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

<table>
<thead>
<tr>
<th>3-argument majority function</th>
<th>v</th>
<th>w</th>
<th>x</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

minterm form: $\text{majority}(v, w, x) = \sum m(0, 1, 3, 4, 5, 7)$

Notice Certain "Adjacencies"

3-argument majority function

$\begin{array}{ccc|c}
  v & w & x & f \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 1 & 1 & 1 \\
  1 & 0 & 0 & 0 \\
  1 & 0 & 1 & 1 \\
  1 & 1 & 0 & 1 \\
  1 & 1 & 1 & 1 \\
\end{array}$

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

simplified majority $(v, w, x) = wx + w'x + wx'$

unsimplified majority $(v, w, x) = vwx + vwx' + wwx$

Any other Adjacencies?

(ok to use a row more than once)

3-argument majority function

$\begin{array}{ccc|c}
  v & w & x & f \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 1 & 1 & 1 \\
  1 & 0 & 0 & 0 \\
  1 & 0 & 1 & 1 \\
  1 & 1 & 0 & 1 \\
  1 & 1 & 1 & 1 \\
\end{array}$

unsimplified majority $(v, w, x) = vwx + vwx' + wwx$

Adjacencies can Overlap

3-argument majority function

$\begin{array}{ccc|c}
  v & w & x & f \\
  0 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 1 & 1 & 1 \\
  1 & 0 & 0 & 0 \\
  1 & 0 & 1 & 1 \\
  1 & 1 & 0 & 1 \\
  1 & 1 & 1 & 1 \\
\end{array}$

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

unsimplified majority $(v, w, x) = vwx + vwx' + wwx + wwx'$
Any More Adjacencies?

3-argument majority function

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

unsimplified majority\( (v, w, x) = v'wx + vx' + vwx \)

unsimplified majority

majority\( (v, w, x) = v'wx + vw'x + vwx' + vxw \)

simplified majority

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

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 call the sub-cubes of the original hypercube.

Which sets are subcubes?
"Plotting" Functions on Hypercubes

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

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
  
  \[n - \text{number of variables in the corresponding product term}\]

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

- Examples: 3 variables total:
  - Dimension 0: 3 variables
  - Dimension 1: 2 variables
  - Dimension 2: 1 variable
  - 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

4-D Hypercube
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

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.

Gray Code on a Hypercube

Path touches every node without retracing any arc (Hamiltionian path).
Other CS uses of Hypercubes

- Distributed parallel computer topologies
- High-speed sorting

Non-CS uses of Hypercubes

Non-CS uses of Hypercubes: chemistry

Non-CS uses of Hypercubes: chemistry

Hypercubes occur wherever independent attributes are found: e.g. genetics, "complexity"

More Hypercube Projections

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
### 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

<table>
<thead>
<tr>
<th>The Cube</th>
<th>The Map</th>
</tr>
</thead>
</table>

- These connections are usually implicit.

#### Covering on a Karnaugh Map

- Try this one
Using a Karnaugh Map to Prove Equalities

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

4-D Karnaugh Map

Minterm Numbering on 4-D Karnaugh Map

Pattern, horizontal and vertical

Usage: Given a function in the form (1, 5, 7, 15), for example
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>x</th>
<th>y</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>u</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>01</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>00</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>1</td>
<td>1</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

- **u’v’xy’**
- **uv’xy’**
- **uxy’**
- **v’uxy’**
- **u’vxy’**
- **uxy**
Plot Function on a K-Map

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

Final expression: \( vy + u'v'x + ux'y' \) vs. original \( u'v'w'x + u'vwx' + x + u'vwx' \)

Do the other half.

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

Final expression: \( u'v'y + vx'y' + ux \) vs. original \( u'v'w'x + u'vwx' + x + u'vwx' \)

The other half.

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

Final expression: \( u'v'y + vx'y' + ux \) vs. original \( u'v'w'x + u'vwx' + x + u'vwx' \)

The other half.

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

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 **and**, **or**, and **not** can be realized with those gate types.
- Actually it is sufficient to show just **and** and **not** can be realized, since **or** can be realized as: 
  \[ a + b = (a' b') \]
  by DeMorgan's law. So if **and** and **not** are realizable, so is **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**

**DeMorgan:**

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

\[ \text{nand}, \text{another way} \]

\[ (a')' = a \]

**SOP form using nand's only**

**Logic Rebus**

- **b**
- **??**

- **??**
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

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

\[ \text{carry}(x, y) = xy \]

mod-2 adder with carry-in/out

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

\[ \text{carry}(x, y, c) = \text{majority}(x, y, c) \]

4-bit ripple-carry adder

4-bit decoder

4-bit encoder

Note: There are other ways to implement this.

unlisted combinations are assumed not to occur.
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?