Lecture 5

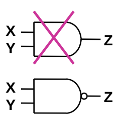
- Logistics
  - HW1 due today
  - HW2 available today, due Wed 1/21
  - Office Hours this week:
    - ➪ Me: Friday 10:00-11:00 CSE 668
    - ➪ TA (Josh): Friday 3:30 CSE 002/3
- Last lecture
  - Logic gates and truth tables
  - Implementing logic functions
  - Canonical forms
- Today's lecture
  - Converting to use NAND and NOR
  - Minimizing functions using Boolean cubes

The "WHY" slide

- Converting to use NAND and NOR
  - NAND and NOR are more efficient gates than AND or OR (and therefore more common). Your computer is built almost exclusively on NAND and NOR gates. It is good to knowhow to convert any logic circuits to a NAND/NOR circuit.
- Pushing bubbles
  - It is always good to remember logical/theoretical concepts visually. This is one way to remember the NAND/NOR conversion and De Morgan's laws easily.
- Logic Simplification
  - If you are building a computer or a cool gadget, you want to optimize on size and efficiency. Having extra unnecessary operations/gates is not great. We teach nice techniques to allow logic simplifications.

NAND/NOR more common/efficient

- CMOS logic gates are more common and efficient in the inverted forms
  - NAND, NOR, NOT
  - Even though Canonical forms discussed so far used AND/OR, NAND/NOR preferred for real hardware implementation

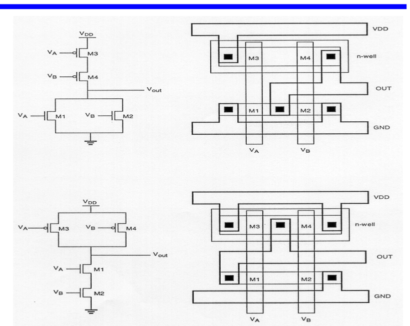


X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

CMOS NAND and NOR Gates

NOR

NAND



NAND and NOR (truth table)

(X + Y)' = X' • Y'
NOR is equivalent to AND with inputs complemented

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

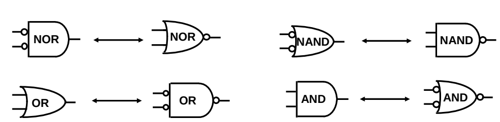
(X • Y)' = X' + Y'
NAND is equivalent to OR with inputs complemented

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

NAND and NOR (logic gates)

- de Morgan's
  - Standard form: A'B' = (A + B)' A' + B' = (AB)'
  - Inverted: A + B = (A'B')' (AB) = (A' + B')'
  - AND with complemented inputs ≡ NOR
  - OR with complemented inputs ≡ NAND
  - OR ≡ NAND with complemented inputs
  - AND ≡ NOR with complemented inputs

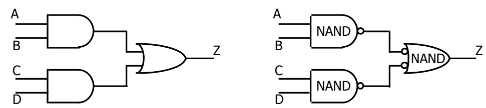
pushing the bubble



Converting to use NAND/NOR

- Introduce inversions ("bubbles")
  - Introduce bubbles in pairs
    - Conserve inversions
    - Do not alter logic function
- Example
  - AND/OR to NAND/NAND

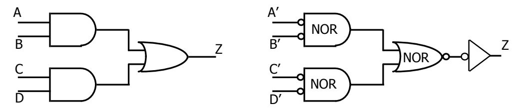
Z = AB + CD
= (A'+B')'+(C'+D')'
= [(A'+B')(C'+D')]'
= [(AB)'(CD)']'



Converting to use NAND/NOR (con't)

- Example: AND/OR network to NOR/NOR

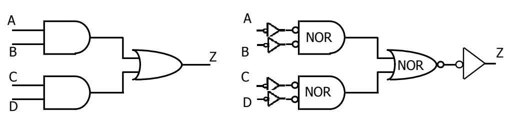
Z = AB+CD
= (A'+B')'+(C'+D')'
= [(A'+B')'+(C'+D')']"
= {[(A'+B')'+(C'+D')']'}'



Converting to use NAND/NOR (con't)

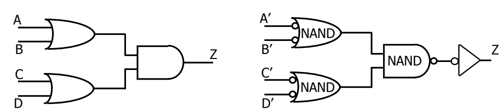
- Example: AND/OR network to NOR/NOR

Z = AB+CD
= (A'+B')'+(C'+D')'
= [(A'+B')'+(C'+D')']"
= {[(A'+B')'+(C'+D')']'}'



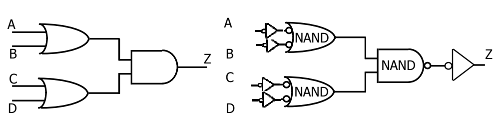
Converting to use NAND/NOR (con't)

- Example: OR/AND to NAND/NAND



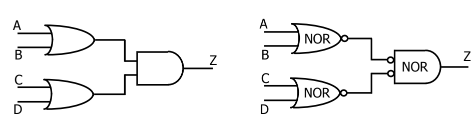
Converting to use NAND/NOR (con't)

- Example: OR/AND to NAND/NAND

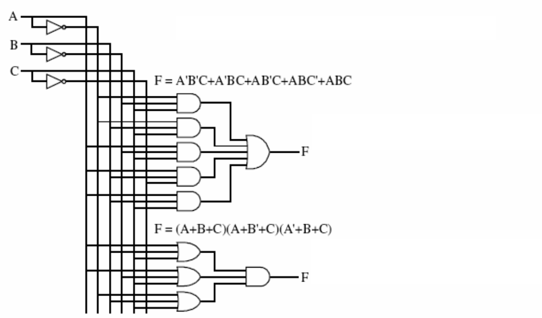


Converting to use NAND/NOR(con't)

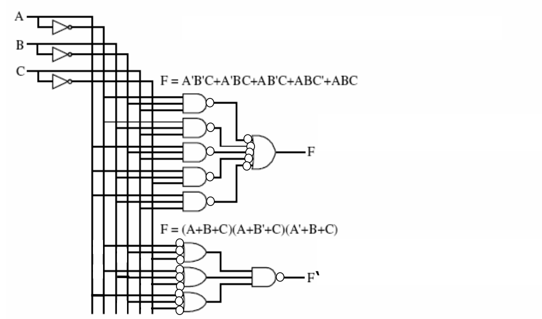
- Example: OR/AND to NOR/NOR



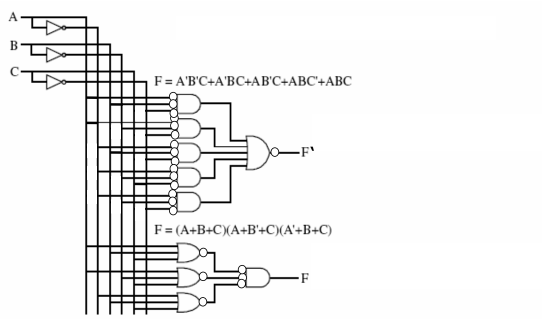
Example of bubble pushing: before pushing



Example of bubble pushing: NAND/NAND



Example of bubble pushing: NOR/NOR



Goal: Minimize two-level logic expression

- Algebraic simplification
  - not an systematic procedure
  - hard to know when we reached the minimum
- Just program it!! Computer-aided design tools
  - require very long computation times (NP hard)
  - heuristic methods employed – "educated guesses"
- Visualization methods are useful
  - our brain is good at figuring simple things out
  - many real-world problems are solvable by hand

Key tool: The Uniting Theorem

- The uniting theorem → A(B'+B) = A
- The approach:
  - Find some variables don't change (the A's above) and others do (the B's above)
  - Eliminate the changing variables (the B's)

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

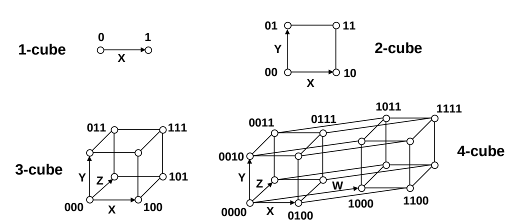
A has the same value in both "on-set" rows ⇒ keep A

B has a different value in the two rows ⇒ eliminate B

F = A'B'+A'B = A'(B+B') = A'

Boolean cubes

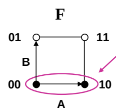
- Visualization tool for the uniting theorem
  - n input variables = n-dimensional "cube"



Mapping truth tables onto Boolean cubes

- ON set = solid nodes
- OFF set = empty nodes

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



Look for on-set adjacent to each other

Sub-cube (a line) comprises two nodes

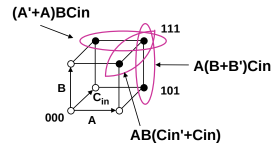
A varies within the sub-cube; B does not

This sub-cube represents B'

Example using Boolean cube

- Binary full-adder carry-out logic
  - On-set is covered by the OR of three 1-subcubes

A	B	Cin	Cout
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	1
1	1	1	1



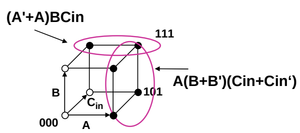
Cout = BCin+AB+ACin

Another example using Boolean cube

Changed one bit from the previous function

- On-set is covered by the OR of one 2-D subcubes and one 3-D subcubes

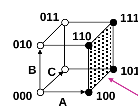
A	B	Cin	D
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



D = BCin+A

M-dimensional cubes in n-dimensional space

- In a 3-cube (three variables):
  - A 0-cube (a single node) yields a term in 3 literals
  - A 1-cube (a line of two nodes) yields a term in 2 literals
  - A 2-cube (a plane of four nodes) yields a term in 1 literal
  - A 3-cube (a cube of eight nodes) yields a constant term "1"



$F(A,B,C) = \Sigma m(4,5,6,7)$

On-set forms a square (a 2-D cube)

A is asserted (true) and unchanging B and C vary

This sub-cube represents the literal A