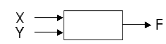
Combinational logic

- Basic logic
  - Boolean algebra, proofs by re-writing, proofs by perfect induction
  - Logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Logic realization
  - two-level logic and canonical forms, incompletely specified functions
  - multi-level logic, converting between ANDs and ORs
- Simplification
  - uniting theorem
  - transformations on networks of Boolean functions
- Time behavior
- Hardware description languages

Possible logic functions of two variables

- There are 16 possible functions of 2 input variables:
  - in general, there are 2**(2**n) functions of n inputs



X	Y	16 possible functions (F0–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 X and Y X Y X xor Y X or Y X nor Y not (X or Y) X = Y not Y not X X nand Y not (X and Y) 1

Cost of different logic functions

- Different functions are easier or harder to implement
  - each has a cost associated with the number of switches needed
  - 0 (F0) and 1 (F15): require 0 switches, directly connect output to low/high
  - X (F3) and Y (F5): require 0 switches, output is one of inputs
  - X' (F12) and Y' (F10): require 2 switches for "inverter" or NOT-gate
  - X nor Y (F4) and X nand Y (F14): require 4 switches
  - X or Y (F7) and X and Y (F1): require 6 switches
  - X = Y (F9) and X ⊕ Y (F6): require 16 switches
  - thus, because NOT, NOR, and NAND are the cheapest they are the functions we implement the most in practice

Minimal set of functions

- Can we implement all logic functions from NOT, NOR, and NAND?
  - For example, implementing X and Y
    is the same as implementing not (X nand Y)
- In fact, we can do it with only NOR or only NAND
  - NOT is just a NAND or a NOR with both inputs tied together

X	Y	X nor Y
0	0	1
1	1	0

X	Y	X nand Y
0	0	1
1	1	0

  - and NAND and NOR are "duals",
    that is, its easy to implement one using the other

X nand Y ≡ not ((not X) nor (not Y))

X nor Y ≡ not ((not X) nand (not Y))

- But lets not move too fast . . .
  - lets look at the mathematical foundation of logic

An algebraic structure

- An algebraic structure consists of
  - a set of elements B
  - binary operations { + , • }
  - and a unary operation { ′ }
  - such that the following axioms hold:

1. the set B contains at least two elements: a, b		
2. closure:	a + b is in B	a • b is in B
3. commutativity:	a + b = b + a	a • b = b • a
4. associativity:	a + (b + c) = (a + b) + c	a • (b • c) = (a • b) • c
5. identity:	a + 0 = a	a • 1 = a
6. distributivity:	a + (b • c) = (a + b) • (a + c)	a • (b + c) = (a • b) + (a • c)
7. complementarity:	a + a′ = 1	a • a′ = 0

Boolean algebra

- Boolean algebra
  - B = {0, 1}
  - variables
  - + is logical OR, • is logical AND
  - ′ is logical NOT
- All algebraic axioms hold

Logic functions and Boolean algebra

- Any logic function that can be expressed as a truth table can be written as an expression in Boolean algebra using the operators: ′, +, and •

X	Y	X • Y
0	0	0
0	1	0
1	0	0
1	1	1

X	Y	X′	X′ • Y
0	0	1	0
0	1	1	1
1	0	0	0
1	1	0	0

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

(X • Y) + (X′ • Y′) ≡ X = Y

Boolean expression that is true when the variables X and Y have the same value and false, otherwise

X, Y are Boolean algebra variables

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')

- 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')

- 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. $X + Y + ... \Leftrightarrow X \bullet Y \bullet ...$
- generalized duality:
  - 17. $f(X_1,X_2,...,X_n,0,1,+,\bullet) \Leftrightarrow f(X_1,X_2,...,X_n,1,0,\bullet,+)$
- Different than deMorgan's Law
  - this is a statement about theorems
  - this is not a way to manipulate (re-write) expressions

Proving theorems (rewriting)

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

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

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

identity (1D)	$X + X \bullet Y$	$= X \bullet 1 + X \bullet Y$
distributivity (8)	$X \bullet 1 + X \bullet Y$	$= X \bullet (1 + Y)$
identity (2)	$X \bullet (1 + Y)$	$= X \bullet (1)$
identity (1D)	$X \bullet (1)$	$= X$ ➨

Activity

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

Proving theorems (perfect induction)

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

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

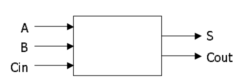
X	Y	X′	Y′	(X + Y)′	X′ • Y′
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

(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

A simple example: 1-bit binary adder

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



A	B	Cin	S	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

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 Boolean expressions
  - e.g., full adder's carry-out function (same rules apply to any function)

Cout = A′ B Cin + A B′ Cin + A B Cin′ + A B Cin
= A′ B Cin + A B′ Cin + A B Cin′ + $\boxed{\text{A B Cin + A B Cin}}$
= 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′ + $\boxed{\text{A B Cin + A B Cin}}$
= 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

Activity

⌘ Fill in the truth-table for a circuit that checks that a 4-bit number is divisible by 2, 3, or 5

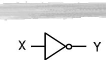
X8	X4	X2	X1	By2	By3	By5
0	0	0	0	1	1	1
0	0	0	1	0	0	0
0	0	1	0	1	0	0
0	0	1	1	0	1	0

⌘ Write down Boolean expressions for By2, By3, and By5

Activity

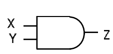
From Boolean expressions to logic gates

⌘ NOT X' $\overline{X}$ ~X



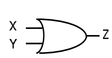
X	Y
0	1
1	0

⌘ AND $X \bullet Y$ XY $X \wedge Y$



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

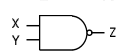
⌘ OR $X + Y$ $X \vee Y$



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

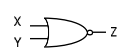
From Boolean expressions to logic gates (cont'd)

⌘ NAND



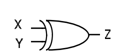
X	Y	Z
0	0	1
0	1	1
1	0	1
1	1	0

⌘ NOR



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

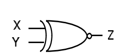
⌘ XOR
$X \oplus Y$



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
$X = Y$

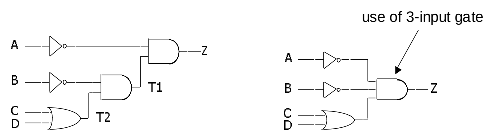


X	Y	Z
0	0	1
0	1	0
1	0	0
1	1	1

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

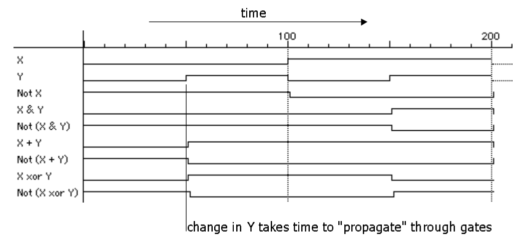
From Boolean expressions to logic gates (cont'd)

- More than one way to map expressions to gates
  - e.g., $Z = A' \bullet B' \bullet (C + D) = (A' \bullet \underbrace{(B' \bullet \underbrace{(C + D)}_{T2}))}_{T1}$



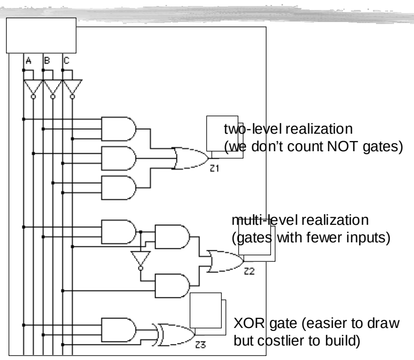
Waveform view of logic functions

- 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



Which realization is best?

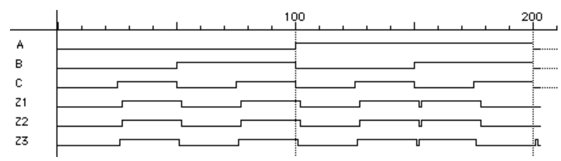
- Reduce number of inputs
  - literal: input variable (complemented or not)
    - can approximate cost of logic gate as 2 transitors per literal
    - why not count inverters?
  - fewer literals means less transistors
    - smaller circuits
  - fewer inputs implies faster gates
    - gates are smaller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

Which is the best realization? (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
- 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

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
  - variations due to differences in number of gate levels and structure
- The three implementations are functionally equivalent



Implementing Boolean functions

- Technology independent
  - canonical forms
  - two-level forms
  - multi-level forms
- Technology choices
  - packages of a few gates
  - regular logic
  - two-level programmable logic
  - multi-level programmable logic

Canonical forms

- Truth table is the unique signature of a Boolean function
- Many alternative gate realizations may have the same truth table
- Canonical forms
  - standard forms for a Boolean expression
  - provides a unique algebraic signature

Sum-of-products canonical forms

- Also known as disjunctive normal form
- Also known as minterm expansion

F = *001* *011* *101* *110* *111*

$F = A'B'C + A'BC + AB'C + ABC' + ABC$

A	B	C	F	F'
0	0	0	0	1
0	0	1	1	0
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	0

$F' = A'B'C' + A'BC' + AB'C'$

Sum-of-products canonical form (cont'd)

- Product term (or minterm)
  - ANDed product of literals – input combination for which output is true
  - each variable appears exactly once, in true or inverted form (but not both)

A	B	C	minterms	
0	0	0	A'B'C'	m0
0	0	1	A'B'C	m1
0	1	0	A'BC'	m2
0	1	1	A'BC	m3
1	0	0	AB'C'	m4
1	0	1	AB'C	m5
1	1	0	ABC'	m6
1	1	1	ABC	m7

short-hand notation for minterms of 3 variables

F in canonical form:

$F(A, B, C) = \Sigma m(1,3,5,6,7)$
$= m1 + m3 + m5 + m6 + m7$
$= A'B'C + A'BC + AB'C + ABC' + ABC$

canonical form $\neq$ minimal form

$F(A, B, C) = A'B'C + A'BC + AB'C + ABC + ABC'$
$= (A'B' + A'B + AB' + AB)C + ABC'$
$= ((A' + A)(B' + B))C + ABC'$
$= C + ABC'$
$= ABC' + C$
$= AB + C$

Product-of-sums canonical form

- Also known as conjunctive normal form
- Also known as maxterm expansion

F = *000* *010* *100*

$F = (A + B + C)\ (A + B' + C)\ (A' + B + C)$

A	B	C	F	F'
0	0	0	0	1
0	0	1	1	0
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	0

$F' = (A + B + C')\ (A + B' + C')\ (A' + B + C')\ (A' + B' + C)\ (A' + B' + C')$

Product-of-sums canonical form (cont'd)

- Sum term (or maxterm)
  - ORed sum of literals – input combination for which output is false
  - each variable appears exactly once, in true or inverted form (but not both)

A	B	C	maxterms	
0	0	0	A+B+C	M0
0	0	1	A+B+C'	M1
0	1	0	A+B'+C	M2
0	1	1	A+B'+C'	M3
1	0	0	A'+B+C	M4
1	0	1	A'+B+C'	M5
1	1	0	A'+B'+C	M6
1	1	1	A'+B'+C'	M7

short-hand notation for maxterms of 3 variables

F in canonical form:

$F(A, B, C) = \Pi M(0,2,4)$
$= M0 \bullet M2 \bullet M4$
$= (A + B + C)\ (A + B' + C)\ (A' + B + C)$

canonical form $\neq$ minimal form

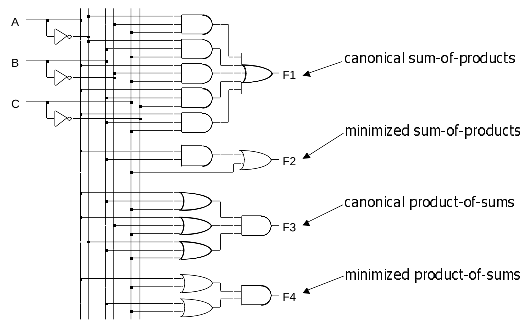
$F(A, B, C) = (A + B + C)\ (A + B' + C)\ (A' + B + C)$
$= (A + B + C)\ (A + B' + C)$
$\quad (A + B + C)\ (A' + B + C)$
$= (A + C)\ (B + C)$

S-o-P, P-o-S, and de Morgan's theorem

- Sum-of-products
  - $F' = A'B'C' + A'BC' + AB'C'$
- Apply de Morgan's
  - $(F')' = (A'B'C' + A'BC' + AB'C')'$
  - $F = (A + B + C) (A + B' + C) (A' + B + C)$

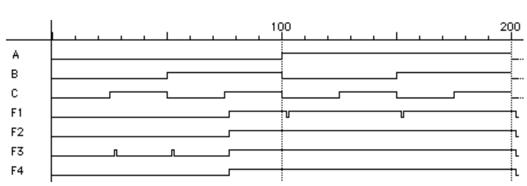
- Product-of-sums
  - $F' = (A + B + C') (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' + C')$
- Apply de Morgan's
  - $(F')' = ((A + B + C')(A + B' + C')(A' + B + C')(A' + B' + C)(A' + B' + C'))'$
  - $F = A'B'C + A'BC + AB'C + ABC' + ABC$

Four alternative two-level implementations of F = AB + C



Waveforms for the four alternatives

- Waveforms are essentially identical
  - except for timing hazards (glitches)
  - delays almost identical (modeled as a delay per level, not type of gate or number of inputs to gate)



Mapping between canonical forms

- Minterm to maxterm conversion
  - use maxterms whose indices do not appear in minterm expansion
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7) = \Pi M(0,2,4)$
- Maxterm to minterm conversion
  - use minterms whose indices do not appear in maxterm expansion
  - e.g., $F(A,B,C) = \Pi M(0,2,4) = \Sigma m(1,3,5,6,7)$
- Minterm expansion of F to minterm expansion of F'
  - use minterms whose indices do not appear
  - e.g., $F(A,B,C) = \Sigma m(1,3,5,6,7)$ $F'(A,B,C) = \Sigma m(0,2,4)$
- Maxterm expansion of F to maxterm expansion of F'
  - use maxterms whose indices do not appear
  - e.g., $F(A,B,C) = \Pi M(0,2,4)$ $F'(A,B,C) = \Pi M(1,3,5,6,7)$

Incompleteley specified functions

- Example: binary coded decimal increment by 1
  - BCD digits encode the decimal digits 0 – 9 in the bit patterns 0000 – 1001

A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	X	X	X	X
1	0	1	1	X	X	X	X
1	1	0	0	X	X	X	X
1	1	0	1	X	X	X	X
1	1	1	0	X	X	X	X
1	1	1	1	X	X	X	X

off-set of W

on-set of W

don't care (DC) set of W

these inputs patterns should never be encountered in practice – **"don't care"** about associated output values, can be exploited in minimization

Notation for incompletely specified functions

- Don't cares and canonical forms
  - so far, only represented on-set
  - also represent don't-care-set
  - need two of the three sets (on-set, off-set, dc-set)
- Canonical representations of the BCD increment by 1 function:
  - Z = m0 + m2 + m4 + m6 + m8 + d10 + d11 + d12 + d13 + d14 + d15
  - Z = Σ [m(0,2,4,6,8) + d(10,11,12,13,14,15)]
  - Z = M1 • M3 • M5 • M7 • M9 • D10 • D11 • D12 • D13 • D14 • D15
  - Z = Π [M(1,3,5,7,9) • D(10,11,12,13,14,15)]

Simplification of two-level combinational logic

- Finding a minimal sum of products or product of sums realization
  - exploit don't care information in the process
- Algebraic simplification
  - not an algorithmic/systematic procedure
  - how do you know when the minimum realization has been found?
- Computer-aided design tools
  - precise solutions require very long computation times, especially for functions with many inputs (> 10)
  - heuristic methods employed – "educated guesses" to reduce amount of computation and yield good if not best solutions
- Hand methods still relevant
  - to understand automatic tools and their strengths and weaknesses
  - ability to check results (on small examples)

The uniting theorem

- Key tool to simplification: A (B' + B) = A
- Essence of simplification of two-level logic
  - find two element subsets of the ON-set where only one variable changes its value – this single varying variable can be eliminated and a single product term used to represent both elements

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

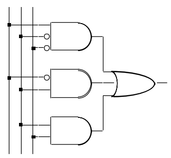
A	B	F
0	0	1
0	1	0
1	0	1
1	1	0

B has the same value in both on-set rows – B remains

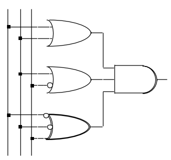
A has a different value in the two rows – A is eliminated

Implementations of two-level logic

- Sum-of-products
  - AND gates to form product terms (minterms)
  - OR gate to form sum

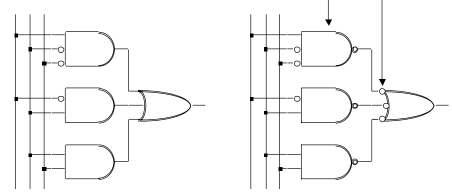


- Product-of-sums
  - OR gates to form sum terms (maxterms)
  - AND gates to form product



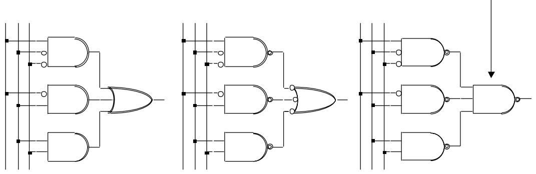
Two-level logic using NAND gates

- Replace minterm AND gates with NAND gates
- Place compensating inversion at inputs of OR gate



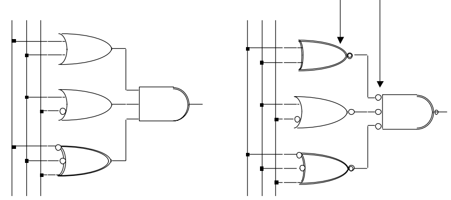
Two-level logic using NAND gates (cont'd)

- OR gate with inverted inputs is a NAND gate
  - de Morgan's: A' + B' = (A • B)'
- Two-level NAND-NAND network
  - inverted inputs are not counted
  - in a typical circuit, inversion is done once and signal distributed



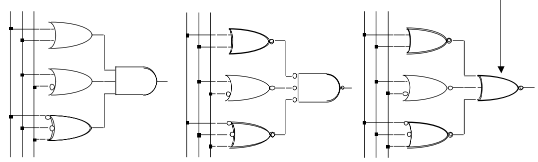
Two-level logic using NOR gates

- Replace maxterm OR gates with NOR gates
- Place compensating inversion at inputs of AND gate



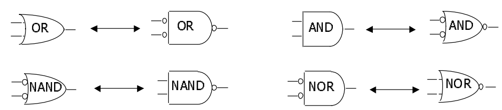
Two-level logic using NOR gates (cont'd)

- AND gate with inverted inputs is a NOR gate
  - de Morgan's: A' • B' = (A + B)'
- Two-level NOR-NOR network
  - inverted inputs are not counted
  - in a typical circuit, inversion is done once and signal distributed



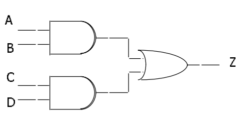
Two-level logic using NAND and NOR gates

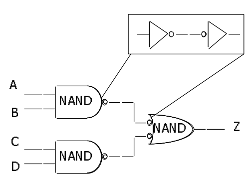
- NAND-NAND and NOR-NOR networks
  - de Morgan's law: (A + B)' = A' • B' (A • B)' = A' + B'
  - written differently: A + B = (A' • B')' (A • B) = (A' + B')'
- In other words ––
  - OR is the same as NAND with complemented inputs
  - AND is the same as NOR with complemented inputs
  - NAND is the same as OR with complemented inputs
  - NOR is the same as AND with complemented inputs



Conversion between forms

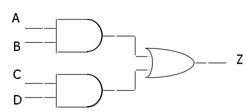
- Convert from networks of ANDs and ORs to networks of NANDs and NORs
  - introduce appropriate inversions ("bubbles")
- Each introduced "bubble" must be matched by a corresponding "bubble"
  - conservation of inversions
  - do not alter logic function
- Example: AND/OR to NAND/NAND

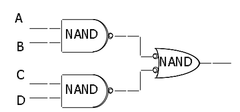




Conversion between forms (cont'd)

- Example: verify equivalence of two forms

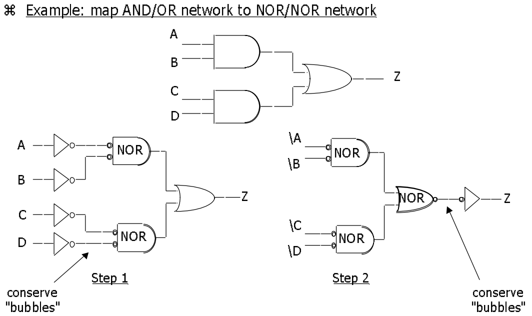




$$
\begin{aligned}
Z &= [\,(A \bullet B)' \bullet (C \bullet D)'\,]' \\
&= [\,(A' + B') \bullet (C' + D')\,]' \\
&= [\,(A' + B')' + (C' + D')'\,] \\
&= (A \bullet B) + (C \bullet D) \;\checkmark
\end{aligned}
$$

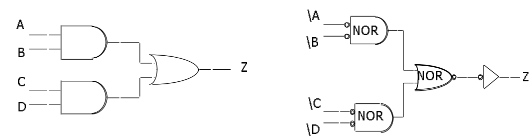
Conversion between forms (cont'd)

⌘ Example: map AND/OR network to NOR/NOR network



Conversion between forms (cont'd)

⌘ Example: verify equivalence of two forms

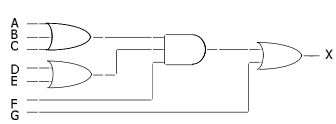


$$
\begin{aligned}
Z &= \{ \; [(A' + B')' + (C' + D')' \;]' \; \}' \\
&= \{ \quad (A' + B') \bullet (C' + D') \quad \}' \\
&= \quad (A' + B')' + (C' + D')' \\
&= \quad (A \bullet B) + (C \bullet D) \;\; ➨
\end{aligned}
$$

Multi-level logic

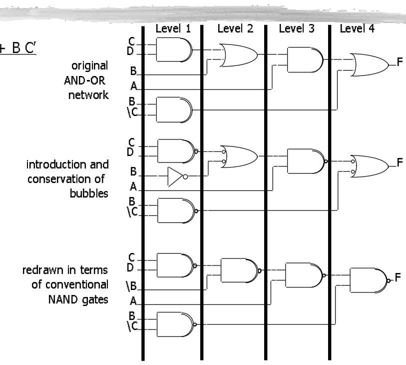
⌘ x = A D F + A E F + B D F + B E F + C D F + C E F + G
- reduced sum-of-products form – already simplified
- 6 x 3-input AND gates + 1 x 7-input OR gate (that may not even exist!)
- 25 wires (19 literals plus 6 internal wires)

⌘ x = (A + B + C) (D + E) F + G
- factored form – not written as two-level S-o-P
- 1 x 3-input OR gate, 2 x 2-input OR gates, 1 x 3-input AND gate
- 10 wires (7 literals plus 3 internal wires)



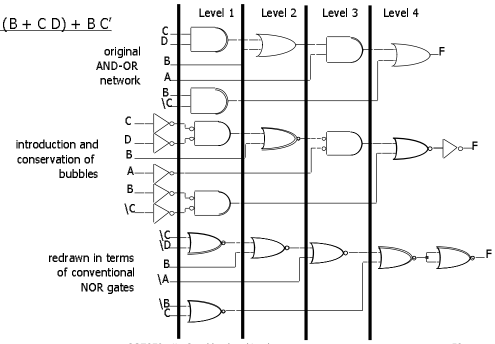
Conversion of multi-level logic to NAND gates

⌘ F = A (B + C D) + B C'



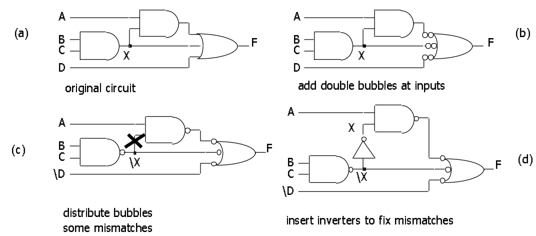
Conversion of multi-level logic to NORs

⌘ F = A (B + C D) + B C'



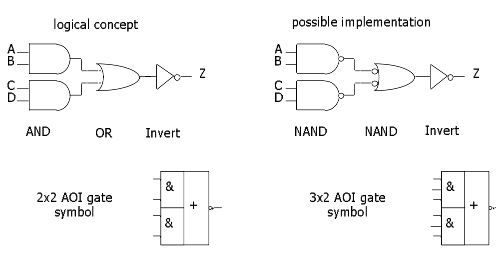
Conversion between forms

⌘ Example



AND-OR-invert gates

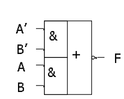
⌘ AOI function: three stages of logic — AND, OR, Invert
- multiple gates "packaged" as a single circuit block



Conversion to AOI forms

⌘ General procedure to place in AOI form
- compute the complement of the function in sum-of-products form
- by grouping the 0s in the Karnaugh map

⌘ Example: XOR implementation — A xor B = A' B + A B'
- AOI form: F = (A' B' + A B)'



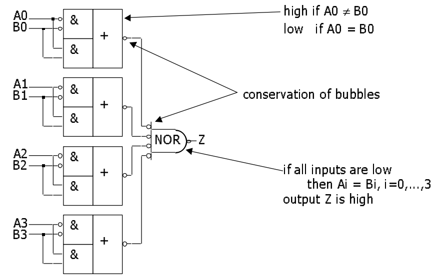
Examples of using AOI gates

- Example:
  - F = B C′ + A C′ + A B
  - F′ = A′ B′ + A′ C + B′ C
  - Implemented by 2-input 3-stack AOI gate
  - F = (A + B) (A + C′) (B + C′)
  - F′ = (B′ + C) (A′ + C) (A′ + B′)
  - Implemented by 2-input 3-stack OAI gate
- Example: 4-bit equality function
  - Z = (A0 B0 + A0′ B0′)(A1 B1 + A1′ B1′)(A2 B2 + A2′ B2′)(A3 B3 + A3′ B3′)

each implemented in a single 2x2 AOI gate

Examples of using AOI gates (cont'd)

- Example: AOI implementation of 4-bit equality function



Summary for multi-level logic

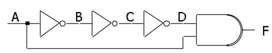
- Advantages
  - circuits may be smaller
  - gates have smaller fan-in
  - circuits may be faster
- Disadvantages
  - more difficult to design
  - tools for optimization are not as good as for two-level
  - analysis is more complex

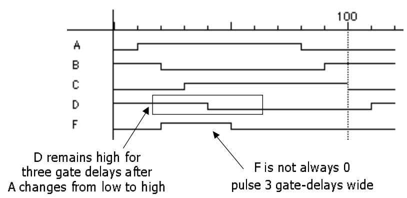
Time behavior of combinational networks

- Waveforms
  - visualization of values carried on signal wires over time
  - useful in explaining sequences of events (changes in value)
- Simulation tools are used to create these waveforms
  - input to the simulator includes gates and their connections
  - input stimulus, that is, input signal waveforms
- Some terms
  - gate delay — time for change at input to cause change at output
    - min delay – typical/nominal delay – max delay
    - careful designers design for the worst case
  - rise time — time for output to transition from low to high voltage
  - fall time — time for output to transition from high to low voltage
  - pulse width — time that an output stays high or stays low between changes

Momentary changes in outputs

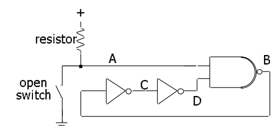
- Can be useful — pulse shaping circuits
- Can be a problem — incorrect circuit operation (glitches/hazards)
- Example: pulse shaping circuit
  - A' • A = 0
  - delays matter in function

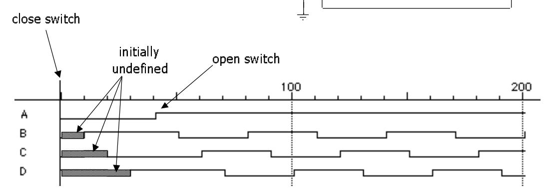




Oscillatory behavior

- Another pulse shaping circuit





Hardware description languages

- Describe hardware at varying levels of abstraction
- Structural description
  - textual replacement for schematic
  - hierarchical composition of modules from primitives
- Behavioral/functional description
  - describe what module does, not how
  - synthesis generates circuit for module
- Simulation semantics

HDLs

- Abel (circa 1983) - developed by Data-I/O
  - targeted to programmable logic devices
  - not good for much more than state machines
- ISP (circa 1977) - research project at CMU
  - simulation, but no synthesis
- Verilog (circa 1985) - developed by Gateway (absorbed by Cadence)
  - similar to Pascal and C
  - delays is only interaction with simulator
  - fairly efficient and easy to write
  - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
  - similar to Ada (emphasis on re-use and maintainability)
  - simulation semantics visible
  - very general but verbose
  - IEEE standard

Verilog

- Supports structural and behavioral descriptions
- Structural
  - explicit structure of the circuit
  - e.g., each logic gate instantiated and connected to others
- Behavioral
  - program describes input/output behavior of circuit
  - many structural implementations could have same behavior
  - e.g., different implementation of one Boolean function
- We'll only be using behavioral Verilog in DesignWorks
  - rely on schematic when we want structural descriptions

Structural model

```
module xor_gate (out, a, b);
  input     a, b;
  output    out;
  wire      abar, bbar, t1, t2;

  inverter invA (abar, a);
  inverter invB (bbar, b);
  and_gate and1 (t1, a, bbar);
  and_gate and2 (t2, b, abar);
  or_gate  or1 (out, t1, t2);

endmodule
```


Simple behavioral model

- Continuous assignment

```
module xor_gate (out, a, b);
  input         a, b;
  output        out;
  reg           out;

  assign #6 out = a ^ b;

endmodule
```

simulation register - keeps track of value of signal

delay from input change to output change

Simple behavioral model

- always block

```
module xor_gate (out, a, b);
  input         a, b;
  output        out;
  reg           out;

  always @(a or b) begin
     #6 out = a ^ b;
  end

endmodule
```

specifies when block is executed ie. triggered by which signals

Driving a simulation

```
module stimulus (x, y);
  output        x, y;
  reg [1:0]     cnt;

  initial begin
    cnt = 0;
    repeat (4) begin
      #10 cnt = cnt + 1;
      $display ("@ time=%d, x=%b, y=%b, cnt=%b",
        $time, x, y, cnt); end
    #10 $finish;
  end

  assign x = cnt[1];
  assign y = cnt[0];
endmodule
```

2-bit vector

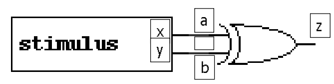
initial block executed only once at start of simulation

print to a console

directive to stop simulation

Complete Simulation

⌘ Instantiate stimulus component and device to test in a schematic



Comparator Example

```
module Compare1 (A, B, Equal, Alarger, Blarger);
  input     A, B;
  output    Equal, Alarger, Blarger;

  assign #5 Equal = (A & B) | (~A & ~B);
  assign #3 Alarger = (A & ~B);
  assign #3 Blarger = (~A & B);
endmodule
```


More Complex Behavioral Model

```
module life (n0, n1, n2, n3, n4, n5, n6, n7, self, out);
  input     n0, n1, n2, n3, n4, n5, n6, n7, self;
  output    out;
  reg       out;
  reg [7:0] neighbors;
  reg [3:0] count;
  reg [3:0] i;

  assign neighbors = {n7, n6, n5, n4, n3, n2, n1, n0};

  always @(neighbors or self) begin
    count = 0;
    for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
    out = (count == 3);
    out = out | ((self == 1) & (count == 2));
  end

endmodule
```

Hardware Description Languages vs. Programming Languages

- Program structure
  - instantiation of multiple components of the same type
  - specify interconnections between modules via schematic
  - hierarchy of modules (only leaves can be HDL in DesignWorks)
- Assignment
  - continuous assignment (logic always computes)
  - propagation delay (computation takes time)
  - timing of signals is important (when does computation have its effect)
- Data structures
  - size explicitly spelled out - no dynamic structures
  - no pointers
- Parallelism
  - hardware is naturally parallel (must support multiple threads)
  - assignments can occur in parallel (not just sequentially)

Hardware Description Languages and Combinational Logic

- Modules - specification of inputs, outputs, bidirectional, and internal signals
- Continuous assignment - a gate's output is a function of its inputs at all times (doesn't need to wait to be "called")
- Propagation delay- concept of time and delay in input affecting gate output
- Composition - connecting modules together with wires
- Hierarchy - modules encapsulate functional blocks
- Specification of don't care conditions (accomplished by setting output to "x")

Combinational logic summary

- Logic functions, truth tables, and switches
  - NOT, AND, OR, NAND, NOR, XOR, . . ., minimal set
- Axioms and theorems of Boolean algebra
  - proofs by re-writing and perfect induction
- Gate logic
  - networks of Boolean functions and their time behavior
- Canonical forms
  - two-level and incompletely specified functions
- Simplification
  - two-level simplification
- Later
  - automation of simplification
  - multi-level logic
  - design case studies
  - time behavior