EECS 151/251A
Spring 2020
Digital Design and Integrated Circuits

Instructor:
John Wawrzynek

Lecture 6: CL
Outline

- Three representations for combinational logic:
  - truth tables,
  - graphical (logic gates), and
  - algebraic equations
- Boolean Algebra
- Boolean Simplification
- Multi-level Logic, NAND/NOR, XOR
Representations of Combinational Logic
Combinational Logic (CL) Defined

\[ y_i = f_i(x_0, \ldots, x_{n-1}), \text{ where } x, y \text{ take on values } \{0,1\}. \]

- \text{Y is a function of only } X, \text{ i.e., it is a “pure function”}.

- If we change } X, \text{ Y will change immediately (well almost!).}
  - There is an \textit{implementation dependent} delay from } X \text{ to } Y.\]
**Truth Table Description:**

<table>
<thead>
<tr>
<th>x0</th>
<th>x1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Boolean Equation:**

\[ y_0 = (x_0 \text{ AND } \neg x_1) \text{ OR } (\neg x_0 \text{ AND } x_1) \]

\[ y_0 = x_0x_1' + x_0'x_1 \]

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

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>NOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Theorem: Any Boolean function that can be expressed as a truth table can be expressed using NAND and NOR.

Proof sketch:

- How would you show that either NAND or NOR is sufficient?
* 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 – 4 Bit Adder - where decomposition helps

\[ R = A + B, \quad c \text{ is carry out} \]

- **Truth Table Representation:**

<table>
<thead>
<tr>
<th>a3</th>
<th>a2</th>
<th>a1</th>
<th>a0</th>
<th>b3</th>
<th>b2</th>
<th>b1</th>
<th>b0</th>
<th>r3</th>
<th>r2</th>
<th>r1</th>
<th>r0</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

256 rows!

**In general:** \(2^n\) rows for \(n\) inputs.

**Is there a more efficient (compact) way to specify this function?**
4-bit Adder Example

- Motivate the adder circuit design by hand addition:

\[
\begin{array}{c|ccc|c}
\hline
 & a_3 & a_2 & a_1 & a_0 \\
\hline
+ & b_3 & b_2 & b_1 & b_0 \\
\hline
 & c & r_3 & r_2 & r_1 & r_0 \\
\end{array}
\]

- Add a0 and b0 as follows:

\[
r = a \oplus b = a \oplus b \\
c = a \land b = ab \\
\]

- Add a1 and b1 as follows:

\[
\begin{array}{c|c|c|c|c}
\hline
\text{ci} & a & b & r & \text{co} \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

\[
r = a \oplus b \oplus c_i \\
\text{co} = ab + ac_i + bc_i \\
\]

\[
\text{carry to next stage}
\]
4-bit Adder Example

- In general:
  \[ r_i = a_i \oplus b_i \oplus c_{in} \]
  \[ c_{out} = a_i c_{in} + a_i b_i + b_i c_{in} = c_{in}(a_i + b_i) + a_i b_i \]

- Now, the 4-bit adder:

  "Full adder cell"

  "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 only 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
Boolean Algebra

Set of elements $B$, binary operators $\{+, \cdot\}$, 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 $B$, $a \cdot b$ in $B$, $a'$ in $B$.

3. Communititive laws:
   
   $a + b = b + a$, $a \cdot b = b \cdot a$.

4. Identities: 0, 1 in $B$

   $a + 0 = a$, $a \cdot 1 = a$.

5. Distributive laws:

   $a + (b \cdot c) = (a + b) \cdot (a + c)$, $a \cdot (b + c) = a \cdot b + a \cdot c$.

6. Complement:

   $a + a' = 1$, $a \cdot a' = 0$. 

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

is a valid Boolean Algebra.
Some Laws (theorems) 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).

\[ \{F(x_1, x_2, \ldots, x_n, 0, 1, +, \cdot)\}^D = \{F(x_1, x_2, \ldots, x_n, 1, 0, \cdot, +)\} \]

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

Operations with 0 and 1:
- \(x + 0 = x\)
- \(x \cdot 1 = x\)
- \(x + 1 = 1\)
- \(x \cdot 0 = 0\)

Idempotent Law:
- \(x + x = x\)
- \(x \cdot x = x\)

Involution Law:
- \((x')' = x\)

Laws of Complementarity:
- \(x + x' = 1\)
- \(x \cdot x' = 0\)

Commutative Law:
- \(x + y = y + x\)
- \(x \cdot y = y \cdot x\)
Some Laws (theorems) of Boolean Algebra (cont.)

**Associative Laws:**
\[(x + y) + z = x + (y + z)\]
\[x y z = x (y z)\]

**Distributive Laws:**
\[x (y + z) = (x y) + (x z)\]
\[x + (y z) = (x + y)(x + z)\]

**“Simplification” Theorems:**
\[x y + x y' = x\]
\[x + x y = x\]
\[x + x' y = x + y\]
\[(x + y) (x + y') = x\]
\[x (x + y) = x\]
\[x(x' + y) = xy\]

**DeMorgan’s Law:**
\[(x + y + z + \ldots)' = x'y'z'\]
\[(x y z \ldots)' = x' + y' + z'\]

**Theorem for Multiplying and Factoring:**
\[(x + y) (x' + z) = x z + x' y\]

**Consensus Theorem:**
\[x y + y z + x' z = (x + y) (y + z) (x' + z)\]
\[x y + x' z = (x + y) (x' + z)\]
DeMorgan's Law

\[(x + y)' = x' \cdot y'
\]

Exhaustive Proof

\[
\begin{array}{ccc|cc}
 x & y & x' & y' & (x + y)' & x' \cdot y' \\
---&---&---&---&---&---
 0 & 0 & 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 0 & 0 & 0 \\
 1 & 0 & 0 & 1 & 0 & 0 \\
 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[(x \cdot y)' = x' + y'
\]

Exhaustive Proof

\[
\begin{array}{ccc|cc}
 x & y & x' & y' & (x \cdot y)' & x' + y' \\
---&---&---&---&---&---
 0 & 0 & 1 & 1 & 1 & 1 \\
 0 & 1 & 1 & 0 & 1 & 1 \\
 1 & 0 & 0 & 1 & 1 & 1 \\
 1 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]
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:

<table>
<thead>
<tr>
<th>Minterms</th>
<th>a b c</th>
<th>f</th>
<th>f'</th>
</tr>
</thead>
<tbody>
<tr>
<td>a'b'c'</td>
<td>0 0 0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'b'c'</td>
<td>0 0 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'bc'</td>
<td>0 1 0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a'bc</td>
<td>0 1 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ab'c'</td>
<td>1 0 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ab'c</td>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ab'c'</td>
<td>1 1 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>abc'</td>
<td>1 1 0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>abc</td>
<td>1 1 1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
  f &= a'bc + ab'c' + ab'c + abc' + abc \\
  f' &= a'b'c' + a'b'c + a'bc'
\end{align*}
\]

(enumerate all the ways the function could evaluate to 1)

What is the cost?
Sum of Products (cont.)

Canonical Forms are usually not minimal:

Our Example:

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

\[ f' = a'b'c' + a'b'c + a'bc' \]
\[ = a'b' + a'bc' \]
\[ = a' ( b' + bc' ) \]
\[ = a' ( b' + c' ) \]
\[ = a'b' + a'c' \]
Canonical Forms

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

Example:

<table>
<thead>
<tr>
<th>maxterms</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>f</th>
<th>f'</th>
</tr>
</thead>
<tbody>
<tr>
<td>a+b+c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a+b+c'</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a+b'+c</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>a+b'+c'</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b+c</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b+c'</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b'+c</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>a'+b'+c'</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

\[
f = (a+b+c)(a+b+c')(a+b'+c)
\]

\[
f' = (a+b'+c')(a'+b+c)(a'+b'+c')
\]

(enumerate all the ways the function could evaluate to 0)

What is the cost?
Boolean Simplification
Algebraic Simplification Example

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

\[ \text{Cout} = a'bc + ab'c + abc' + abc \]

<table>
<thead>
<tr>
<th>ci</th>
<th>a</th>
<th>b</th>
<th>r</th>
<th>co</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Algebraic Simplification

\[ \text{Cout} = a'bc + ab'c + abc' + abc \]

\[ = a'bc + ab'c + abc' + abc + abc \]
\[ = a'bc + abc + ab'c + abc' + abc \]
\[ = (a' + a)bc + ab'c + abc' + abc \]
\[ = (1)bc + ab'c + abc' + abc \]
\[ = bc + ab'c + abc' + abc + abc \]
\[ = bc + ab'c + abc' + abc \]
\[ = bc + a(b' + b)c + abc' + abc \]
\[ = bc + a[1]c + abc' + abc \]
\[ = bc + ac + ab[c' + c] \]
\[ = bc + ac + ab[1] \]
\[ = bc + ac + ab \]
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 Simplification**

**Key tool:** **The Uniting Theorem:**

\[ xy' + xy = x (y' + y) = x \cdot 1 = x \]

<table>
<thead>
<tr>
<th>(ab)</th>
<th>(f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(0)</td>
</tr>
<tr>
<td>01</td>
<td>(0)</td>
</tr>
<tr>
<td>10</td>
<td>(1)</td>
</tr>
<tr>
<td>11</td>
<td>(1)</td>
</tr>
</tbody>
</table>

- \(f = ab' + ab = a(b'+b) = a\)
- \(b\) values change within the on-set rows
- \(a\) values don’t change
- \(b\) is eliminated, \(a\) remains

<table>
<thead>
<tr>
<th>(ab)</th>
<th>(g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>(1)</td>
</tr>
<tr>
<td>01</td>
<td>(0)</td>
</tr>
<tr>
<td>10</td>
<td>(1)</td>
</tr>
<tr>
<td>11</td>
<td>(0)</td>
</tr>
</tbody>
</table>

- \(g = a'b' + ab' = (a'+a)b' = b'\)
- \(b\) 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 visualize the adjacencies.

Note: “gray code” labeling.

5 & 6 variable k-maps possible
Karnaugh Map Method

- Adjacent groups of 1’s represent product terms

\[
\begin{array}{ccc}
\text{a} & 0 & 1 \\
\text{b} & 0 & 0 & 1 \\
& 1 & 0 & 1 \\
\end{array}
\]

\[f = a\]

\[
\begin{array}{cccc}
\text{ab} & 00 & 01 & 11 & 10 \\
\text{c} & 0 & 0 & 1 & 1 \\
\end{array}
\]

\[\text{cout} = ab + bc + ac\]
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.)

**Example 1:**

<table>
<thead>
<tr>
<th>cd</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ f = b'c' + ac \]

**Example 2:**

<table>
<thead>
<tr>
<th>cd</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ f = c + a'bd + b'd' \]

*(bigger groups are better)*
**Product-of-Sums K-map**

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.

\[
f = (b' + c + d)(a' + c + d')(b + c + d')
\]
### Binary Coded Decimal

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>w</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>y</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **{a,b,c,d}**
  - 4
  - \(+1\)

- **{w,x,y,z}**
  - 4
  - \(-\)
  - \(-\)
  - \(-\)

**BCD incrementer example**
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**

<table>
<thead>
<tr>
<th>ab</th>
<th>(00)</th>
<th>(01)</th>
<th>(11)</th>
<th>(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(00)</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>(01)</td>
<td>0</td>
<td>0</td>
<td>-0</td>
<td>0</td>
</tr>
<tr>
<td>(11)</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(10)</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**X**

<table>
<thead>
<tr>
<th>ab</th>
<th>(00)</th>
<th>(01)</th>
<th>(11)</th>
<th>(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(00)</td>
<td>0</td>
<td>1</td>
<td>-0</td>
<td>0</td>
</tr>
<tr>
<td>(01)</td>
<td>0</td>
<td>1</td>
<td>-0</td>
<td>0</td>
</tr>
<tr>
<td>(11)</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(10)</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\(w = \) 

**Y**

<table>
<thead>
<tr>
<th>ab</th>
<th>(00)</th>
<th>(01)</th>
<th>(11)</th>
<th>(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(00)</td>
<td>0</td>
<td>0</td>
<td>-0</td>
<td>0</td>
</tr>
<tr>
<td>(01)</td>
<td>1</td>
<td>1</td>
<td>-0</td>
<td>0</td>
</tr>
<tr>
<td>(11)</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(10)</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Z**

<table>
<thead>
<tr>
<th>ab</th>
<th>(00)</th>
<th>(01)</th>
<th>(11)</th>
<th>(10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(00)</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>(01)</td>
<td>0</td>
<td>0</td>
<td>-0</td>
<td>0</td>
</tr>
<tr>
<td>(11)</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(10)</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

\(x = \) 

\(y = \) 

\(z = \)
Higher Dimensional K-maps

For a = 1:
- de 00 01 11 10
- 10 11

For a = 0:
- bc 00 01 11 10
- 10 11 10
Boolean Simplification
– Multi-level Logic
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
  - \( \Rightarrow \) \( \sim \)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
  - \( \Rightarrow \) \( \sim \)20 transistors
  - **delay:** 3-input OR + 3-input AND + 2-input OR

**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.

Footnote: NAND would be used in place of all ANDs and ORs.
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
   Guess what? These problems tend to be NP-complete

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

\[(x')' = x\]
Mapping from AND/OR to NAND/NAND

a) 

b) 

c) 

d)
Multi-level Networks

Convert to NANDs:

\[ F = a(b + cd) + bc' \]
EXOR Function Implementations

Parity, addition mod 2

\[ x \oplus y = x'y + xy' \]

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>xor</th>
<th>xnor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Another approach:

if \( x = 0 \) then \( y \) else \( y' \)