Simplification

- 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

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

Boolean cubes

- Visual technique for indentifying when the uniting theorem can be applied
- $n$ input variables = $n$-dimensional "cube"
Mapping truth tables onto 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>

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

ON-set = solid nodes
OFF-set = empty nodes

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

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>

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

- In general,
  - 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

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

13 = 1101 = ABCD
Adjacencies in Karnaugh maps

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

Karnaugh map examples

- \( F = \)
- \( \text{Cout} = \)
- \( f(A,B,C) = \Sigma m(0,4,6,7) \)

obtain the complement of the function by covering 0s with subcubes
More Karnaugh map examples

\[ F(A,B,C) = \Sigma m(0,4,5,7) \]

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

\[ F' \] simply replace 1's with 0's and vice versa

\[ G(A,B,C) = A \]

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

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

Karnaugh map: 4-variable example

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

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

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
    - \( f = A'D + B'C'D \)

  - with don't cares
    - \( f = A'D + C'D \)

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

  don’t cares can be treated as
  1s or 0s
  depending on which is more advantageous
Example: two-bit comparator

we'll need a 4-variable Karnaugh map for each of the 3 output functions

Example: two-bit comparator

LT = \( A' B' D + A' C + B' C D \)
EQ = \( A'B'C'D' + A'BC'D + ABCD + AB'CD' = (A \text{xnor} C) \cdot (B \text{xnor} D) \)
GT = \( B C' D' + A C' + A B D' \)

LT and GT are similar (flip A/C and B/D)
Example: two-bit comparator

two alternative implementations of EQ with and without XOR

XNOR is implemented with at least 3 simple gates

Example: 2x2-bit multiplier

<table>
<thead>
<tr>
<th>A2</th>
<th>A1</th>
<th>B2</th>
<th>B1</th>
<th>P8</th>
<th>P4</th>
<th>P2</th>
<th>P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

4-variable K-map for each of the 4 output functions
Example: 2x2-bit multiplier

K-map for P8

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

P8 = A2A1B2B1

K-map for P4

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


K-map for P2

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


K-map for P1

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

P1 = A1B1

Example: BCD increment by 1

I8 I4 I2 I1 | O8 O4 O2 O1
-------------|-------------
0 0 0 0   | 0 0 0 1
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 0
1 1 1 1   | 1 1 1 1

<table>
<thead>
<tr>
<th>I1</th>
<th>I2</th>
<th>I4</th>
<th>I8</th>
<th>O1</th>
<th>O2</th>
<th>O4</th>
<th>O8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4-variable K-map for each of the 4 output functions
Example: BCD increment by 1

O8 = I4 I2 I1 + I8 I1'
O4 = I4 I2' + I4 I1' + I4' I2 I1
O2 = I8' I2' I1 + I2 I1'
O1 = I1'

Definition of terms

- **Implicant**: Single element of ON-set or any group of these elements that can be combined to form a subcube
- **Prime implicant**: Implicant that can't be combined with another to form a larger subcube
- **Essential prime implicant**: Prime implicant is essential if it alone covers an element of ON-set
- **Objective**: Grow implicant into prime implicants (minimize literals per term)
  - Cover the ON-set with as few prime implicants as possible (minimize number of product terms)
Examples to illustrate terms

6 prime implicants:
- $A'B'D$, $BC'$, $AC$, $A'C'D$, $AB$, $B'CD$

Essential:
- $AC$ + $BC'$ + $A'B'D$

5 prime implicants:
- $BD$, $ABC'$, $ACD$, $A'BC$, $A'C'D$

Essential:
- $AC$ + $BC'$ + $A'B'D$

Minimum cover: 4 essential implicants

Two-level simplification algorithm

- Algorithm: minimum sum-of-products expression from a Karnaugh map

  - Step 1: choose an element of the ON-set
  - Step 2: find "maximal" groupings of 1s and Xs adjacent to that element
    - consider top/bottom row, left/right column, and corner adjacencies
    - this forms prime implicants (number of elements are power of 2)

  - Repeat Steps 1 and 2 to find all prime implicants

  - Step 3: revisit the 1s in the K-map
    - if covered by single prime implicant, it is essential, and participates in final cover
    - 1s covered by essential prime implicant do not need to be revisited
  
  - Step 4: if there remain 1s not covered by essential prime implicants
    - select the smallest number of prime implicants that cover the remaining 1s
Finite State Machine Optimization

- State Minimization
  - Fewer states require fewer state bits
  - Fewer bits require fewer logic equations
- Encodings: State, Inputs, Outputs
  - State encoding with fewer bits has fewer equations to implement
    - However, each may be more complex
  - State encoding with more bits (e.g., one-hot) has simpler equations
    - Complexity directly related to complexity of state diagram
  - Input/output encoding may or may not be under designer control
Algorithmic Approach

- **Goal** – identify and combine states that have equivalent behavior
- **Equivalent States:**
  - Same output
  - For all input combinations, states transition to same or equivalent states
- **Algorithm Sketch**
  1. Place all states in one set
  2. Initially partition set based on output behavior
  3. Successively partition resulting subsets based on next state transitions
  4. Repeat (3) until no further partitioning is required
     - states left in the same set are equivalent
  - Polynomial time procedure

---

State Minimization Example

- **Sequence Detector for 010 or 110**

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State $X=0$</th>
<th>Next State $X=1$</th>
<th>Output $X=0$</th>
<th>Output $X=1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S3</td>
<td>S4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S5</td>
<td>S6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S3</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Method of Successive Partitions

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>S0</td>
<td>S1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>S1</td>
<td>S3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S2</td>
<td>S5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>S3</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>S4</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>S5</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>S6</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

( S0 S1 S2 S3 S4 S5 S6 )  
S1 is equivalent to S2
( S0 S1 S2 S3 S5 )  ( S4 S6 )  
S3 is equivalent to S5
( S0 S3 S5 )  ( S1 S2 )  ( S4 S6 )  
S4 is equivalent to S6
( S0 )  ( S3 S5 )  ( S1 S2 )  ( S4 S6 )

Minimized FSM

- State minimized sequence detector for 010 or 110

### Minimized FSM Table

<table>
<thead>
<tr>
<th>Input Sequence</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>S0</td>
<td>S1'</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 + 1</td>
<td>S1'</td>
<td>S3'</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X0</td>
<td>S3'</td>
<td>S0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>X1</td>
<td>S4'</td>
<td>S0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
More Complex State Minimization

- Multiple input example

*inputs here*

<table>
<thead>
<tr>
<th>present state</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S3</td>
<td>1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S3</td>
<td>S1</td>
<td>S4</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S3</td>
<td>S2</td>
<td>S4</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>S1</td>
<td>S0</td>
<td>S4</td>
<td>S5</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td>S5</td>
<td>1</td>
</tr>
<tr>
<td>S5</td>
<td>S1</td>
<td>S4</td>
<td>S0</td>
<td>S5</td>
<td>0</td>
</tr>
</tbody>
</table>

*symbolic state transition table*

Minimized FSM

- Implication Chart Method
  - Cross out incompatible states based on outputs
  - Then cross out more cells if indexed chart entries are already crossed out

*minimized state table* $$(S0'=S4) (S3'=S5)$$
Incompletely Specified FSMs

- Equivalence of states is transitive when machine is fully specified
- But its not transitive when don't cares are present
  
  e.g.,
  
<table>
<thead>
<tr>
<th>state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>0</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>1</td>
</tr>
</tbody>
</table>

  S1 is compatible with both S0 and S2
  but S0 and S2 are incompatible

- No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states