Minimization of Boolean logic

- Minimization
  - uniting theorem
  - grouping of terms in Boolean functions
- Alternate representations of Boolean functions
  - cubes
  - Karnaugh maps

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

\[
F = A'B' + AB' = (A' + A)B' = B'
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>

- B has the same value in both on-set rows
  - B remains, actually B' because B is 0 in both cases
- A has a different value in the two rows
  - A is eliminated

Boolean cubes

- Visual technique for identifying when the uniting theorem can be applied
- n input variables = n-dimensional "cube"
Mapping truth tables onto Boolean cubes

- Uniting theorem combines two "faces" of a cube into a larger "face"
- Example:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</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>

ON-set = solid nodes
OFF-set = empty nodes
DC-set = ×d nodes

two faces of size 0 (nodes) combine into a face of size 1(line)

A varies within face, B does not this face represents the literal B'

Three variable example

- Binary full-adder carry-out logic

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>Cout</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ (A'+A)BCin \]
\[ AB(Cin'+Cin) \]
\[ A(B+B')Cin \]

the on-set is completely covered by the combination (OR) of the subcubes of lower dimensionality - note that "111" is covered three times

\[ Cout = BCin + AB + ACin \]
Higher dimensional cubes

- Sub-cubes of higher dimension than 2

\[ F(A,B,C) = \Sigma m(4,5,6,7) \]

- on-set forms a square
  - i.e., a cube of dimension 2

- represents an expression in one variable
  - i.e., 3 dimensions – 2 dimensions

- A is asserted (true) and unchanged
  - B and C vary

This subcube represents the literal A

m-dimensional cubes in a n-dimensional Boolean space

- In a 3-cube (three variables):
  - a 0-cube, i.e., a single node, yields a term in 3 literals
  - a 1-cube, i.e., a line of two nodes, yields a term in 2 literals
  - a 2-cube, i.e., a plane of four nodes, yields a term in 1 literal
  - a 3-cube, i.e., a cube of eight nodes, yields a constant term "1"

- In general,
  - an m-subcube within an n-cube (m < n) yields a term
    with n – m literals
Karnaugh maps

- Flat map of Boolean cube
  - wrap-around at edges
  - hard to draw and visualize for more than 4 dimensions
  - virtually impossible for more than 6 dimensions
- Alternative to truth-tables to help visualize adjacencies
  - guide to applying the uniting theorem
  - on-set elements with only one variable changing value are adjacent unlike the situation in a linear truth-table

\[
\begin{array}{c|cc}
A & B & F \\
\hline
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

Karnaugh maps (cont’d)

- Numbering scheme based on Gray–code
  - e.g., 00, 01, 11, 10
  - only a single bit changes in code for adjacent map cells

A = 1101 = ABC'D
Adjacencies in Karnaugh maps

- Wrap from first to last column
- Wrap top row to bottom row

Karnaugh map examples

- \( F = \)
- \( C_{out} = \)
- \( f(A,B,C) = \Sigma m(0,4,5,7) \)
More Karnaugh map examples

G(A,B,C) = A

F(A,B,C) = \Sigma m(0,4,5,7) = AC' + B'C'

F' simply replace 1’s with 0’s and vice versa

F'(A,B,C) = \Sigma m(1,2,3,6) = BC' + A'C

Karnaugh map: 4-variable example

F(A,B,C,D) = \Sigma m(0,2,3,5,6,7,8,10,11,14,15)

F = C + A'B'D + B'D'

find the smallest number of the largest possible subcubes to cover the ON-set
(fewer terms with fewer inputs per term)
Karnaugh maps: don’t cares

- \( f(A,B,C,D) = \Sigma m(1,3,5,7,9) + d(6,12,13) \)
  - without don’t cares
    - \( f = A'D + B'C'D \)

\[
\begin{array}{cccc}
0 & 0 & \times & 0 \\
1 & 1 & \times & 1 \\
1 & 1 & 0 & 0 \\
0 & \times & 0 & 0 \\
\end{array}
\]

Karnaugh maps: don’t cares (cont’d)

- \( f(A,B,C,D) = \Sigma m(1,3,5,7,9) + d(6,12,13) \)
  - \( f = A'D + B'C'D \) without don’t cares
  - \( f = A'D + C'D \) with don’t cares

by using don't care as a "1"
a 2-cube can be formed rather than a 1-cube to cover this node

\[
\begin{array}{cccc}
0 & 0 & \times & 0 \\
1 & 1 & \times & 1 \\
1 & 1 & 0 & 0 \\
0 & \times & 0 & 0 \\
\end{array}
\]

don't cares can be treated as 1s or 0s depending on which is more advantageous
Activity

- Minimize the function \( F = \sum m(0, 2, 7, 14, 15) + d(3, 6, 9, 12, 13) \)

\[
F = AC' + A'C + BC + AB + A'B'D' + B'C'D'
\]

\[
F = BC + A'B'D' + B'C'D'
\]

\[
F = A'C + AB + B'C'D'
\]

Does \( BC + A'B'D' + B'C'D' = A'C + AB + B'C'D' \)?

- NO! Not in general, only if we ignore the cells with don’t cares

\[
F_1 = BC + A'B'D' + B'C'D'
\]

\[
F_2 = A'C + AB + B'C'D'
\]

\( F_1 \neq F_2 \)

\[
F_1 + d(3, 6, 9, 12, 13) = F_2 + d(3, 6, 9, 12, 13)
\]

(don’t cares all 1)

\[
F_1 \cdot D(3, 6, 9, 12, 13) = F_2 \cdot D(3, 6, 9, 12, 13)
\]

(don’t cares all 0)
Combinational logic summary (so far)

- 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
  - a start at understanding two-level simplification