Chapter 3

Gate-Level Minimization

Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language
Why Logic Minimization?

- Minimize the number of gates used
  - Reduce gate count = reduce cost
- Minimize total delay (critical path delay)
  - Reduce delay = improve performance
- Satisfy design constraints
  - Maximum fanins and fanouts, ...
- Remove undesired circuit behavior
  - Hazard, race, ...

The Map Method

- The map method is also known as the Karnaugh map or K-map
- Provide a straightforward procedure for minimizing Boolean functions
- The simplified expressions are always in one of the two standard forms:
  - Sum of Products (SOP)
  - Product of Sums (POS)
Two-Variable Map (1/2)

- Two-variable function has **four** minterms
  - Four squares in the map for those minterms
- The corresponding minterm of each square is determined by the bit status shown outside

```

```

Two-Variable Map (2/2)

```

x + y = x(y + y') + y(x + x')
= xy (m_3) + xy' (m_2) + x'y (m_1)
= m_1 + m_2 + m_3
```
Three-Variable Map (1/2)

Gray Code

\[
\begin{array}{c|cccc}
 & 00 & 01 & 11 & 10 \\
\hline
x & x' y' z' & x' y' z & x' y z & x' y z' \\
\hline
y' z & y' z & x y z & x y z' \\
\end{array}
\]

(a)

Three-Variable Map (2/2)

\[F(x, y, z) = \sum (3, 4, 6, 7) = yz + xz'\]
\[F(x, y, z) = \sum (0, 2, 4, 5, 6) = z' + xy'\]

\[
\begin{array}{c|cccc}
 & 00 & 01 & 11 & 10 \\
\hline
x & 0 & 1 & 1 & 1 \\
\hline
y' z & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|cccc}
 & 00 & 01 & 11 & 10 \\
\hline
x & 0 & 1 & 1 & 1 \\
\hline
y' z & 1 & 1 & 1 & 1 \\
\end{array}
\]

(x, y covered) (all z covered)

(all y covered) (all x covered)

3-7

3-8
Four-Variable Map (1/2)

Gray Code

\[
\begin{array}{cccc}
wx & 00 & 01 & 11 & 10 \\
\hline
00 & w'y'z' & w'y'y'z & w'y'yz & w'y'y'z' \\
01 & w'xy'z' & w'xy'y'z & w'xy'yz & w'xy'y'z' \\
11 & wxy'z' & wxy'y'z & wxy'yz & wxy'y'z' \\
10 & wx'y'z' & wx'y'y'z & wx'y'yz & wx'y'y'z' \\
\end{array}
\]

(a) & (b)

\[F = A'B'C' + B'CD' + A'BCD' + AB'C' = \sum_{0,1,2,6,8,9,10}(A'B'C')\]

= B'D' + B'C' + A'CD'

Four-Variable Map (2/2)

\[
\begin{array}{cccc}
CD & 00 & 01 & 11 & 10 \\
\hline
A'B'C' & 1 & 1 & & \\
A'BCD' & 1 & & 1 & \\
AB'C' & 1 & & & 1 \\
AB'CD' & & 1 & & 1 \\
\end{array}
\]

A'CD' \quad A'C'D' \quad B'D' \quad B'C
Maps with six or more variables need too many squares and are impractical to use.

\[ F(A,B,C,D,E) = (0,2,4,6,9,13,21,23,25,29,31) \]
\[ = A'B'E' + BD'E + ACE \]
Prime Implicants

- Implicant (cube): A group of minterms that form a cube
- Prime implicant: Combine maximum possible number of adjacent squares in the map

Essential Prime Implicants

- If a minterm is covered by only one prime implicant, that prime implicant is essential and must be included
Systematic Simplification

- Identify all prime implicants on the k-map
- Select all essential prime implicants
- Select a minimum subset of the remaining prime implicants that cover all 1’s
- Ex: \( F(A, B, C, D) = \sum (0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15) \)

Product of Sums Simplification

- The complement of a function is represented in the map by the squares not marked by 1’s
- Choose 1 \( \Rightarrow \) sum of products (minterms)
- Choose 0 \( \Rightarrow \) product of sums (Maxterms)

\[ F(A, B, C, D) = \sum (0, 1, 2, 5, 8, 9, 10) \]

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

\[ = (A' + B')(C' + D')(B' + D) \]
Two Gate Implementations

- Sometimes product-of-sums representations may have smaller implementations

**7 literals, 4 gates**

**6 literals, 4 gates**

(a) \( F = B'D' + B'C + A'CD \)

(b) \( F = (A' + B')(C' + D')(B' + D) \)

Fig. 3-15 Gate Implementation of the Function of Example 3-8

---

Don’t Care Conditions

- \( X = \) don’t care (can be 0 or 1)
- Don’t cares can be included to form a larger cube, but not necessary to be completely covered
- Ex: \( F(w, x, y, z) = \sum(1,3,7,11,15) \) \( d(w, x, y, z) = \sum(0,2,5) \)

larger cube with don’t cares
**Outline**

- Karnaugh Map Method
- **NAND and NOR Implementations**
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

---

**NAND and NOR Implementation**

- Digital circuits are frequently constructed with **NAND** or **NOR** gates rather than with AND and OR gate
  - NAND and NOR gates are much easier to fabricate
  - NAND or NOR gates are both universal gates
    - Any digital system can be implemented with only NAND gates or NOR gates

---

![Logic Gates Diagram](image-url)
Alternative Graphic Symbols

- To facilitate the conversion to NAND or NOR logic, it is convenient to define alternative graphic symbols.
  - “Bubble” means complement.

![](image1)

Two-Level Implementation (NAND)

- It’s easy to implement a Boolean function with only NAND gates if converted from a **sum of products** form.
- Ex: $F = AB + CD = ((AB)'(CD))'$

![](image2)
Example 3-10

Procedures:
1. Simplify the function in sum of products
2. Draw NAND gates for the first level
3. Draw a single AND-invert or invert-OR in the second level
4. Add an inverter at the first level for the term with a single literal

\[ F(x, y, z) = \sum (1, 2, 3, 4, 5, 7) \]

Two-Level Implementation (NOR)

- It's easy to implement a Boolean function with only NOR gates if converted from a product of sums form.
- Ex: \( F = (A+B)(C+D)E \)

1. add two bubbles at the ends
2. complement this input to add the third bubble (if required)
3. convert to NOR gate using DeMorgan's theorem
Multilevel NAND Circuits

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

- Procedures:
  1. Convert all AND gates to NAND gates with AND-invert symbols
  2. Convert all OR gates to NAND gates with invert-OR symbols
  3. Check all bubbles and insert an inverter for the bubble that are not compensated by another bubble

Multilevel NOR Circuits

- For NOR gates, AND → invert-AND, OR → OR-invert
- Other procedures are the same as those for NAND

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

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

Wired Logic

- Wired logic: direct wire connection that results in a specific logic function
  - Wired-AND
  - Wired-OR
- Some NAND and NOR implementations (not all) have such a property

(a) Wired-AND in open-collector TTL NAND gates

(b) Wired-OR in ECL gates

F = (AB + CD)'

F = [(A + B) (C + D)]'
Nondegenerate Forms

- There are 16 possible combinations of two-level forms
  - Eight of these combinations will degenerate to a single operation
  - The other eight nondegenerate forms produce an implementation in SOP or POS

<table>
<thead>
<tr>
<th>AND-OR</th>
<th>NAND-NAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOR-OR</td>
<td>OR-NAND</td>
</tr>
</tbody>
</table>

The two forms at the same line are dual to each other

<table>
<thead>
<tr>
<th>OR-AND</th>
<th>NOR-NOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND-AND</td>
<td>AND-NOR</td>
</tr>
</tbody>
</table>

AND-OR-INVERT Implementation

- NAND-AND and AND-NOR are equivalent and both perform the AND-OR-INVERT (AOI) function
- Require sum-of-products form in nature
  - When starting from product-of-sums form, complement it using DeMorgan's theorem to obtain sum-of-products form
- Ex: \( F = (AB + CD + E)' \)
**OR-AND-INVERT Implementation**

- OR-NAND and NOR-OR are equivalent and both perform the OR-AND-INVERT (OAI) function.
- Require product-of-sums form in nature
  - When starting from sum-of-products form, complement it using DeMorgan's theorem to obtain product-of-sums form.
- Ex: \( F = [(A+B)(C+D)E]' \)

![Implementation diagrams]

**Implement with Two-Level Forms**

**Table 3-3:** Implementation with Other Two-Level Forms

<table>
<thead>
<tr>
<th>Equivalent Form</th>
<th>Implements the Function</th>
<th>Simplify ( F ) in</th>
<th>To Get an Output of</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(b)*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND-NOR</td>
<td>NAND-AND</td>
<td>AND-OR-INVERT</td>
<td></td>
</tr>
<tr>
<td>OR-NAND</td>
<td>NOR-OR</td>
<td>OR-AND-INVERT</td>
<td></td>
</tr>
</tbody>
</table>

*Form (b) requires an inverter for a single literal term.*
Example 3-11

- \( F = x'y'z' + xyz' \)
  (choose 1 in K-map)
  \( = (x'y + xy' + z)' \)
  (choose 0 in K-map)

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

(a) Map simplification in sum of products.

Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language
Exclusive-OR (XOR) Function

- XOR is often denoted by the symbol ⊕
- Logic operation of XOR
  - $X \oplus Y = XY' + X'Y$
  - Equal to 1 if only $x$ is equal to 1 or if only $y$ is equal to 1, but not when both are equal to 1
- It's complement, exclusive-NOR (XNOR), is often denoted by the symbol ⊗
- Logic operation
  - $X \odot Y = XY + X'Y'$
  - It is equal to 1 if both $x$ and $y$ are equal to 1 or if both are equal to 0
- Seldom used in general Boolean functions
  - Particularly useful in arithmetic operations and error detection and correction circuits

Exclusive-OR Implementations

(a) With AND-OR-NOT gates

(b) With NAND gates
Odd Function

- The multiple-variable XOR operation is defined as an **odd function**
  - TRUE when no. of “1” in inputs is odd

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</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>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>

(a) 3-input odd function

Even Function

- The multiple-variable XNOR operation is defined as an **even function**
  - TRUE when no. of “1” in inputs is even

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</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>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Even function

\[ F = (A \oplus B \oplus C)' \]
Four-Variable XOR Function

<table>
<thead>
<tr>
<th>CD</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Odd function
\[ F = A \oplus B \oplus C \oplus D \]

<table>
<thead>
<tr>
<th>CD</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(b) Even function
\[ F = (A \oplus B \oplus C \oplus D)' \]

Fig. 3-35 Map for a Four-variable Exclusive-OR Function

Parity Generation and Checking

- An extra **parity** bit is often added and checked at the receiving end for error
- The circuit that generates the parity bit in the transmitter is called a **parity generator**
- The circuit that checks the parity in the receiver is called a **parity checker**
- Exclusive-OR functions are very useful to construct such circuits
Parity Generator

- For even parity:
  - The total number of “1” (including P) is even
  - The number of “1” at inputs is odd
  - Generated with an XOR gate (odd function)
  - \( P = x \oplus y \oplus z \) (for 3-bit message)
- Similarly, odd parity can be generated with an XNOR gate

<table>
<thead>
<tr>
<th>Three-Bit Message</th>
<th>Parity Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ( \oplus ) y ( \oplus ) z</td>
<td>P</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) 3-bit even parity generator

Parity Checker

- For even parity, the total number of “1” in the message is even
  - An error occurs when the received number of “1” is odd
  - An XOR gate (odd function) can detect such an error
- Has \( n+1 \) inputs

<table>
<thead>
<tr>
<th>Four Bits Received</th>
<th>Parity Error Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ( \oplus ) y ( \oplus ) z</td>
<td>P</td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) 4-bit even parity checker
Outline

- Karnaugh Map Method
- NAND and NOR Implementations
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language

Hardware Description Language

- Have high-level language constructs to describe the functionality and connectivity of the circuit
- Can describe a design at some levels of abstraction
  - Behavioral, RTL, Gate-level, Switch
- Can describe functionality as well as timing
- Can model the concurrent actions in real hardware
- Can be used to document the complete system design tasks
  - testing, simulation ... related activities
- Comprehensive and easy to learn
- Two popular languages: Verilog & VHDL
  - Will be taught in another course
Why Use an HDL?

- Hard to design directly for complex systems
- Formal description using HDL
  - Verify the specification through simulation or verification
  - Easy to change
  - Enable automatic synthesis
- Allow architectural tradeoffs with short turnaround
- Reduce time for design capture
- Encourage focus on functionality
- Shorten the design verification loop

*HDL = Hardware Description Language