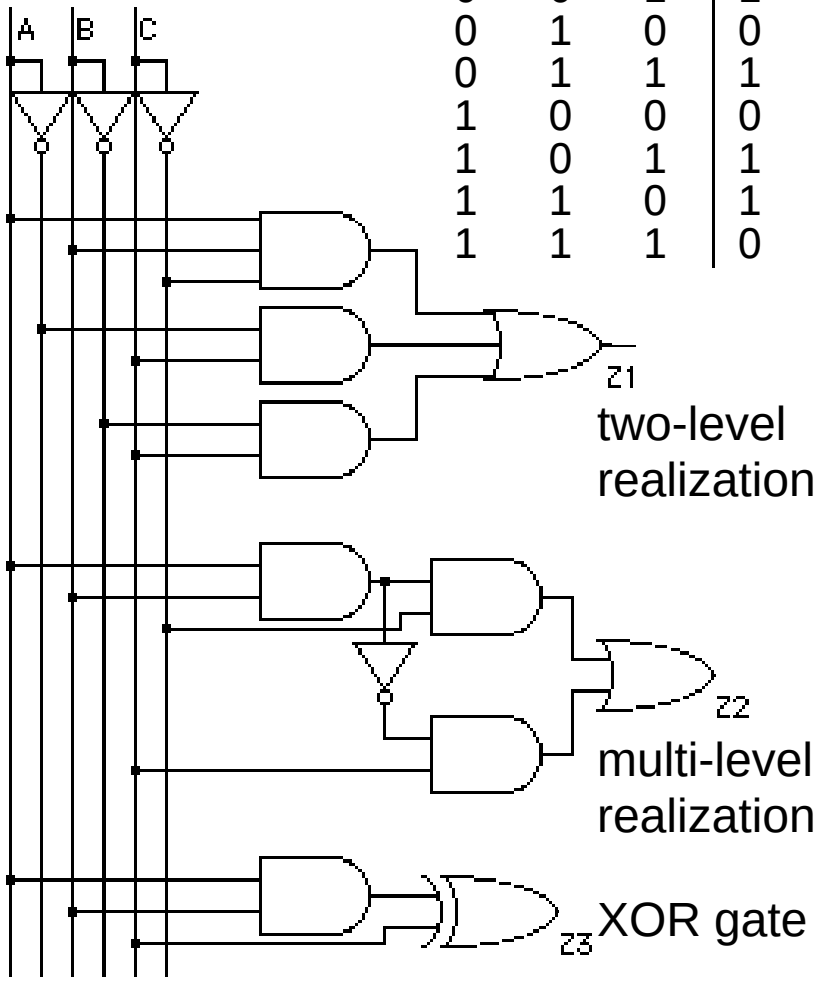




(a)

Choosing different realizations of a function

A	B	C	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0



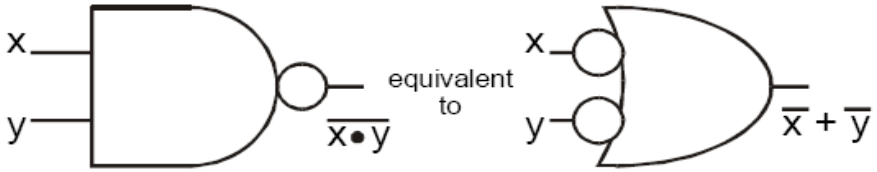
Which is the best realization and why?

- Reduce number of levels of gates
  - fewer level of gates implies reduced signal propagation delays
  - minimum delay configuration typically requires more gates
- Reduce number of inputs
  - fewer inputs implies smaller and thus faster gates; fan-in can be limited
- Reduce number of gates
  - fewer gates means smaller circuits - directly influences manufacturing costs
- Exploring tradeoffs between increased circuit delay and size
  - tools to generate different solutions
  - logic minimization: reduce number of gates and complexity
  - logic optimization: reduction while trading off against delay

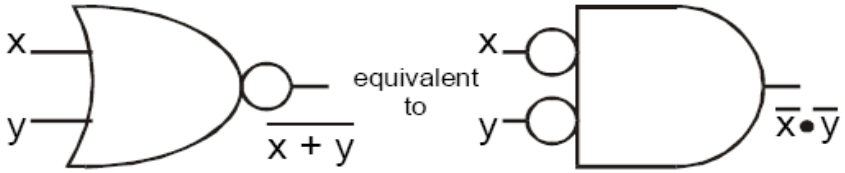
DeMorgan and CMOS

- Use it to figure out P & NMOS parts of gates

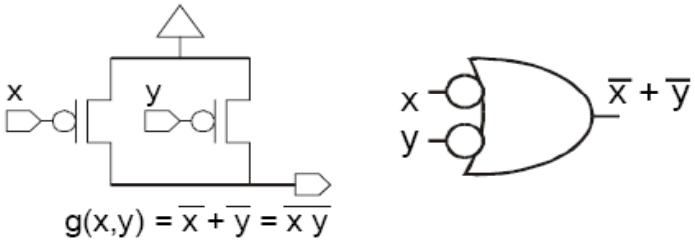
- DeMorgan Relations
  - NAND-OR rule $\overline{a \cdot b} = \overline{a} + \overline{b}$
    - bubble pushing illustration



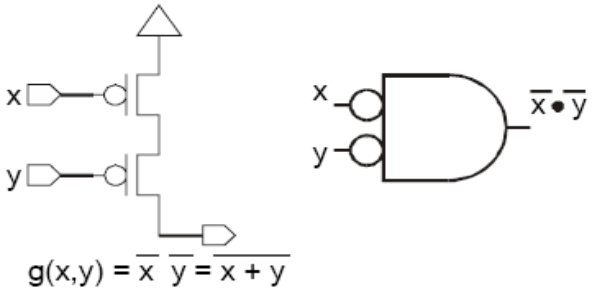
    - bubbles = inversions
  - NOR-AND rule $\overline{a + b} = \overline{a} \cdot \overline{b}$



- pMOS and bubble pushing
  - Parallel-connected pMOS

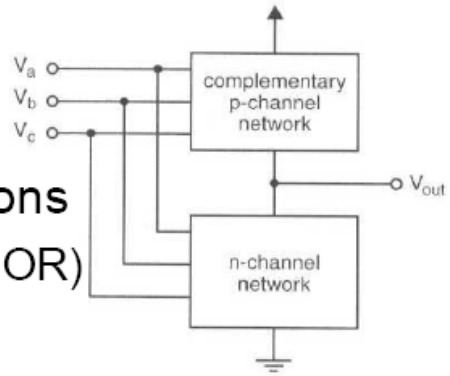


    - assert-low OR
    - creates NAND function
  - Series-connected pMOS

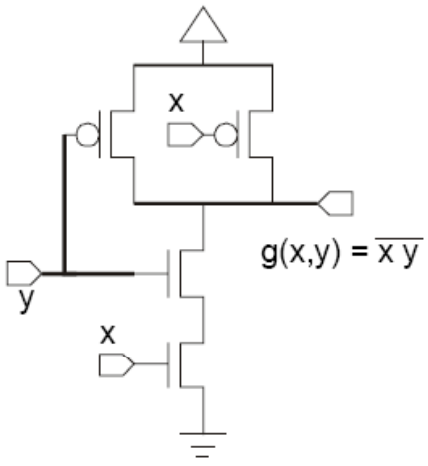


Rules for making gates

The Mathematical Method



- Given a logic function
  F = f(a, b, c)
- Reduce (using DeMorgan) to eliminate inverted operations
  - inverted variables are OK, but not operations (NAND, NOR)
- Form pMOS network by complementing the **inputs**
  $Fp = f(\overline{a}, \overline{b}, \overline{c})$
- Form the nMOS network by complementing the **output**
  $Fn = \overline{f(a, b, c)} = \overline{F}$
- Construct Fn and Fp using AND/OR series/parallel MOSFET structures
  - series = AND, parallel = OR



•EXAMPLE:

$F = \overline{ab} \Rightarrow$

$Fp = \overline{\overline{a}\ \overline{b}} = a+b$; OR/parallel

$Fn = \overline{\overline{ab}} = ab$: AND/series

Another way of making CMOS gates

- Reducing Logic Functions
  - fewest operations ⇒ fewest txs
  - minimized function to eliminate txs
  - Example: $x\,y + x\,z + x\,v = x\,(y + z + v)$

    5 operations: 3 AND, 2 OR — # txs =

    3 operations: 1 AND, 2 OR — # txs =

- Suggested approach to implement a CMOS logic function
  - create nMOS network
    - invert output
    - reduce function, use DeMorgan to eliminate NANDs and NORs
    - implement using **series for AND** and **parallel for OR**
  - create pMOS network
    - complement each operation in nMOS network

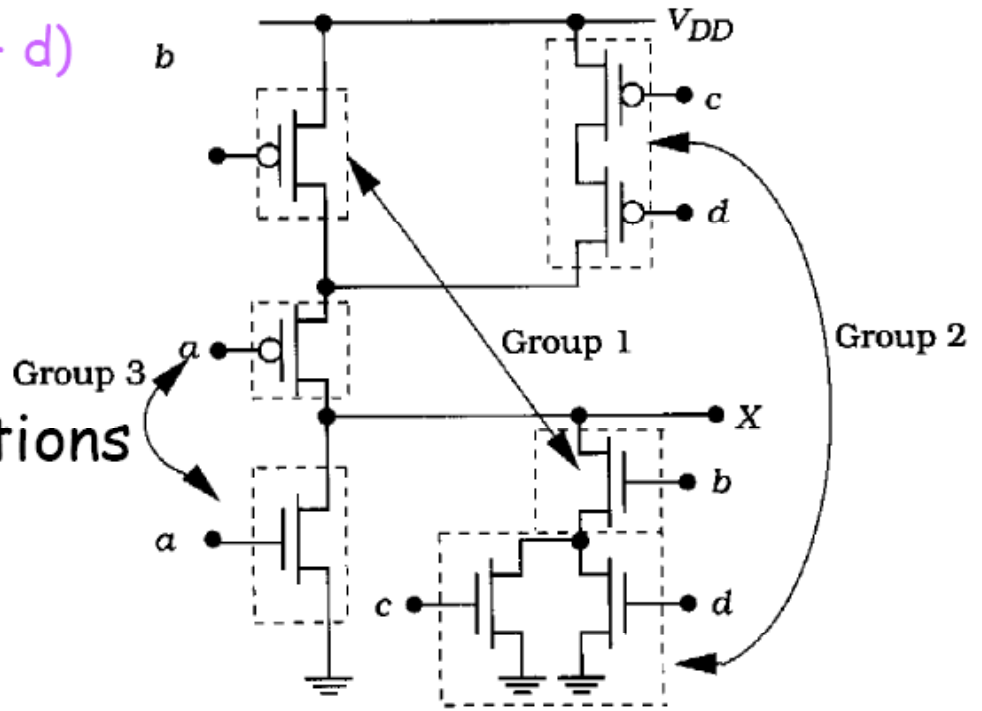
CMOS Example

- Construct the function below in CMOS

  $F = \overline{a + b \cdot (c + d)}$; remember AND operations occur before OR

- Step 1, invert output and find nMOS
  - nMOS; implement $a + b \cdot (c + d)$
  - Group 1: c & d in parallel
  - Group 2: b in series with G1
  - Group 3: a parallel to G2

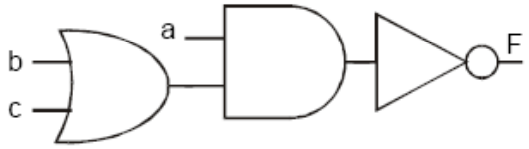
- Step 2, complement operations
  - pMOS
  - Group 1: c & d in series
  - Group 2: b parallel to G1
  - Group 3: a in series with G2



Another example

- Construct a CMOS logic gate which implements the function:

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



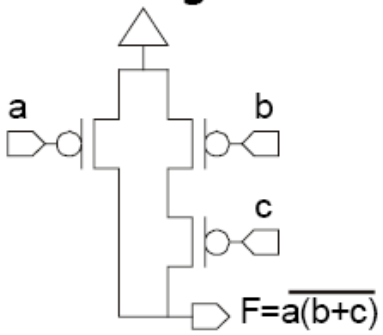
14 transistors

- pMOS
  - Apply DeMorgan expansions

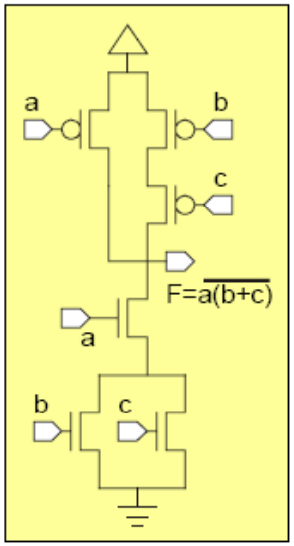
    $F = \overline{a} + \overline{(b + c)}$

    $F = \overline{a} + (\overline{b} \cdot \overline{c})$
  - Invert inputs for pMOS

    $Fp = a + (b \cdot c)$
  - Resulting Schematic



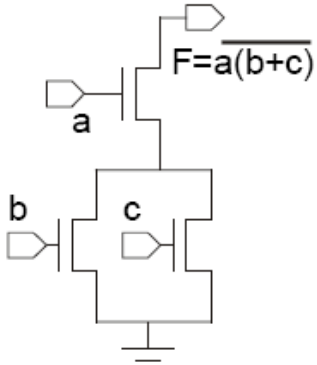
6 transistors



- nMOS
  - Invert output for nMOS

    $Fn = a \cdot (b + c)$
  - Apply DeMorgan

    none needed
  - Resulting Schematic

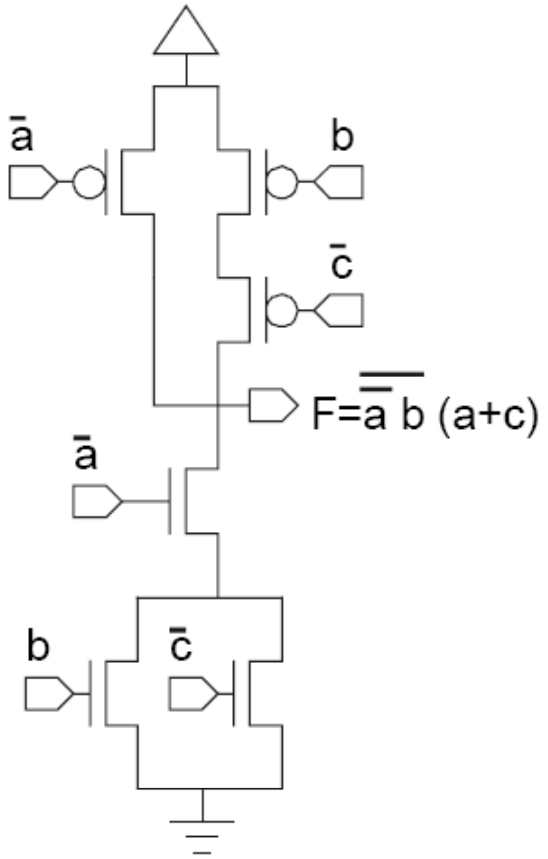


One more example

- Implement the function below by constructing the nMOS network and complementing operations for the pMOS:

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

- nMOS
  - Invert Output
    - $Fn = \overline{\bar{a} \cdot b \cdot (a + c)} = \bar{a} \cdot b + \overline{(a + c)}$
  - Eliminate NANDs and NORs
    - $Fn = \bar{a} \cdot b + (\bar{a} \cdot c)$
  - Reduce Function
    - $Fn = \bar{a} \cdot (b + \bar{c})$
  - Resulting Schematic ?
- Complement operations for pMOS
  - $Fp = a + (b \cdot c)$



CMOS Example

- The rules:
  - NMOS connects to GND, PMOS to power supply Vdd
  - Duality of NMOS and PMOS
  - Rp ~ 2 Rn => PMOS in series is much slower than NMOS in series
- Implement Z using CMOS: Z = (A + BC)’

Another CMOS Example

- Implement F using CMOS: F=A*(B+C)

Summary

- What we covered thus far:
  - Number representations
  - Switches, MOS transistors, Logic gates
  - Boolean algebra
  - Supplement: NMOS and PMOS transistor characteristics
- What is next:
  - Combinational logic:
    - Representations
    - Minimization
    - Implementations