EECS150 - Digital Design

Lecture 19 - Combinational Logic Circuits : A Deep Dive

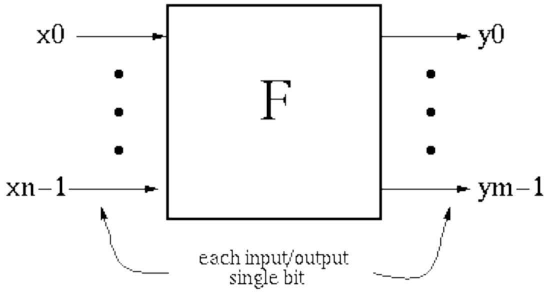
March 28, 2011

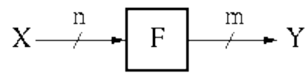
John Wawrzynek

Outline

- Review of three representations for combinational logic:
  - truth tables,
  - graphical (logic gates), and
  - algebraic equations
- Relationship among the three
- Adder example
- Laws of Boolean Algebra
- Canonical Forms
- Boolean Simplification

Combinational Logic (CL) Defined

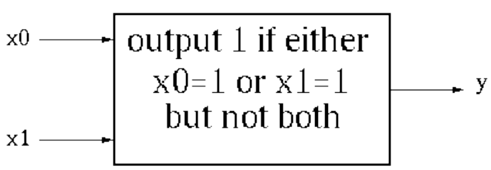




$y_i = f_i(x0 , ... , xn\text{-}1)$, where x, y are {0,1}.
Y is a function of only X.

- If we change X, Y will change immediately (well almost!).
- There is an implementation dependent delay from X to Y.

CL Block Example #1



Boolean Equation:

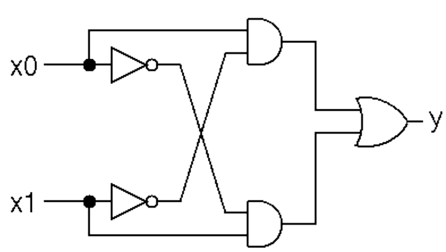
$y_0 = (x_0 \text{ AND not}(x_1))$
$\text{OR } (\text{not}(x_0) \text{ AND } x_1)$

$y_0 = x_0x_1' + x_0'x_1$

Truth Table Description:

x0	x1	y
0	0	0
0	1	1
1	0	1
1	1	0

Gate Representation:

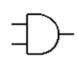


How would we *prove* that all three representations are equivalent?

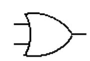
Boolean Algebra/Logic Circuits

- Why are they called "logic circuits"?
- Logic: The study of the principles of reasoning.
- The 19th Century Mathematician, George Boole, developed a math. system (algebra) involving logic, Boolean Algebra.
- His variables took on TRUE, FALSE
- Later Claude Shannon (father of information theory) showed (in his Master's thesis!) how to map Boolean Algebra to digital circuits:
- Primitive functions of Boolean Algebra:

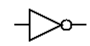
a b	AND
0 0	0
0 1	0
1 0	0
1 1	1



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

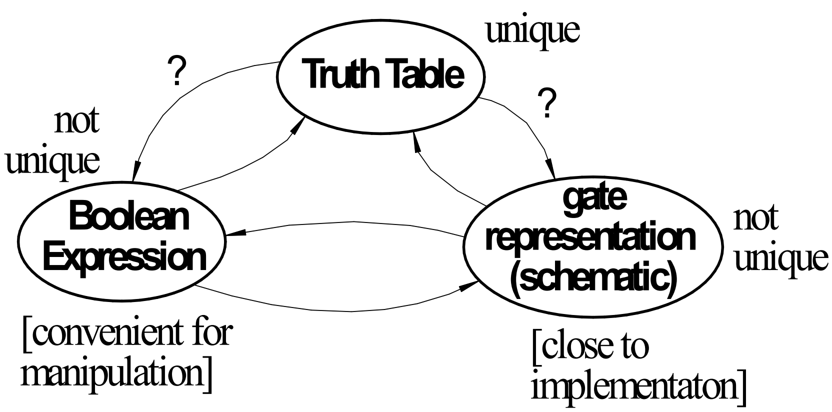


a	NOT
0	0
1	0



Relationship Among Representations

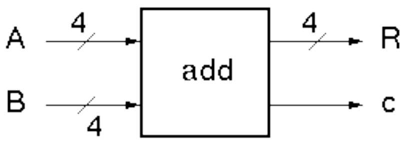
* Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.



How do we convert from one to the other?

CL Block Example #2

- 4-bit adder:



R = A + B,
c is carry out

- Truth Table Representation:

a3	a2	a1	a0	b3	b2	b1	b0	r3	r2	r1	r0	c
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	0	1	1	0	0	1	1	0
0	0	0	0	0	1	0	0	0	1	0	0	0
⋮												
0	0	1	0	0	0	1	0	0	1	0	0	0
0	0	1	0	0	0	1	1	0	1	0	1	0
⋮												
0	0	0	1	1	1	1	1	0	0	0	0	1

In general: 2^n rows for n inputs. 256 rows!
Is there a more efficient (compact) way to specify this function?

4-bit Adder Example

- Motivate the adder circuit design by hand addition:

```
    a3  a2  a1 | a0
  + b3  b2  b1 | b0
  -----------------
  c r3  r2  r1 | r0
```

- Add a0 and b0 as follows:

a	b	r	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

c ← carry to next stage

r = a XOR b = $a \oplus b$
c = a AND b = ab

```
    a3  a2 | a1 | a0
  + b3  b2 | b1 | b0
  ------------------
  c r3  r2 | r1 | r0
```

- Add a1 and b1 as follows:

ci	a	b	r	co
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

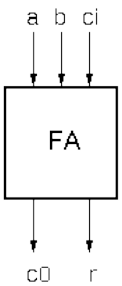
$r = a \oplus b \oplus c_i$
$co = ab + ac_i + bc_i$

4-bit Adder Example

- In general:

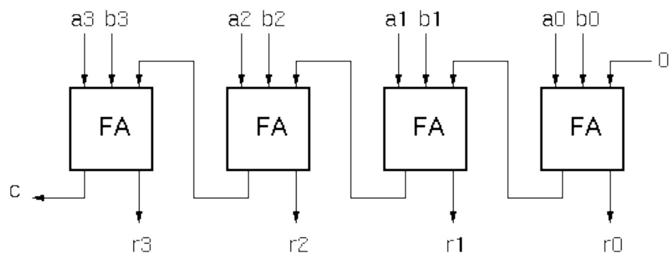
  $r_i = a_i \oplus b_i \oplus c_{in}$

  $c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} = c_{in}(a_i + b_i) + a_i b_i$



"Full adder cell"

- Now, the 4-bit adder:



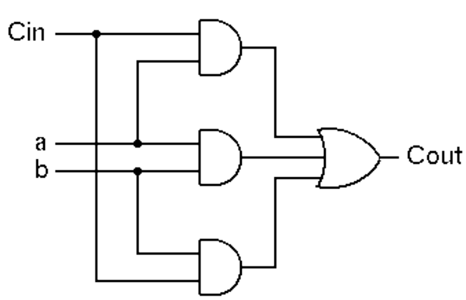
"ripple" adder

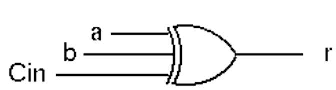
4-bit Adder Example

- Graphical Representation of FA-cell

  $r_i = a_i \oplus b_i \oplus c_{in}$

  $c_{out} = a_i c_{in} + a_i b_i + b_i c_{in}$

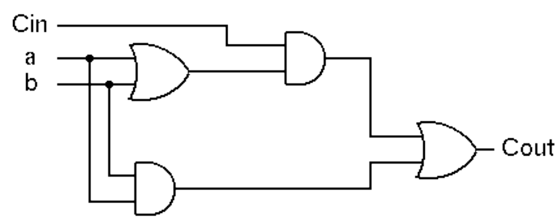


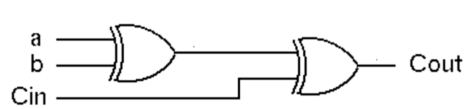


- Alternative Implementation (with 2-input gates):

  $r_i = (a_i \oplus b_i) \oplus c_{in}$

  $c_{out} = c_{in}(a_i + b_i) + a_i b_i$





Boolean Algebra

Defined as:

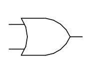
Set of elements B, binary operators $\{+,\bullet\}$, unary operation $\{'\}$, such that the following axioms hold:

1. B contains at least two elements a, b such that $a \neq b$.
2. Closure: a, b in B,
   $a + b$ in B, $a \bullet b$ in B, a' in B.
3. Communitive laws:
   $a + b = b + a$, $a \bullet b = b \bullet a$.
4. Identities: 0, 1 in B
   $a + 0 = a$, $a \bullet 1 = a$.
5. Distributive laws:
   $a + (b \bullet c) = (a + b) \bullet (a + c)$, $a \bullet (b + c) = a \bullet b + a \bullet c$.
6. Complement:
   $a + a' = 1$, $a \bullet a' = 0$.

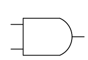
Logic Functions

$B = \{0,1\}$, $+ =$ OR, $\bullet =$ AND, $' =$ NOT

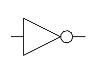
is a valid Boolean Algebra.



00	0
01	1
10	1
11	1



00	0
01	0
10	0
11	1



0	1
1	0

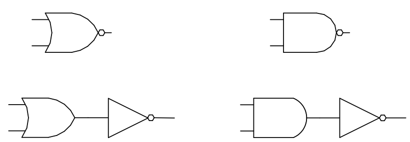
Do the axioms hold?

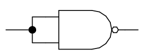
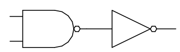
- Ex: communitive law: 0+1 = 1+0?

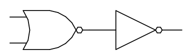
Other logic functions of 2 variables (x,y)

x y	f0	f1														
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
01	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
10	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
11	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
	0	AND		X		Y	⊕	OR	NOR	XNOR					NAND	1

Look at NOR and NAND:



- **Theorem**: Any Boolean function that can be expressed as a truth table can be expressed using NAND and NOR.
  - Proof sketch:  = NOT  = AND

     = OR

  - How would you show that either NAND or NOR is sufficient?

Laws of Boolean Algebra

Duality: A dual of a Boolean expression is derived by interchanging OR and AND operations, and 0s and 1s (literals are left unchanged).

$$\{F(x_1, x_2, ..., x_n, 0, 1, +, \bullet)\}^D = \{F(x_1, x_2, ..., x_n, 1, 0, \bullet, +)\}$$

Any law that is true for an expression is also true for its dual.

Operations with 0 and 1:
1. x + 0 = x   x * 1 = x
2. x + 1 = 1   x * 0 = 0

Idempotent Law:
3. x + x = x   x x = x

Involution Law:
4. (x')' = x

Laws of Complementarity:
5. x + x' = 1   x x' = 0

Commutative Law:
6. x + y = y + x   x y = y x

Laws of Boolean Algebra (cont.)

Associative Laws:

$(x + y) + z = x + (y + z)$ $\quad$ $x\,y\,z = x\,(y\,z)$

Distributive Laws:

$x\,(y + z) = (x\,y) + (x\,z)$ $\quad$ $x + (y\,z) = (x + y)(x + z)$

"Simplification" Theorems:

$x\,y + x\,y' = x$ $\quad$ $(x + y)\,(x + y') = x$

$x + x\,y = x$ $\quad$ $x\,(x + y) = x$

DeMorgan's Law:

$(x + y + z + \ldots)' = x'y'z'$ $\quad$ $(x\,y\,z\,\ldots)' = x' + y' + z'$

Theorem for Multiplying and Factoring:

$(x + y)\,(x' + z) = x\,z + x'\,y$

Consensus Theorem:

$x\,y + y\,z + x'\,z = (x + y)\,(y + z)\,(x' + z)$

$x\,y + x'\,z = (x + y)\,(x' + z)$

Proving Theorems via axioms of Boolean Algebra

Ex: prove the theorem: $x\,y + x\,y' = x$

$x\,y + x\,y' = x\,(y + y')$ distributive law

$x\,(y + y') = x\,(1)$ complementary law

$x\,(1) = x$ identity

Ex: prove the theorem: $x + x\,y = x$

$x + x\,y = x\,1 + x\,y$ identity

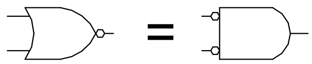
$x\,1 + x\,y = x\,(1 + y)$ distributive law

$x\,(1 + y) = x\,(1)$ identity

$x\,(1) = x$ identity

DeMorgan's Law

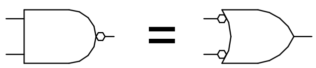
$(x + y)' = x' y'$



Exhaustive Proof

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

$(x y)' = x' + y'$

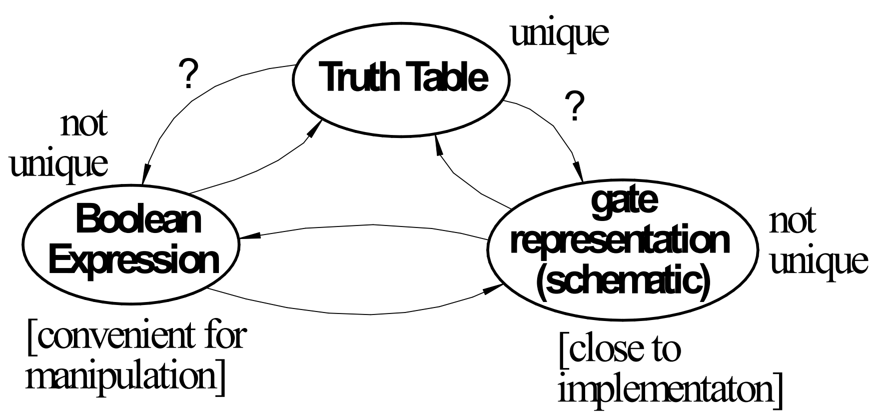


Exhaustive Proof

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

Relationship Among Representations

* Theorem: Any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using AND, OR, NOT.



How do we convert from one to the other?

Canonical Forms

- Standard form for a Boolean expression - unique algebraic expression directly from a true table (TT) description.
- Two Types:
  - * Sum of Products (SOP)
  - * Product of Sums (POS)
- **Sum of Products** (disjunctive normal form, minterm expansion).
  Example:

minterms	a b c	f f'
a'b'c'	0 0 0	0 1
a'b'c	0 0 1	0 1
a'bc'	0 1 0	0 1
a'bc	0 1 1	1 0
ab'c'	1 0 0	1 0
ab'c	1 0 1	1 0
abc'	1 1 0	1 0
abc	1 1 1	1 0

One product (**and**) term for each 1 in f:

$f = a'bc + ab'c' + ab'c + abc' + abc$

$f' = a'b'c' + a'b'c + a'bc'$

Sum of Products (cont.)

Canonical Forms are usually not minimal:

Our Example:

$$f = a'bc + ab'c' + ab'c + abc' + abc \quad (xy' + xy = x)$$
$$= a'bc + ab' + ab$$
$$= a'bc + a \qquad (x'y + x = y + x)$$
$$= a + bc$$

$$f' = a'b'c' + a'b'c + a'bc'$$
$$= a'b' + a'bc'$$
$$= a' (b' + bc')$$
$$= a' (b' + c')$$

Canonical Forms

- **Product of Sums** (conjunctive normal form, maxterm expansion). Example:

maxterms	a b c	f	f'
a+b+c	0 0 0	0	1
a+b+c'	0 0 1	0	1
a+b'+c	0 1 0	0	1
a+b'+c'	0 1 1	1	0
a'+b+c	1 0 0	1	0
a'+b+c'	1 0 1	1	0
a'+b'+c	1 1 0	1	0
a'+b'+c'	1 1 1	1	0

One sum (**or**) term for each 0 in f:

f = (a+b+c)(a+b+c')(a+b'+c)

f' = (a+b'+c')(a'+b+c)(a'+b+c')
(a'+b'+c)(a+b+c')

Mapping from SOP to POS (or POS to SOP): Derive truth table then proceed.

Algebraic Simplification Example

Ex: full adder (FA) carry out function (in canonical form):

Cout = a'bc + ab'c + abc' + abc

Algebraic Simplification

Cout = a'bc + ab'c + abc' + abc

= a'bc + ab'c + abc' + **abc + abc**

= a'bc + **abc** + ab'c + abc' + **abc**

= **(a' + a)bc** + ab'c + abc' + abc

= **(1)bc** + ab'c + abc' + abc

= bc + ab'c + abc' + **abc + abc**

= bc + ab'c + **abc** + abc' + **abc**

= bc + **a(b' +b)c** + abc' +abc

= bc + **a(1)c** + abc' + abc

= bc + ac + **ab(c' + c)**

= bc + ac + **ab(1)**

= bc + ac + ab

Outline for remaining CL Topics

- K-map method of two-level logic simplification
- Multi-level Logic
- NAND/NOR networks
- EXOR revisited

Algorithmic Two-level Logic Simplication

Key tool: The Uniting Theorem:

$$xy' + xy = x(y' + y) = x(1) = x$$

ab	f
00	0
01	0
10	1
11	1

$f = ab' + ab = a(b'+b) = a$

b values change within the on-set rows

a values don't change

b is eliminated, a remains

ab	g
00	1
01	0
10	1
11	0

$g = a'b'+ab' = (a'+a)b' = b'$

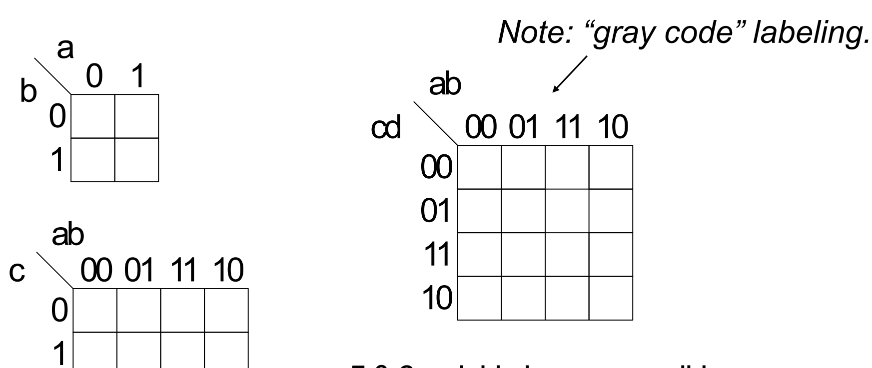
b values stay the same

a values changes

b' remains, a is eliminated

Karnaugh Map Method

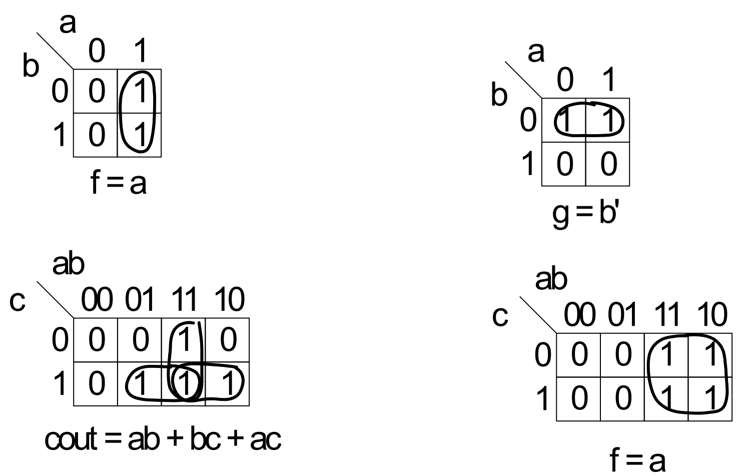
- K-map is an alternative method of representing the TT and to help visual the adjacencies.



5 & 6 variable k-maps possible

Karnaugh Map Method

- Adjacent groups of 1's represent product terms



K-map Simplification

1. Draw K-map of the appropriate number of variables (between 2 and 6)
2. Fill in map with function values from truth table.
3. Form groups of 1's.
   - ✓ Dimensions of groups must be even powers of two (1x1, 1x2, 1x4, …, 2x2, 2x4, …)
   - ✓ Form as large as possible groups and as few groups as possible.
   - ✓ Groups can overlap (this helps make larger groups)
   - ✓ Remember K-map is periodical in all dimensions (groups can cross over edges of map and continue on other side)
4. For each group write a product term.
   - the term includes the "constant" variables (use the uncomplemented variable for a constant 1 and complemented variable for constant 0)
5. Form Boolean expression as sum-of-products.

K-maps (cont.)

c \ ab	00	01	11	10
0	1	0	0	1
1	0	0	1	1

$f = b'c' + ac$

cd \ ab	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

$f = c + a'bd + b'd'$

(bigger groups are better)

Product-of-Sums Version

1. Form groups of 0's instead of 1's.
2. For each group write a sum term.
   - the term includes the "constant" variables (use the uncomplemented variable for a constant 0 and complemented variable for constant 1)
3. Form Boolean expression as product-of-sums.

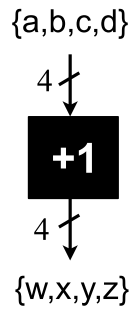
cd \ ab	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

$f = (b' + c + d)(a' + c + d')(b + c + d')$

BCD incrementer example

Binary Coded Decimal

	a b c d	w x y z
0	0 0 0 0	0 0 0 1
1	0 0 0 1	0 0 1 0
2	0 0 1 0	0 0 1 1
3	0 0 1 1	0 1 0 0
4	0 1 0 0	0 1 0 1
5	0 1 0 1	0 1 1 0
6	0 1 1 0	0 1 1 1
7	0 1 1 1	1 0 0 0
8	1 0 0 0	1 0 0 1
9	1 0 0 1	0 0 0 0
	1 0 1 0	- - - -
	1 0 1 1	- - - -
	1 1 0 0	- - - -
	1 1 0 1	- - - -
	1 1 1 0	- - - -
	1 1 1 1	- - - -



BCD Incrementer Example

- Note one map for each output variable.
- Function includes "don't cares" (shown as "-" in the table).
  - These correspond to places in the function where we don't care about its value, because we don't expect some particular input patterns.
  - We are free to assign either 0 or 1 to each don't care in the function, as a means to increase group sizes.
- In general, you might choose to write product-of-sums or sum-of-products according to which one leads to a simpler expression.

BCD incrementer example

w

cd \ ab	00	01	11	10
00	0	0	-	1
01	0	0	-	0
11	0	1	-	-
10	0	0	-	-

x

cd \ ab	00	01	11	10
00	0	1	-	0
01	0	1	-	0
11	1	0	-	-
10	0	1	-	-

y

cd \ ab	00	01	11	10
00	0	0	-	0
01	1	1	-	0
11	0	0	-	-
10	1	1	-	-

z

cd \ ab	00	01	11	10
00	1	1	-	1
01	0	0	-	0
11	0	0	-	-
10	1	1	-	-

w =

x =

y =

z =

BCD incrementer example

w

cd \ ab	00	01	11	10
00	0	0	-	1
01	0	0	-	0
11	0	1	-	-
10	0	0	-	-

x

cd \ ab	00	01	11	10
00	0	1	-	0
01	0	1	-	0
11	1	0	-	-
10	0	1	-	-

y

cd \ ab	00	01	11	10
00	0	0	-	0
01	1	1	-	0
11	0	0	-	-
10	1	1	-	-

z

cd \ ab	00	01	11	10
00	1	1	-	1
01	0	0	-	0
11	0	0	-	-
10	1	1	-	-

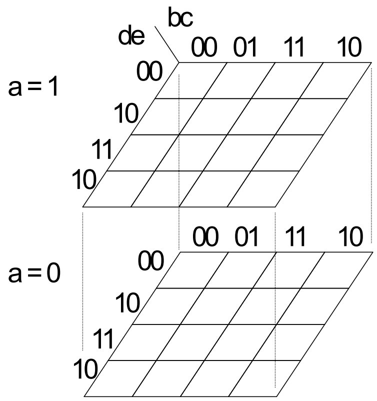
w =

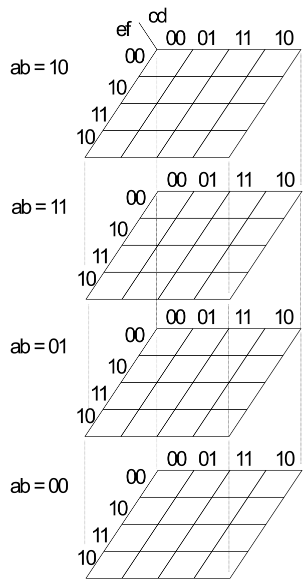
x =

y =

z =

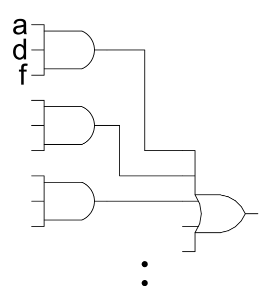
Higher Dimensional K-maps



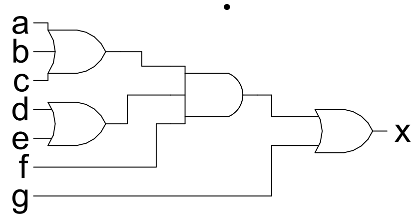


Multi-level Combinational Logic

- Example: reduced sum-of-products form
  x = adf + aef + bdf + bef + cdf + cef + g
- Implementation in 2-levels with gates:
  cost: 1 7-input OR, 6 3-input AND
  => 50 transistors
  delay: 3-input OR gate delay + 7-input AND gate delay



- Factored form:
  x = (a + b +c)(d + e)f + g
  cost: 1 3-input OR, 2 2-input OR, 1 3-input AND
  => 20 transistors
  delay: 3-input OR + 3-input AND + 2-input OR



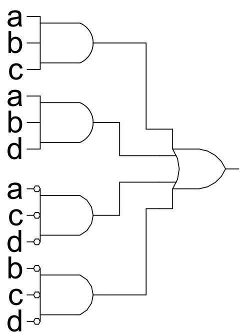
Footnote: NAND would be used in place of all ANDs and ORs.

Which is faster?

In general: Using multiple levels (more than 2) will reduce the cost. Sometimes also delay. Sometimes a tradeoff between cost and delay.

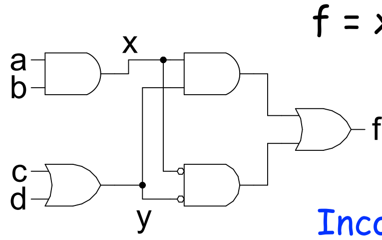
Multi-level Combinational Logic

Another Example: $F = abc + abd + a'c'd' + b'c'd'$



let $x = ab \quad y = c+d$

$f = xy + x'y'$



Incorporates fanout.

No convenient hand methods exist for multi-level logic simplification:

a) CAD Tools use sophisticated algorithms and heuristics

b) Humans and tools often exploit some special structure (example adder)

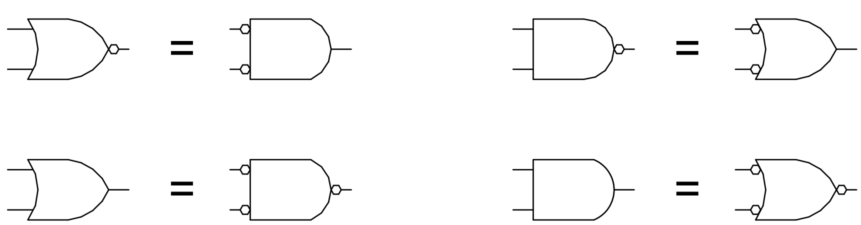
Are these optimizations still relevant for LUT implementations?

NAND-NAND & NOR-NOR Networks

DeMorgan's Law Review:

$(a + b)' = a' b' \qquad (a b)' = a' + b'$

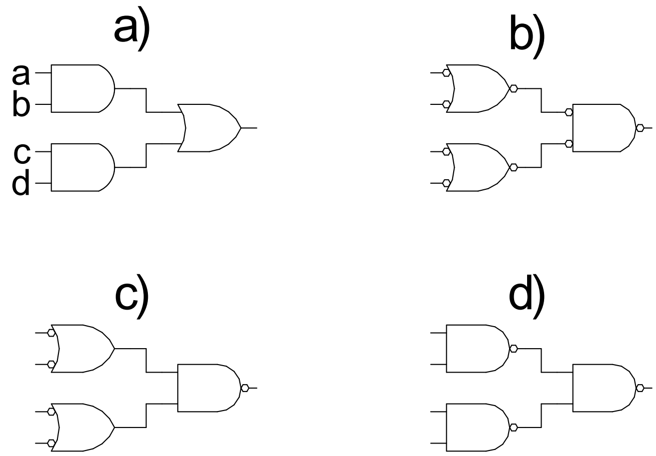
$a + b = (a' b')' \qquad (a b) = (a' + b')'$



push bubbles or *introduce in pairs* or *remove pairs:*
(x')' = x.

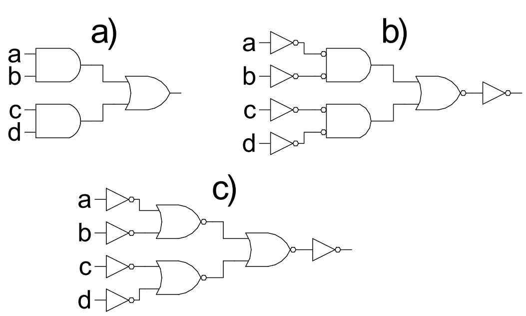
NAND-NAND & NOR-NOR Networks

- Mapping from AND/OR to NAND/NAND

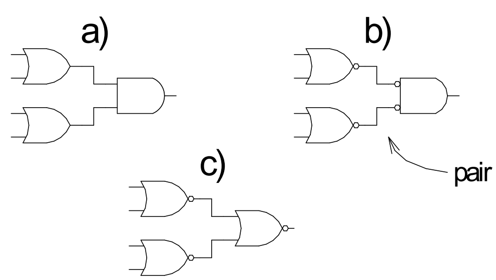


NAND-NAND & NOR-NOR Networks

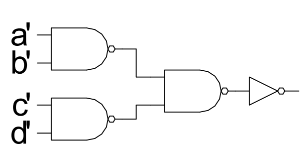
- Mapping AND/OR to NOR/NOR



- Mapping OR/AND to NOR/NOR



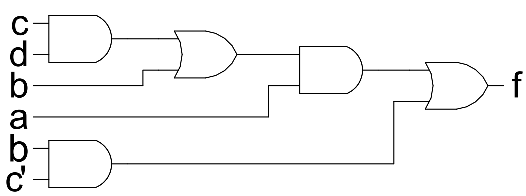
- OR/AND to NAND/NAND (by symmetry with above)



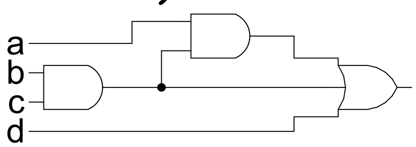
Multi-level Networks

Convert to NANDs:

F = a(b + cd) + bc'



(note fanout)

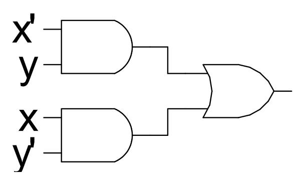


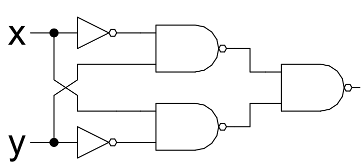
EXOR Function

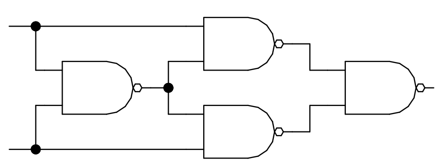
Parity, addition mod 2

$x \oplus y = x'y + xy'$

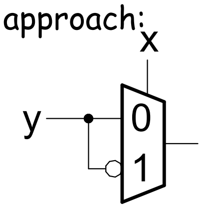
x y	xor	xnor
0 0	0	1
0 1	1	0
1 0	1	0
1 1	0	1







Another approach:



if x=0 then y else y'