L4: Karnaugh diagrams, two-, and multi-level minimization

Elena Dubrova
KTH / ICT / ES
dubrova@kth.se
A combinatorial system has no memory - its output depends therefore **ONLY** on the **PRESENT** value of the input signal.
A sequential system has a built-in memory - its output depends therefore **BOTH** on the **PRESENT** and **PREVIOUS** value(s) of the input signal.
This lecture covers ...

- BV pp. 168-211
Example: Minterms of the function are \( m(1,2,3) \) or

\[
f = \overline{x_1}x_0 + x_1\overline{x_0} + x_1x_0
\]

Using Boolean algebra, we can shorten the expression to:

\[
f = \overline{x_1}x_0 + x_1\overline{x_0} + x_1x_0 = \overline{x_1}x_0 + x_1(\overline{x_0} + x_0)
\]

\[
= \overline{x_1}x_0 + x_1 (1) = \overline{x_1}x_0 + x_1 (1 + x_0) = \\
= \overline{x_1}x_0 + x_1 + x_1x_0 = x_0 (\overline{x_1} + x_1) + x_1 = x_0 (1) + x_1 \\
= x_1 + x_0
\]
Minterms

• A minterm MUST contain all variables, otherwise it is not a minterm
• A minterm represents the combination of values of function's variables for which the function evaluates to 1

\[ f = \sum m(0,2,3) = \overline{x_1}x_2x_3 + \overline{x_1}x_2x_3 + x_1x_2x_3 \]
Maxterms

• A maxterm MUST contain all variables, otherwise it is not a maxterm
• A minterm represents the combination of values of function's variables for which the function evaluates to 0

\[ f = \prod \text{M (0,2,3)} = (x_1+x_2+x_3) (x_1+x_2+x_3) (x_1+x_2+x_3) \]
**Commonly used functions in two-dimensional table-form**

<table>
<thead>
<tr>
<th>x_0</th>
<th>x_1</th>
<th>x_0</th>
<th>x_1</th>
<th>x_0</th>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

AND

<table>
<thead>
<tr>
<th>x_0</th>
<th>x_1</th>
<th>x_0</th>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NAND

<table>
<thead>
<tr>
<th>x_0</th>
<th>x_1</th>
<th>x_0</th>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOR

<table>
<thead>
<tr>
<th>x_0</th>
<th>x_1</th>
<th>x_0</th>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

XNOR

<table>
<thead>
<tr>
<th>x_0</th>
<th>x_1</th>
<th>x_0</th>
<th>x_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NOT
3-dimensional Boolean space

Size: abc

0-dimensional subspace

1-dimensional subspace

2-dimensional subspace
Karnaugh map

Differ in the value of only one variable
Minimizing

\[ f(a, b, c) = \Sigma m (0, 2, 3, 5, 7) \]
Implicants

Circle the minterms located "next" to each other

\[ f(a,b,c) = \Sigma m(0,2,3,5,7) \]
A given product-term in a sum-of-products form consists of several variables
  – Each of the variables may appear complemented or non-complemented
  – These variables are called literals

\[ f(a,b,c) = \overline{a} \overline{b} \overline{c} + ab + bc \]
Minterms and product-terms

• A minterm always contains all variables of a function
• A product-term may contain less variables
  – Examples of minterms for three variable functions:
    abc, abc
  – Examples of product-terms for three variable functions:
    abc, ab, a
Implicants and Covers

- **Implicant** - A product-term for which the function evaluates to 1
- **Prime implicant** - An implicant which is not contained in any other implicant
  - A prime implicants cannot be expanded into a larger implicant
- **Cover** is a set of implicants which contains all minterms for which the function evaluates to 1
More implicant treminology

• A prime implicant is **essential** if there is a minterm covered by that implicant, but no other prime implicant
  – Essential imlicants will always be included in a cover of a function

• If we can remove an implicant, but all minterms are still covered, then such an implicant is called **redundant**
Example

Redundant implicants - both are not necessary to cover the function

\[ f(a, b, c) = \Sigma m(0, 2, 3, 5, 7) \]
Covers

- **Minimum** cover is a cover with the minimum number of cubes
- **Prime cover** is a cover consisting of only prime implicants
- **Irredundant** cover is a cover which does not contain any redundant implicants
Example

Example: \( f(a, b, c, d) = abc + bd + cd \) is minimum, prime and irredundant cover for \( f \)
Quine's Theorem: For any Boolean function, there exists a minimum cover which is prime

Consequence: The search for a minimum cover can be restricted to covers with prime implicants only.
Two-level minimization
Minimum sum-of-product implementation

\[ f = \overline{bc} + ac + bc \]
Both AND and OR arrays are programmable.
Programmable Array Logic (PAL)

- Only the AND arrays are programmable
- Which functions $P_1$, $P_2$, $P_3$ and $P_4$ represent?
Karnaugh map with 4 variables

We always circle an entire sub-space (as large as possible) !!!
Karnaugh map with 5 variables

\[ \overline{x_3 x_2 x_1 x_0} \]
Example

\[ \overline{x_3 x_2 x_1 x_0} \]
Karnaugh map with 6 variables

\[
\begin{array}{ccccccc}
\text{x}_4 \text{x}_1 \text{x}_0 & & & & & & \\
\text{x}_5 \text{x}_3 \text{x}_2 & 000 & 001 & 011 & 010 & 110 & 111 & 101 & 100 \\
000 & & & & & & & & \\
001 & & & & & & & & \\
011 & 1 & & & & & & \\
010 & & & & & & & & \\
110 & & & & & & & & \\
111 & & & & & & & & \\
101 & & & & & & & & \\
100 & & & & & & & & \\
\end{array}
\]

\[\text{x}_3 \text{x}_2 \text{x}_1 \text{x}_0\]
Example

\[ x_3 \bar{x}_2 x_1 x_0 \]
Alternative: Circle 0s

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>x1x0</td>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x3x2</td>
<td>1 0 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1 1 1 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Circle the zeros is zeros is less than ones !!!

\[
x_3 \cdot x_2 \cdot x_1 \cdot x_0 = x_3 + \overline{x_2} + x_1 + \overline{x_0}
\]
Incomplete functions with don’t cares

• Often you can simplify a specification of the logic function knowing that some input combinations never occur
• For these combinations, we use the value "don’t care"
• There are different symbols for "don't care"
  – 'd', 'D', '-', 'ϕ', 'x'
Specification of incomplete functions

(A) SOP implementation

(B) POS implementations

Two implementations of the function

\[ f(x_3, \ldots, x_0) = \Sigma m(2, 4, 5, 6, 10) + D(12, 13, 14, 15). \]
Two implementations of the function

\[ f(x_3, \ldots, x_0) = \Sigma m(2, 4, 5, 6, 10) + D(12, 13, 14, 15). \]
Functions with multiple outputs

Different outputs can share implicants!

\[ f_1 = f_0 + x_3x_0 \]
Multi-level minimization
Do we need multi-level logic?

- One can realize all the combinational circuits with two-level (AND-OR, OR-AND)
  - The assumption is that all inputs are also available in the inverted form (as in PAL, PLA)
Why multi-level logic?

- A multi-level implementation may have considerably less gates than a two-level implementation
Two strategies for multi-level logic

1. Factorisation
2. Functional Decomposition
• Suppose the following function is to be implemented

\[ f = x_1 \bar{x}_2 x_3 \bar{x}_4 x_5 x_6 + x_1 x_2 \bar{x}_3 \bar{x}_4 x_5 \bar{x}_6 \]
Factorisation

- If we use the distributive law (L2, s.21, 12a), we can get the following multi-level implementation:

\[
\begin{align*}
  f &= x_1 \bar{x}_2 x_3 \bar{x}_4 x_5 x_6 + x_1 x_2 \bar{x}_3 \bar{x}_4 \bar{x}_5 x_6 \\
  &= x_1 \bar{x}_4 x_6 \cdot (\bar{x}_2 x_3 x_5 + x_2 \bar{x}_3 \bar{x}_5)
\end{align*}
\]
Factorization

\[ f = x_1 \overline{x}_4 x_6 \cdot (\overline{x}_2 x_3 x_5 + x_2 \overline{x}_3 \overline{x}_5) \]
Functional decomposition

- One can often reduce the complexity of a logic function by reusing its sub-functions several times.
- For implementation, it means to share sub-circuits in its construction.
Functional Decomposition

• How can the following function implemented?

\[
f = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 x_4 + \overline{x}_1 \overline{x}_2 x_4
\]
Functional Decomposition

• Factorisation gives us

\[ f = \overline{x}_1 x_2 x_3 + x_1 \overline{x}_2 x_3 + x_1 x_2 x_4 + \overline{x}_1 \overline{x}_2 x_4 \]

\[ = (\overline{x}_1 x_2 + x_1 \overline{x}_2) x_3 + (x_1 x_2 + \overline{x}_1 \overline{x}_2) x_4 \]

• If we define a sub-function \( g \) as

\[ g = \overline{x}_1 x_2 + x_1 \overline{x}_2 \]
Functional Decomposition

- So, we get

\[ f = gx_3 + \overline{g}x_4 \]

- where

\[ g = \overline{x}_1x_2 + x_1\overline{x}_2 = x_1x_2 + \overline{x}_1\overline{x}_2 \]
Functional Decomposition

Implementation

\[ f = gx_3 + \overline{g}x_4 \]

\[ g = \overline{x}_1x_2 + x_1\overline{x}_2 = x_1x_2 + \overline{x}_1\overline{x}_2 \]
Algorithms for minimization

• Karnaugh map minimization gives a good insight into how to minimize logic functions
• But to minimize the complex functions with the help of computer, there are better algorithms
• Chapter 4.9 and 4.10 in Brown/Vranesic provides an introduction to the minimization algorithms (for the interested student)
Summary

• Karnaugh maps are a good tool for minimizing logic functions with a few variables
• There are algorithms for both two-level and multi-level minimization
• Next lecture: BV pp. 250-280