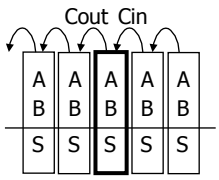
Realizing Boolean logic

- Algebraic expressions to gates
- Mapping between different gates
- Discrete logic gate components (used in lab 1)

A simple example: 1-bit binary adder

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



A	B	Cin	Cout	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

S = A' B' Cin + A' B Cin' + A B' Cin' + A B Cin

Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin

Apply the theorems to simplify expressions

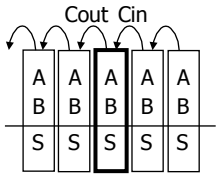
- The theorems of Boolean algebra can simplify expressions
  - e.g., full adder's carry-out function

Cout = A' B Cin + A B' Cin + A B Cin' + A B Cin
= A' B Cin + A B' Cin + A B Cin' + A B Cin + A B Cin
= A' B Cin + A B Cin + A B' Cin + A B Cin' + A B Cin
= (A' + A) B Cin + A B' Cin + A B Cin' + A B Cin
= (1) B Cin + A B' Cin + A B Cin' + A B Cin
= B Cin + A B' Cin + A B Cin' + A B Cin + A B Cin
= B Cin + A B' Cin + A B Cin + A B Cin' + A B Cin
= B Cin + A (B' + B) Cin + A B Cin' + A B Cin
= B Cin + A (1) Cin + A B Cin' + A B Cin
= B Cin + A Cin + A B (Cin' + Cin)
= B Cin + A Cin + A B (1)
= B Cin + A Cin + A B

adding extra terms creates new factoring opportunities

A simple example: 1-bit binary adder

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



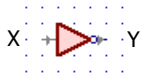
A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Cout = B Cin + A Cin + A B

S = A' B' Cin + A' B Cin' + A B' Cin' + A B Cin
= A' (B' Cin + B Cin') + A (B' Cin' + B Cin)
= A' Z + A Z'
= A xor Z = A xor (B xor Cin)

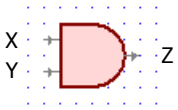
From Boolean expressions to logic gates

- NOT X' $\overline{X}$ ~X X/



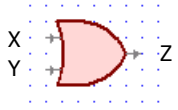
X	Y
0	1
1	0

- AND X • Y XY X ∧ Y



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

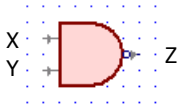
- OR X + Y X ∨ Y



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

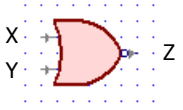
From Boolean expressions to logic gates (cont'd)

- NAND



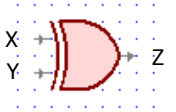
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

- NOR



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

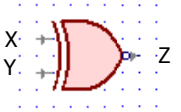
- XOR
  X ⊕ Y



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

X xor Y = X Y' + X' Y
X or Y but not both
("inequality", "difference")

- XNOR
  X = Y



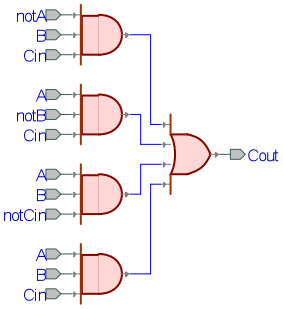
X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

X xnor Y = X Y + X' Y'
X and Y are the same
("equality", "coincidence")

Full adder: Carry-out

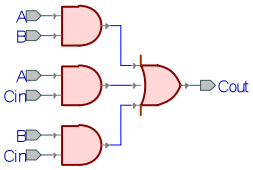
Before Boolean minimization

Cout = A'BCin + AB'Cin
+ ABCin' + ABCin



After Boolean minimization

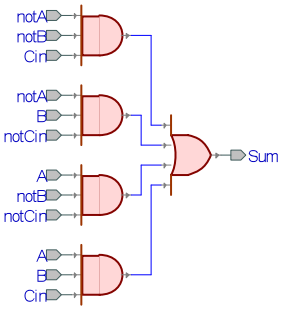
Cout = BCin + ACin + AB



Full adder: Sum

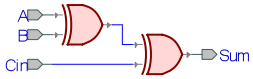
Before Boolean minimization

Sum = A'B'Cin + A'BCin'
+ AB'Cin' + ABCin

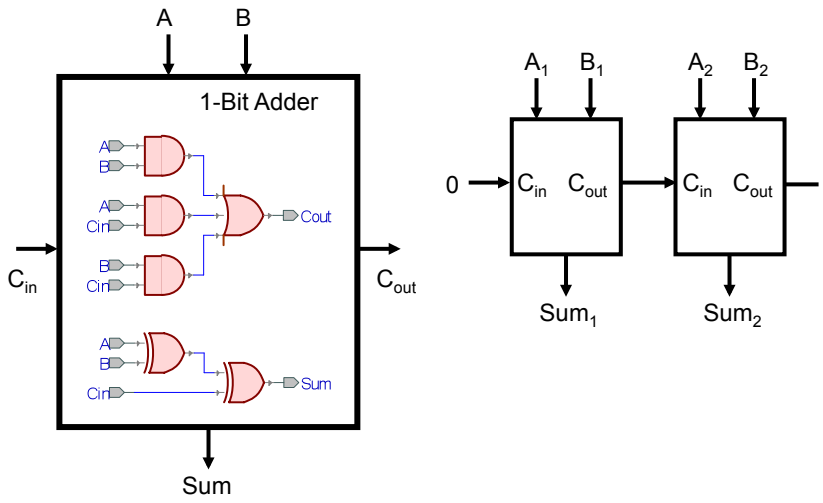


After Boolean minimization

Sum = (A⊕B) ⊕ Cin



Preview: A 2-bit ripple-carry adder



Mapping truth tables to logic gates

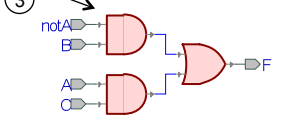
- Given a truth table:
  1. Write the Boolean expression
  2. Minimize the Boolean expression
  3. Draw as gates
  4. Map to available gates

A	B	C	F
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

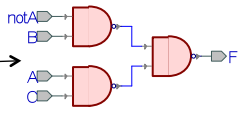
(1)

(2) F = A'BC'+A'BC+AB'C+ABC
= A'B(C'+C)+AC(B'+B)
= A'B+AC

(3)

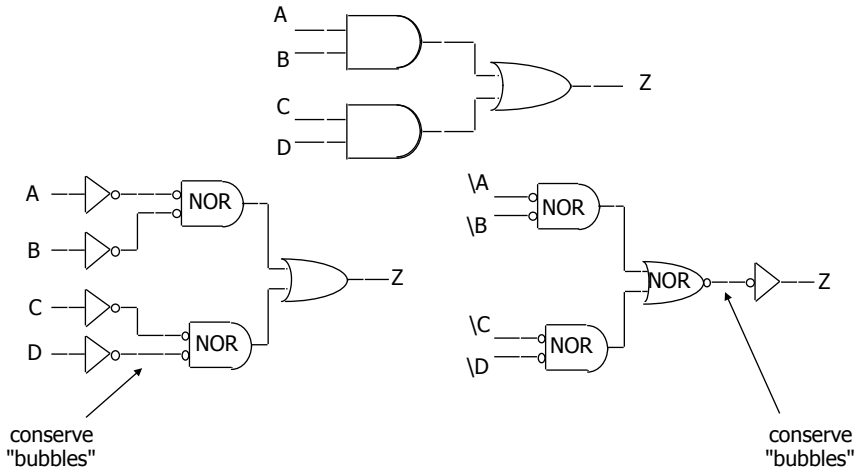


(4)



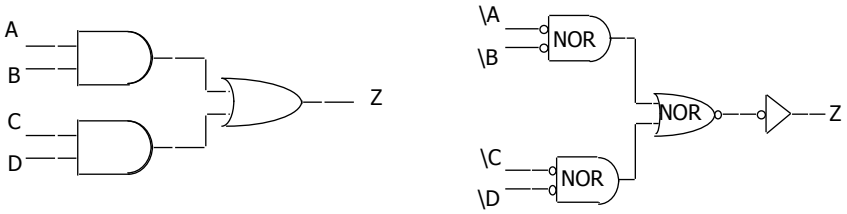
Conversion between gate types

- Example: map AND/OR network to NOR-only network



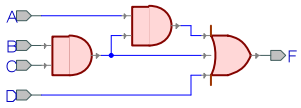
Conversion between gate types (cont'd)

- Example: verify equivalence of two forms



$$
\begin{aligned}
Z &= \{\ [\ (A' + B')' + (C' + D')'\ ]'\ \}' \\
&= \{\ \ (A' + B')\ \bullet\ (C' + D')\ \ \}' \\
&= \ \ (A' + B')' + (C' + D')' \\
&= \ \ (A\ \bullet\ B)\ +\ (C\ \bullet\ D)
\end{aligned}
$$

Activity: convert to NAND gates



Example: tally circuit (outputs # of 1s in inputs)

X1	X2	X3	T2	T1
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

T1 = X1′ X2′ X3 + X1′ X2 X3′
+ X1 X2′ X3′ + X1 X2 X3

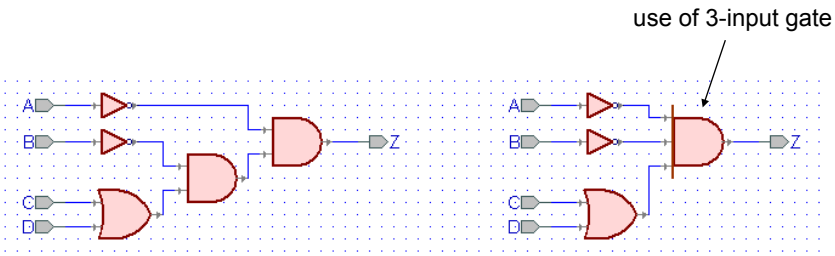
= (X1′ X2′ + X1 X2) X3
+ (X1′ X2 + X1 X2′) X3′
= (X1 xor X2)′ X3
+ (X1 xor X2) X3′
= (X1 xor X2) xor X3

T2 = X1′ X2 X3 + X1 X2′ X3
+ X1 X2 X3′ + X1 X2 X3

= X1′ (X2 X3) + X1 (X2 + X3)

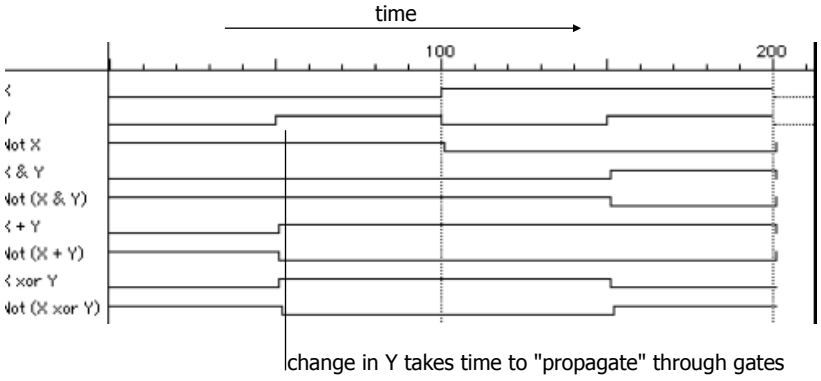
From Boolean expressions to logic gates

- More than one way to map expressions to gates
  - e.g., Z = A' • B' • (C + D) = (A' • (B' • (C + D)))



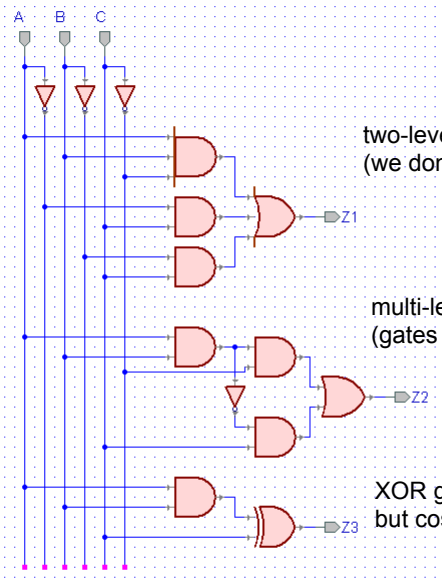
Waveform view of logic functions

- Just a sideways truth table
  - but note how edges don't line up exactly
  - it takes time for a gate to switch its output!



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



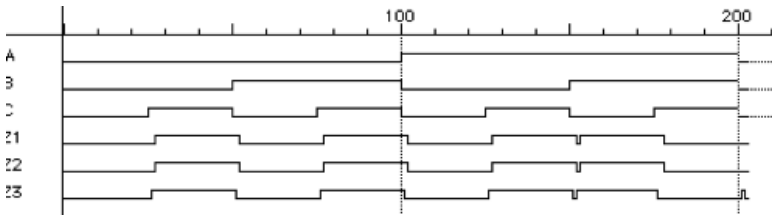
two-level realization
(we don't count NOT gates)

multi-level realization
(gates with fewer inputs)

XOR gate (easier to draw
but costlier to build)

Are all realizations equivalent?

- Under the same input stimuli, the three alternative implementations have almost the same waveform behavior
  - delays are different
  - glitches (hazards) may arise – these could be bad, it depends
  - variations due to differences in number of gate levels and structure
- The three implementations are functionally equivalent



Which realization is best?

- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transistors per literal
    - why not count inverters?
  - fewer literals means less transistors
    - smaller circuits
  - fewer inputs implies faster gates
    - gates are smaller and thus also faster
  - fan-ins (# of gate inputs) are limited in some technologies
    - the programmable logic we'll be using later in the quarter
- Reduce number of gates
  - fewer gates (and the packages they come in) means smaller circuits
    - directly influences manufacturing costs

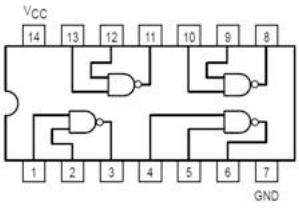
Which realization is best? (cont'd)

- Reduce number of levels of gates
  - fewer level of gates implies reduced signal propagation delays
  - minimum delay configuration typically requires more gates
    - wider, less deep circuits
- Hazards/glitches
  - one without hazards may be preferable/necessary
- How do we explore tradeoffs between increased circuit delay and size?
  - automated tools to generate different solutions
  - logic minimization: reduce number of gates and complexity
  - logic optimization: reduction while trading off against delay

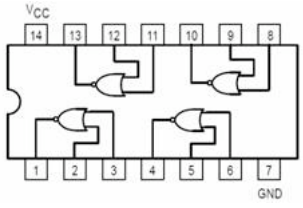
Random logic gates

- Transistors quickly integrated into logic gates (1960s)
- Catalog of common gates (1970s)
  - Texas Instruments Logic Data Book – the yellow "bible"
  - all common packages listed and characterized (delays, power)
  - typical packages:
    - in 14-pin IC: 6-inverters, 4 NAND gates, 4 XOR gates
- Today, very few of these parts are still in use
- However, parts libraries exist for chip design
  - designers reuse already characterized logic gates on chips
  - same reasons as before
  - difference is that the parts don't exist in physical inventory – created as needed

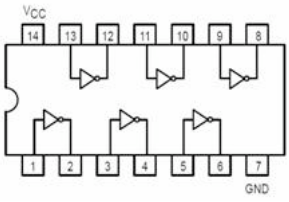
Some logic gate components



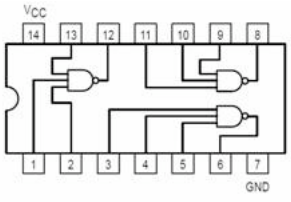
Quad 2-input NANDs – '00



Quad 2-input NORs – '02



6 inverters (NOTs) – '04



3 3-input NANDs – '10