Logic Simplification
Often it is possible to simplify function expressions from, say, their minterm form.

This may be done by using various identities, as we know. But more systematic methods will be shown.
Simplification Example

3-argument majority function

<table>
<thead>
<tr>
<th>v w x</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

minterm form: majority(v, w, x) =
Notice Certain “Adjacencies”

3-argument majority function

<table>
<thead>
<tr>
<th>v</th>
<th>w</th>
<th>x</th>
<th>f</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>

These rows are “adjacent”: their variable combinations differ in only 1 bit

simplified majority(v, w, x) = \[ wx + vw'x + vwx' \]

unsimplified majority(v, w, x) = \[ v'wx + vw'x + vwx' + vwx \]
Any other Adjacencies?

(ok to use a row more than once)

<table>
<thead>
<tr>
<th>v</th>
<th>w</th>
<th>x</th>
<th>f</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>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>
Adjacencies can Overlap

3-argument majority function

<table>
<thead>
<tr>
<th>v</th>
<th>w</th>
<th>x</th>
<th>f</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>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>0</td>
<td>1</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>

simplified majority \((v, w, x) = wx + \boxed{vx} + \text{vw}'x\)

unsimplified majority \((v, w, x) = v'wx + \boxed{vw'x} + \text{vwx}' + \boxed{vwx}\)
Any More Adjacencies?

3-argument majority function

<table>
<thead>
<tr>
<th>v</th>
<th>w</th>
<th>x</th>
<th>f</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>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>

simplified majority(v, w, x) = wx + vx + vwx’

ununsimplified majority(v, w, x) = v’wx + vw’x + vwx’ + vwx
Any More Adjacencies?

3-argument majority function

<table>
<thead>
<tr>
<th>v</th>
<th>w</th>
<th>x</th>
<th>f</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>

simplified majority(v, w, x) = wx + vx + \( \overline{vw} \)

unsimplified majority(v, w, x) = v’wx + vw’x + \( \overline{vwx’} \) + vw
majority(v, w, x) = v'wx + vw'x + vx' + vx

unsimplified majority
simplified majority

\[ \text{majority}(v, w, x) = wx + vx + vw \]
Simplified/Unsimplified Comparison for Majority

- Draw the diagrams
- Unsimplified:
  4 and-gates, 3 inputs each,
  some with negation,
  1 4-input or-gate
- Simplified:
  3 and-gates, 2 inputs each,
  none with negation,
  1 3-input or-gate
Implications for Simplified Functions

- If communicated by human, easier to understand, transfer
- If implemented as hardware, fewer gates, wires
- If implemented as software, fewer tests, execution steps
Visualizing Adjacencies

- Hypercube representation of truth table
- Karnaugh map representation
Hypercubes
(connected vertices = adjacent)

1-dimension = 1 logical variable
2-dimensions = 2 logical variables
Hypercubes

3-dimensions = 3 logical variables

![Diagram of a hypercube with labels w, w', v, v', x, x']
Sub-cubes

- An n-dimensional hypercube, for $n > 0$, has embedded in it a set of $m$-dimensional hypercubes, for each $m < n$.

- These are called the sub-cubes of the original hypercube.
Sub-cubes
Which sets are subcubes?
“Plotting” Functions on Hypercubes

\[ \text{majority}(v, w, x) = v'wx + vw'x + vx' + vw'x' \]

Each minterm is a vertex
Simplified Functions on Hypercubes

\[ \text{majority}(v, w, x) = wx + vx + vw \]

each term is a sub-cube
Sub-Cube Dimensions

- The dimension of a sub-cube is
  
n minus (number of variables in the corresponding product term)

- where \( n \) is the total number of variables.

- Examples: 3 variables total:
  - dimension 0: 3 variables
  - dimension 1: 2 variables
  - dimension 2: 1 variables
  - dimension 3: 0 variables (= constant 1)

- “More is less” (Greater dimension, fewer variables)
SOP Circuits
(sum-of-products)

- An SOP will always correspond to
  - a set of sub-cubes
  - Collectively the vertices in the sub-cubes cover the “on” vertices.
  - The vertices in the sub-cubes don’t cover any “off” vertices.
- Such a set is called a cover for the function.
- Each cover corresponds to a different gate implementation.
Simplified Covers

- In a fully-simplified SOP, each sub-cube will be *maximal*, in that it is not contained within another sub-cube.

- If instead it were properly contained in another sub-cube, then it could be replaced with that sub-cube.

- The replacement would have *fewer* variables, i.e. be simpler.
Maximality

This set of 2 sub-cubes covers the function.
Maximality

However the red sub-cube is not maximal. It should be extended to the blue sub-cube as shown for greater simplification. The resulting cover is simpler, even though one vertex is covered twice.
More-is-Less Principle

Making the sub-cubes as large as possible, makes the resulting product term as small as possible.
Einstein’s Principle:

Explanations should be made as simple as possible, but no simpler.

Including x by itself will not work!
Non-Uniqueness of Simplest Cover
Non-Uniqueness of Simplest Cover
4-D Hypercube

$H_4$

$(H_3)$

$(H_3)$
Alternate 4-D hypercube representation
(= “shadow” of a Tesseract)
Plotting Function on 4-D Hypercube
Plotting Function on 4-D Hypercube
Plotting Function on 4-D Hypercube
Plotting Function on 4-D Hypercube
Plotting Function on 4-D Hypercube
Plotting Function on 4-D Hypercube
Plotting Function on 4-D Hypercube
Plotting Function on 4-D Hypercube
Hypercube Function-Plotting Applet

An applet conceived by R. Keller and implemented by Ian Weiner, HMC '01 as a CS60 project.

http://www.cs.hmc.edu/~keller/javaExamples/hypercube

Check out dimensions > 4.
Path touches every node without retracing any arc (Hamiltonian path).
Other CS uses of Hypercubes

- Distributed parallel computer topologies
- High-speed sorting
Non-CS uses of Hypercubes

Rotating hypercube from http://casa.colorado.edu/~ajsh/sr/hypercubel.html
Hypercubes occur wherever independent attributes are found: e.g. genetics, “complexity”.


More Hypercube Projections

From: http://www.cs.reading.ac.uk/archive/hypercubes/
Non-CS uses of Hypercubes: chemistry
Hypercube Game

“Sudden Death in the 4th Dimension”:

http://kbs.cs.tu-berlin.de/~jutta/swd/hyper-game.html

2 players,

Start:

Move:

Win = all of your pieces in sequence or star

(Grey wins.)  (Grey wins.)  (Black wins.)
Rubick’s Hypercube

http://www.rose-hulman.edu/~berglunb/Rubik/index.html
Karnaugh Maps

- Invented by Maurice Karnaugh, 1953, at Bell Labs

- A way to **visualize** hypercubes of up to 4 (and stretching to 5 or 6) dimensions

- Approach by “flattening” a hypercube

- A structured form of Venn Diagram
“3-D” Karnaugh Map

These connections are usually implicit.
Covering on a Karnaugh Map
Covering on a Karnaugh Map
Try this one

\[ X \]

\[ W \]

\[ V \]
Answer
Using a Karnaugh Map to Prove Equalities

Prove that $x(v+w) = xv + xw$
Using a Karnaugh Map to Prove Equalities

Prove that \( x(v+w) = xv + xw \)

\[
\begin{align*}
\text{X} & \quad \text{V+W} & \quad \text{X(V+W)} \\
\text{XV} & \quad \text{XW} & \quad \text{XV+XW}
\end{align*}
\]
4-D Karnaugh Map
Implied connections on 4-D Karnaugh Map
Minterm Numbering on 4-D Karnaugh Map

Usage: Given a function in the form \{1, 5, 7, 15\}, for example

Pattern, horizontal and vertical
Assorted Sub-Cubes on 4-D Karnaugh Maps
Try this one
Exercises, etc.

- **Project:** Translate the hypercube game into a Karnaugh-map game.

- **Exercise:** Why is a Gray code sometimes called a “snake-in-the-box” code?
Simplification with “Don’t Care” Combinations

- For certain problems, certain combinations (truth-table rows) never arise in actual operation.

- These are called “don’t care” combinations.

- Because their input combinations never occur, the function value can be either 0 or 1. This means we can elect which value to use at our discretion.

- Usually we elect whichever value achieves the best simplification of the result.
Example: mod 3 adder

<table>
<thead>
<tr>
<th>xy</th>
<th>uv</th>
<th>uv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Only 9 of 16 combinations actually occur, leaving 7 don’t care ones. The don’t cares are the same for both functions.
Plot Function on a K-Map

<table>
<thead>
<tr>
<th>xy</th>
<th>uv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

○ = don’t care

u’vx’y

u’v’xy’
Plot Function on a K-Map

○ = don’t care

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

Final expression: \( vy + u'v'x + ux'y' \)

vs. original

\( u' v' w x' + u' v w' x + u v' w x' \)

We elect not to cover these.
Do the other half.

○ = don’t care

<table>
<thead>
<tr>
<th>xy</th>
<th>uv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
</tr>
<tr>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>
The other half.

○ = don’t care

<table>
<thead>
<tr>
<th>xy</th>
<th>uv</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0 1 0</td>
</tr>
<tr>
<td>01</td>
<td>1 0 0</td>
</tr>
<tr>
<td>10</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>
The other half.

\[ u'v'y + vx'y' + ux \]

vs. original

\[ u' v' w' x + u' v w' x' + u v' w x' \]
Implementation using NAND gates

- In some families of logic, *and* - and *or* - gates might not be available.

- Instead the only gate is *nand* (negated *and*).

- The question of sufficiency arises.
Sufficiency

- We know that functions **and**, **or**, and **not** are sufficient to realize any logic function.

- Two proofs:
  - minterm expansion: a sum of minterms, and each minterm uses **and** and **not** only
  - Boole-Shannon expansion (applied recursively)
Sufficiency

- To show that a given set of gate types is sufficient, it is enough to show that \textit{and}, \textit{or}, and \textit{not} can be realized with those gate types.

- Actually it is sufficient to show just \textit{and} and \textit{not} can be realized, since \textit{or} can be realized as:

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

  by DeMorgan's law. So if \textit{and} and \textit{not} are realizable, so is \textit{or}. 
**nand alone is sufficient**

- \( a' = \text{nand}(a, a) \)
- \( \text{and}(a, b) = \text{nand}(a, b)' \)
- \( \text{or}(a, b) = \text{nand}(a', b') \)
  \[= \text{nand} (\text{nand}(a, a), \text{nand}(b, b))\]
Exercises

- *nor* alone is sufficient.
- Is *xor* alone sufficient?
Bubble Logic

nand

or

DeMorgan:

\[ \text{nand}(a, b) = \text{and}(a, b)' = \text{or}(a', b') \]

nand, another way

\[ (a')' = a \]
SOP form using nand's only
Logic Rebuses

b
B

??

v
w
x

moral

??
Common Logic Packages

- mod-2 adder
- mod-2 adder with carry-in
- 4-bit adder
- decoder (binary to one-hot)
- encoder (one-hot to binary)
- (MUX) multiplexor
- (DMUX) demultiplexor
mod-2 adder with carry-out

\[
\text{sum}(x, y) = x \oplus y = xy' + x'y
\]

\[
\text{carry}(x, y) = x y
\]
mod-2 adder with carry-in/out

\[ \text{sum}(x, y, c) = x \oplus y \oplus c \]

\[ \text{carry}(x, y, c) = x \cdot y + x \cdot c + y \cdot c = \text{majority}(x, y, c) \]
4-bit ripple-carry adder

Note: There are other ways to implement this.
4-bit decoder

binary code
\[ x_0 \quad x_1 \]

\{ \]

\text{one-hot code}
\[ x_0' \quad x_1' \quad x_0 x_1' \quad x_0' x_1 \]
4-bit encoder

one-hot code

\[ \begin{align*}
&x_0 \\
&x_1 \\
&x_2 \\
&x_3 \\
\end{align*} \]

binary code

\[ \begin{align*}
&y_0 \\
&y_1 \\
\end{align*} \]

unlisted combinations are assumed not to occur

\[
\begin{array}{c|cc|cc}
\text{x}_0 \text{x}_1 \text{x}_2 \text{x}_3 & \text{y}_1 & \text{y}_0 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 \\
\end{array}
\]
multiplexor

\[ a_0' a_1' x_0 + a_0' a_1 x_1 + a_0 a_1' x_2 + a_0 a_1 x_3 \]

address lines
demultiplexor

address lines

$\begin{align*}
\text{x} & : a_0 a_1 x \\
a_0 a'_1 x & \\
a'_0 a_1 x & \\
a_0 a'_1 x & \\
a_0 a_1 x & \end{align*}$
Exercises

- How would you build a decoder out of a demultiplexor?
- How would you build a 16-way multiplexor out of 4-way multiplexors?
- How would you build a 16-way demultiplexor out of 4-way demultiplexors?
- How would you use a decoder to build a decoder ring?