Chapter 3 Gate-level Minimization

3-1 The Map Method
Two-variable map and Three-variable map

3-2 Four-Variable Map

3-3 Five-variable Map

3-4 Product of Sums Simplification

3-5 Don’t-care Conditions

3-6 NAND and NOR Implementation

3-7 Other Two-Level Implementations

3-8 Exclusive-OR Function

3-9 Hardware Description Language (HDL)

3-1 The Map Method

- Simplification of Boolean Expression
  - Minimum # of terms, minimum # of literals
  - To reduce complexity of digital logic gates
  - The simplest expression is not unique

- Methods:
  - Algebraic minimization ⇒ lack of specific rules
    - Section 2.4
  - Karnaugh map or K-map
    - Combination of 2, 4, … adjacent squares

Logic circuit ⇔ Boolean function ⇔ Truth table ⇔ K-map
Canonical form (sum of minterms, product of maxterms)
⇔ (Simplified) standard form (sum of products, product of sums)

m_1 + m_2 + m_3 = x'y + xy' + xy = x + y
Three-Variable Map

- 8 minterms for 3 binary variables
- Any two adjacent squares differ by only one variable (Gray code; wrap)

\[
m_0 + m_2 = x'y'z + x'y'z' = x'z(y' + y) = x'z
\]

![Three-variable Map](image)

Examples 3-3 and 3-4

- One square represents one minterm, giving a term of three literals
- Two adjacent squares represent a term of two literals
- Four adjacent squares represent a term of one literal

![Examples 3-3 and 3-4](image)

Examples 3-1 and 3-2

- Two adjacent squares represent a term of three literals
- Four adjacent squares represent a term of two literals
- Eight adjacent squares represent a term of one literal

The larger the number of squares combined, the smaller the number of literals in the term

![Examples 3-1 and 3-2](image)

3-2 Four-Variable Map

- Two adjacent squares represent a term of three literals
- Four adjacent squares represent a term of two literals
- Eight adjacent squares represent a term of one literal

The larger the number of squares combined, the smaller the number of literals in the term

![3-2 Four-Variable Map](image)
**Examples 3-5 and 3-6**

- **prime implicant**: a product term obtained by combining the *maximum possible number* of adjacent squares in the map
  - A single 1 on a map represents a prime implicant if it is not adjacent to any other 1’s
  - Two adjacent 1’s form a prime implicant, provided that they are *not within* a group of four adjacent squares
  - Four adjacent 1’s form a prime implicant, provided that they are *not within* a group of eight adjacent squares
  - and so on
- If a minterm in a square is covered by *only one* prime implicant, that prime implicant is said to be *essential*

**Simplification Using Prime Implicants**

- First determine all the *essential* prime implicants
- The simplified expression is obtained from the logical sum of all the *essential* prime implicants plus other prime implicants that may be needed to cover any remaining minterms not covered by the *essential* prime implicants

**3-3 Five-Variable Map**
Any $2^k$ adjacent squares, for $k=(0, 1, 2, \ldots, n)$ in an $n$-variable map, will represent an area that gives a term of $n-k$ literals.

**3-4 Product of Sums Simplification**

- Boolean functions can be expressed in sum of products or product of sums
  - Recall from Table 2-4
    
    $f_1 = m_1 + m_4 + m_7 = M_2 M_3 M_5 M_6$
    
    $= x'y'z + xy'z' + xyz$
    
    $= (x+y+z)(x+y'+z')(x'+y+z')(x'+y'+z)$
    
    $= \Sigma(1, 4, 7) = \Pi(0, 2, 3, 5, 6)$

  - Complement of $f'$ (by DeMorgan’s Theorem)
    
    $f_1' = m_0 + m_2 + m_3 + m_5 + m_6 = M_1 M_4 M_7$
    
    $= x'y'z' + x'yz' + x'yz + xy'z + xyz'$
    
    $= (x+y+z')(x'+y+z')(x'+y'+z')$
    
    $= \Sigma(0, 2, 3, 5, 6) = \Pi(1, 4, 7)$

- Simplifying Boolean function in product of sums
  1. Derive $f'$ in sum of products from the map
  2. Complement of $f'$

**Example 3-7**

Simply Boolean function $F(A, B, C, D) = S(0, 1, 2, 5, 8, 9, 10)$ in (a) sum of products and (b) product of sums

(a) $F = B'D'$

(b) 1. Obtain simplified complemented function:

   $F' = AB + CD + BD'$

   2. Applying DeMorgan’s theorem to obtain $F$

   $F = (F')' = (A' + B')(C' + D')(B' + D)$
3-5 Don’t-Care Condition

- In practice, there are some applications where the function is not specified for certain combinations of the variables.
- Functions that have unspecified outputs for some input combinations are called incompletely specified functions.
- It’s customary to call the unspecified minterms of a function don’t-care conditions.
  - marked as X, indicating that we don’t care where 0 or 1 is assigned to $F$ for the particular minterm.
  - can be used on a map to provide further simplification.
    - may be assumed to be either 0 or 1.

\[ F(x, y, z) = \Sigma (1, 3, 4, 6) = \Pi (0, 2, 5, 7) \]

<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>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</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>

\[ F = x'z + xz' \]
\[ F' = xz + x'z' \]
\[ F = (x' + z')(x + z) \]

**Example 3-9**

Simplify the Boolean function:
\[ F(w, x, y, z) = \Sigma (1, 3, 7, 11, 15) \]

which has the don’t-care conditions:
\[ d(w, x, y, z) = \Sigma (0, 2, 5) \]

\[ F(w, x, y, z) = yz + w'z' \]
\[ F(w, x, y, z) = yz + w'z \]

Product of sums?
Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates – easier to fabricate, basic gates used in all IC

**NAND gate: a universal gate**
- Any digital system can be implemented with it
  - including AND, OR and complement

```
NAND
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
NOR
<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

\[ F = (xy)' \]
\[ F = (x + y)' \]

Implement the following Boolean function with NAND gates

**Example 3-10**
Implement the following Boolean function with NAND gates

\[ F = xy' + x'y + z \]

```
\[ F = (xy)' \]
```

```
\[ (xy)' \]
```

```
\[ x' + y' + z' = (xy)' \]
```

\[ (AB + CD)' \]
\[ = [(AB + CD)]' \]
\[ = [(AB)'*(CD)']' \]

Fig. 3-20 Three Ways to Implement \( F = AB + CD \)

Fig. 3-19 Two Graphic Symbols for NAND Gate

Fig. 3-21 Solution to Example 3-10
Procedures of Implementation with two levels of NAND gates

1. Express simplified function in sum of products
2. Draw a NAND gate for each product term that has at least two literals to constitute a group of first-level gates
3. Draw a single gate using AND-invert or invert-OR in the second level
4. A term with a single literal requires an inverter in the first level

Multilevel NAND Circuits

1. Convert all AND gates to NAND gates with AND-invert graphic symbols
2. Convert all OR gates to NAND gates with invert-OR graphic symbols
3. Check all the bubbles in the diagrams. For a single bubble, insert an inverter (one-input NAND gate) or complement the input literal

NOR Circuits

- The NOR operation is the dual of the NAND operation
- The NOR gate is another universal gate to implement any Boolean function
- Easy for OR-AND (product of sums)
Transformation from OR-AND diagram to NOR diagram
• OR gates => OR-invert
• AND gate => invert-AND

wired logic: some NAND or NOR gates allow a direct wire connection between the outputs of two gates to provide a specific logic function
– The wired-AND gate or wired-OR gate is not a physical second-level gate, but only a symbol

F = (AB)′ · (CD)′ = (AB + CD)′
F = (A + B)′ + (C + D)′ = [(A + B)(C + D)]′

3-7 Other Two-Level Implementations

<table>
<thead>
<tr>
<th>1st</th>
<th>2nd</th>
<th>AND</th>
<th>OR</th>
<th>NAND</th>
<th>NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>AND</td>
<td>3-4</td>
<td>NAND</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td>3-4</td>
<td>OR</td>
<td>x</td>
<td>NOR</td>
<td></td>
</tr>
<tr>
<td>NAND</td>
<td>=</td>
<td>NAND</td>
<td>3-6</td>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>NOR</td>
<td>NOR</td>
<td>x</td>
<td>OR</td>
<td>3-6</td>
<td></td>
</tr>
</tbody>
</table>

AND-OR-INVERT Implementation

AND-NOR = NAND-AND = AND-OR-INVERT
F = (AB + CD + E)′
Similar to AND-OR, AND-OR-INVERT requires an expression in sum of products
Given F, we can implement F′ with AND-OR-INVERT
OR-AND-INVERT Implementation

OR-NAND = NOR-OR = OR-AND-INVERT

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

Similar to OR-AND, OR-AND-INVERT requires an expression in products of sum.

Given \( F \), we can implement \( F' \) with OR-AND-INVERT.

Example 3-11
Other Two-level Implementations

\[ F = x'y'z' + xyz' \]
\[ F' = x'y + xy' + z \]

\[ F = (F')' = (x'y + xy' + z)' = (x + y') (x' + y) z \]

\[ F' = (F')' = (x'y'z' + xyz')' = (x + y + z)(x' + y' + z) \]

3-8 Exclusive-OR (XOR) Function

XOR: \( x \oplus y = xy' + x'y \)

\[ (x \oplus y)' = (xy' + x'y)' = (x' + y)(x + y') = x'y' + xy \]

Communtative: \( A \oplus B = B \oplus A \)

Associative: \( (A \oplus B) \oplus C = A \oplus (B \oplus C) = A \oplus B \oplus C \)

Odd Function

- The 3-variable XOR function is equal to 1 if only one variable is equal to 1 or if all three variables are equal to 1.
- Multiple-variable exclusive OR operation = odd function: odd number of variables be equal to 1.
Figure 3-35 Map for a 4-variable XOR Function

\[ A \oplus B \oplus C \oplus D = \left( AB' + A'B \right) \oplus \left( CD' + C'D \right) = \left( AB' + A'B \right) \left( CD + C'D' \right) + \left( AB + A'B' \right) \left( CD' + C'D \right) = \sum \left( 1, 2, 4, 7, 8, 11, 13, 14 \right) \]

- (a) Odd function: \( F = A \oplus B \oplus C \oplus D \)
- (b) Even function: \( F = (A \oplus B \oplus C \oplus D)' \)

Parity Generation and Checking

parity bit: an extra bit included with a binary message to make the number of 1's either odd or even

3-bit even-parity-generator

\[ P = x \oplus y \oplus z \]

Table 3-4

<table>
<thead>
<tr>
<th>Three-Bit Message</th>
<th>Parity Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<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>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>0</td>
</tr>
</tbody>
</table>

4-bit Even-Parity-Checker

\[ C = x \oplus y \oplus z \oplus P \]

Table 3-5

<table>
<thead>
<tr>
<th>Four Bits Received</th>
<th>Parity Error Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Digital Gates of TTL or CMOS series 7400

(Various diagrams of digital gates are shown.)
Put it All Together

Logic circuit
  • AND-OR, OR-AND
  • NAND-NAND, NOR-NOR
⇔ Truth table
⇔ K-map
⇔ Boolean function
⇔ Canonical form
  • sum of minterms, Σ
  • product of maxterms, Π
⇔ (Simplified) standard form
  • sum of products
  • product of sums

Problem 3-6
(a) A'B'C'D'+AC'D'+B'CD'+A'BCD+BC'D
(b) x'z+w'xy'+w(x'y + xy')

Σ(0,2,5,7,8,10,12,13) = Π(1,3,4,6,9,11,14,15)
Σ(1,3,4,5,9,10,11,12,13) = Π(0,2,6,7,8,14,15)

Summary

Chapter 3  Gate-level Minimization
3-1 The Map Method
  Two-variable map and Three-variable map
3-2 Four-Variable Map
3-3 Five-variable Map
3-4 Product of Sums Simplification
3-5 Don’t-care Conditions
3-6 NAND and NOR Implementation
3-7 Other Two-Level Implementations
3-8 Exclusive-OR Function
3-9 Hardware Description Language (HDL)