Chapter #2: Two-Level Combinational Logic

Motivation

Further Amplification on the Concepts of Chapter #1:
- Rapid prototyping technology
  Use of computer aided design tools: espresso
- Design Techniques that Spanning Multiple Technologies
  Transistor-Transistor Logic (TTL)
  Complementary Metal on Oxide Silicon (CMOS)
- Multiple Design Representations
  Truth Tables
  Static gate descriptions
  Dynamic waveform descriptions

Chapter Overview

- Logic Functions and Switches
  Not, AND, OR, NAND, NOR, XOR, XNOR
- Gate Logic
  Laws and Theorems of Boolean Algebra
  Two Level Canonical Forms
  Incompletely Specified Functions
- Two Level Simplification
  Boolean Cubes
  Karnaugh Maps
  Quine-McClusky Method
  Espresso Methods

Logic Functions: Boolean Algebra

Algebraic structure consisting of:
set of elements \( B \)
binary operations \( \{+\,\cdot\} \)
unary operation \( \{'\} \)
such that the following axioms hold:

1. \( B \) contains at least two elements, \( a, b \), such that \( a \cdot b \)

2. Closure \( a, b \) in \( B \),
   (i) \( a + b \) in \( B \)
   (ii) \( a \cdot b \) in \( B \)

3. Commutative Laws: \( a, b \) in \( B \),
   (i) \( a + b = b + a \)
   (ii) \( a \cdot b = b \cdot a \)

4. Identities: \( 0, 1 \) in \( B \)
   (i) \( a + 0 = a \)
   (ii) \( a \cdot 1 = a \)

5. Distributive Laws:
   (i) \( a + (b \cdot c) = (a + b) \cdot (a + c) \)
   (ii) \( a \cdot (b + c) = a \cdot b + a \cdot c \)

6. Complement:
   (i) \( a + a' = 1 \)
   (ii) \( a \cdot a' = 0 \)
Logic Functions: Boolean Algebra

must verify that the axioms hold:

E.g., Commutative Law:

$0 + 1 = 1 + 0$?
$0 \cdot 1 = 1 \cdot 0$?
$1 = 1$?

Theorem: any Boolean function that can be expressed as a truth table can be written as an expression in Boolean Algebra using $', +, \cdot$

E.g., Commutative Law:

$X, X', Y, Y', X \cdot Y, X + Y, 0, 1$ only

half of the possible functions

Two-Level Logic

Listed

More than one way to map an expression to gates

E.g.,

$Z = A' \cdot B' \cdot (C + D) = (A' \cdot (B' \cdot (C + D)))$

use of 3-input gate

Literal: each appearance of a variable or its complement in an expression

E.g.,

$Z = AB' + A'B + A'B'C + B'C$

3 variables, 10 literals

Contemporary Logic Design

Two-Level Logic

NAND, NOR gates far outnumber AND, OR in typical designs
easier to construct in the underlying transistor technologies

Any Boolean expression can be implemented by NAND, NOR, NOT gates

In fact, NOT is superfluous

(NOT = NAND or NOR with both inputs tied together)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X NOR Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X NAND Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Logic Functions: XOR, XNOR

XOR: X or Y but not both ("inequality", "difference")

XNOR: X and Y are the same ("equality", "coincidence")

Description
Z = 1 if X has a different value than Y
Z = 1 if X has the same value as Y

Gates
\[ X \oplus Y = X Y' + X' Y \]

Truth Table
\[
\begin{array}{ccc}
X & Y & Z \\
0 & 0 & 0 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

(a) XOR

\[ X \oplus Y = X Y' + X' Y \]

Description
Z = 1 if X has a different value than Y

Gates
\[ X \oplus \bar{Y} = X Y + X' Y' \]

Truth Table
\[
\begin{array}{ccc}
X & Y & Z \\
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 1 \\
\end{array}
\]

(b) XNOR

Logic Functions: Rationale for Simplification

Logic Minimization: reduce complexity of the gate level implementation
- reduce number of literals (gate inputs)
- reduce number of gates
- reduce number of levels of gates

fewer inputs implies faster gates in some technologies
fan-ins (number of gate inputs) are limited in some technologies
fewer levels of gates implies reduced signal propagation delays
minimum delay configuration typically requires more gates
number of gates (or gate packages) influences manufacturing costs

Traditional methods:
reduce delay at expense of adding gates

New methods:
trade off between increased circuit delay and reduced gate count

Logic Functions: Waveform View

Logic Functions: Alternative Gate Realizations

Two-Level Realization
(inverters don't count)

Multi-Level Realization
Advantage: Reduced Gate Fan-ins

Complex Gate: XOR
Advantage: Fewest Gates

TTL Package Counts:
Z1 - three packages (1x 6-inverters, 1x 3-input AND, 1x 3-input OR)
Z2 - three packages (1x 6-inverters, 1x 2-input AND, 1x 2-input OR)
Z3 - two packages (1x 2-input AND, 1x 2-input XOR)
Logic Functions: Waveform Verification

Under the same input stimuli, the three alternative implementations have essentially the same waveform behavior.

Slight variations due to differences in number of gate levels

The three implementations are equivalent

Gate Logic: Laws of Boolean Algebra

Duality: a dual of a Boolean expression is derived by replacing AND operations by ORs, OR operations by ANDs, constant 0s by 1s, and 1s by 0s (literals are left unchanged).

Any statement that is true for an expression is also true for its dual!

Useful Laws/Theorems of Boolean Algebra:

Operations with 0 and 1:
1. \( X + 0 = X \)
2. \( X + 1 = 1 \)

Idempotent Law:
3. \( X + X = X \)

Involution Law:
4. \( (X')' = X \)

Laws of Complementarity:
5. \( X + X' = 1 \)
6. \( X + Y = Y + X \)

Commutative Law:
7. \( X \cdot Y = Y \cdot X \)
8. \( X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z) \)

Associative Laws:
9. \( (X + Y) + Z = X + (Y + Z) \)
10. \( X \cdot (Y \cdot Z) = (X \cdot Y) \cdot Z \)

Distributive Laws:
11. \( X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z) \)
12. \( X + (Y \cdot Z) = (X + Y) \cdot (X + Z) \)

Simplification Theorems:
13. \( X \cdot Y + X \cdot Y' = X \)
14. \( X + X \cdot Y + X \cdot Y' = X \)

DeMorgan's Law:
15. \( (X + Y + Z + ...) = X' \cdot Y' \cdot Z' + ... \)
16. \( (X + Y + Z + ...) = X \cdot Y \cdot Z \)

Theorems for Multiplying and Factoring:
17. \( (X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X \cdot Z \)
18. \( (X \cdot Y) \cdot (Y + Z) \cdot (X' + Z) = X' \cdot Y \cdot Z \)

Consensus Theorem:
19. \( (X \cdot Y) + (Y \cdot Z) + (X' \cdot Z) = X \cdot Y + X \cdot Z \)
20. \( (X \cdot Y) \cdot (Y + Z) \cdot (X' + Z) = X \cdot Y \cdot Z \)

Gate Logic: Laws of Boolean Algebra (cont)

Proving theorems via axioms of Boolean Algebra:

E.g., prove the theorem: \( X \cdot Y + X \cdot Y' = X \)

E.g., prove the theorem: \( X + Y = X \)
Gate Logic: Laws of Boolean Algebra

Proving theorems via axioms of Boolean Algebra:

E.g., prove the theorem: \( X \cdot Y + X \cdot Y' = X \)

- **distributive law (8)** \( X \cdot Y + X \cdot Y' = X \cdot (Y + Y') \)
- **complementary law (5)** \( X \cdot (Y + Y') = X \cdot (1) \)
- **identity (1D)** \( X \cdot (1) = X \)

E.g., prove the theorem: \( X + X \cdot Y = X \)

- **identity (1D)** \( X + X \cdot Y = X \cdot 1 + X \cdot Y \)
- **distributive law (8)** \( X \cdot 1 + X \cdot Y = X \cdot (1 + Y) \)
- **identity (2)** \( X \cdot (1 + Y) = X \cdot (1) \)
- **identity (1)** \( X \cdot (1) = X \)

**Example:**

\[
\begin{align*}
\text{Z} &= A' \cdot B' + A' \cdot B + A' \cdot C + A \cdot B' + A' \cdot C' + A \cdot B + C \\
\text{Z}' &= (A \cdot B + C') \cdot (A + B' + C') \cdot (A' + B + C)
\end{align*}
\]

**De Morgan's Law**

\[
(X + Y)' = X' \cdot Y'
\]

NOR is equivalent to AND with inputs complemented

\[
\begin{array}{c|cc|cc|cc}
X & Y & X & Y & X & Y \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
\]

De Morgan's Law can be used to convert AND/OR expressions to OR/AND expressions

**Example:**

\[
\text{Z} = A' \cdot B' \cdot C + A' \cdot B \cdot C + A \cdot B' \cdot C + A \cdot B \cdot C'
\]

\[
\text{Z}' = (A \cdot B + C') \cdot (A + B' + C') \cdot (A' + B + C) \\
\]

Apply the laws and theorems to simplify Boolean equations

**Example:**

**full adder's carry out function**

\[
\text{Cout} = A' \cdot B \cdot \text{Cin} + A \cdot B' \cdot \text{Cin} + A \cdot B \cdot \text{Cin}' + A \cdot B \cdot \text{Cin}
\]

- **identity**
- **associative**
Gate Logic: Two-Level Canonical Forms

Sum of Products, Products of Sums, and DeMorgan's Law

F' = A'B'C' + A'B'C + A'BC

Apply DeMorgan's Law to obtain F:

(F')' = (A'B'C' + A'B'C + A'BC)'

F = (A + B + C)(A + B + C')(A + B' + C)

F' = (A + B' + C')(A' + B + C)(A' + B' + C)(A' + B' + C)

Apply DeMorgan's Law to obtain F:

(F')' = ((A + B' + C')(A' + B + C)(A' + B' + C)(A' + B' + C))'

F = A'B'C + A'B'C' + A'B'C + A BC

Four Alternative Implementations of F:

Mapping Between Forms

1. Minterm to Maxterm conversion:
   rewrite minterm shorthand using maxterm shorthand
   replace minterm indices with the indices not already used

   E.g., F(A,B,C) = Σm(3,4,5,6,7) = ΠM(0,1,2)

2. Maxterm to Minterm conversion:
   rewrite maxterm shorthand using minterm shorthand
   replace maxterm indices with the indices not already used

   E.g., F(A,B,C) = ΠM(0,1,2) = Σm(3,4,5,6,7)

3. Minterm expansion of F to Minterm expansion of F':
   in minterm shorthand form, list the indices not already used in F

   E.g., F(A,B,C) = Σm(3,4,5,6,7) = ΠM(0,1,2) → F'(A,B,C) = Σm(0,1,2) = ΠM(3,4,5,6,7)

4. Minterm expansion of F to Maxterm expansion of F':
   rewrite in Maxterm form, using the same indices as F

   E.g., F(A,B,C) = Σm(3,4,5,6,7) = ΠM(0,1,2) → F'(A,B,C) = ΠM(3,4,5,6,7) = Σm(0,1,2)
Use OR gate if input polarities are neg.-logic

Voltage Table

\[
\begin{array}{cccc}
A & B & F & \overline{F} \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

Positive Logic

Negative Logic

Two Alternative Interpretations

Positive Logic: Active High

Negative Logic: Active Low

Conversion from Positive to Negative Logic

Two-Level Logic

Alternative Convention sometimes used: Negative Logic/Active Low

Low Voltage = 0; High Voltage = 1

Mismatches in logic polarities

Change Request

Request

Alternative Conventions sometimes used: Negative Logic / Active

Change Lights

(active high)

(active low)

Timer

Expired

Voltage Table

\[
\begin{array}{cccc}
A & B & F & \overline{F} \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
\end{array}
\]

Positive Logic

Negative Logic

Dual Operations

Behavior in terms of Electrical Levels

E.g.: Binary Coded Decimal (BCD) Increment by 1

On-set of W

Don't care (DC) set of W

These input patterns should never be encountered in practice

Associated output values are don't cares

Dual Operations:

AND becomes OR, OR becomes AND

Invert complements remain unchanged
**Gate Logic: Incompletely Specified Functions**

Don’t Cares and Canonical Forms

Canonical Representations of the BCD Increment by 1 Function:

\[ Z = m_0 + m_2 + m_4 + m_6 + m_8 + d_10 + d_11 + d_12 + d_13 + d_14 + d_15 \]

\[ Z = \sum m(0, 2, 4, 6, 8) + d(10, 11, 12, 13, 14, 15) \]

\[ Z = m_1 \cdot m_3 \cdot m_5 \cdot m_7 \cdot m_9 \cdot d_{10} \cdot d_{11} \cdot d_{12} \cdot d_{13} \cdot d_{14} \cdot d_{15} \]

\[ Z = \prod m(1, 3, 5, 7, 9) \cdot d(10, 11, 12, 13, 14, 15) \]

**Gate Logic: Two-Level Simplification**

Key Tool: The Uniting Theorem — \( A (B' + B) = A \)

\[ F = A B' + A B = A (B' + B) = A \]

B’s values change within the on-set rows

\( B \) is eliminated, \( A \) remains

A’s values don’t change within the on-set rows

**Essence of Simplification:**
find two element subsets of the ON-set where only one variable changes its value. This single varying variable can be eliminated!

**Gate Logic: Two-Level Simplification**

Algebraic Simplification:
not an algorithm/systematic procedure
how do you know when the minimum realization has been found?

Computer-Aided Tools:
precise solutions require very long computation times, especially for functions with many inputs (>10)

heuristic methods employed — "educated guesses" to reduce the amount of computation
good solutions not best solutions

Still Relevant to Learn Hand Methods:
insights into how the CAD programs work, and their strengths and weaknesses
ability to check the results, at least on small examples
don’t have computer terminals during exams

**Gate Logic: Two-Level Simplification**

Boolean Cubes
Visual technique for identifying when the Uniting Theorem can be applied

Just another way to represent the truth table
n input variables = n dimensional "cube"
Gate Logic: Two-Level Simplification

Mapping Truth Tables onto Boolean Cubes

ON-set = filled-in nodes
OFF-set = empty nodes
DC-set = X’d nodes

Cube of n-1 dimensions
Reduced expression contains n-1 variables

A asserted and unchanged
B varies within loop

A varies within loop
B complemented and unchanged

Subcubes of Higher Dimensions than 2

F(A,B,C) = \Sigma m(4,5,6,7)
On-set forms a rectangle,
i.e., a cube of two dimensions
represents an expression in one variable
i.e., 3 dimensions - 2 dimensions

A is asserted and unchanged
B and C vary

This subcube represents the literal A

Gate Logic: Two-Level Simplification

Three variable example: Full Adder Carry Out

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

In general,
an m-subcube within an n-cube (m < n) yields a term with
n - m literals

\( C_{out} = B \ C_{in} + A \ B + A \ C_{in} \)
Gate Logic: Two-Level Simplification

Karnaugh Map Method

- Hard to draw cubes of more than 4 dimensions
- K-map is an alternative method of representing the truth table that helps visualize adjacencies in up to 6 dimensions
- Beyond that, computer-based methods are needed

2-variable K-map

3-variable K-map

4-variable K-map

Numbering Scheme: 00, 01, 11, 10
Gray Code — only a single bit changes from code word to next code word

Gate Logic: Two-Level Simplification

Karnaugh Map Method Examples

F = A asserted, unchanged B varies

G = B complemented, unchanged A varies

Cout = A B + B Cin + A Cin

F(A,B,C) = A

© R.H. Katz Transparency No. 2-41

© R.H. Katz Transparency No. 2-42

© R.H. Katz Transparency No. 2-43

© R.H. Katz Transparency No. 2-44
Gate Logic: Two-Level Simplification

More K-Map Method Examples, 3 Variables

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

\[ F = B'C' + AC \]

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

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

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

In the K-map, adjacency wraps from left to right and from top to bottom

Compare with the method of using DeMorgan's Theorem and Boolean Algebra to reduce the complement!

K-map Method Examples: 4 variables

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

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

Find the smallest number of the largest possible subcubes that cover the ON-set

K-map Corner Adjacency Illustrated in the 4-Cube
**Gate Logic: Two-Level Simplification**

**K-map Method: Circling Zeros**

Replace $F$ by $\overline{F}$, 0's become 1's and vice versa

$$\overline{F} = B \overline{C} \overline{D} + A \overline{C} D + B C D$$

$$F = B \overline{C} D + A \overline{C} D + B C D$$

$$F = (B + C + D)(\overline{A} + C + D)(\overline{B} + C + \overline{D})$$

**K-map Example: Don't Cares**

Don't Cares can be treated as 1's or 0's if it is advantageous to do so

$F(A,B,C,D) = \Sigma m(1,3,5,7,9) + \Sigma d(6,12,13)$

$F = A'D + B'C'D$ w/o don't cares

$F = C'D + A'D$ w/ don't cares

By treating this DC as a "1", a 2-cube can be formed rather than one 0-cube

In PoS form: $F = D(A' + C')$

Same answer as above, but fewer literals

**Gate Logic: Two-Level Simplification**

**Design Example: Two Bit Comparator**

A 4-Variable K-map for each of the 3 output functions

Block Diagram and Truth Table
Contemporary Logic Design

Gate Logic: Two-Level Simplification

Design Example: Two Bit Comparator

F1 = \(A'BC'D' + A'BCD + ABCD + AB'C'D'\)
F2 = \(AB'D' + A'C\)
F3 = \(BC'D' + A'C' + AB'D'\)

A \text{\ xor} B \text{\ xor} C \text{\ xor} D

© R.H. Katz Transparency No. 2-54

Contemporary Logic Design

Gate Logic: Two-Level Simplification

Design Example: Two Bit Adder

X = \(A'B'C'D' + A'BCD + ABCD + AB'C'D'\)
Y = \(A'B'D' + A'C\)
Z = \(BC'D' + A'C' + AB'D'\)

© R.H. Katz Transparency No. 2-56
**Gate Logic: Two-Level Simplification**

**Design Example (Continued)**

\[ X = A \overline{C} + B \overline{C} D + A B D \]
\[ Z = B \overline{D}' + B' D = B \text{xor} D \]
\[ Y = A' B' C + A B' C' + A' B C' D + A' B C D' + A B C' D' + A B C D \]
\[ = B' (A \text{xor} C) + A' B (C \text{xor} D) + A B (C \text{xnor} D) \]
\[ = B' (A \text{xor} C) + B (A \text{xor} B \text{xor} C) \]

Note: XOR typically requires 4 NAND gates to implement!

**Gate Logic: Two-Level Simplification**

**Design Example: BCD Increment By 1**

\[ W = B C D + A D' \]
\[ Y = A' C' D + CD' \]
\[ X = B C' + B D' + B' C D \]
\[ Z = D' \]
Definition of Terms

**implicant**: single element of the ON-set or any group of elements that can be combined together in a K-map

**prime implicant**: implicant that cannot be combined with another implicant to eliminate a term

**essential prime implicant**: if an element of the ON-set is covered by a single prime implicant, it is an essential prime

**Objective**

grow implicants into prime implicants
cover the ON-set with as few prime implicants as possible
essential primes participate in ALL possible covers

### More Examples

#### 6 Prime Implicants:

- \( A' B' D \)
- \( B C' \)
- \( A C \)
- \( A' C' D \)
- \( A B \)
- \( B' C D \)

Minimum cover = \( B C' + A C + A' B' D \)

#### 5 Prime Implicants:

- \( B D \)
- \( A B C' \)
- \( A C D \)
- \( A' B C \)
- \( A' C' D \)

Essential primes form the minimum cover

### Algorithm: Minimum Sum of Products Expression from a K-Map

**Step 1**: Choose an element of ON-set not already covered by an implicant

**Step 2**: Find “maximal” groupings of 1’s and X’s adjacent to that element. Remember to consider top/bottom row, left/right column, and corner adjacencies. This forms prime implicants (always a power of 2 number of elements).

**Repeat Steps 1 and 2 to find all prime implicants**

**Step 3**: Revisit the 1’s elements in the K-map. If covered by single prime implicant, it is essential, and participates in final cover. The 1’s it covers do not need to be revisited

**Step 4**: If there remain 1’s not covered by essential prime implicants, then select the smallest number of prime implicants that cover the remaining 1’s
Example: $f(A, B, C, D) = \cdot m(4, 5, 6, 8, 9, 10, 13) + d(0, 7, 15)$

Initial K-map

Example Continued

Primes around $A'B'C'D'$
GATE LOGIC: TWO-LEVEL SIMPLIFICATION
EXAMPLE CONTINUED

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>01</td>
<td>01</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>01</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>10</td>
<td>01</td>
</tr>
</tbody>
</table>

PRIMES AROUND A B C' D
PRIMES AROUND A B' C' D'

5-Variable K-maps

\[ f(A, B, C, D, E) = \Sigma m(2, 5, 7, 8, 10, 13, 15, 17, 19, 21, 23, 24, 29, 31) \]

EXAMPLE CONTINUED

<table>
<thead>
<tr>
<th>AB</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

PRIMES AROUND A B C' D
PRIMES AROUND A B' C' D'

ESSENTIAL PRIMES WITH MIN COVER

\[ f(A, B, C, D, E) = C E + A B' E + B C' D' E' + A' C' D E' \]
**Gate Logic: Two Level Simplification**

6-Variable K-Maps

\[ f(A, B, C, D, E, F) = \Sigma m(2, 8, 10, 18, 24, 26, 34, 37, 42, 45, 50, 53, 58, 61) \]

= \(D' \cdot E' + A \cdot D' \cdot E' + A' \cdot C \cdot D' \cdot F'\)

**Contemporary Logic Design**

Two-Level Logic

**Gate Logic: CAD Tools for Simplification**

Quine-McCluskey Method

Tabular method to systematically find all prime implicants

\[ f(A, B, C, D) = \Sigma m(4, 5, 6, 8, 9, 10, 13) + \Sigma d(0, 7, 15) \]

Stage 1: Find all prime implicants

Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

<table>
<thead>
<tr>
<th>Implication Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column I</td>
</tr>
<tr>
<td>0000</td>
</tr>
<tr>
<td>0100</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>0101</td>
</tr>
<tr>
<td>0110</td>
</tr>
<tr>
<td>1010</td>
</tr>
<tr>
<td>0111</td>
</tr>
<tr>
<td>1101</td>
</tr>
<tr>
<td>1111</td>
</tr>
</tbody>
</table>

Step 2: Apply Uniting Theorem—

- Compare elements of group w/ N 1's against those w/ N+1 1's.
- Differ by one bit implies adjacent. Eliminate variable and place in next column.
- E.g., 0000 vs. 0100 yields 0-00
  - 0000 vs. 1000 yields -000

When used in a combination, mark with a check. If cannot be

- When used in a combination, mark with a star. These are the prime implicants.
- Repeat until no further combinations can be made.

© R.H. Katz Transparency No. 2-73

© R.H. Katz Transparency No. 2-74

© R.H. Katz Transparency No. 2-75

© R.H. Katz Transparency No. 2-76
**Quine-McCluskey Method**

Tabular method to systematically find all prime implicants

\[ f(A,B,C,D) = \Sigma m(4,5,6,8,9,10,13) + \Sigma d(0,7,15) \]

### Stage 1: Find all prime implicants

#### Step 1: Fill Column 1 with ON-set and DC-set minterm indices. Group by number of 1's.

<table>
<thead>
<tr>
<th>Column I</th>
<th>Column II</th>
<th>Column III</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0-00 *</td>
<td>01-- *</td>
</tr>
<tr>
<td>0100</td>
<td>010-</td>
<td>-1-1 *</td>
</tr>
<tr>
<td>1000</td>
<td>100-</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>100-</td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>011-</td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>011-</td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>-111</td>
<td></td>
</tr>
</tbody>
</table>

E.g., 0000 vs. 0100 yields 0-00
0000 vs. 1000 yields -00
0010 vs. 1010 yields -1-1

When used in combination, mark with a check. If cannot be combined, mark with a star. These are the prime implicants.

Repeat until no further combinations can be made.

### Stage 2: Find smallest set of prime implicants that cover the ON-set

Recall that essential prime implicants must be in all covers another tabular method – the prime implicant chart
Gate Logic: CAD Tools for Simplification

Prime Implicant Chart

<table>
<thead>
<tr>
<th>Column</th>
<th>04(0-00)</th>
<th>08(0-00)</th>
<th>09(100-1)</th>
<th>0A(10-0)</th>
<th>0B(1-01)</th>
<th>45,6,7(01-)</th>
<th>57,13,15(-1-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>05</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>06</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>09</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

rows = prime implicants

columns = ON-set elements

place an "X" if ON-set element is covered by the prime implicant

If column has a single X, then the implicant associated with the row is essential. It must appear in minimum cover.

Eliminate all columns covered by essential primes

Find minimum set of rows that cover the remaining columns

f = A'B'D' + A'C'D + A'B

ESPRESSO Method

Problem with Quine-McCluskey: the number of prime implicants grows rapidly with the number of inputs

upper bound: 3^n/n, where n is the number of inputs

finding a minimum cover is NP-complete, i.e., a computational expensive process not likely to yield to any efficient algorithm

Espresso: trades solution speed for minimality of answer

don't generate all prime implicants (Quine-McCluskey Stage 1)

judiciously select a subset of primes that still covers the ON-set

operates in a fashion not unlike a human finding primes in a K-map.
Espresso Method: Overview

1. Expands implicants to their maximum size
   Implicants covered by an expanded implicant are removed from further consideration
   Quality of result depends on order of implicant expansion
   Heuristic methods used to determine order
   Step is called EXPAND

2. Irredundant cover (i.e., no proper subset is also a cover) is extracted from the expanded primes
   Just like the Quine-McCluskey Prime Implicant Chart
   Step is called IRREDUNDANT COVER

3. Solution usually pretty good, but sometimes can be improved
   Might exist another cover with fewer terms or fewer literals
   Shrink prime implicants to smallest size that still covers ON-set
   Step is called REDUCE

4. Repeat sequence REDUCE/EXPAND/IRREDUNDANT COVER to find alternative prime implicants
   Keep doing this as long as new covers improve on last solution

5. A number of optimizations are tried, e.g., identify and remove essential primes early in the process

---

Espresso Input

\[ f(A,B,C,D) = \sum(4,5,6,8,9,10,13) + \prod(0,7,15) \]

Espresso Output

\[ f = A' C' D' + A B' D' + A' B \]

---

Second EXPAND generates a different set of prime implicants

IRREDUNDANT COVER found by final step of espresso

Only three prime implicants!
Two-Level Logic: Summary

Primitive logic building blocks
  INVERTER, AND, OR, NAND, NOR, XOR, XNOR

Canonical Forms
  Sum of Products, Products of Sums
  Incompletely specified functions/don't cares

Logic Minimization
  Goal: two-level logic realizations with fewest gates and fewest
  number of gate inputs
  Obtained via Laws and Theorems of Boolean Algebra
  or Boolean Cubes and the Uniting Theorem
  or K-map Methods up to 6 variables
  or Quine-McCluskey Algorithm
  or Espresso CAD Tool