EECS150 - Digital Design

Lecture 20 - Combinational Logic Circuits (Part 2)

April 1, 2010

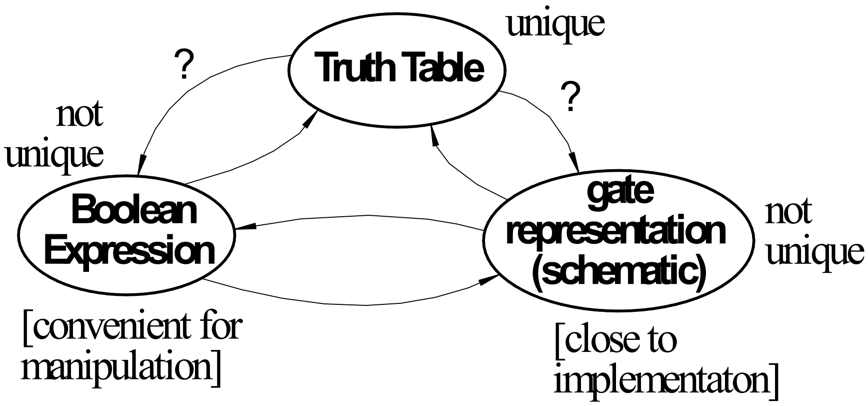
John Wawrzynek

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

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?

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

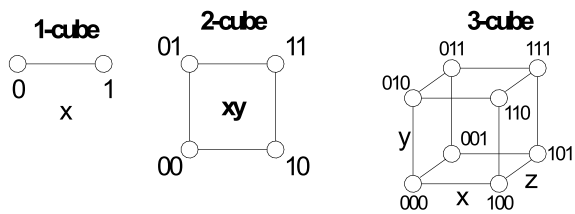
a values changes

b' remains, a is eliminated

Boolean Cubes

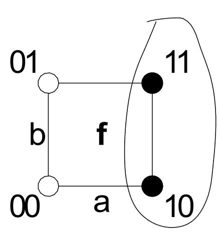
Visual technique for identifying when the Uniting Theorem can be applied

Alternative way to represent boolean functions.
Filled in nodes represent a 1 in the function.
Moving between adjacent nodes represents changing only one input.



- Sub-cubes of on-nodes can be used for simplification.
  - On-set: filled in nodes, off-set: empty nodes

a b	f	g
0 0	0	1
0 1	0	0
1 0	1	1
1 1	1	0



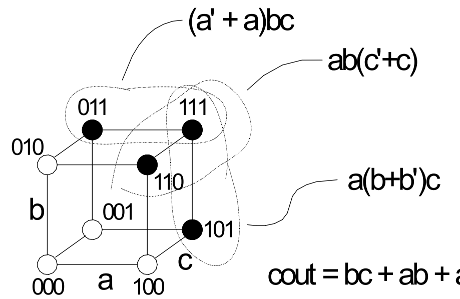
a asserted & unchanged
b varies

$ab + ab' = a$

3-variable cube example

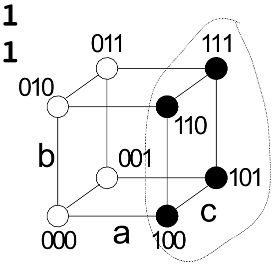
FA carry out:

a	b	c	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 = bc + ab + ac

What about larger sub-cubes?



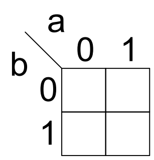
ab'c' + ab'c + abc' + abc

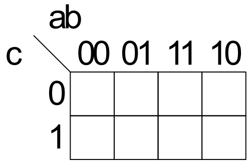
ac' + ac + ab = a + ab = a

- Both b & c change, a is asserted & remains constant.

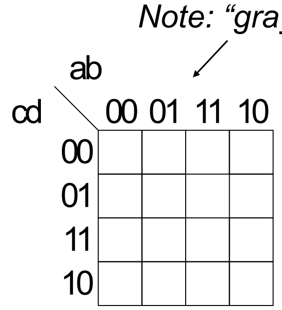
Karnaugh Map Method

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





Note: "gray code" labeling.



5 & 6 variable k-maps possible

Karnaugh Map Method

- Adjacent groups of 1's represent product terms

b \ a	0	1
0	0	1
1	0	1

f = a

b \ a	0	1
0	1	1
1	0	0

g = b'

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

cout = ab + bc + ac

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

f = a

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, …)
   - ✓ <u>Form as large as possible groups and as few groups as possible</u>.
   - ✓ 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.

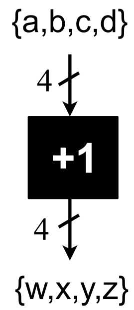
cd \ ab	00	01	11	10
00	1	0	0	1
01	0	1	0	0
11	1	1	1	1
10	1	1	1	1

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

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	-	-

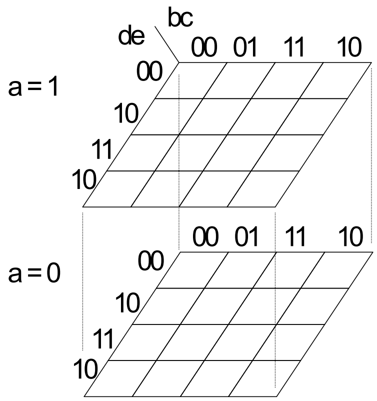
w =

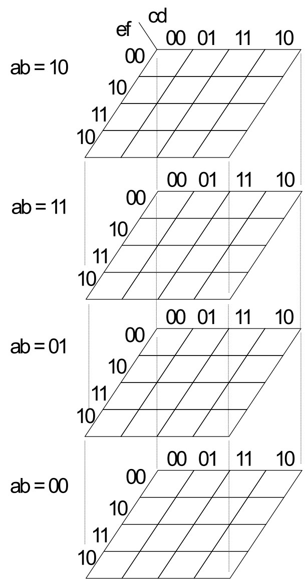
x =

y =

z =

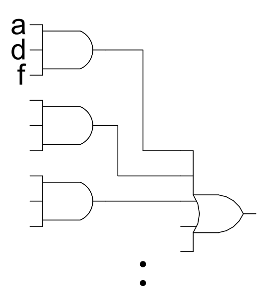
Higher Dimensional K-maps

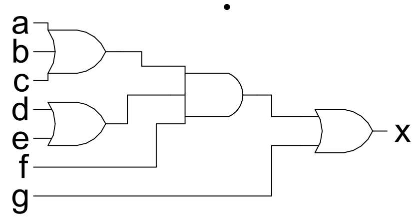




Multi-level Combinational Logic

- 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

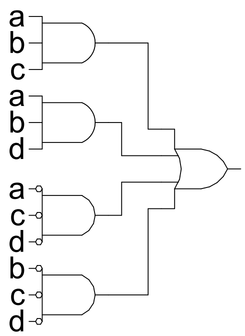
Footnote: NAND would be used in place of all ANDs and ORs.

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.

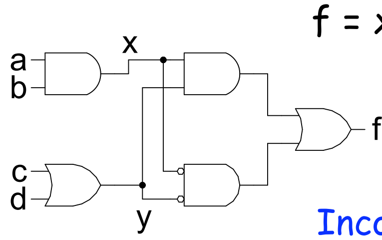
Multi-level Combinational Logic

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



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

$f = xy + x'y'$



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)

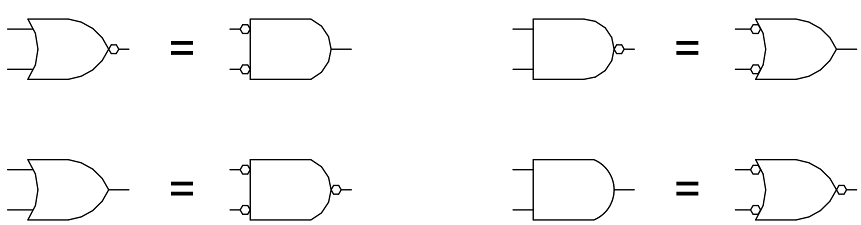
Are these optimizations still relevant for LUT implementations?

NAND-NAND & NOR-NOR Networks

DeMorgan's Law Review:

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

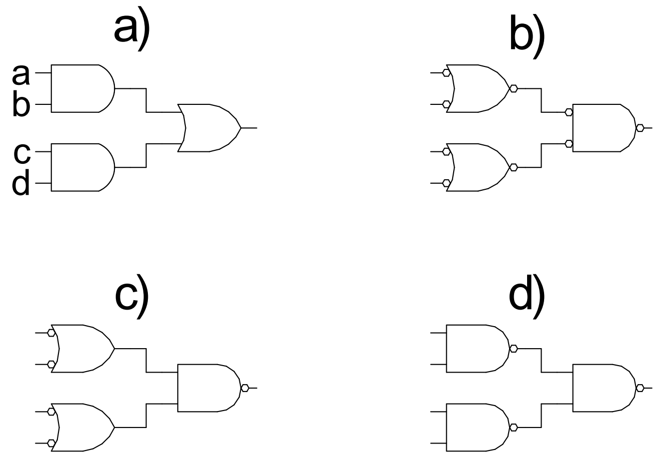
$a + b = (a'\, b')'$ $(a\, b) = (a' + b')'$



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

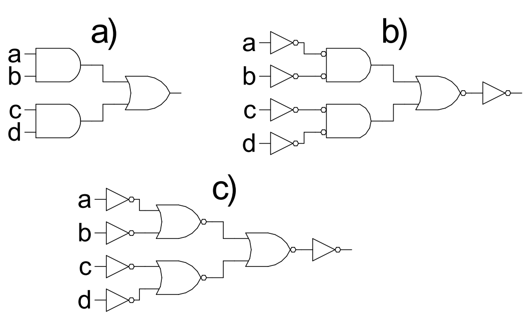
NAND-NAND & NOR-NOR Networks

- Mapping from AND/OR to NAND/NAND

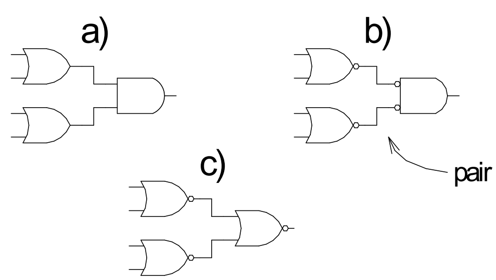


NAND-NAND & NOR-NOR Networks

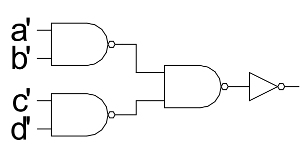
- Mapping AND/OR to NOR/NOR



- Mapping OR/AND to NOR/NOR



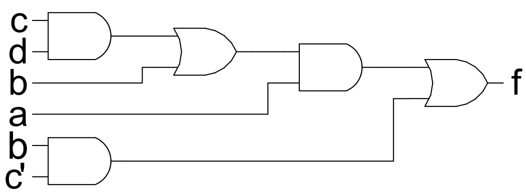
- OR/AND to NAND/NAND (by symmetry with above)



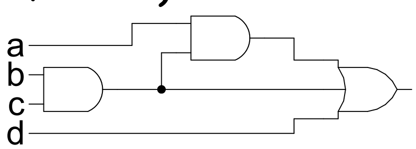
Multi-level Networks

Convert to NANDs:

F = a(b + cd) + bc’



(note fanout)

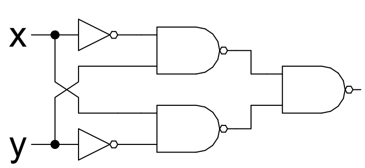


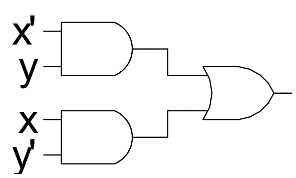
EXOR Function

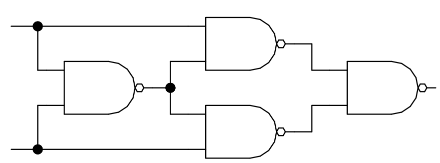
Parity, addition mod 2

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

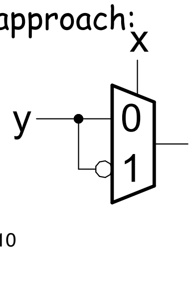
x y	xor	xnor
0 0	0	1
0 1	1	0
1 0	1	0
1 1	0	1







Another approach:



if x=0 then y else y'