

## **Combinational circuits**

- 1 Logic expressions
- 2 Truth table
- 3 Maps and their use
- 4 Minimization of logic functions
- 5 Elimination of redundant implicants
- 6 Group minimization
- 7 Minimization with maps
- 8 Quine - McCluskey minimization method
- 9 Espresso minimization methods
- Appendix - basic rules of Boolean algebra

## Combinational circuits

It is characteristic of combinational circuits that they always react to a certain combination of input signal values with the same combination of output signal values, regardless of past signal values. The combination of signal values is called a state - there is an input state and an output state.

The operation of combinational circuits can be partially described by **logic functions**. Partly because logic functions work with logic variables which can take only the values '0' or '1'. They cannot describe transient phenomena, but only steady states. Logical functions are expressed in several ways:

- logic expression
- truth table
- logic map
- programming language

### 1 Logic expressions

The logic expression consists of logic **variables**, **operators** and logic **constants** '0' and '1' - beware of confusion with numbers 0 and 1 of the integer type.

The most commonly used system of operators is the logic sum  $a + b$  (OR function), the logic product  $a \cdot b$  (AND function), and negation  $\bar{a}$ . The labeling of negation is not uniform in the literature. The best readable is the bar over the variable, you can often see the notation of negation as  $/a$  or  $a'$ . Both make it easier to write text with a regular editor, but complicate the writing of more complex expressions, as it requires the use of parentheses.

**Expressions** are created and modified according to the basic rules of Boolean algebra. It is supposed they are sufficiently known – see **Appendix** to this chapter if it is necessary to fill the gaps.

### 2 Truth table

The truth table assigns values to the output (s) of the circuit to all possible combinations of values at the inputs of the circuit. The example of the function of three variables  $f(x, y, z)$  is given in Table 1.

DEK	x	y	z	$f(x, y, z)$
0	0	0	0	1
1	0	0	1	1
2	0	1	0	-
3	0	1	1	0
4	1	0	0	1
5	1	0	1	-
6	1	1	0	0
7	1	1	1	1

Tab. 1. Example of truth table

In some rows of the output column, there are **dashes** that indicate **unspecified** output states. Unspecified status results from the situation when a certain set of values at the input **cannot occur**, most often for some technical limitations. If this input file cannot occur, it will not matter how we fill the status at the output. It should be noted that there are no unspecified states on the outputs in a **real** circuit. The circuit always generates some combination of values of output signals - in unspecified states, however, it does not matter what signals it emits. In practice, unspecified states are filled so that the designed combinational circuit is as simple as possible. At the end of the design, all the unspecified states will be completely known and the behavior of the circuit will be determined in all states. Unspecified conditions occur **very often** in practice not only in combinational, but also in sequential circuits (see other chapters).

If we read the individual input signals from left to right, we get a sequence of zeros and ones, which we can consider as a binary number - we can also convert it to a decimal number. Then each row of the table has its own **decimal index** DEK. If we follow this convention, we do not have to draw a table, but it is enough to list the indexes of the rows in which the functional value is '1'. It is a very economical notation. In the first parenthesis are specified states; in the second parenthesis are unspecified states:

$$f(x, y, z) = \sum m(0, 1, 4, 7(2, 5))$$

The symbol  $\sum$  indicates that the function  $f$  is expressed in a **sum of products** or **disjunctive** form - **SPF**. It is, of course, a **logic** sum.

Any logic function can be expressed as the sum of elementary logic functions  $m_i$ , each of which has the value '1' on only one row of the table. Such elementary functions must contain all variables of the function  $f$ , whether non-negated or negated. For example, an elementary function has a value of '1' only in the line DEK = 0, a function has a value of '1' only in the line DEK = 1, a function has a value of '1' only in the line DEK = 2, etc. The elementary functions defined in this way are called **minterms**. The resulting sum of minterms is called the **initial sum of products** form (or initial disjunctive form) of the function. Each minterm **covers** exactly one row of the truth table with a functional value of '1', or one **point** of the function.

For the above example from table 1 we get SPF as:

$$f(x, y, z) = \bar{x}\bar{y}\bar{z} + \bar{x}\bar{y}z + x\bar{y}\bar{z} + xyz (+ \bar{x}y\bar{z} + x\bar{y}z)$$

Minterms in the brackets belong to the unspecified states. Unfortunately, this initial form is too complicated for further work - it contains too many letters. For simplification or **minimization** of functions exist well documented systematic processes.

As well as listing the rows with function value '1', it would be possible to enumerate the rows with function value '0'. Then we get the function from the above table 1 in the form

$$f(x, y, z) = \prod M(3, 6(2, 5))$$

The symbol  $\prod$  indicates that the function  $f$  is expressed in a **product of sums form or conjunctive** form - **PSF**. Any logical function can be expressed as the product of elementary logical functions  $M_i$ , each of which has the value '0' on only one row of the table. The elementary functions defined in this way are called **maxterms**. Like minterms, maxterms contain all variables, but in a logic sum. Thus, e.g. the row with DEC = 3, when  $f$  is '0', there would be a maxterm  $M_3 = x + \bar{y} + z$ . It has a value of '0' only when  $x = '0'$ ,  $y = '1'$ ,  $z = '1'$ . The resulting product of maxterms is called the **initial product of sums form** (or initial conjunctive form) of the function. Each maxterm covers exactly one point of the function (i.e. the row of the truth table). Initial PSF is too complicated to implement the circuit. There are also systematic procedures for simplifying it.

For the above example from table 1 we get PSF as:

$$f(x, y, z) = (x + \bar{y} + \bar{z}) \cdot (\bar{x} + \bar{y} + z) \cdot ((x + \bar{y} + z)(\bar{x} + y + \bar{z}))$$

If the truth table is filled with "ones" too densely (there are more ones than zeros), there will be a large number of terms in the resulting SPF. Then it is worth solving a function like PSF - on the contrary, it will be filled with zeros sparsely and it will be simpler.

### 3 Maps and their use

One of the options for describing a function is a **map**. The most used form of map is a **Karnaugh map**. It is arranged in a square or rectangle so that the **adjacent** fields always differ in only **one variable**. Fig. 1 shows a map for 3 variables, Fig. 2 a map for 4 variables.

	1		

Fig. 1: Karnaugh map for 3 variables

1	0	0	-
1	1	1	1
1	0	0	-
1	0	0	-

Fig. 2: Karnaugh map for 4 variables

Each individual **field** (cell) of the map corresponds to a combination of values of all variables. If the field is below the bar marked next to a variable, then that variable will be non-negated in the minterm. If it is outside the bar, the variable will be negated in the minterm. For example, the field marked X in the map in Fig. 1 will correspond to a combination  $\overline{a}b\overline{c}$ .

From the truth table of the function, it is easy to build its map and vice versa. The rows of the truth table in which the functional value is '1' correspond to the map fields with the states '1'; similarly for zeros. The map can also contain **unspecified** states like the truth table.

The map in Fig. 2 shows a logic function

$$f(a, b, c, d) = \sum m(0, 1, 2, 3, 6, 10, 14, (4, 5, 7))$$

Maps can be extended to a larger number of variables. Fig. 3 shows a map for 5 variables and Fig. 4 a map for 6 variables. The map for 5 variables is obtained by flipping the map for 4 variables along the vertical axis of symmetry and adding the fifth variable, the map for 6 variables is obtained by flipping the map for 5 variables along the horizontal axis of symmetry and adding the sixth variable.


Fig. 3: Karnaugh map for 5 variables


Fig. 4: Karnaugh map for 6 variables

Maps for 3 and 4 variables are very clear, the map for 5 variables is partly losing clarity, and with a map for 6 or more variables it is practically impossible to work. This implies the importance of maps for processing logic functions - they are only suitable for **small** tasks. But they are very suitable for **explanation of principles** and, therefore, they are widely used in textbooks. The map for 4 variables is particularly suitable for this purpose.

#### 4 Minimization of logic functions

The initial SPF, obtained from the truth table, can be further modified, completely systematically. Modifications to simplify a logical function are called **minimization**. The **number of letters** in an expression for a function is generally accepted as a criterion of minimality. If a letter occurs several times, we count it each time. This criterion has a very

**realistic** basis. Logic devices (CMOS) have 2 transistors per input, whether AND, OR, or NOT. The number of inputs corresponds to the number of variables in each logic product, so that the number of letters. Thus, the total number of letters roughly corresponds to the **number of transistor pairs** (NMOS-PMOS) in the integrated circuit.

We denote the minterms of the function by numbers and try to perform the operation of **combining** between each pair. The combining operation is defined as

$$a.\bar{b} + a.b = a ,$$

where  $a$  can be both a single variable and a logical expression and  $b$  is a single variable. The combining operation apparently **excluded the** variable  $b$ . Note that the condition for combining minterms is that they must be **adjacent**, i.e. they must differ only in **one** variable (here  $b$ ) - in one minterm it is direct (non- negated ), in the other it is negated.

As an example, let's have a following function with SPF which has been extracted from the truth table:

$$f(a, b, c, d) = \begin{array}{cccccc} \text{1} & \text{2} & \text{3} & \text{4} & \text{5} & \text{6} \\ \overline{a}\overline{b}\overline{c}\overline{d} & \overline{a}\overline{b}c\overline{d} & \overline{a}b\overline{c}\overline{d} & \overline{a}b\overline{c}d & \overline{a}bc\overline{d} & \overline{a}bcd \\ \hline \end{array}$$

The minterms are numbered from 1 to 6. As the next step the minterms will be combined each with each. The results are as follows:

$$\begin{array}{ll} 1-2 \dots & \overline{a}\overline{b}\overline{d} \\ 1-4 \dots & \overline{a}\overline{b}c \\ 2-3 \dots & \overline{a}b\overline{c} \\ 2-5 \dots & \overline{a}c\overline{d} \\ 3-4 \dots & \overline{a}b\overline{d} \end{array}$$

The following minterms cannot be combined:

$$1-5, 1-6, 2-4, 2-6, 3-5, 3-6, 4-5, 4-6, 5-6$$

Each minterm that has been combined with another is appropriately marked (e.g. by underlining). Minterms that could not be combined (are unmarked) will be left **without change** (here it is member 6).

The combining of minterms 1 and 2 worked as follows:

$$\overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}c\overline{d} = \overline{a}\overline{b}\overline{d}.(c + \bar{c}) = \overline{a}\overline{b}\overline{d}$$

A new product was created, 1 letter shorter. The question is what happened to the original two minterms - if they disappeared, member 1 would be missing for further combining 1-4. Fortunately, in Boolean algebra, the following holds:

$$X = X + X$$

Thus, the original minterms could be arbitrarily "multiplied" and at the appropriate time cancelled, because:

$$X + X = X$$

We combine newly created products with a logic sum. The result of the first "round" of combining is:

$$f(a, b, c, d) = a\bar{b}\bar{d} + a\bar{b}\bar{c} + a\bar{b}c + ac\bar{d} + a\bar{b}d + \bar{a}b\bar{c}\bar{d}$$

The simplification against the original expression is obvious. The products of combining are called **implicants** of the 1st order. Each implicant was created by two minterms and so, it covers two rows of the original truth table or, in other words, it **covers two points** of the original function. We will continue in the simplification in the second "round", using the same rules. As in the first round, the 1st order implicants that differ in only one variable can be combined. It is important that both implicants must have the **same variables** - in the first round, some variables disappeared.

Again we number the implicants and try to connect each one with each other. The results of the combining are as follows:

$$\begin{array}{l} 1-5 \dots a\bar{b} \\ 2-3 \dots a\bar{b} \end{array}$$

The following implicants cannot be combined:

$$1-2, 1-3, 1-4, 1-5, 2-3, 2-4, 2-5, 3-4, 3-5, 4-5$$

The members that could not be combined are  $ac\bar{d}$  and the original unconnected  $\bar{a}b\bar{c}\bar{d}$ . Two identical members  $a\bar{d}$  merge into one.

The result of the second round will be:

$$f(a, b, c, d) = a\bar{b} + ac\bar{d} + \bar{a}b\bar{c}\bar{d}$$

Further simplification is no longer possible.

If the SPF can no longer be simplified by the above procedure, we get the **terminal SPF**. Its members are called **prime implicants**. Unlike a minterm which covers only one point of a function, each implicant **covers** more than one point of a function. If one variable was excluded during the operation of combining, the implicant will cover two points. If two variables have been excluded, the implicant will cover four points, etc. We consider an implicant to be **larger** when it covers a larger number of points, i.e. when it consists of a **smaller** number of variables.

In case the logic function is **unspecified states**, first they should be filled in as '1'. It gives more opportunities for simplification of specified minterms. Those of the unspecified states that could not be combined with other minterms should then be dropped (i.e. filled in as '0') so that they do not complicate the result.

## 5 Elimination of redundant implicants

We often do not need all the implicants of terminal SPF to fully cover all initial SPF minterms. Some may prove **redundant** and may be eliminated without changing the coverage of the truth of the function - remaining implicants already cover all minterms. This further reduces the number of letters. Conversely, some implicant is **essential** - it cannot be excluded if there is at least one minterm covered only by that implicant. Finally, the third group of implicants is optional and can be formed in **various** combinations of implicants which are sufficient to fully cover the remaining minterms.

The table of implicants gives clear information.

### Example:

Let's have a function given as:

$$f(a, b, c, d) = \sum m(0, 1, 2, 5, 7, 8, 9, 10, 13, 15)$$

The function was converted to a truth table and a SPF was found. Minterms and implicants marked with capital letters are listed in Table 2. The coverage of minterms by individual implicants is shown by crosses.

		minterms									
	implicant	0	1	2	5	7	8	9	10	13	15
A	$\bar{b}\bar{c}$	x	x				x	x			
B	$\bar{b}\bar{d}$	<u>x</u>		<u>x</u>			<u>x</u>		<u>x</u>		
C	$\bar{c}d$		x		x			x		x	
D	$bd$				<u>x</u>	<u>x</u>				<u>x</u>	<u>x</u>

Tab. 2: Example of a table of implicants

It is obvious that minterms 2 and 10 are covered only by the implicant B, so this is **essential**. Similarly, the essential implicant D covers minterms 7 and 15. Therefore, implicants B and D **must** be selected. Due to this choice, implicants 0, 8, 5, 13 are also covered. Coverage of minterms is marked by underlining the crosses. Now only minterms 1 and 9 remain (not underlined). To cover them, we can choose either A or C. We also could select both, but then we would not make any reduction. If there is a choice, the implicants with the least number of letters are chosen (in this case, they are equal). The implicant which was not selected is **redundant**.

The resulting **minimum form** of the function is therefore:

$$f(a, b, c, d) = \bar{b}\bar{d} + bd + \bar{b}\bar{c} \quad \text{or} \\ f(a, b, c, d) = \bar{b}\bar{d} + bd + \bar{c}d$$



In the case of functions with **unspecified states**, only those minterms that belong to fully specified states are listed in the table of implicants (there is 1 in the truth table). This simplifies coverage.

This procedure is quite laborious and error-prone for large tables. In this case, the **Petricks method** can be used, which makes the whole procedure easier. The method is based on the expression of coverage conditions using Boolean algebra. The individual implicants are again named by letters, e.g. A, B, C, ... etc. The coverage of minterms is expressed by logical conditions. For example, minterm 0 may be covered by implicant A or B, or both. We then write the condition as a logical sum ( $A + B$ ). Similarly to cover minterm 1, the condition is ( $A + C$ ), etc. In the case of coverage of a minterm by only one (essential) implicant, the condition is expressed as a single letter - e.g. B. Since all the minterms must be covered, then all conditions must be in force **simultaneously** - we express this by their logic **product**. For the above table we get the product column by column:

$$(A + B) (A + C) B (C + D) D (A + B) (A + C) B (C + D) D$$

We process the expression as a normal logic expression with known operations:

By excluding repeated members ...  $X + X = X$

By multiplying ...  $X (Y + Z) = XY + XZ$

By absorbing ...  $XY + X = X$

We will remain:

$$\begin{aligned} BD (A + B) (A + C) (C + D) &= BD (A + AC + AB + BC) (C + D) = BD (A + BC) (C + D) = \\ &= BD (AC + AD + BC + BCD) = BD (AC + AD + BC) = ABCD + ABD + BCD = \\ &= \mathbf{ABD + BCD} \end{aligned}$$

In order for this expression to have the value "true", it is sufficient to select only one of the members in the final sum, generally the one with the smallest number of letters - in this case arbitrary. This corresponds to a selection of implicants as it was derived above.

$$bd + bd + \bar{b}\bar{c} \quad \text{or} \quad b d + bd + \bar{c}d$$

The Petrick method is suitable for **larger** tasks; it can be easily **programmed** for a computer.

Knowledge of the procedure for solving the problem of optimal coverage is useful not only in electronics, but also in many other areas of human activity.

## 6 Group minimization

Combinational circuits usually have several output variables. It is possible to design the whole circuit as a set of **completely isolated** logic functions that have only input variables in common. The functions are designed and minimized separately. This guarantees the required function of the whole circuit, but often rather **uneconomically**. The implicants of one function could be used in other functions which would make the whole circuit simpler. A simple method is to minimize each function separately and then make a **list** of all implicants of all functions. From this list, the implicants are inserted into individual functions in their proper

combinations. This ensures that each implicant is implemented only once in the whole circuit.

## 7 Minimization in maps

The construction of the Karnaugh map is significant in that the coordinates of the fields are arranged so that they differ in only one variable for adjacent fields. Thus, geometrically adjacent fields are also adjacent in an algebraic sense (i.e. they differ in a single variable). Each field with a value of '1' corresponds to a **minterm** from the truth table. Adjacent fields thus correspond to minterms differing in only one variable can be combined into **implicants**. Adjacent are also the fields on the edges of the map, because they also differ in only one variable (end of rows, end of columns and corners of the map). For the map for 5 and 6 variables, adjacent are also the fields symmetrically placed around the axis of symmetry. The connection of the fields is marked by a **loop**. Fields in adjacent pairs can be combined into larger loops, those again into larger loops, etc. Therefore, each loop must have a side exactly  $2^k$  fields long, where  $k$  is a positive integer. Loops include 2 fields, 4 fields, 8 fields, etc.

Each loop in the map corresponds to an **implicant** of the function. The principle of **minimization** consists in covering all fields with a value of '1' (and any number of unspecified fields) with a system of loops, whereby:

- loops must be as large as possible
- number of loops must be as small as possible

This principle is illustrated on the function of four variables in the map according to Fig. 5.

$$f(a, b, c, d) = \sum m(0, 1, 2, 3, 6, 10, 14, (4, 5, 7, 15))$$

The minterm  $\bar{a}\bar{b}\bar{c}\bar{d}$  corresponds to the index 0 and in the map the box at the top left, the minterm  $\bar{a}\bar{b}cd$  with the index 1 corresponds to the box at the bottom left, etc.

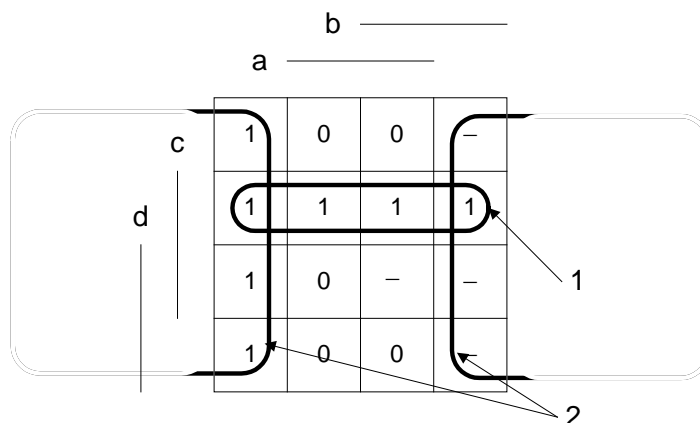


Fig. 5: Example of minimizing a function of 4 variables

The loops have been constructed so that each included as many fields as possible, including those with unspecified values. Loop 1 corresponds to the implicant  $c\bar{d}$ , loop 2 corresponds to the implicant  $\bar{a}$ . Although the loop 2 is composed of two halves in the map, it meets the conditions for creating loops. We can imagine the map as originally drawn on the surface of a sphere, so that the fields at the ends of the rows and columns were geometrically adjacent. The map was then cut and leveled. Originally, the adjacent fields are now at the ends of the rows and columns, but in the algebraic sense (they differ only in one variable) they remained neighboring.

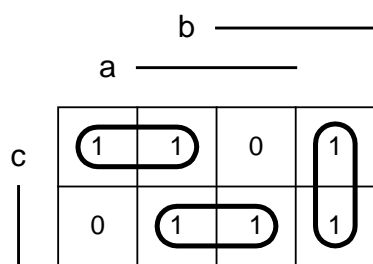
The condition of creating the largest possible loops is fulfilled by including unspecified states in the loop. Thus, they were preferably filled as '1'. The remaining unspecified state would require the creation of a new loop, which would unnecessarily complicate the function. So it was filled as '0'. The loop 1 could only include the two middle fields of the line, but the implicant would have one letter more ( $acd$  instead of  $c\bar{d}$ ). The two outer fields are thus covered twice, but this does not affect the functional value - this corresponds to **multiple coverage** of some points of the function. The resulting function in the minimal form is:

$$f(a, b, c, d) = \bar{a} + c\bar{d}$$

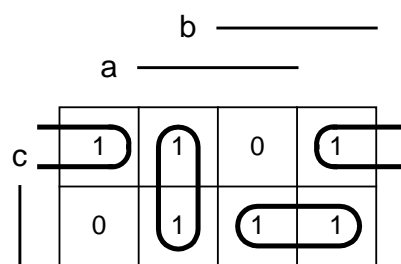
Minimizing functions in maps corresponds to minimizing SPF as follows :

minterm .....	map field
implicant .....	loop
minimum number of letters in implicant .....	maximum size of loop
minimum number of implicants .....	minimum number of loops

The elimination of redundant implicants in maps is not necessary, because redundant loops are not introduced at all due to the clarity of the map. It should be taken into account that the function can have several SPFs, which in the maps means several possible variants of loop selection. This illustrate maps in Fig. 6 and Fig. 7, where the loops were created in two ways, but both provide equally complex expressions:



$$f(a, b, c) = \bar{b}\bar{c} + ac + \bar{a}b$$



$$f(a, b, c) = \bar{a}\bar{c} + \bar{a}b + bc$$

Fig. 6: First variant of function minimization

FIG. 7: Second variant of function minimization

Both functions are **equivalent** - their truth tables are identical, they can be both written in a single table 3:

$a$	$b$	$c$	$f(a,b,c)$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Tab. 3: Truth table of functions according to Fig. 6 and 7

For simple cases, it is not necessary to compile a table as an intermediate stage, but a map can be compiled directly from the requirements for the behavior of the circuit. Maps give a very clear view of logical functions and their minimization. That is their main meaning. However, **their practical use** in the design of digital systems is very **small**. This is related to the concept of neighborhood. Up to four variables it is easy to create an optimal set of loops. For 5 variables, more imagination is needed in the use of axial symmetry, for 6 variables it is even more difficult. For a larger number of variables, maps are completely meaningless. It can therefore be stated that their applicability ends with five variables. However, this range is too small for practice.

### Example

Using the map to minimize the function of 5 variables - see Fig. 8. The use of axial symmetry makes possible to create an  $abd$  loop, which is composed of two symmetrically placed components. Their connection is marked with arrows. The resulting expression for the function would be:

$$f(a, b, c, d) = a\bar{c}d + abd + \bar{d}e + \bar{a}\bar{b}c\bar{d}$$

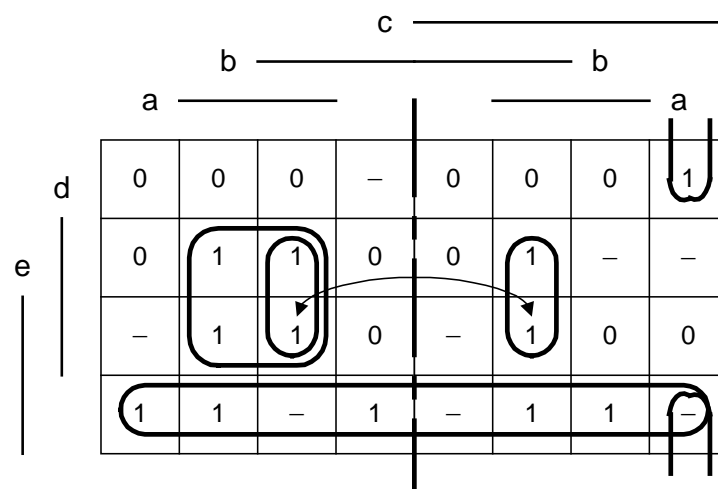


Fig.8: Example of minimizing the function of 5 variables

## 8 Quine - McCluskey minimization method

The method was developed in 1956 and is suitable for computer use - it has been programmed many times. The basic procedure is based on the minimization of the truth table in the Sum of Product Form, as described in paragraph 4. For functions with large number of variables, however, the set of minterms is also large and the number of their combinations becomes enormous. The Q-MC methods simplifies the phase of minterm-to-minterm combining and makes it possible to process functions with large number of variables.

The reduction starts by creating a table with only those rows in which the functional value is 1 instead of the whole truth table - a table of minterms will be created. Minterms in the form of the product of variables and their negations are replaced by their **binary representations** - for example, instead of  $ab\bar{c}d$  it is possible to write 1101. A combination of zeros and ones can be assigned a binary number and its decimal equivalent - in this case 13.

Combining minterms is possible only when they differ in only one variable - one is negated and the other is not, e.g.  $ab\bar{c}d$  and  $a\bar{b}\bar{c}d$ , in binary representation 1101 and 1001. To avoid the need trying the combinations by method "each with each", which is factually correct but very laborious, we sort the minterms **into groups** according to the number of **ones**. The number of ones in a group is called an **index**. Only minterms from neighboring groups, which differ in index by one, can be combined. This condition is not enough - minterms must differ in **the same position**. Thus e.g. minterms  $ab\bar{c}d$  and  $a\bar{b}c\bar{d}$  can be combined with the result  $a\bar{c}d$ , while e.g.  $ab\bar{c}d$  and  $a\bar{b}c\bar{d}$  cannot be combined - however, in both cases the minterms belong to groups with subscripts 2 and 3.

### Example:

We have a table of minterms - Tab. 4. In Tab. 5 are minterms in decimal representation and sorted into groups with indices 0 to 4 depending on the number of ones in the binary representation.

dec.	a	b	c	d	
0	0	0	0	0	$\bar{a}\bar{b}\bar{c}\bar{d}$
1	0	0	0	1	$\bar{a}\bar{b}\bar{c}d$
2	0	0	1	0	$\bar{a}\bar{b}c\bar{d}$
5	0	1	0	1	$\bar{a}b\bar{c}d$
7	0	1	1	1	$\bar{a}bcd$
8	1	0	0	0	$a\bar{b}\bar{c}\bar{d}$
9	1	0	0	1	$a\bar{b}\bar{c}d$
10	1	0	1	0	$a\bar{b}c\bar{d}$
13	1	1	0	1	$ab\bar{c}d$
15	1	1	1	1	$abcd$

Tab. 4: Reduced truth table

dec.	ind.	a	b	c	d	combined
0	0	0	0	0	0	*
1	1	0	0	0	1	*
2		0	0	1	0	*
8		1	0	0	0	*
5	2	0	1	0	1	*
9		1	0	0	1	*
10		1	0	1	0	*
7	3	0	1	1	1	*
13		1	1	0	1	*
15	4	1	1	1	1	*

Tab. 5: Table with minterms sorted by index

In facilitating and mechanizing the combining process, further progress can be made and the **decimal representation of minterms** can be used instead of their binary representation in all phases of processing. Combining minterms is only possible between minterms with indices that **differ by one** and at the same time differ in a **single variable**. Rather than comparing zeros and ones, this can be verified by comparing the decimal numbers of minterms. If two minterms differ in exactly one bit, their decimal numbers must differ by the **power of two**. For example, minterm 3 (0011) can be combined with minterm 7 (0111), as it differs by 4, which is caused by the difference in the third place on the right with the weight  $2^2 = 4$ . Tab. 5 shows a procedure with the same set of minterms as in Tab. 2. We will perform the merging by gradually calculating the differences between the numbers in group 0 and group 1, then between the numbers in group 1 and group 2, group 2 and group 3, etc. If the difference is other than the power of two, the merging did not take place. If it is equal to the power of two, the combining took place and a **new product** was created (it is an implicant). Implicants which were successfully combined are labeled (e.g. with an asterisk). The results are written in Table 6. The numbers in parentheses indicate the weights of the variables that were excluded during the connection - e.g. (8) means the weight  $2^3$  and thus the fourth variable from the right.

For example, minterm No. 2 (with index 1) in tab. 5 can only be combined with minterms with indices 2 (5, 9, 10), but their decimal equivalents should differ only by 1, 2, 4, 8. This is satisfied only by minterm 10, which differs from minterm 2 by the number 8. At the same time, the variable with this weight, i.e. the fourth from the right, was eliminated. The resulting implicant is thus written as 2,10 (8) with index 1.

For a better view of the procedure, the shapes of implicants after the first coupling are given in Table 7. A hyphen is written in place of the excluded variable.

dec.	ind.	product	combined
0	0	0,1 (1)	*
		0,2 (2)	*
		0,8 (8)	*
1	1	1,5 (4)	*
2		1,9 (8)	*
8		2,10 (8)	*
		8,9 (1)	*
		8,10 (2)	*
5	2	5,7 (2)	*
9		5,13 (8)	*
10		9,13 (4)	*
7	3	7,15 (8)	*
13		13,15 (2)	*

Tab. 6: Combining minterms with decimal representation

dec.	ind.	a	b	c	d	combined
0,1	0	0	0	0	-	*
0,2		0	0	-	0	*
0,8		-	0	0	0	*
1,5	1	0	-	0	1	*
1,9		-	0	0	1	*
2,10		-	0	1	0	*
8,9		1	0	0	-	*
8,10		1	0	-	0	*
5,7	2	0	1	-	1	*
5,13		-	1	0	1	*
9,13		1	-	0	1	*
7,15	3	-	1	1	1	*
13,15		1	1	-	1	*

Tab. 7: Implicants after the first coupling

There is another condition for combining - a minterm with a lower index and a higher decimal value cannot be combined with a minterm with a **higher** index and a **lower** decimal value. This is clear in tab. 5 for minterms 9 (binary 1001) and 7 (binary 0111) - they differ in two variables.

Further combining, i.e. combining of first order implicants, shows tab. 8. For a better view of the procedure, the shape of the implicants after the second connection is given in Table 9. In addition to the already explained condition on permissible differences in decimal equivalents, the condition on **identical positions** of excluded variables ( i.e. numbers in parentheses in tab. 6) must be applied. Thus, for example, the implicant 0,2 (2) can be combined only with the implicants with the number (2), e.g. 8,10 (2), which meets the condition of permissible differences in dec. equivalents - here is the difference 8. The resulting implicants of the second order have two variables less and thus cover 4 minterms. No other second-order implicants can be combined in this example.

dec.	ind .	combined
0,1,8,9 (1,8)	0	
0,2,8,10 (2,8)		
1,5,9,13 (4,8)	1	
5,7,13,15 (2,8)	2	

Tab. 8: Combining implicants  
(already the second combining)

dec.	ind .	a	b	c	d	combined
0,1,8,9	0	-	0	0	-	
0,2,8,10		-	0	-	0	
1,5,9,13	1	-	-	0	1	
5,7,13,15	2	-	1	-	1	

Tab. 9: Implicants after the second combining

Now we need to convert the decimal representation to the sum of products of the variables. Suitable for this purpose is any of the minterms covered by the selected implicant. Variables that are not represented in the selected implicant are omitted. E.g. from the implicant 5,7,13,15 (2,8) we choose minterm 5, binary 0101. We omit the variables with weights 2 and 8 and we get -1-1. Taking into account the set of variables  $a, b, c, d$  defined at the beginning of the example we get  $bd$ . Using the same procedure, we obtain the resulting system of implicants

$$\bar{b}\bar{c}, \bar{b}\bar{d}, \bar{c}d, bd ,$$

which is the terminal SPF. Furthermore, redundant implicants should be ruled out, as already explained above. This way we would obtain the minimum SPF.

## 9 Espresso minimization method

Method Quine-McCluskey has been great progress in solving the minimization of large scale combinational circuits as it allowed intensive use of computers. Nevertheless, over time it reached its limit. With the development of technology, the number of variables in combinational circuits gradually increased, and for example in computers, they moved from the common standard of 16 bits to the common standard of 32 bits (and 64 bits). The complexity of circuit modifications (minimization) has thus increased exponentially. As a criterion of labor, we can accept the maximum number of **primary implicants** of a given function. In the literature, an approximate value of  $3^n/n$  is given, where  $n$  is the number of variables. For 8 variables it is 820, for 16 variables already  $2.69 \cdot 10^6$ , for 32 variables it is  $5.79 \cdot 10^{13}$ , etc.. Probably the problem is in too many minterms that have to be compared with each other. Therefore, methods have been developed that work differently. In principle, these are **heuristic** methods - that is, approximate - which do not emphasize **absolute** minimization, but rather **sufficient** minimization. This significantly saves the number of operations and thus

the time for solutions. The best known and most common method is the method **Espresso**. This method was invented in the 1980s and has been refined many times.

The first novelty is coding of the states of the variables. Here the variable  $x$  is not encoded by one bit (0 and 1), but by two bits as follows:

$x = 0 \dots$	code 10
$x = 1 \dots$	code 01
$x$ missing ...	code 11
invalid ...	code 00

Minterm  $\bar{a}bc\bar{d}$  is written in two-bit code as 10 01 11 10, implicant  $b\bar{c}d$  as 11 01 10 01, etc. The invalid code will be explained later. The 2-bit code allows to know by simple means whether two implicants have common points, or similarly, whether two loops have common fields in maps. For example, do the loops  $\bar{b}\bar{d}$  and  $a\bar{b}\bar{c}$  have common fields? Sure, we can put the loops in a map and find out if they intersecting - but we prefer using a computer (we will only use maps here as a means of easier understanding).

We perform a simple operation of a logic product column by column:

$a$	$b$	$c$	$d$	
-----				
11	10	11	10	corresponds to $b\bar{d}$
01	11	10	10	corresponds to $a\bar{c}\bar{d}$
-----				
01	10	10	10	corresponds to $a\bar{b}\bar{c}\bar{d}$ , which is the common point of both loops.

We will try to find the intersection of the implicants  $\bar{b}\bar{d}$  and  $a\bar{b}\bar{d}$ .

11	10	11	10
01	10	11	01
-----			
01	10	11	00

!!! invalid result, common points do not exist.

In Espresso, this procedure is used to automate the search for the largest possible implicants (loops). First, the coverage of zeros is constructed by a system of 0-loops, covering the **zeros** of the function - this coverage does not have to be minimal at all and is assembled as simply as possible - it is just an auxiliary construction to the beginning. Of course, the coverage by ones is also known, but no operations with minterms (combining) are performed. Conversely, we start from one point of the function (one row of the truth table with value 1), which corresponds to a loop around a selected single field in the map with value 1.

Next step is the **expansion** of the loop, i.e. the loop is enlarged in a randomly selected direction (imagine the situation in a map); this corresponds to elimination of one variable from the minterm. This creates a first-order implicant - a loop in the map over two fields with ones. There are more variables and therefore they are chosen randomly. However, it is necessary to find out whether the experimentally created implicant **does not interfere with** the zeros of the function - the new loop must not contain zeros. To do this, we will use the above described operation and test whether the implicant has an intersection with the whole coverage of zeros - that is, the loop over two ones is compared successively with all loops



over zeros. If there is at least one intersection, the selected direction of expansion is impossible and removed variable is returned. Then expansion in another direction is tried and tested, etc..

After successful expansion, **further expansion is** attempted by removing another variable and an admissibility check is performed again. When all possibilities are exhausted, the implicant is **maximal** (a prime implicant). There is a lot of simple operations, but much less than trying each minterm with each as in the methods previously described. The expansion procedure of the minterm  $abc\bar{d}$  is shown in Fig. 9. The third image from the left shows a failed expansion.

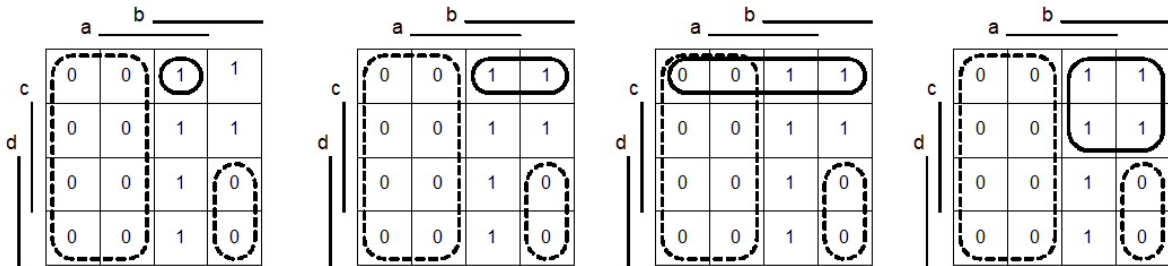


Fig. 9: Gradual expansion of the minterm  $abc\bar{d}$  (from left to right)

The same procedure is performed with other minterms of the function that do not yet belong to the already created implicants. In the end, this results in the **terminal** form of the function, which no more can be modified by the above procedure.

The function can be simplified further by eliminating **redundant** implicants. However, the Petrick method explained above is not suitable for too large tasks, so a different procedure was included in the Espresso method. It involves the following steps:

- **Reduction** - eliminatio the overlap of implicants by reducing their size - an apparent paradox, before that they were expanded. But it makes sense.
- **Re-expansion** of implicants.

The procedure is repeated until the desired simplification is achieved. It is not necessarily a minimal form. The re-expansion is carried out, if possible, in **different directions** (with different variables omitted) than in the previous rounds. Fig. 10 shows the procedure in a simple case.

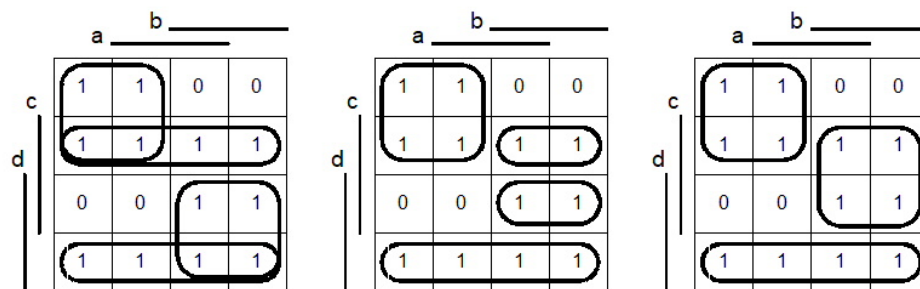


Fig. 10: Expansion - reduction - expansion (from left to right)

The picture on the left shows the initial state - it does not contain any redundant implicant and it would not be possible to modify it using the Patrick method described above. But when implicants in the original group have been suitably reduced (middle picture), another pattern appears (right). It has one less implicant.

Some Espresso operations may evoke the impression of random selection. However, at each point where the choice of the next step is decided, the authors of the method included appropriate evaluations of alternatives according to their probability of success. This significantly increased the speed of finding a good result. The word **good**, not the absolute **best**, needs to be emphasized.

The Espresso method is complicated, but it will easily solve the minimization of a 16-bit combinational circuit - on a regular PC it only takes a few seconds. It can handle larger circuits as well.

**Appendix** - basic rules of Boolean algebraInfluence of **constants** 0 and 1:

$$0 + a = a + 0 = a$$

$$1 + a = a + 1 = 1$$

$$1 \cdot a = a \cdot 1 = a$$

$$0 \cdot a = a \cdot 0 = 0$$

$$a + a = a$$

$$a \cdot a = a$$

**Negation** properties:

$$\overline{0} = 1, \overline{1} = 0$$

$$a + \overline{a} = 1$$

$$a \cdot \overline{a} = 0$$

$$\overline{\overline{a}} = a$$

In the following relations, the symbols  $a, b, \dots$  may denote both individual variables and whole **expressions**. The use of **parentheses** is the same as in ordinary algebra.

**Commutative law** :  $a + b = b + a,$   
 $a \cdot b = b \cdot a$

**Associative law** :  $a + b + c = a + (b + c),$   
 $a \cdot b \cdot c = a \cdot (b \cdot c)$

**Distributive law** :  $a \cdot (b + c) = a \cdot b + a \cdot c,$   
 $a + b \cdot c = (a + b) \cdot (a + c)$

**Duality** - each identity also applies after the mutual exchange of operators '+' and '·', and also of the constants '0' and '1'. This is clearly seen in all other rules.

**Absorption:**  $a + a \cdot b = a$  since:  $a + a \cdot b = a \cdot (1 + b) = a \cdot 1 = a$   
 dual:  $a \cdot (a + b) = a$  since:  $a \cdot (a + b) = a \cdot a + a \cdot b = a + a \cdot b = a$

**Combining:**  $a \cdot \overline{b} + a \cdot b = a$   
 dual:  $(a + \overline{b}) \cdot (a + b) = a$

**Simplification:**  $a \cdot (\overline{a} + b) = a \cdot b$   
 dual:  $a + \overline{a} \cdot b = a + b$

**Consensus** (exclusion of the third):  $a \cdot b + \overline{a} \cdot c + b \cdot c = a \cdot b + \overline{a} \cdot c$   
 dual:  $(a + b) \cdot (\overline{a} + c) \cdot (b + c) = (a + b) \cdot (\overline{a} + c)$

**DeMorgan's laws:**  $\overline{a + b} = \overline{a} \cdot \overline{b}$   
 dual:  $\overline{a \cdot b} = \overline{a} + \overline{b}$