1 The Map Method

Gate-level minimization refers to the design task of finding an optimal gate-level implementation of the Boolean function describing a digital circuit. In this section, we will discuss the manual design of simple circuits.

The complexity of a digital logic-gate circuit that implements a Boolean function directly depends on the complexity of the corresponding algebraic expression. Although the truth-table representation of a function is unique, its algebraic form can take many different, but equivalent, forms. Minimization of Boolean function using the algebraic method is awkward. The map method provides a well-structured method of minimizing Boolean functions. The map method is also known as the Karnaugh map or k-map method.

The simplified expression produced by the map are always in two standard forms: sum of products or product of sums. We will assume that the simplest algebraic expression is an algebraic expression with a minimum number of terms and with the smallest possible number of literals in each term. This expression produces a circuit diagram with a minimum number of gates and the minimum number of inputs to each gate. However, this simplest expression is not unique: It is possible to sometimes find two or more expressions that satisfy the minimization criteria. In that case, each solution is satisfactory.
1.1 Two-Variable Map

In the two-variable map shown in Figure 1(a), there are four minterms for two variables. Hence the map consists of four squares one for each minterm. The map is redrawn in 1(b) to show the relationship between the squares and the two variables $x$, and $y$.

If we mark the squares whose minterms belong to a given function, the two-variable map becomes another useful way to represent any one of the 16 Boolean functions of two variables.

*Example 1.* Represent $f = xy$ in a map.

*Example 2.* Represent $f = x + y$ in a map.
1.2 Three-Variable Map

There are eight minterms for three binary variables; therefore the map consists of eight squares. Note that minterms are arranged not in a binary sequence but in a sequence similar to the Gray code, where only one bit changes from one adjacent column to the next. Figure 2 shows the four-variable map.

Simplifying

Any two adjacent squares in the map differ by only one variable. Form the postulates of Boolean algebra it follows that the sum of two minterms in adjacent squares can be simplified to a single AND term consisting of only two literals. For example,

\[ m_5 + m_7 = x'y'z + xyz = xz(y' + y) = xz. \]

*Example 3.* Simplify the Boolean function \( F(x, y, z) = \sum(2, 3, 4, 5). \)
In certain cases, two squares in the map are considered to be adjacent even though they do not touch each other. For example, in the three-variable map \( m_0 \) is adjacent to \( m_2 \), and \( m_4 \) is adjacent to \( m_6 \), because the minterms differ by only one variable.

Example 4. Simplify the Boolean function \( F(x, y, z) = \sum(3, 4, 6, 7) \).

Example 5. Simplify the Boolean function \( F(x, y, z) = \sum(0, 2, 5, 6) \).
1.3 *Four-Variable Map*

In the four-variable map there are 16 minterms. In addition to the columns, the rows are numbered such that only one variable changes between two adjacent rows or columns. Adjacent squares are defined to be squares next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as right and left edges touching each other. Figure 3 shows the four-variable map.

*Example 6.* Simplify the Boolean function $F(w, x, y, z) = \sum(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)$.

---

**Figure 3:** Four-variable map.
Example 7. Simplify the Boolean function \( F(A, B, C, D) = A'B'C' + B'CD' + A'BCD' + AB'C' \).

1.4 Five-Variable Map

Maps for more than four variables are not simple to use as maps for four or fewer variables. A five-variable map needs 32 squares and a six-variable map needs 64. When the number of variables become large, the number of squares become excessive and the geometry for combining adjacent squares become more involved.

The five-variable map consists of 2 four-variable maps with variables \( A, B, C, D, \) and \( E \). Variable \( A \) distinguishes between the two maps. The left-hand four-variable map represents the 16 squares in which \( A = 0 \), and the other four-variable map represents the squares in which \( A = 1 \). Minterms 0 through 15 belongs with \( A = 0 \) and
minterms 16 through 31 with $A = 1$. In addition to the adjacency that we already discussed, each square in the $A = 0$ map is adjacent to the corresponding square in the $A = 1$ map. Figure 4 shows the five-variable map.

Maps with six or more variables need too many squares and are impractical to use. The alternative is to use computer-based methods.

Example 8. Simplify the Boolean function $F(A, B, C, D, E) = \Sigma(0, 2, 4, 6, 9, 13, 21, 23, 25, 29, 31)$.
2 Product-of-Sums Simplification

The minimized Boolean function derived from the map in all previous examples were expressed in sum-of-products form. With a minor modification, the products-of-sums form can be obtained.

The procedure for obtaining a minimized function of product-of-sums form follows from the basic properties of Boolean functions. The minterms not included in the standard sum-of-products form of a function denote the complement of the function. From this observation, we see that the complement of the function is represented in the map by the squares not marked by 1s. If we mark the empty squares by 0s and combine them into valid adjacent squares, we obtain a simplified expression of the complement of the function, $F'$. The complement of $F'$ gives back $F$. Because of the generalized DeMorgan's theorem, the function so obtained is automatically in product-of-sums form.

Example 9. Simplify the Boolean function $F(A, B, C, D) = \sum(0, 1, 2, 5, 9, 10)$ into (a) sums-of-product form and (b) product-of-sums form.
3 Don't-Care Conditions

The logical sum of minterms associated with a Boolean function specifies the conditions under which the function is equal to 1. The function is equal to zero for the rest of the minterms. This assumes that all the combinations of the values for the variables of the function are valid. In some applications, the function is not specified for certain combinations of variables. Functions that have unspecified outputs for some input conditions are called incompletely specified functions. In most applications, we simply don’t care what value is assumed by the minterms of a function’s don’t-care conditions.

To distinguish don’t-care conditions from 1s and 0s, an x is used. In choosing adjacent squares to simplify the function in a map, the don’t-care minterms may be assumed to be either 0 or 1.

Example 10. Simplify the Boolean function

\[ F(w, x, y, z) = \sum (1, 3, 7, 11, 15) \]

which has the don’t-care conditions

\[ d(w, x, y, z) = \sum (0, 2, 5). \]

4 NAND and NOR Implementations

Digital circuits are frequently constructed with NAND or NOR gates rather than with AND or OR gates. NAND and NOR gates are easier to fabricate and are basic gates used in IC digital logic families.
NAND Circuits

The NAND gate is said to be a universal gate, because any digital system can be implemented using NAND gates. To show that any circuit can be implemented with NAND gates we need only to show that logical operations, AND, OR, and NOT, can be obtained by NAND gates alone.

A convenient way to implement a Boolean function with NAND gates is to obtain the simplified Boolean function and then convert the function to NAND logic.

Two-Level Implementation

The NAND implementation requires the function to be in sum-of-products form.

Example 11. Implement the Boolean function $F = AB + CD$ using NAND gates.

Example 12. Implement the following Boolean function using NAND gates:

$$F(x, y, z) = \sum (1, 2, 3, 4, 5, 7).$$
NAND Implementation Procedure

1. Simplify the function and express it in sum-of-products form.

2. Draw a NAND gate for each product term of the expression that has at least two literals. The inputs to each NAND gate are the literals of the term. This procedure produces a group of first-level gates.

3. Draw a single gate using AND-invert or the invert-OR graphic symbol in the second level, with inputs coming from the outputs of the first-level gates.

4. A term with a single literal requires an inverter in the first level. However, if the single literal is complemented, it can be connected directly to an input of the second-level NAND gate.

Example 13. Implement the following Boolean function using NAND gates:

\[ F = (AB' + A'B')(C + D'). \]
NOR Implementation

The NOR operation is the dual of the NAND operation. Therefore, all procedures and rules for NOR logic gates are the duals of the corresponding procedures and rules developed for the NAND logic. The NOR gate is another universal gate that can be used to implement any Boolean function.

A two-level implementation with NOR gates require that the function be simplified into product-of-sum form.

Example 14. Implement the following Boolean function using NOR gates:

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