Chapter 3

Gate-Level Minimization

Outlines

- Introduction
- The Map Method
- Four-Variable Map
- Five-Variable Map
- Product of Sums Simplification
- Don’t-Care Conditions
- NAND and NOR Implementation
- Other Two-Level Implementations
- Exclusive-OR Function
- Hardware Description Language (HDL)
3.1 Introduction

Why do we need logic minimization?
- Minimize the number of gates used (gate count)
- Minimize the total delay (critical path delay) in order to improve speed
- Satisfy design constraints such as max fanins and fanouts
- Remove undesired circuit behavior such as hazard and race

THE MAP METHOD

The map method is also known as the Karnaugh map or K-map

The map method provides a simple straightforward procedure for minimizing Boolean function.

The simplified expressions produced by the map are always in one of the two standard forms:
- Sum of Products (SOP)
- Product of Sums (POS)
Two Variable Map

\( \begin{array}{cc}
m_0 & m_1 \\
m_2 & m_3 \\
\end{array} \)  
(a)

\[ \begin{array}{cc}
y & y' \\
x & x' \\
0 & 0' \\
1 & 1' \\
\end{array} \]  
(b)

Two Variable Map

\[ \begin{array}{cc}
y & y \\
x & x \\
0 & 0 \\
1 & 1 \\
\end{array} \]  
(a) \( xy \)

\[ \begin{array}{cc}
y & y \\
x & x \\
0 & 0 \\
1 & 1 \\
\end{array} \]  
(b) \( x + y \)
K-map three variable

(a) 

<table>
<thead>
<tr>
<th>( m_0 )</th>
<th>( m_1 )</th>
<th>( m_3 )</th>
<th>( m_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m_4 )</td>
<td>( m_5 )</td>
<td>( m_7 )</td>
<td>( m_6 )</td>
</tr>
</tbody>
</table>

(b) 

<table>
<thead>
<tr>
<th>( x ) ( y )</th>
<th>( 00 )</th>
<th>( 01 )</th>
<th>( 11 )</th>
<th>( 10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x'y'z' )</td>
<td>( x'y'z )</td>
<td>( xy'z )</td>
<td>( xyz' )</td>
<td></td>
</tr>
<tr>
<td>( xy'z' )</td>
<td>( xy'z )</td>
<td>( xyz )</td>
<td>( xyz' )</td>
<td></td>
</tr>
</tbody>
</table>

\[ y' = \left( \sum_{x=0} (0,2,4,5,6) \right) = z' + xy' \]

\[ F(x, y, z) = \sum (3,4,6,7) = yz + xz' \]
K-map four variable

(a)  

(b)  

F(A, B, C, D) = \sum (0, 1, 2, 6, 8, 9, 10) = B' C' + A' CD' + B' D'
K-map five variable

\[
\begin{array}{c|cc|c|c}
\text{BC} & \text{DE} & & \\
00 & 0 & 1 & 3 & 2 \\
01 & 4 & 5 & 7 & 6 \\
11 & 12 & 13 & 15 & 14 \\
10 & 8 & 9 & 11 & 10 \\
\end{array}
\]

A = 0

\[
\begin{array}{c|cc|c|c}
\text{BC} & \text{DE} & & \\
00 & 0 & 1 & 3 & 2 \\
01 & 4 & 5 & 7 & 6 \\
11 & 12 & 13 & 15 & 14 \\
10 & 8 & 9 & 11 & 10 \\
\end{array}
\]

A = 1

\[
\begin{array}{c|cc|c|c}
\text{BC} & \text{DE} & & \\
00 & 0 & 1 & 3 & 2 \\
01 & 4 & 5 & 7 & 6 \\
11 & 12 & 13 & 15 & 14 \\
10 & 8 & 9 & 11 & 10 \\
\end{array}
\]

K-map five variable

\[
\begin{array}{c|cc|c|c}
\text{BC} & \text{DE} & & \\
00 & 0 & 1 & 3 & 2 \\
01 & 4 & 5 & 7 & 6 \\
11 & 12 & 13 & 15 & 14 \\
10 & 8 & 9 & 11 & 10 \\
\end{array}
\]

A = 0

\[
\begin{array}{c|cc|c|c}
\text{BC} & \text{DE} & & \\
00 & 0 & 1 & 3 & 2 \\
01 & 4 & 5 & 7 & 6 \\
11 & 12 & 13 & 15 & 14 \\
10 & 8 & 9 & 11 & 10 \\
\end{array}
\]

A = 1

\[
\begin{array}{c|cc|c|c}
\text{BC} & \text{DE} & & \\
00 & 0 & 1 & 3 & 2 \\
01 & 4 & 5 & 7 & 6 \\
11 & 12 & 13 & 15 & 14 \\
10 & 8 & 9 & 11 & 10 \\
\end{array}
\]

\[
F = A'B'E' + BD'E + ACE
\]
Terminologies

- Implicates (cubes)
  - A group of minterms that form cube
- Prime implicates (cubes)
  - Implicants that are not contained by other implicates
- Essential Implicates (cubes)
  - Implicates contain minterms that are not included by other implicates
- Cover
  - All the 1’s are included by the selected implicates

Prime Implicates

- Circle all prime implicates on the k-map
- Identify and select all essential prime implicates
- Select a minimum subset of the remaining prime
- Implicates to complete the cover
Prime Implicates

\[ F(A, B, C, D) = \sum (0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15) \]

Five – Variable Map

- Variable A distinguishes between the two maps
**Five – Variable Map**

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

\[
\begin{array}{c|cc}
A = 0 & DE & D \\
\hline
B' & 00 & 01 & 11 & 10 \\
00 & 1 & 1 & \\
01 & 1 & 1 & 1 & \\
11 & 1 & 1 & 1 & 1 \\
10 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{c|cc}
A = 1 & DE & D \\
\hline
B' & 00 & 01 & 11 & 10 \\
00 & 1 & 1 & \\
01 & 1 & 1 & 1 & \\
11 & 1 & 1 & 1 & 1 \\
10 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
F = A'B'E' + BD'E + ACE
\]

---

**Product of Sums Simplification**

- The complement of a function is represented in the map by the squares not marked by 1’s.
- We obtain a simplified expression of the complement of the function.

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

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

\[
= (A' + B')(C' + D')(B' + D)
\]
Don’t Care Conditions

- An X inside a square in the map indicates that we don’t care whether the value of 0 or 1 is assigned to F for the particular minterm.

Ex:
\[ F(w, x, y, z) = \sum (1,3,7,11,15) \]
\[ d(w, x, y, z) = \sum (0,2,5) \]

NAND and NOR Implementation

- Digital circuits are frequently constructed with NAND or NOR gates rather than with AND and OR gates.

- NAND gate ➔ NOT operation (Inverter)
  
  AND operation (AND gate)
  
  OR operation (OR gate)
  
  NOR operation (NOR gate)

- NOR gate ➔ NOT operation (Inverter)
  
  AND operation (AND gate)
  
  OR operation (OR gate)
  
  NAND operation (NAND gate)
NAND Circuits

Logic operations with NAND gate

- Inverter: $x \rightarrow x'$
- AND: $x \land y \rightarrow xy$
- OR: $(x'y')' = x + y$

(a) AND-invert
(b) Invert-OR
Two level implementation

- It implements can be easily converted to a sum of products form by using DeMorgan's theorem
  \[ F = ((AB)'(CD)')' = AB + CD \]

Multilevel NAND Circuits

- The general procedure for converting a multilevel AND-OR diagram into an all-NAND diagram using mixed notation is as follows:
  - Convert all AND gates to NAND gates with AND-invert graphic symbols
  - Convert all OR gates to NAND gates with invert-OR graphic symbols
  - Check all the bubbles in the diagram
**Multilevel NAND Circuits**

EXAMPLE: \[ F = (AB' + A'B)(C + D') \]

---

**NOR Implementation**

Logic Operations with NOR Gates

- NOR: \[ x' + y' = xy \]
NOR Implementation

Example: $F = (A + B)(C + D)E$

(a) OR−invert

(b) Invert−AND
Other Two-Level Implementations

- Some NAND or NOR gates (but not all) allow the possibility of a wire connection between the outputs of two gates to provide a specific logic function. This type of logic is called *wired logic*.

![Diagram of wired logic](image)

Nondegenerate forms

- The eight nondegenerate forms are as follows:
  - AND-OR
  - NAND-NAND
  - NOR-OR
  - OR-NAND
  - OR-AND
  - NOR-NOR
  - NAND-AND
  - AND-NOR
AND-OR-INVERT Implementation

(a) AND-NOR

(b) AND-NOR

(c) NAND-AND

Fig. 3.29 AND-OR-INVERT Circuits: $F = (AB + CD + E)'$

OR-AND-INVERT Implementation

(a) OR-NAND

(b) OR-NAND

(c) NOR-OR

Fig. 3.30 OR-AND-INVERT Circuits: $F = [(A + B)(C + D)]'$
OR-AND-INVERT Implementation

Table 3-3
Implementation with Other Two-Level Forms

<table>
<thead>
<tr>
<th>Equivalent Nondegenerate 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>Sum of products</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>by combining 0's</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in the map</td>
</tr>
<tr>
<td>OR-NAND</td>
<td>NOR-OR</td>
<td>OR-AND-INVERT</td>
<td>Product of sums</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>by combining 1's</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>in the map and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>then complementing</td>
</tr>
</tbody>
</table>

*Form (b) requires an inverter for a single literal term.

Exclusive-OR Function

- The exclusive-OR (XOR), denoted by the symbol ⊕
- Logic operation
  - \( X \oplus Y = XY' + X'Y \)
- It is 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
- Truth table & K-map:
Exclusive-OR Function

Truth table & K-map:

<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) Odd function

\[ F = A \oplus B \oplus C \]

Exclusive-OR Implementations

(a) With AND-OR-NOT gates

(b) With NAND gates
Exclusive-NOR Function

- The exclusive-NOR (XNOR), denoted by the symbol $\odot$
- 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

Exclusive-NOR Function

- Truth table & K-map:

<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
$F = (A \oplus B \oplus C)'$
Odd Function

- The multiple-variable exclusive-OR operation is defined as an *odd function*.

<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>

Parity Generation and Checking

- Exclusive-OR functions are very useful in systems requiring error-detection and correction codes.
- 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*. 
Parity Generation and Checking

Example:

- 3-bit message to be transmitted together with an even parity bit

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

(a) 3-bit even parity generator

(a) 4-bit even parity checker

Parity Generation and Checking

Example:

- 3-bit message to be transmitted together with an even parity bit

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

Table 3-4: Even-Parity-Generator Truth Table

<table>
<thead>
<tr>
<th>Three-Bit Message</th>
<th>Parity Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</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>

Table 3-5: Even-Parity-Checker Truth Table

<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>

42
A Hardware Description Language (HDL) is a high-level programming language with special constructs used to model the function of hardware logic circuits. The special language constructs provide you the ability to:

- Describe The Connectivity Of The Circuit
- Describe The Functionality Of A Circuit
- Describe A Circuit At Various Levels Of Abstraction
- Describe The Timing Of A Circuit
- Express Concurrency

Hardware Description Languages (HDL) share several features in common:

- Typically, an HDL contains some high-level programming language constructs, along with constructs to describe the connectivity of the hardware design.
- An HDL allows you to describe the design at various levels of abstraction using structural or behavioral constructs.
- An HDL allows you to describe the functionality of the hardware, along with its timing constraints.
- Concurrency, which is the ability to perform multiple tasks at the same time. Typically, programming languages are not concurrent, but in hardware a number of operations happen at the same time. Thus, an HDL must be concurrent.