Combinational Circuit Minimization
Circuit minimization

• All logic functions can be implemented with two-level logic networks
  – Canonical sum (minterms) or canonical product (maxterms)
  – This becomes intractable when there are a lot of inputs, e.g. \( n = 64 \)
• Minimize circuit: reduce the cost to build the circuit while satisfying all the constraints
  – Constraints: time (circuit delay), area, power, etc.
  – Cost: area, time to market, etc.
• The minimization method we will study reduces the cost of two-level AND-OR, OR-AND, NAND-NAND, and NOR-NOR circuits
• A minimal sum of a logic function \( F \) is a sum-of-products expression for \( F \) such that no sum-of-products expression for \( F \) has fewer product terms; and any sum-of-products expression with the same number of product terms has at least as many literals. Basically this is a sum with fewest product terms and fewest literals in the expression.
Goals and assumption

- Minimizing the number of first-level gates
- Minimizing the number of inputs on each first-level gates
- Minimizing the number of inputs on the second-level gate
- We assume both the true and complemented forms of all the input signals are available.
- Now let’s start with Section 4.3.4 on Karnaugh Maps, also known as K-maps. A K-map is but a graphical representation of the truth table.
- The K-map for an $n$-input logic function is an array with $2^n$ cells, one for each minterm.
Karnaugh maps

Karnaugh map of function of one variable.

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>f(0)</td>
</tr>
<tr>
<td>1</td>
<td>f(1)</td>
</tr>
</tbody>
</table>

\[ (a) \]

<table>
<thead>
<tr>
<th>x</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(0)</td>
<td>f(1)</td>
<td></td>
</tr>
</tbody>
</table>

\[ (b) \]
Visualizing T10 – Karnaugh maps (1)

T10: \[ X \cdot Y + X \cdot Y' = X \]

Adjacent cells can be merged (reduced)

Minterm 5: \[ W' \cdot X \cdot Y' \cdot Z \]
Minterm 13: \[ W \cdot X \cdot Y' \cdot Z \]

\[ X \cdot Y' \cdot Z \]
Visualizing T10 – Karnaugh maps (2)

T10: \( X \cdot Y + X \cdot Y' = X \)

Minterm 1: \( W' \cdot X' \cdot Y' \cdot Z \)
Minterm 9: \( W \cdot X' \cdot Y' \cdot Z \)

\[ X' \cdot Y' \cdot Z \]
Visualizing T10 – Karnaugh maps (3)

Minterm 5 and Minterm 13:
\[ X \cdot Y' \cdot Z \]

Minterm 1 and Minterm 9:
\[ X' \cdot Y' \cdot Z \]
\[ Y' \cdot Z \]

Each mergence removes one literal
\[ 2^i \text{ cells } \rightarrow (n - i) \text{ literals} \]

Corresponding product terms:
covers only 1: variables
covers only 0: complement of variables
covers both 0 and 1: not included
Merging cells
Example: $F = \Sigma(1,2,5,7)$

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</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>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Karnaugh-map usage

- Plot 1’s corresponding to minterms of function
- Circle largest possible rectangular sets of 1’s
  - Number of 1’s in a set must be power of 2
  - OK to cross edges
- Read off product terms, one per circled set
  - Variable is 1 ==> include variable
  - Variable is 0 ==> include complement of variable
  - Variable is both 0 and 1 ==> variable not included
- Circled sets and corresponding product terms are called “implicants”
  - The largest circles are called “prime implicants”
Concepts

• **Imply**: Logic expression $F_1$ implies $F_2$ if any assignment of values to the variables involved makes $F_1 = 1$, it also makes $F_2 = 1$.
  - $F_1 = X' \cdot Z + Y' \cdot Z$ implies $F_2 = X' \cdot Y + Y' \cdot Z$

• A product term is said to be an **implicant** of a logic function if it implies the function
  - In SOP, any product is an implicant

• A term $T_1$ **subsumes** $T_2$ if any literal that occurs in $T_2$ also appears in $T_1$
  - $W \cdot X' \cdot Y \cdot Z$ subsumes $X' \cdot Z$
  - $X + Y$ subsumes $X$

• An implicant is said to be a **prime implicant** if the implicant does not subsume any other implicant with fewer literals of that function

• Example: $F(X, Y, Z) = X' \cdot Y + Z$.
  - Both $X' \cdot Y$ and $X \cdot Z$ are implicant of $F$.
  - However, $X \cdot Z$ subsumes $Z$. So $X \cdot Z$ is **not** a prime implicant.
Complete sum

• The sum of all the prime implicants of a logic function is called the complete sum.
  – The complete sum is not necessarily a minimal sum.
Prime Implicant Theorem

A minimal sum is a sum of prime implicants.

Proof: (By contradiction). Think about this like a logician.
Suppose a product term P in a minimal sum is not a prime implicant.
According to the definition of prime implicant, if P is not a prime implicant, it is possible to remove some literals from P to obtain a new product term P* that still implies F.
If we replace P with P* in the minimal sum, the resulting sum still equals to F but has fewer literals.
Therefore, the presumed minimal sum P is not minimal.

• See http://www.engr.uconn.edu/~stc11012/courses/CSE2300W/K-map_Terms.pdf for terminology related to prime implicants.
Which prime implicants to choose?

- **Distinguished 1-cell** of a logic function is a 1 cell that is covered by *only one prime implicant*
- **An essential prime implicant** of a logic function is a prime implicant that covers one or more distinguished 1-cells
- Choose all the essential prime implicants first
- What if some 1’s are not covered?
Example:

What’s left after we remove the EPIs and the distinguished 1-cells they cover? Use A’•D or B•C•D?
Secondary essential prime implicants

- Removed the 1-cells covered by essential prime implicants
- Remove some prime implicants
  - Given two prime implicants P and Q, P is said to \textit{eclipse} Q (written \( P \triangleright Q \)) if P covers at least all 1-cells covered by Q
  - A prime implicant Q is removed if there exists P
    - P costs no more than Q
    - \( P \triangleright Q \) (i.e., P covers all 1 cells covered by Q)
- After removing some prime implicants, some 1-cells are only covered by one implicant. Those implicants are called \textit{secondary essential prime implicants}.
  - Secondary essential prime implicants must be included in the minimal sum (assuming we can find such prime implicants)
Example: Remove essential prime implicants

\[ F = A' \cdot C' + A' \cdot B' + A \cdot B \cdot C \]
Example: Remove more prime implicants

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

\((A' \cdot D) \ldots (B \cdot C \cdot D)\)

\((B \cdot C \cdot D)\) is more expensive

Remove (B \cdot C \cdot D) !!!
Example: Include secondary essential prime implicants

\[
\begin{align*}
A' \cdot C' \\
A' \cdot D \\
A' \cdot B' \\
B \cdot C \cdot D \\
A \cdot B \cdot C \\
F = \\
A' \cdot C' + \\
A' \cdot B' + \\
A \cdot B \cdot C + \\
A' \cdot D
\end{align*}
\]
Example 2:

Include essential prime implicants (blue circles)

Remove 1 cells covered by essential prime implicant
Example 2 (continued)

Remove red prime implicants

Can you remove the green prime implicant?
What if no essential prime implicants can be found?

- Trial and error
- Branching method
Two minimal sums

Both equations have the same cost.

Not always true.
Prime-number detector (again)

\[ F = \Sigma_{N_3,N_2,N_1,N_0}(1,2,3,5,7,11,13) \]

\[ F = N_3 \cdot N_0 + N_3 \cdot N_2' \cdot N_1 + N_2' \cdot N_1 \cdot N_0 + N_2 \cdot N_1' \cdot N_0 \]
When we solved algebraically, we missed one simplification -- the circuit below has three less gate inputs.
Another example

F = $\Sigma_{W,X,Y,Z}(5,7,12,13,14,15)$

F = $X \cdot Z + W \cdot X$
Yet another example

\[ F = \Sigma_{W,X,Y,Z}(1, 3, 4, 5, 9, 11, 12, 13, 14, 15) \]

\[ F = X \cdot Y' + X' \cdot Z + W \cdot X \]
Yet another example

\[ F = \Sigma_{W,X,Y,Z}(2,3,4,5,6,7,11,13,15) \]

\[ F = W' \cdot Y + W' \cdot X + X \cdot Z + Y \cdot Z \]
Steps in K-map reduction

1. Merge 1 cells to get the largest circles (identify prime implicants)
2. Identify distinguished 1 cells
   • Find essential prime implicants
     • Secondary, tertiary, …
     • If no essential prime implicants can be found → Trial and error
3. Include essential prime implicants in the minimal sum
4. Are all 1 cells covered?
   • If yes, you are done.
5. Remove all 1 cells already covered
6. Remove some prime implicants
   • Remove Q if
     • P covers all the 1 cells covered by Q, i.e., (P … Q)
     • Cost(P) ≤ Cost (Q)
7. Goto step 2
K-map with 5 variables

V=0

V=1
Example of 5-variable K-maps

\[ D_1 = Q_1 + Q_2' \cdot Q_3' \]

\[ D_2 = Q_1 \cdot Q_3' \cdot A' + Q_1 \cdot Q_3 \cdot A + Q_1 \cdot Q_2 \cdot B \]

\[ D_3 = Q_1 \cdot A + Q_2' \cdot Q_3' \cdot A \]
K-map with 6 variables
Simplifying with Products of Sum (i.e. maxterms)

\[
F = (A' + B') \cdot (C' + D') \cdot (B + D')
\]

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

\[
F = \text{(using 1-minterms)}
\]

\[
B' \cdot D' + A' \cdot D' + A' \cdot B \cdot C'
\]

Which gives you a minimal sum?
Don’t-care minimization

- **Don’t cares**: the output does not matter for these input combinations, or they never appear as valid inputs

\[ F = \Sigma(4, 12, 13, 14, 15) + d(0, 5, 8) \]

**Minimal Sum:**

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

or

\[ F = A \cdot B + C' \cdot B \]

**Minimal Product (from bottom):**

\[ F = B \cdot (A + C') \]
Differences when don’t cares are considered

- When identifying the prime implicants, consider x as 1 if doing so gives larger circles
  - Each prime implicant covers at least one 1-cell

- When identifying essential prime implicants
  - The definition of distinguished 1-cell is the same
  - X’s do not make a prime implicant essential
Another example: BCD Prime

\[ F = \Sigma(2, 3, 5, 7) + d(10, 11, 12, 13, 14, 15) \]
\[ = B \cdot D + B' \cdot C \]
Real-world logic design

- Lots have more than 6 inputs $\Rightarrow$ can’t use K-maps practically
- Design correctness more important than gate minimization
  - Use “higher-level language” to specify logic operations
- Use programs to manipulate logic expressions and minimize logic circuit implementation
- PALASM, ABEL, CUPL – developed for PLD’s
- VHDL, Verilog – developed for ASIC’s
- Chapter 5 of the textbook contains details.
Quine-McCluskey algorithm

- This process can be made into a program, using appropriate algorithms and data structures.
  - Guaranteed to find the “minimal” solution
- Required computation has exponential complexity (run time and storage) – works well for functions with up to 8-12 variables, but blows up (space & run time) for larger problems.
  - Because min-cover problem is NP-hard (? See algorithm literature)
- Heuristic programs (e.g., Espresso) used for larger problems, usually give minimal results.
Multiple-level circuit

- More cost saving is obtained with multiple-level circuit optimization.
  - $G = A \cdot B \cdot C + A \cdot B \cdot D + E + A \cdot C \cdot F + A \cdot D \cdot F$  
  - $G = A \cdot B \cdot (C + D) + E + A \cdot (C + D) \cdot F$  
  - $G = (A \cdot B + A \cdot F) \cdot (C + D) + E$  
  - $G = A \cdot (B + F) \cdot (C + D) + E$

- Common techniques:
  - Factoring  
    Find the factored form
  - Decomposition  
    Express a function as a set of new functions
  - Extraction  
    Express a set of function as a set of new functions
  - Substitution  
    Substituting $G$ into a function $F$: express $F$ as a function of $G$ and some or all variables in $F$. 
  - Elimination  
    Inverse of substitution. Replace $G$ with its expression in function $F$. Also called flattening or collapsing.

- Algorithms: Boolean networks, factored form
Timing hazards

- We have studied **steady-state** behavior
  - Assuming inputs have been stable for a long time
- The **transient behavior** of circuits may differ
  - A circuit’s output may produce a pulse, called a **glitch**
- A **hazard** is said to exist when a circuit has a possibility of producing such a glitch in transient
Static and dynamic hazards

- **Static-1 hazard**: circuit may produce a 0 when we expect a steady 1 at the output (thus the glitch)
  - Find a pair of input combinations
    - Both result in a 1 output value for the function, and
    - Differ only in one input variable, two PI’s next to each other without “overlapping” 1-cells

- **Static-0 hazard**: circuit may produce a 1 when we expect a steady 0 at the output (thus the glitch)
  - Properly designed AND-OR circuits do not have static-0 hazards

- **A dynamic hazard** is the possibility of an output changing more than once as the result of a single input transition.
A circuit with a static-1 hazard

Inputs X, Y, Z change from 111 to 110
ZP changes from 0 to 1 after one NOT gate delay
YZ changes from 1 to 0 after one AND gate delay
X \cdot ZP changes from 0 to 1 after two gate (NOT+AND) delays
F = X \cdot ZP + YZ changes on 3\textsuperscript{rd} cycle
Finding the hazard for the circuit on K-map

Suppose X and Y are 1 and Z changes from 1 to 0.

\[ F = X \cdot Z' + Y \cdot Z \]

<table>
<thead>
<tr>
<th>Y\cdot Z</th>
<th>X\cdot Z'</th>
<th>X\cdot Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

On K-map, any adjacent 1’s need to be covered by a PI.
Circuit with static-1 hazard eliminated