Chapter 1 Boolean functions and combinational circuits

1.1 Short introduction into Boolean functions

1.1.1 Basic Boolean functions. There are two classes of digital circuits – combinational circuits and sequential circuits. In the first Chapter we will talk about combinational circuits. In such circuits, outputs at a time *t* depend only on the inputs in this time *t* and do not depend on the prehistory, that is, on what was at their inputs at the preceding times *t-1*, *t-2*, etc. Boolean functions are used to describe the behavior of combinational circuits. These functions can have only two values (we will use *0* and *1* for these values) and they depend on the variables with two values as well. First of all, we will discuss three basic Boolean functions – *AND*, *OR* and *NOT*.

Function *AND* with two variables is presented in the truth Table 1. In the left part of such a table we write all possible different vectors with components *0* and *1*. In general, a truth table has 2^n rows for a function with *n* variables (only four rows in our case). In the right part of such a table, in each row we write a value of a function at the corresponding vector. Function *AND* is equal to one when both variables are equal to one (the fourth row of Table 1), otherwise (rows from the first to the third) it is equal to zero.

Table 1. Function *AND*

a	*b*	*a and b*
0	*0*	*0*
0	*1*	*0*
1	*0*	*0*
1	*1*	*1*

Table 2 presents the truth table for function *OR*. This function is equal to zero only when both variables are equal to zero (the first row), otherwise it is equal to one. Function *NOT* (Table 3) has only two rows since it always has only one variable.

Table 2. Function *OR*

a	*b*	*a or b*
0	*0*	*0*
0	*1*	*1*
1	*0*	*1*
1	*1*	*1*

Table 3. Function *NOT*

a	*not a*
0	*1*
1	*0*

Any Boolean function can be realized by a combinational circuit containing logic gates. The simplest basic logic gates implementing Boolean function *AND*, *OR* and *NOT* are presented in Fig. 1–3. Gates *AND* and *OR* have two images each – rectangular and semi-oval. In our logic circuits we will use the former.

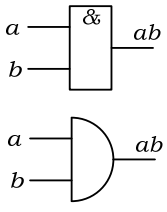


Figure 1. *AND* - gates

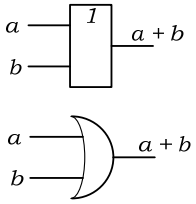


Figure 2. *OR* - gates

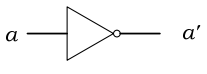


Figure 3. *NOT*-gate

1.1.2 The main laws and theorems of Boolean algebra

Operations with 0 and 1:

1. A + 0 = A; *A·1 = A*
2. A + 1 = 1; *A·0 = 0*

Idempotent theorem:

3. A + A = A; *A·A = A*

Involution theorem:

4. (A')' = A

Theorem of complementarity:

5. A + A' = 1; *A·A' = 0*

Commutative law:

6. A + B = B + A; *A·B = B·A*

Associative law:

7. (A + B) + C = A + (B + C) = A + B + C; *(AB)C = A(BC) = ABC*

Distributive law:

8. A(B + C) = AB + AC ; *A + (BC) = (A + B)(A + C)*

Simplification theorems:

9. AB + AB' = A; *(A + B)(A + B') = A*
10. A + AB = A; *A(A + B) = A*
11. (A + B')B = AB; *(AB') + B = A + B*

It is easy to define the functions *AND* and *OR* with three and more variables. We illustrate it in Table 4 for three variable functions. This table contains $2^3 = 8$ rows.

Table 4. Truth table for AND and OR with three variables

a b c	*abc*	*a + b + c*
0 0 0	*0*	*0*
0 0 1	*0*	*1*
0 1 0	*0*	*1*
0 1 1	*0*	*1*
1 0 0	*0*	*1*
1 0 1	*0*	*1*
1 1 0	*0*	*1*
1 1 1	*1*	*1*

Truth tables can be used to prove laws and theorems of Boolean algebra. Let us show this for the distributive law *a + bc = (a + b)(a + c)* in Table 5. The first column of this table contains all possible combinations of values *a*, *b* and *c*. In the next two columns

we constructed the left part of an expression for the distributive law – the product *bc* and the sum *a + bc.* In the same way we constructed the right part *(a + b) (a + c).*

To prove the equation *a + bc = (a + b)(a + c)* we should compare the grey columns of Table 5. Two functions are *equivalent* if they have the same values at the same bit-vectors. Thus, the distributive law *a + bc = (a + b) (a + c)* is proven.

However, if two functions are equivalent their circuit implementations may have different costs. Let us take it that the cost of a gate is equal to the number of inputs into this gate and that the cost of a circuit is equal to the sum of costs of gates in this circuit (to the total number of inputs into all gates in the circuit).

Table 5. Example of proving *a + bc = (a + b)(a + c)*

a b c	*bc*	*a + bc*	*a + b*	*a + c*	*(a + b)(a + c)*
0 0 0	*0*	*0*	*0*	*0*	*0*
0 0 1	*0*	*0*	*0*	*1*	*0*
0 1 0	*0*	*0*	*1*	*0*	*0*
0 1 1	*1*	*1*	*1*	*1*	*1*
1 0 0	*0*	*1*	*1*	*1*	*1*
1 0 1	*0*	*1*	*1*	*1*	*1*
1 1 0	*0*	*1*	*1*	*1*	*1*
1 1 1	*1*	*1*	*1*	*1*	*1*

The logic circuits corresponding to the left (f_1) and to the right (f_2) parts of the proven law are drawn in Fig. 4. These circuits are equivalent because they realize the equvalent functions (in reality, this is the same function presented by different expressions) but the costs of these circuits are different. Here for the first time, we have met the problem of logic circuit minimization. It is evident that some circuit is more minimized than the other one if the cost of the first of them is lower than the cost of the second one.

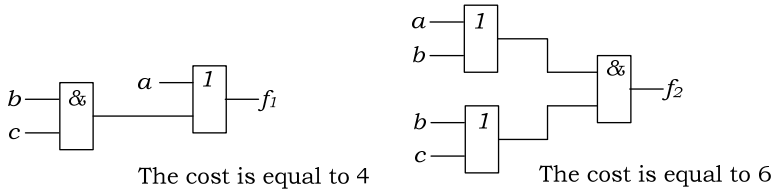


Figure 4. Two circuits for two parts of the distributive law

1.1.3 DeMorgan's theorems. The complement of the sum of two variables is equal to the product of complements of these variables:

$$(a + b)' = a'b' \qquad (1)$$

The complement of the product of two variables is equal to the sum of complements of these variables:

$$(ab)' = a' + b' \qquad (2)$$

It is very simple to expand these laws to any number of variables. Here we show that for three variables:

$$(a + b + c)' = ((a + b) + c)' = (a + b)'c' = a'b'c'$$
$$(abc)' = ((ab)c)' = (ab)' + c' = a' + b' + c'$$

Formula (3) presents DeMorgan's theorem in the general form: to complement function f with variables $x_1, \ldots, x_m$ and operators AND, OR and NOT, we must complement each variable (replace each x_p by x'_p and each x'_q by x_q) and replace each operator AND by OR and each operator OR by AND.

$$f'(x_1, \ldots, x_m, \&, +) = f(x_1', \ldots, x_m', +, \&) \qquad (3)$$

Examples:

$f_1 = x'y + w'z;$ $\qquad f_1' = (x + y')(w + z').$

$f_2 = x_1'(x_3' + x_4'x_5x_7 + x_7'(x_4 + x_5'x_6' + x_5x_6));$

$f_2' = x_1 + x_3(x_4 + x_5' + x_7')(x_7 + x_4'(x_5 + x_6)\ (x_5' + x_6')).$

1.1.4 Canonical forms. Let us look at the Table 6. This table describes Boolean function f with three variables a, b and c. In the column *Minterm,* we constructed *minterms* – the products containing all three variables, in the following way. If in the column *abc,* some variable is equal to zero, this variable is complemented in the corresponding product. Otherwise (variable is equal to one), this variable is written without inversion.

Table 6. Function *f* with three variables

a b c	Minterm	f	f'
0 0 0	a'b'c'	0	1
0 0 1	a'b'c	1	0
0 1 0	a'bc'	1	0
0 1 1	a'bc	0	1
1 0 0	ab'c'	1	0
1 0 1	ab'c	0	1
1 1 0	abc'	1	0
1 1 1	abc	1	0

$$f = a'b'c + a'bc' + ab'c' + abc' + abc \qquad (4)$$

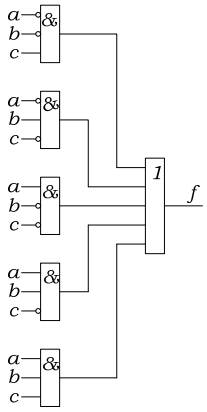


Figure 5. The circuit for canonical sum-of-products (4)

It is easy to show that we can immediately get the expression for Boolean function from its truth table. For this, we must write a sum of minterms written in the rows where the function f is equal to one (expression (4) under Table 6 in our example). We name such expression a *canonical sum-of-products*. The logic circuit corresponding to this expression is presented in Fig. 5.

The last column of Table 6 contains function f' – the inversion of function f. Let us construct the canonical sum-of-product for this function using the "ones" in the last column:

$$f' = a'b'c' + a'bc + ab'c.$$

Using De-Morgan's law, we can return to the initial function f presented as a product-of-sums where each sum contains all variables or their inversions:

$$f = (f')' = (a + b + c)(a + b' + c')(a' + b + c')$$

As above, for a sum-of-products, we can get this expression immediately from the truth table of function *f* (Table 7). For this, we write a product of maxterms

$$f = (a + b + c)(a + b' + c')(a' + b + c') \quad (5)$$

Table 7. The same function *f*

a b c	*Maxterm*	*f*
0 0 0	$a + b + c$	*0*
0 0 1	$a + b + c'$	*1*
0 1 0	$a + b' + c$	*1*
0 1 1	$a + b' + c'$	*0*
1 0 0	$a' + b + c$	*1*
1 0 1	$a' + b + c'$	*0*
1 1 0	$a' + b' + c$	*1*
1 1 1	$a' + b' + c'$	*1*

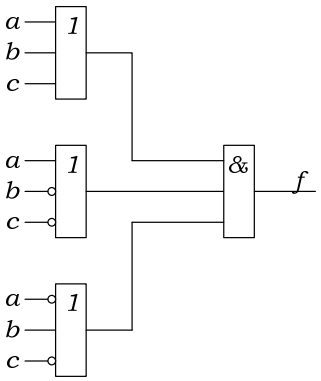


Figure 6. The circuit for a canonical product-of-sums

written in the rows where the function *f* is equal to zero (equation (5) in our example). In the column *Maxterm* in this table, we constructed the sums containing all three variables, in the following way. If in column *abc* some variable is equal to zero, this variable is not complemented in the corresponding sum. Otherwise (variable is equal to one) this variable is written with inversion. We name such expression a *canonical product-of-sums*. The logic circuit for this expression is presented in Fig. 6.

The costs of circuits in Fig. 5 and Fig. 6 are equal to 20 and 12 respectively. Of course, looking at these two circuits, you should not think that the circuit corresponding to the product-of-sums is always better than the circuit for the sum-of-products. First of all, these circuits are not minimized. Second, it is reasonable to begin from sum-of-products if the number of "ones" in the column for function value is less than the number of "zeroes", and vice versa.

1.1.5 Cover of Boolean function. Let us discuss one more example of the function *f* presented in Table 8. Really, we can at once construct its logic circuit from this truth table. For this, we should go along the column *f,* and each time when the function is equal to one in some row, we construct an AND-gate implementing a minterm for this row (there are six AND-gates in this circuit because Table 8 has six "ones" in the column *f).* Our last step is to construct a six-input OR-gate, each its input is connected with one of the outputs of AND-gates (see the logic circuit in Fig. 7).
In such a design we use only rows with '1' in the last column of Table 8, so we do not need columns with '0' for this procedure. Let a *cover* of function *f* be a set of input vectors where function *f* is equal to one. Fig. 8 contains an initial cover for function *f* from Table 8. We call it an *initial cover* because it was obtained directly from the truth table of this function. Of course, we can construct the circuit in Fig. 7 immediately from the cover in Fig. 8.

Table 8. Function *f* with four variables

x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	`1
1	1	1	0	0
1	1	1	1	1

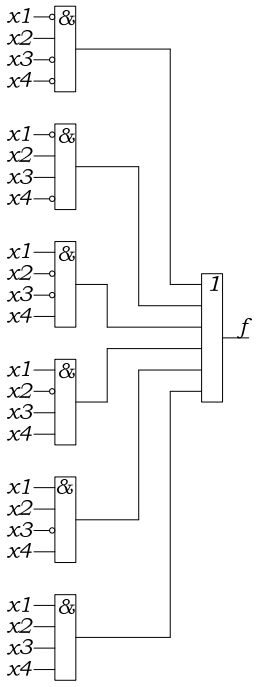


Figure 7. Circuit for the function in Table 8

$$f = \begin{array}{|cccc|} x_1 & x_2 & x_3 & x_4 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{array}$$

Figure 8. Initial cover for the function in Table 8

Now let us construct sum-of-products of function *f* from the initial cover in Fig. 8 and make some trivial transformations. We take common terms from pairs of minterms and use theorem 5 ($A + A' = 1$) and theorem1 ($A \cdot 1 = A$) from the list of main laws and theorems of Boolean algebra found in the beginning of this Chapter.

$$f = x'_1x_2x'_3x'_4 + x'_1x_2x_3\,x'_4 + x_1x'_2x'_3x_4 + x_1x'_2x_3x_4 + x_1x_2\,x'_3x_4 + x_1x_2x_3x_4 =$$
$$= x'_1x_2x'_4(x'_3 + x_3) + x_1x'_2x_4(x'_3 + x_3) + x_1x_2x_4(x'_3 + x_3) =$$
$$= x'_1x_2x'_4 + x_1x'_2x_4 + x_1x_2x_4 = x'_1x_2x'_4 + x_1x_4(x'_2 + x_2).$$

The result of these transformations is presented in expression (6) and the corresponding logic circuit – in Fig. 9. The cost of this circuit is equal to 7; this is much lower than the cost of the circuit in Fig. 7.

$$f = x'_1x_2x'_4 + x_1x_4. \qquad (6).$$

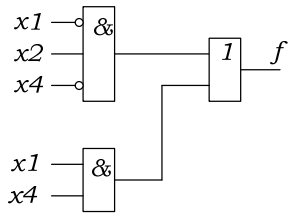


Figure 9. The circuit for expression (6)

Let us think, whether it is possible to construct the circuit in Fig. 9 without transformations of Boolean expressions. In the first two vectors (*0100* and *0110*) all components except one are equal. We can replace these two vectors by one (*01x0*) where symbol *x* takes the place of non-equal components in the initial vectors. We call *x* a *free component* and '*0*' or '*1*' – *bound components*. We can immediately return from vector *01x0* to the initial vectors replacing the free component with '*0*' and '*1*':

$$\left.\begin{matrix}0100\\0110\end{matrix}\right\}01x0$$

For the next four vectors in the cover we can make two steps:

$$\left.\begin{matrix}\left.\begin{matrix}1001\\1011\end{matrix}\right\}10x1\\\left.\begin{matrix}1101\\1111\end{matrix}\right\}11x1\end{matrix}\right\}1xx1$$

After the first step we get two vectors with one free component (*10x1* and *11x1*). These vectors also contain all equal components except the third bound component, and we replace two vectors by one vector *1xx1* with two free components. As before, it is easy to return from vector *1xx1* to the four initial vectors *1001, 1011, 1101* and *1111* replacing free components with all possible combinations of "zeroes" and "ones" (*00*, *01*, *10* and *11*).

Let us define a vector as an *m-cube* (or as having *rank m*), if this vector contains *m* free components. So, the cube *1xx1* is a 2-cube, vectors *01x0, 10x1* and *11x1* are 1-cubes and all vectors in the initial cover are 0-cubes. Since cubes *01x0* and *1xx1* present all 0-cubes in the initial cover we can construct a new cover for function *f* containing only these two cubes:

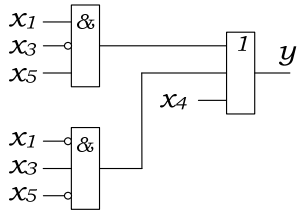
$$f = \left|\begin{matrix}x_1 & x_2 & x_3 & x_4\\0 & 1 & x & 0\\1 & x & x & 1\end{matrix}\right|$$

For this minimized cover we can get the minimized sum-of-product (6). Each product in this expression corresponds to one cube. Here we have two products: $x'_1x_2x'_4$ for cube *01x0* and x_1x_4 for cube *1xx1*. In such product, a variable is non-complemented if the corresponding bound component is equal to one. Otherwise, the variable is complemented. There are no variables in the product for free components.

Moreover, to build a circuit such as in Fig. 9 we do not need a Boolean expression. We must construct as many AND-gates as the number of cubes in the cover, and connect the outputs of these AND-gates to the inputs of OR-gate. In such a design there exists some exceptions – if a cube contains only one bound component (the second cube in cover *y*):

$$y = \left| \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \\ 1 & x & 0 & x & 1 \\ x & x & x & 1 & x \\ 0 & x & 1 & x & 0 \end{array} \right|$$

AND-gate for this cube is not constructed but OR-gate has input x_4 for this bound component:



From the examples above it is evident that to construct a minimal logic circuit we must get the cover with the minimal number of maximal cubes – cubes with the maximal number of free components. We did not show yet how to do it. It will be the topic of the next item.

1.2 Minimization with Karnaugh maps

1.2.1 Two variable Karnaugh maps. If we have two cubes of rank *m* (each contains *m* free components) with all equal components except for one bound component, we can combine these two cubes to get one cube with rank *m+1*. To increase the rank of cubes in such a way we will use *Karnaugh maps*. A Karnaugh map is a graphical representation of a truth table. In a Karnaugh map we can easily find cubes to be combined to get a cube with a higher rank. We will begin from the simplest Karnaugh maps with two variables.

A Karnaugh map with two variables is a square with four cells (Fig. 10). Rows in such Karnaugh map correspond to the first variable *a*, columns – to the second variable *b*. Rows and columns are denoted by zeros and "ones", and the cell at the intersection of row *0* and column *0* corresponds to vector *00* of the truth table, at the intersection of row *0* and column *1* – to vector *01*, etc.

a \ b	0	1
0	00	01
1	10	11

Figure 10. Correspondence between a Karnaugh map and a truth table

Let us discuss several examples. Function f_1 is presented by its cover in Fig. 11,a. To insert this function into the Karnaugh map we put "ones" in the cells for 0-cubes in the function cover (Fig. 11,b). The two cells with *ones* in one row or one column are *adjacent* and can be combined into one 1-cube. To construct such 1-cube we must check which variable has the same value in this cube (*a = 0* in our example) and which variable changes its value (*b* in our example: *b = 0* in cube *00* and *b = 1* in cube *01*). Then, in the 1-cube we write a bound component *0* for unchanging variable *a* and

a free component *x* for changing variable *b*. This cube *0x* is written in the minimized cover in Fig. 11,c. The similar example with two adjacent cells in the column is shown in Fig 12.

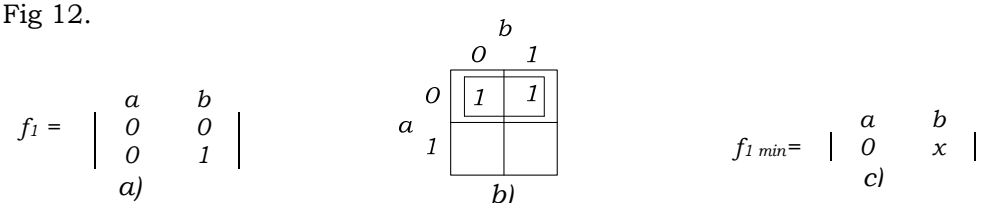


Figure 11. Function f_1: initial cover (a), Karnaugh map (b) and minimized cover (c)

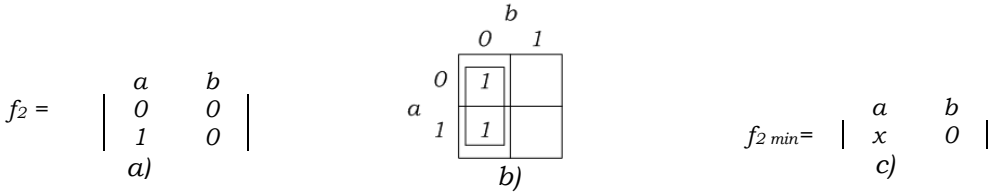


Figure 12. Function f_2: initial cover (a), Karnaugh map (b) and minimized cover (c)

In Fig. 13, two possible one cube have one common cell *00*. Since our goal to get maximal cubes, we cover this cell twice.

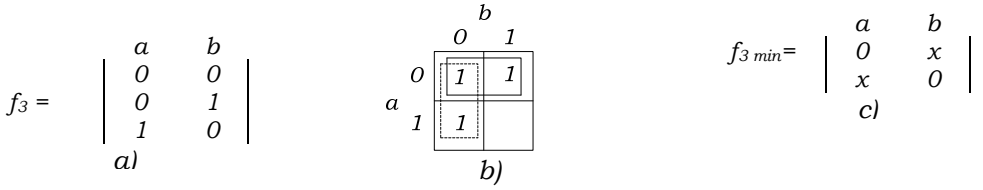


Figure 13. Function f_3: initial cover (a), Karnaugh map (b) and minimized cover (c)

In Fig. 14, two zero cubes are not adjacent and cannot be combined.

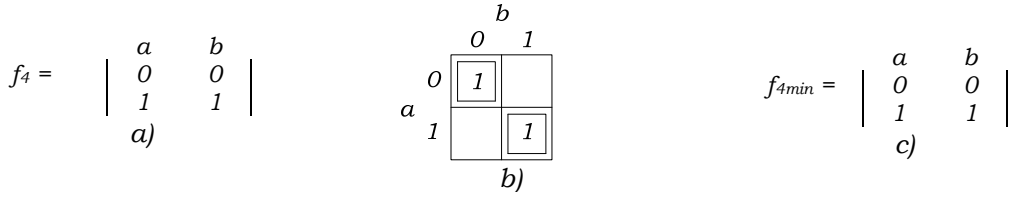


Figure 14. Function f_4: initial cover (a), Karnaugh map (b) and minimized cover (c)

In Fig. 15, four adjacent cells form 2-cube *xx*:

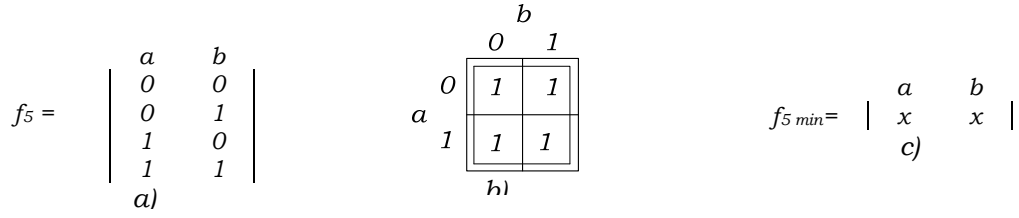


Figure 15. Function f_5: initial cover (a), Karnaugh map (b) and minimized cover (c)

1.2.2 Three variable Karnaugh maps. First two examples for three variable Karnaugh maps are rather simple. In both cases, four combined cells give 2-cubes. In

Fig. 16, variables *a* and *c* are changed but variable *b* is equal to *0* in all four of these cells (*b* is not changed). Thus, the final cube contains two free and one bound components. In Fig. 17, 2-cube contains two free and one bound components as well.

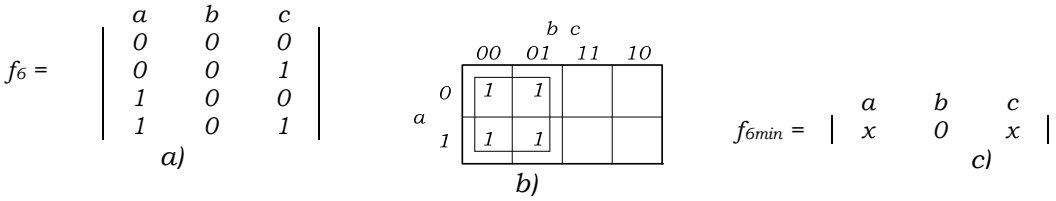


Figure 16. Function f_6: initial cover (a), Karnaugh map (b) and minimized cover (c)

Fig. 18 illustrates the only specific feature of Karnaugh maps with three variables – the "edge effect". It is clear from the cover of function f_8 (Fig. 18, a) that two 0-cubes *000* and *010* can be combined into one 1-cube *0x0*. To find such a combining in a Karnaugh map we should combine cells at the opposite edges of the map.

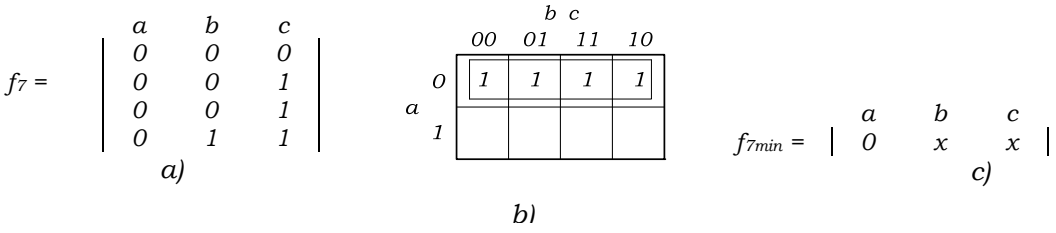


Figure 17. Function f_7: initial cover (a), Karnaugh map (b) and minimized cover (c)

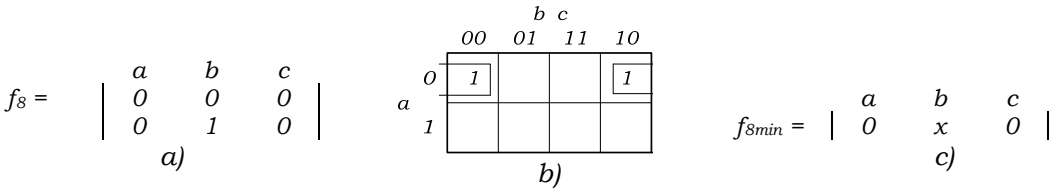


Figure 18. Function f_8: initial cover (a), Karnaugh map (b) and minimized cover (c)

A similar example with a combination of four cells is shown in Fig. 19. Fig. 20 illustrate closely related example with twice covered two cells.

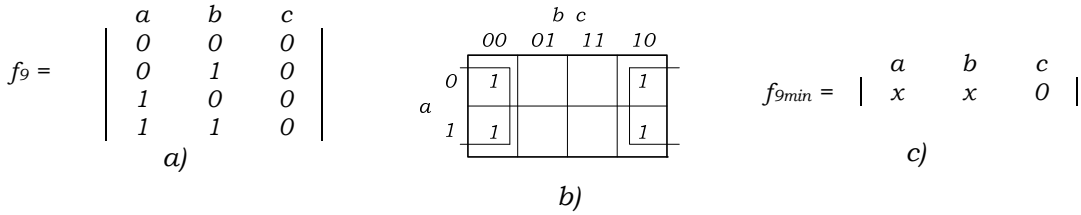


Figure 19. Function f_9: initial cover (a), Karnaugh map (b) and minimized cover (c)

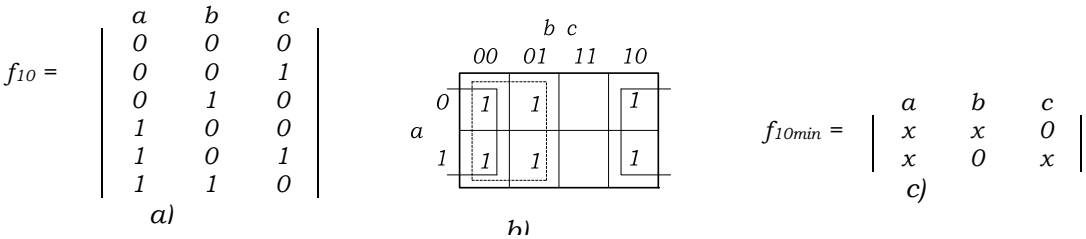


Figure 20. Function f_{10}: initial cover (a), Karnaugh map (b) and minimized cover (c)

1.2.3 Four variable Karnaugh maps. The specific feature of a Karnaugh map with four variables is shown in Fig. 21 – it is possible co combine four corner "ones" in one 2-cube. You can check it yourself if you combine "ones" in each column separately at the first step and then get the 2-cube at the second. Of course, we should do it in one step, checking which variables are changed (free components correspond to these variables) and which variables are not changed (bound components correspond to these variables). In our example we get cube x0x0 immediately from Fig. 21,b (variables *a* and *c* are changed and variables *b* and *d* are not changed).

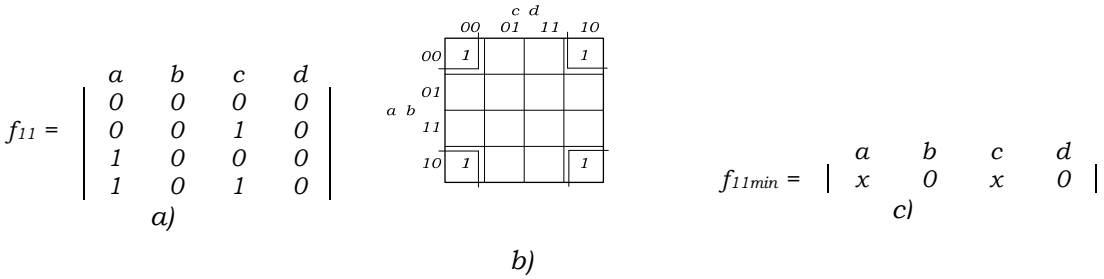


Figure 21. Function f_{11}: initial cover (a), Karnaugh map (b) and minimized cover (c)

In Fig. 22 we have one more example for a four variable Karnaugh map:

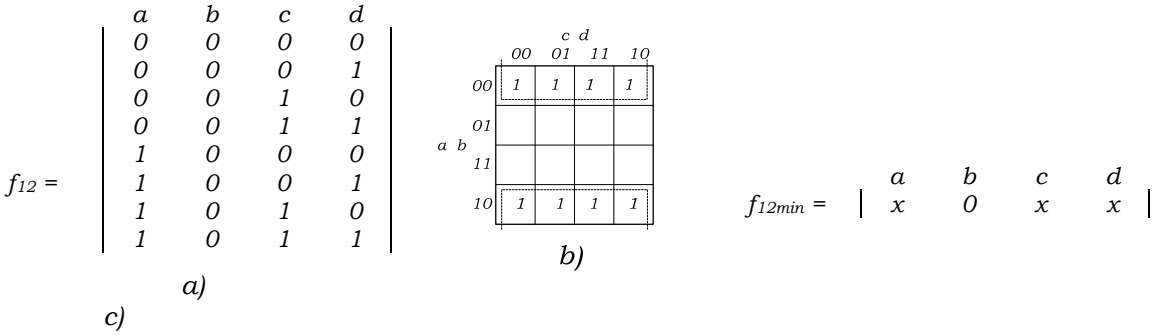


Figure 22. Function f_{12}: initial cover (a), Karnaugh map (b) and minimized cover (c)

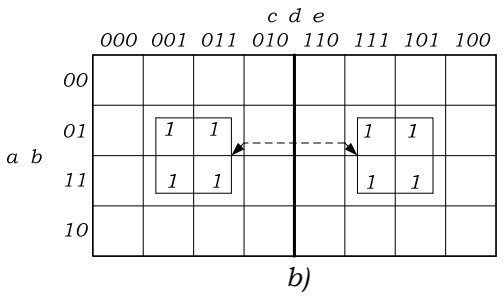
1.2.4 Five variable Karnaugh maps. A five variable Karnaugh map is formed from two four variable Karnaugh maps. The only specific feature of this map is the possibility to combine cells which are symmetric along the vertical line which divides this map into two four variable maps (along the "*main meridian*").

Let us turn to the Karnaugh map in Fig. 23,b. If we combine "four" ones at the left side of this map we will get 2-cube *x10x1*. The same goes for the right four "ones", which will give us 2-cube *x11x1,* and we can combine these two 2-cubes and get 3-cube *x1xx1*. To make this in one step we should find at once that these two figures are symmetric along the "main meridian". Thus, we can get 3-cube *x1xx1* immediately from Fig. 23,b (variables *a*, *c* and *d* are changed, variables *b* and *e* are not changed in the combined figure and both are equal to one).

f_{13} =

a	*b*	*c*	*d*	*e*
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

a)



b)

f_{13min} =

a	*b*	*c*	*d*	*e*
x	1	x	x	1

c)

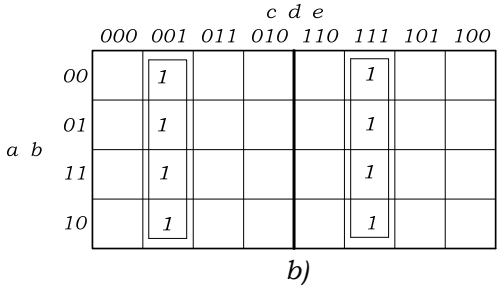
Figure 23. Function f_{13}: initial cover (a), Karnaugh map (b) and minimized cover (c)

In the example in Fig. 24,b two figures on the left and on the right are not symmetric so they are not combinable. In our further examples of five variables maps we will discuss the same topic – how to find out the symmetry along the "main meridian" and at once combine two distant figures in a Karnaugh map into one cube with a higher rank.

f_{14} =

a	*b*	*c*	*d*	*e*
0	0	0	0	1
0	0	1	1	1
0	1	0	0	1
0	1	1	1	1
1	1	0	0	1
1	1	1	1	1
1	0	0	0	1
1	0	1	1	1

a)



b)

f_{14min} =

a	*b*	*c*	*d*	*e*
x	x	0	0	1
x	x	1	1	1

c)

Figure 24. Function f_{14}: initial cover (a), Karnaugh map (b) and minimized cover (c)

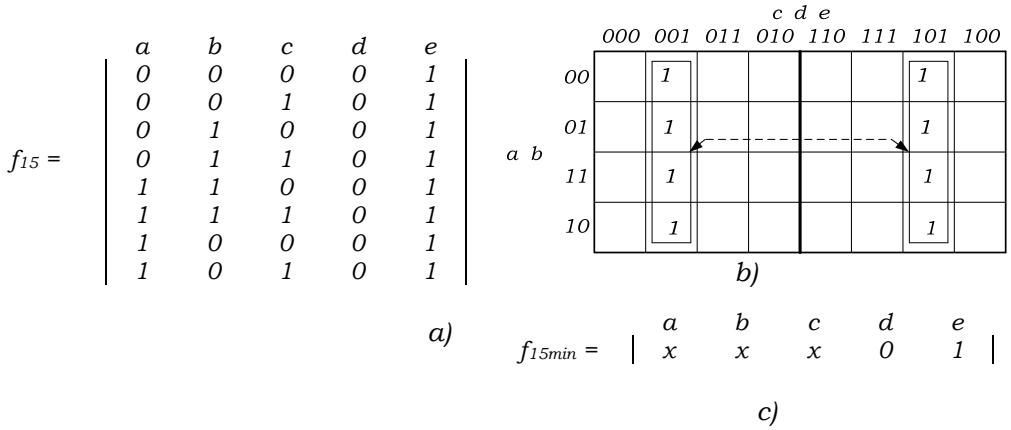


Figure 25. Function f_{15}: initial cover (a), Karnaugh map (b) and minimized cover (c)

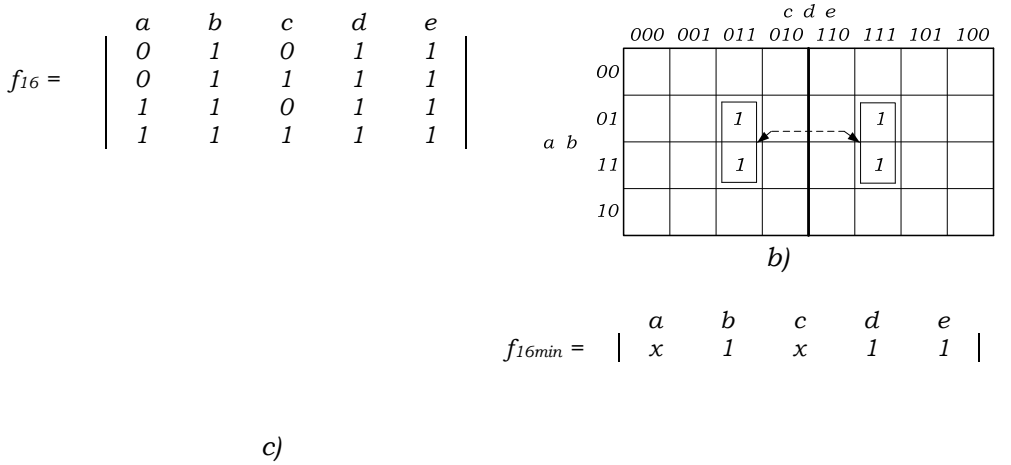
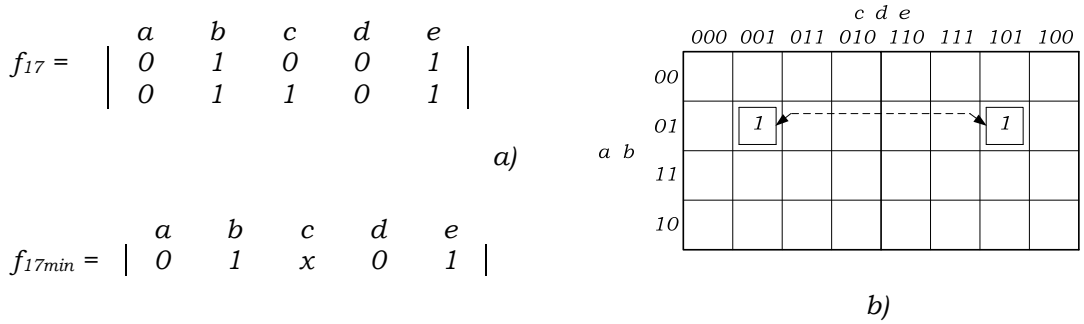


Figure 26. Function f_{16}: initial cover (a), Karnaugh map (b) and minimized cover (c)



c)

Figure 27. Function f_{17}: initial cover (a), Karnaugh map (b) and minimized cover (c)

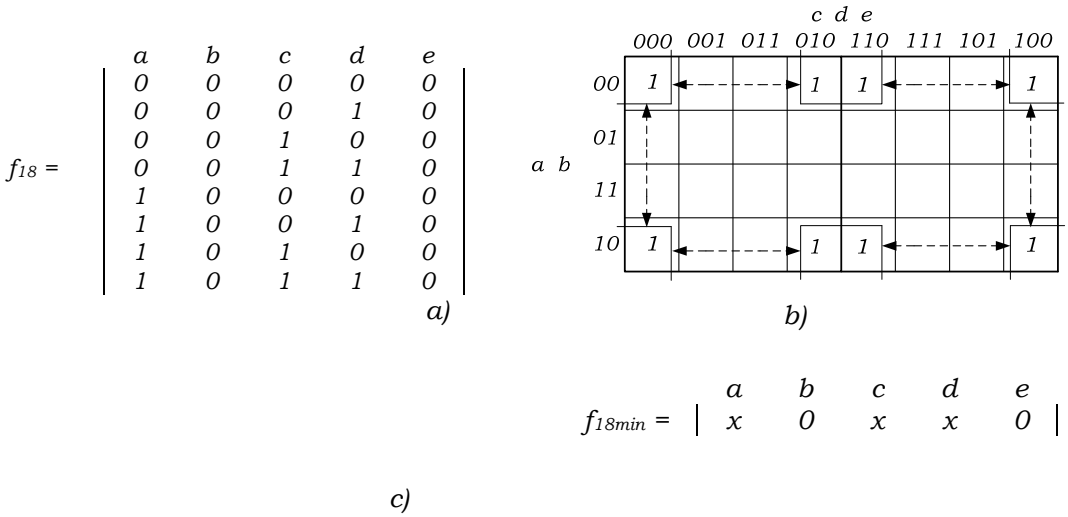


Figure 28. Function f_{18}: initial cover (a), Karnaugh map (b) and minimized cover (c)

1.2.5 Karnaugh maps with don't care. To explain what *don't care* is, let us suppose that the circuit at Fig. 29 is used in decimal arithmetics, i.e. only the vectors corresponding to the decimal digits from zero to nine can appear at its input. Thus, the vectors from *1010* to *1111* will never come to the input of this circuit. This means that this function is not defined at these vectors and we use term *"don't care"* to name this set of inputs. We can define the value of function *y* as one or zero at don't care and use it to enlarge cubes in the minimized cover. The symbol Ø will be used to mark the cell in a Karnaugh map where a function is not defined.

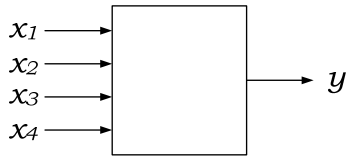


Figure 29. Circuit in decimal arithmetic

Two rules should be used in minimization with don't care:

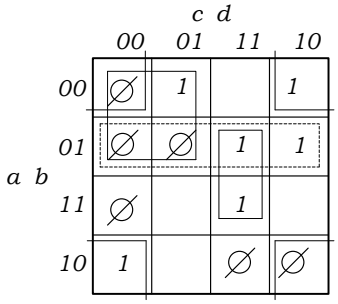
1. We must cover only "ones" in Karnaugh maps, not don't cares Ø;
2. We can use the cell with Ø as the cell with '1' if we construct a larger cube in the minimized cover.

Let us illustrate this by the example of the cover in Fig. 30,a. In this cover, the function f_{19} is equal to one at the vectors written over the dotted line, and it is not defined (domain of don't care) at the vectors under the dotted line. The covering process is shown in Fig. 30,b. As we can see from this Karnaugh map, only cells with don't care, which help us to enlarge the cubes in the final cover, were used to get the minimized cover (Fig. 30,c).

f_{19} =

a	b	c	d
0	0	0	1
0	0	1	0
0	1	1	0
0	1	1	1
1	0	0	0
1	1	1	1
0	0	0	0
0	1	0	0
0	1	0	1
1	0	1	0
1	0	1	1
1	1	0	0

a)



b)

f_{19min} =

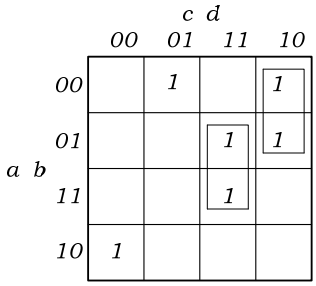
a	b	c	d
0	x	0	x
0	1	x	x
x	0	x	0
x	1	1	1

c)

Figure 30. Function f_{19}: initial cover (a), Karnaugh map (b) and minimized cover (c)

In Fig. 31 we minimized the same function without taking don't care into consideration (function f_{20}). Logic circuits for minimized functions f_{19min} and f_{20min} are presented in Fig. 32. The cost of the first of them is equal to 13 (the total number of inputs in gates), the cost of the second – 18. We will return to the using of don't care in Chapter 3 where we will show how it can help us to minimize logic circuits of simple Finite State Machines (FSM).

Now we will show how to shorten the specification of a Boolean function. Let us return to the Boolean function f with four variables $x_1, \ldots, x_4$. Its truth table was presented in Table 8, we illustrated the minimization of this function in Fig. 9.



f_{20min} =

a	b	c	d
0	x	1	0
x	1	1	1

0	0	0	1
1	0	0	0

a)

b)

Figure 31. Function f_{20}: Karnaugh map (a) and minimized cover (b)

We repeated this truth table in Table 9 and added one (the first) column to this table. In this column we wrote decimal numbers from *0* to *15* corresponding to the binary vectors of variables $x_1, \ldots, x_4$ in each row. It is clear that we can define this function writing down the sequence of decimal numbers corresponding to the vectors where Boolean function is equal to one (these vectors should be in the cover of this function):

$$f(x_1, x_2, x_3, x_4) = \Sigma(4, 6, 9, 11, 13, 15).$$

Here Σ means that we used sum-of-products for our function. We will use such function description in many our examples.

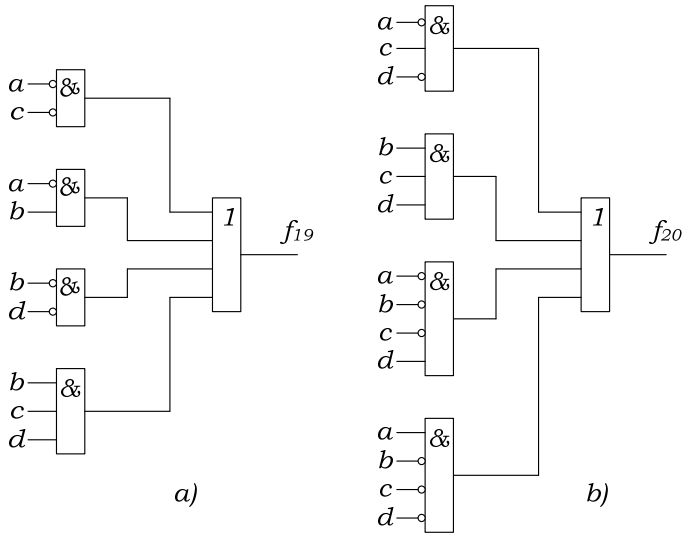


Figure 32. Logic circuits after minimization with don't care (a) and without it (b)

Table 9. Truth table of function *f*

#	x_1	x_2	x_3	x_4	f_{12}
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	1

7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	0
13	1	1	0	1	`1
14	1	1	1	0	0
15	1	1	1	1	1

$f =$

x_1	x_2	x_3	x_4
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Figure 33. Function *f – initial cover*

1.2.6 Examples. In this section we will discuss several examples of Boolean function minimization using Karnaugh maps.

Example 1. The function in this example

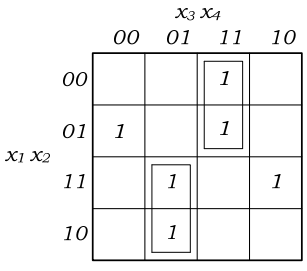
$$f_{21}(x_1, x_2, x_3, x_4) = \Sigma(3, 4, 7, 9, 13, 14)$$

is defined as a list of decimal numbers. We begin with the initial cover (Fig. 34,a) writing vectors (0-cubes) for each decimal number in the function representation. The next steps of minimization are the same as above.

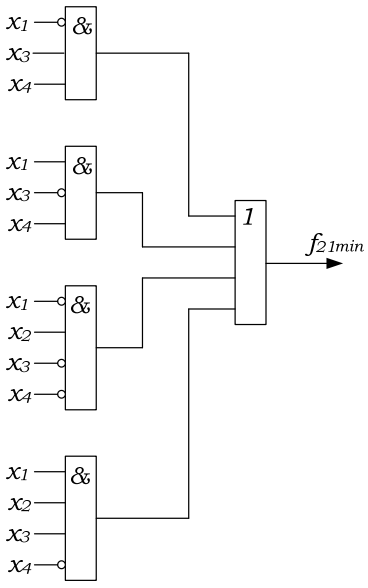
$f_{21} =$

x_1	x_2	x_3	x_4
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	1	0	1
1	1	1	0

a)



b)



d)

x_1	x_2	x_3	x_4
0	x	1	1

f_{21min} =

1	x	0	1
0	1	0	0
1	1	1	0

c)

Figure 34. Function f_{21}: initial cover (a), Karnaugh map (b), minimized cover (c) and minimized logic circuit (d)

Example 2. The function in this example

$$f_{22}(a, b, c) = a'b + bc' + b'c'$$

has three input variables and is defined as a Boolean expression. To construct the initial cover (Fig. 35,a) we must present each product in this expression as a corresponding cube (1-cube for each product in our example because products contain two variables). Then we insert these cubes into Karnaugh map and find the minimized cover (Fig. 35,c). The logic circuit implementing this function is shown in (Fig. 35,d).

f_{22} =

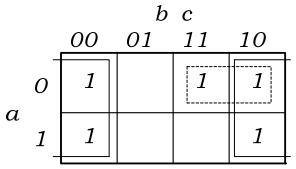
a	b	c
0	1	x
x	1	0
x	0	0

a)

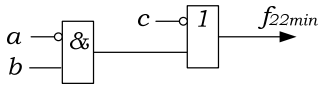
f_{22min} =

a	b	c
x	x	0
0	1	x

b)



c)



d)

Figure 35. Function f_{22}: initial cover (a), Karnaugh map (b), minimized cover (c) and minimized logic circuit (d)

Example 3. The function in this example is also presented as Boolean expression:

$$f_{23} = w'z + xz + x'y + wx'z$$

As above, we can present each product in this expression as a corresponding cube in the initial cover (Fig. 36,a). Here we have three 2-cubes and one 1-cube. If, for some reason, it is difficult to insert this cover in Karnaugh map immediately, it is possible to construct an initial cover f^0_{23} with 0-cubes only. For this, we must replace free components in each cube in the previous cover by all possible combinations of zeroes and ones (Fig. 36,b). Repeating 0-cubes (grey color in this figure) can be deleted. Karnaugh map, minimized cover and minimized logic circuit are presented in Fig. 36, c – e.

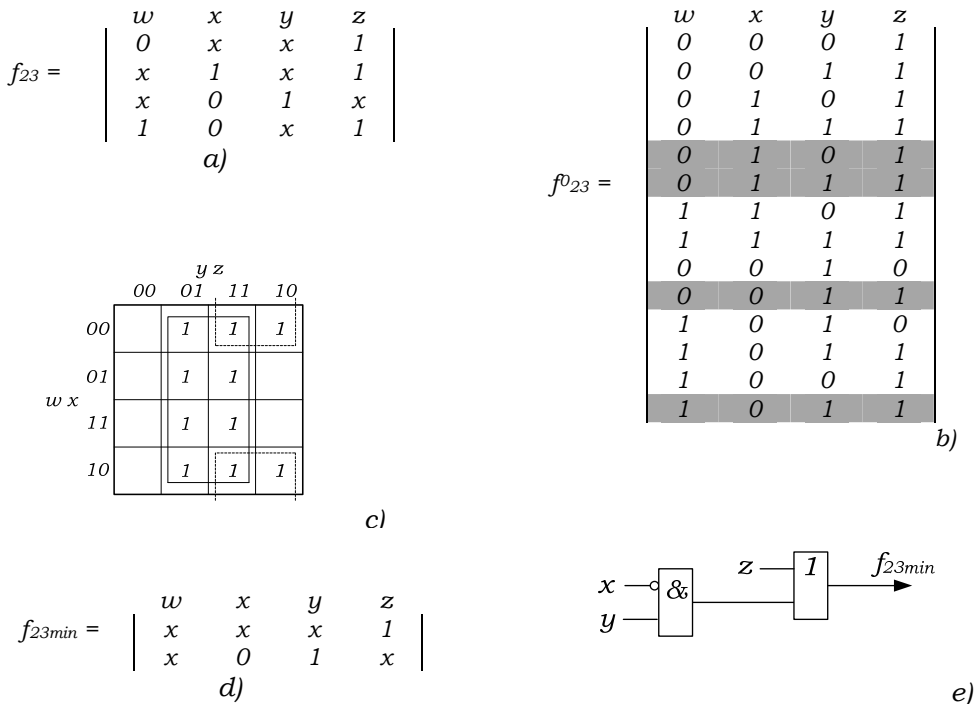
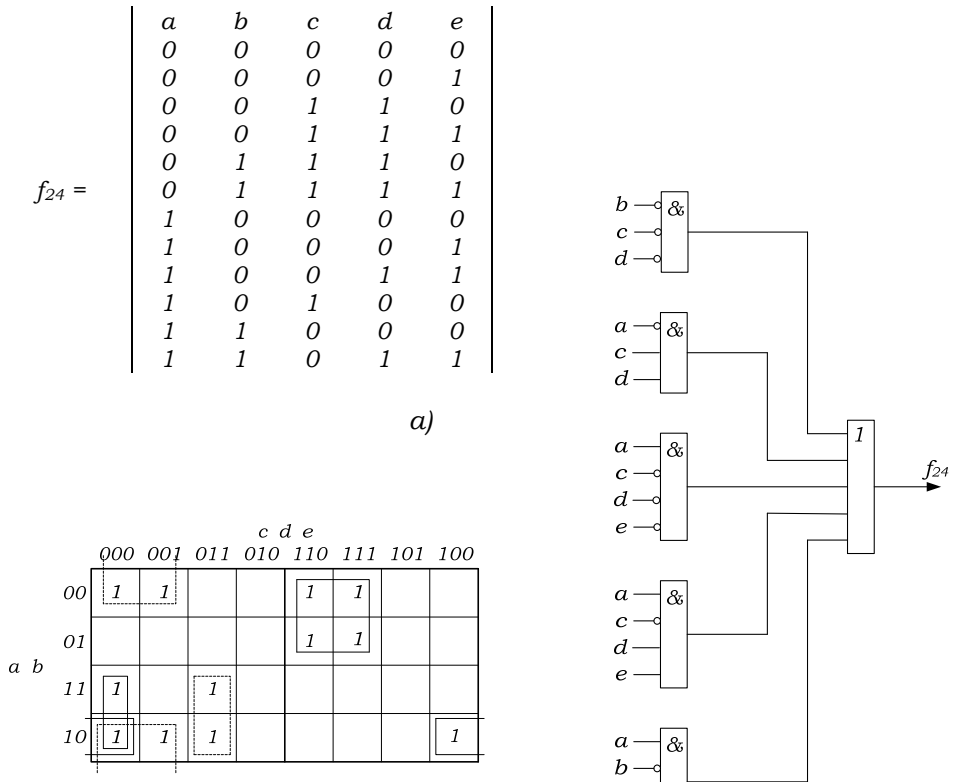


Figure 36. Function f_{23}: initial cover (a), initial cover with 0-cubes (b), Karnaugh map (c), minimized cover (d) and minimized logic circuit (e)

Example 4. The function with five variables in this example

$$f_{24}(a, b, c, d, e) = \Sigma(0, 1, 6, 7, 14 - 17, 19, 20, 24, 27).$$

is defined as a list of decimal numbers. The sequence of operations to minimize a logic circuit for this function is the same as in *Example 1*.



f_{24min} =

a	b	c	d	e
x	0	0	0	x
0	x	1	1	x
1	x	0	0	0
1	x	0	1	1
1	0	x	0	0

c)

Figure 37. Function f_{24}: initial cover (a), Karnaugh map (b), minimized cover (c) and minimized logic circuit (d)

Example 5. The function with five variables and don't care (Fig. 38)

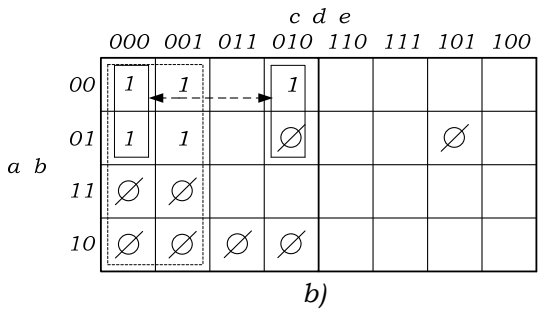
$$f_{25}(a,b,c,d,e) = \Sigma(0, 1, 2, 8, 9,[10, 13, 16 – 19, 24, 25])$$

is defined as a list of decimal numbers. The decimal number, corresponding to vectors in don't care are enclosed into square brackets. The "care" and "don't care" cubes in the initial cover are separated by the dotted line.

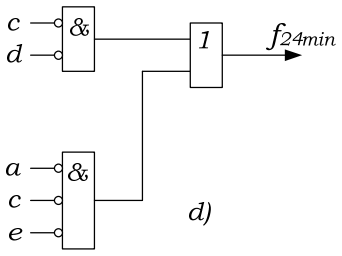
f_{25} =

a	b	c	d	e
0	0	0	0	0
0	0	0	0	1
0	0	0	1	0
0	1	0	0	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
1	0	0	0	0
1	0	0	0	1
1	0	0	1	0
1	0	0	1	1
1	1	0	0	0
1	1	0	0	1

a)



b)



d)

f_{25min} =

a	b	c	d	e
x	x	0	0	x
0	x	0	x	0

c)

Figure 38. Function f_{25}: initial cover (a), Karnaugh map (b), minimized cover (c) and minimized logic circuit (d)

1.3 Logic circuits with NORs and NANDs

In this section we will discuss the synthesis of logic circuits with *NOR* and *NAND* gates. Such gates are more popular in logic synthesis. Moreover, it is possible to say that most VLSI (Very Large Scale Integrated) circuits are constructed from these gates. However, people are used to think on the basis *AND-OR-NOT*, it is impossible to think on the basis *NOR-NAND*. The simplest methods to construct logic circuits from a truth table, a Karnough map or a cubical cover give us an expression in sum-of-products or product-of-sums form, which can be implemented as a two-level circuit. All the known methods for minimization of logic circuits use circuits on the *AND-OR-NOT* basis and produce results on the same basis. Only after this the special mapping algorithms are used to cover circuit by librarian elements, *NOR* and *NAND* gates as well.

We will discuss very simple algorithms for transformation of any multilevel circuit into the circuit with NOR gates, with *NAND* gates and with *NOR* and *NAND* gates together. Let us start with the circuits with *NOR* gates.

1.3.1 Circuits with NOR gates. Table 10 presents the truth table for function *NOR*. This function is equal to one only when both variables are equal to zero (the first row), otherwise it is equal to zero. The logic gate *NOR* is shown in Fig. 39.

Table 10. Function *NOR*

x_1	x_2	$x_1 + x_2$	$(x_1 + x_2)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

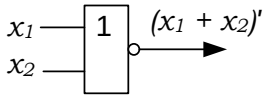


Figure 39. Gate NOR

Implementation of functions *OR* and *AND* with *NOR* gates is shown in Fig. 40. Thus, to realize OR-function $f = x_1 + x_2$ with *NOR* gates we must use the same inputs x_1, x_2 as inputs for *NOR* gate and invert its output (mnemonics: *Cover – Invert*). To realize AND-function $f = x_1x_2$ with *NOR* gates we must use the inverted inputs x_1, x_2 for *NOR* gate (mnemonics: *Invert – Cover*).

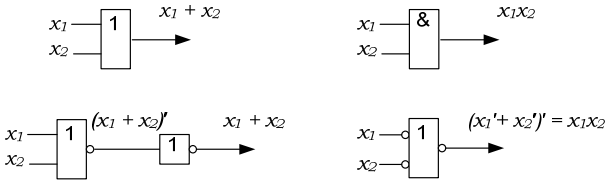


Figure 40. Implementation of *OR* and *AND* with *NOR* gates

As the first example, we will discuss mapping of a logic circuit in Fig. 41 with *NOR* gates. Here we will use gate by gate transformation. Thus, gate OR_1 in this figure is replaced by gates *NOR* and INV_1 in Fig. 42 (*Cover – Invert)*. Gate AND_2 in Fig. 41 is replaced by one gate NOR_2 in Fig. 42 (*Invert – Cover*) etc. In the circuit thus

constructed, two sequential inverters may be found (such cases are dotted in Fig. 42). The final step consists of deleting such pairs of inverters (Fig. 43).

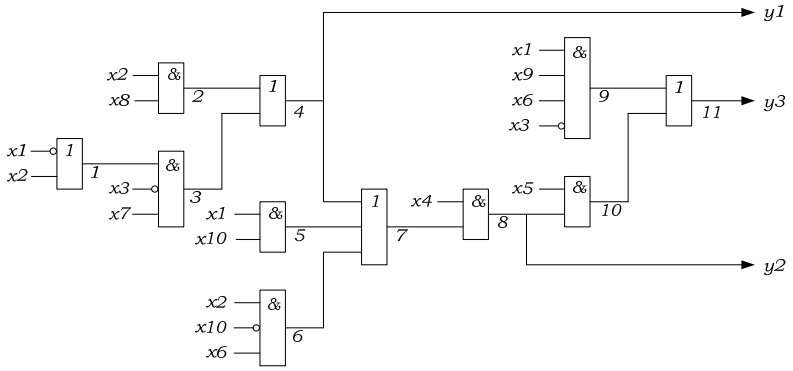


Figure 41. Example 1 with AND and OR gates

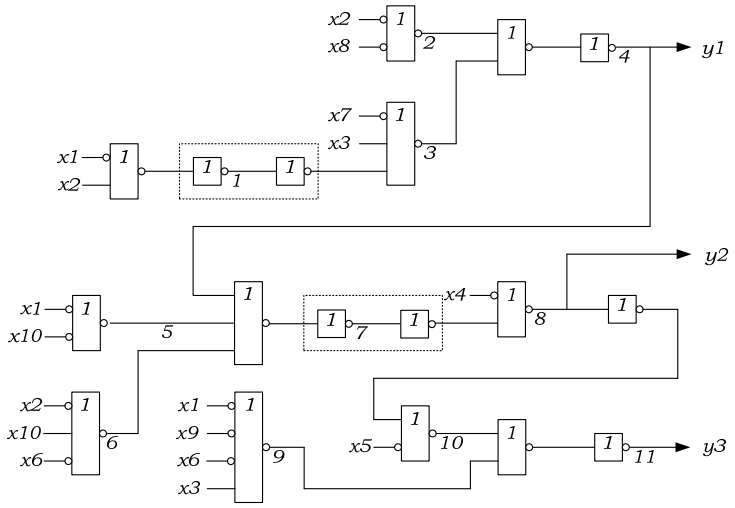


Figure 42. *Gate by gate* mapping of the circuit in Fig. 41 with *NOR* gates

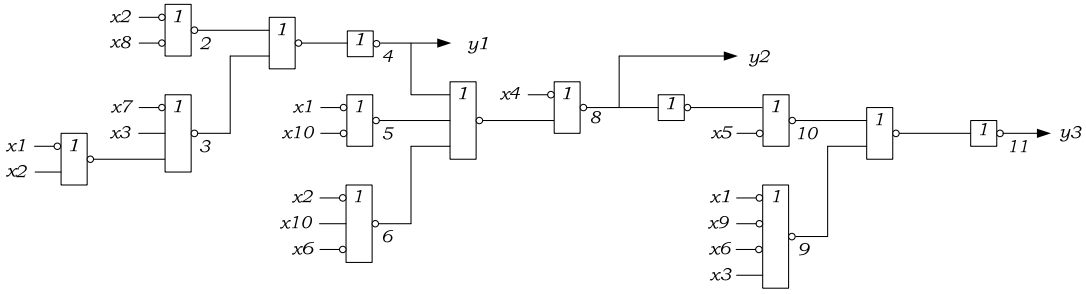


Figure 43. Final step of mapping with NOR gates

1.3.2 Circuits with NAND gates. Table 11 presents the truth table for function *NAND*. This function is equal to zero only when both variables are equal to one (the last row), otherwise it is equal to one. The logic gate *NAND* is shown in Fig. 44.

Implementation of functions *AND* and *OR* with *NAND* gates is evident from Fig. 45. Thus, to realize AND-function $f = x_1x_2$ with *NAND* gates we must use the same inputs x_1, x_2 as inputs for *NAND* gate and invert its output (mnemonics: *Cover – Invert*). To realize OR-function $f = x_1 + x_2$ with *NAND* gates we must use the inverted inputs x_1, x_2 for *NAND* gate (mnemonics: *Invert – Cover*).

Table 11. Function *NAND*

x_1	x_2	x_1x_2	$(x_1x_2)'$
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

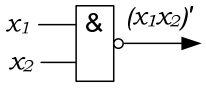


Figure 44. Gate NAND

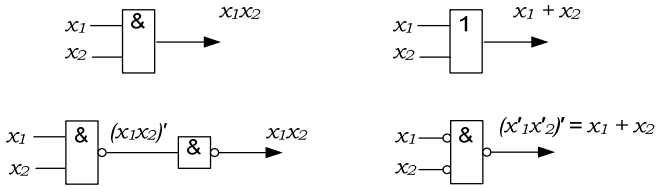


Figure 45. Implementation of AND and OR with gates NAND

As an example we will discuss the mapping of the same logic circuit (Fig. 41) with *NAND* gates. Once again, here we use a gate by gate transformation. Thus, gate OR_1 in this figure is replaced by gate $NAND_1$ in Fig. 46 (*Invert – Cover).* Gate AND_2 in Fig. 41 is replaced by two gates *NAND and* INV_2 in Fig. 46 (*Cover – Invert*) etc. As above, in the circuit thus constructed, two sequential inverters may be found (such cases are dotted in Fig. 46). The final step consists of deleting such pairs of inverters (Fig. 47).

We can generalize the rule for mapping of *AND-OR* circuits by *NOR (NAND)* gates. Let us say that gates AND and NAND (OR and NOR) are consonant (with similarly articulated gates in both occurences). At the same time, gates AND and NOR (OR and NAND) are not consonant with. Then, in a gate by gate transformation, the *consonant gates* are replaced by the rule *Cover – Invert,* the *non-consonant gates* are replaced according with the rule *Invert – Cover.*

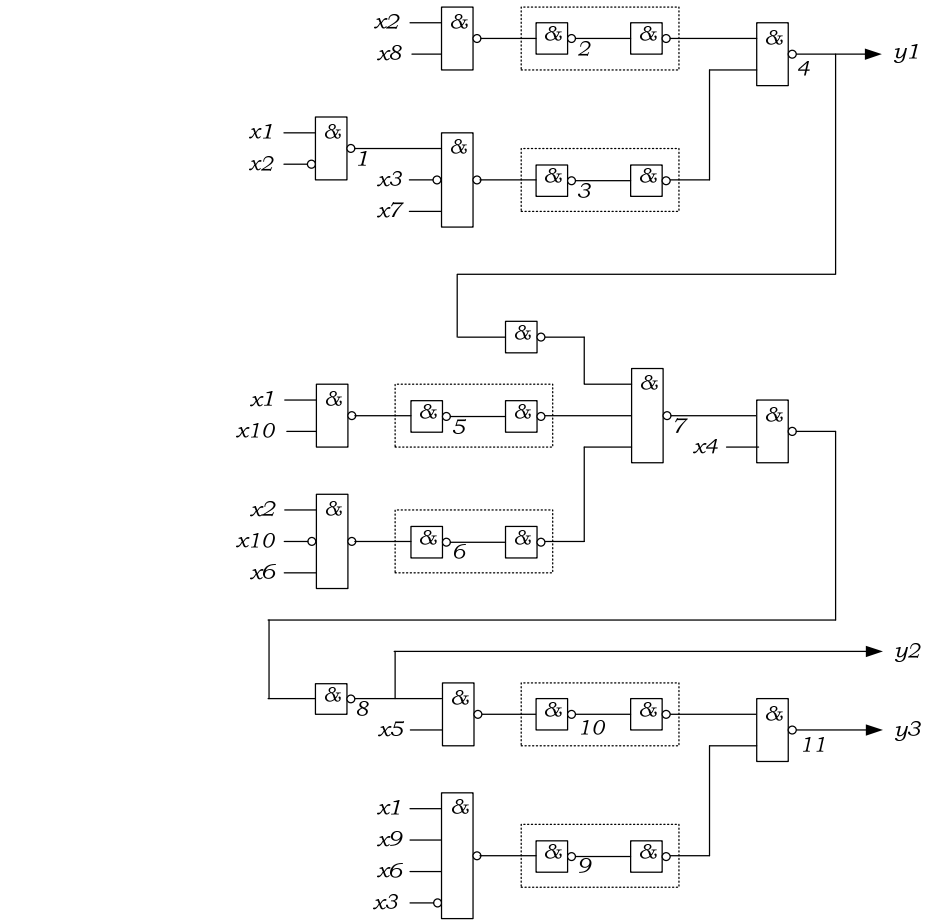


Figure 46. Gate by gate mapping of the circuit in Fig. 46 with *NAND* gates

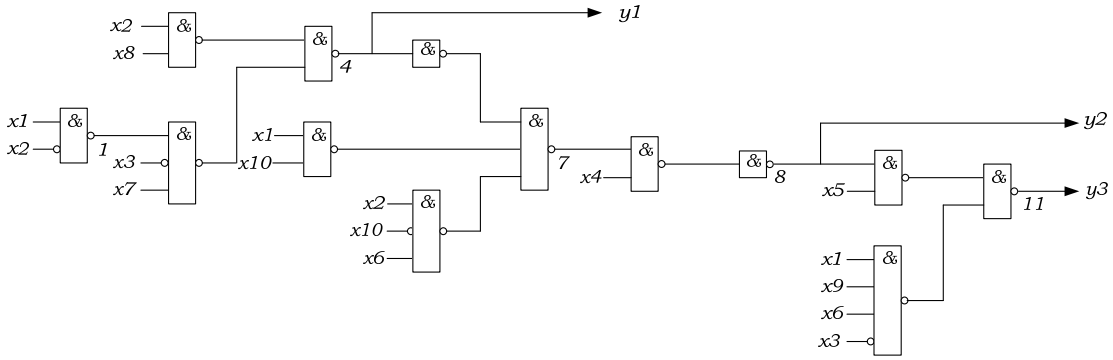


Figure 47. Example 1 with NAND gates

1.3.3 Circuits with NOR and NAND gates. Before we define the rules for transformation of any logic circuit into the circuit with NOR and NAND gates, let us first conduct some experiments. In Fig. 48,a we have four copies of the same two-level circuit implementing function f_1. In these circuits we numbered gates by all possible combinations *0-0, 0-1, 1-0* and *1-1* of zeroes and ones. Then, in Fig. 48,b we have implemented these circuits with NOR and NAND gates according the rules in Fig. 40

and Fig. 45. To decide what gate (NOR or NAND) should cover the gates OR and AND on the left side of this figure, we have used the following very simple rules:

1. We realized AND gate by NAND gate (consonant gates: NAND for AND) if the AND gate is marked by 1;
2. We realized OR gate by NOR gate (consonant gates: NOR for OR) if the OR gate is marked by 1;
3. We realized AND gate by NOR gate (non-consonant gates: NOR for AND) if the AND gate is marked by 0;
4. We realized OR gate by NAND gate (non-consonant gates: NAND for OR) if the OR gate is marked by 0.

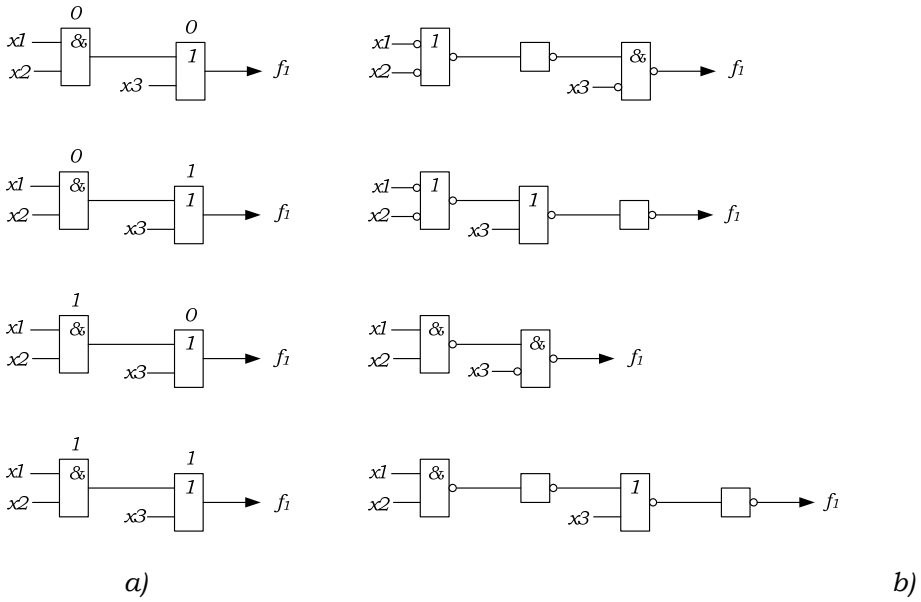


Figure 48. Four implementation of the same circuit with NOR – NAND gates

It follows immediately from Fig. 48 that:

1. If two gates in the sequence are marked by different numbers (*0-1* in the second circuit and *1-0* in the third circuit) there is no inverter between NOR and NAND gates;
2. If two gates in the sequence are marked by the same numbers (*0-0* in the first circuit and *1-1* in the last one) there is an inverter between NOR and NAND gates;
3. If the right (last gate) in the AND-OR circuit is marked by one (the second and the fourth circuits in our example) there is an inverter at the output of the NOR-NAND circuit.

If you repeat our experiment for three other possible combinations of two-level circuits (Fig. 49) you will get the same results.

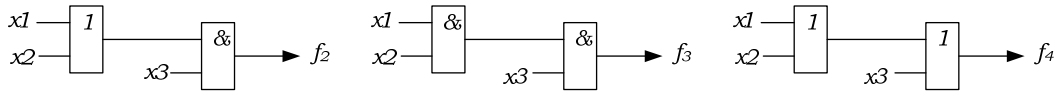


Figure 49. Three other possible two-level circuits

The rules for transformation of any logic circuits with AND-OR gates into the circuit with NAND-NOR gates can be briefly formulated in the following way:

Step1 – Marking. At this step we go from right to left in the OR-AND circuit and mark each gate by *1* or *0*, minimizing the number of cases in which two connected gates are marked by the same number. If we consider the circuit as a graph with gates as its vertices, it would be equivalent to the coloring of this graph by two colors (*0* and *1*) with minimization of the number of failures in such coloring – with minimization of the number of cases in which two connected vertices are colored by the same colors. One of the possible markings for the circuit in Fig. 41 is presented in Fig. 50.

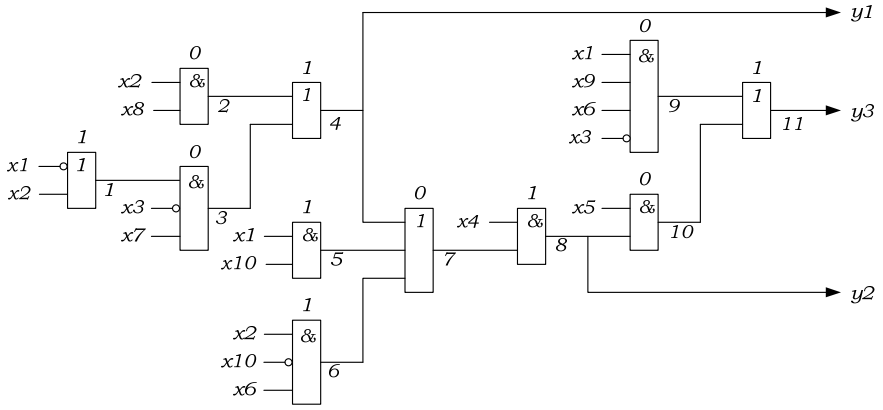


Figure 50. Example 1 with *AND* and *OR* gates – the first version of marking

Step2 – Mapping. At this step we go from left to right. Each gate marked by *1,* should be replaced with the consonant gate (OR by NOR, AND by NAND) in accordance with the rule *Cover – Invert*. Each gate marked by *0,* should be replaced with the non-consonant gate (OR by NAND, AND by NOR) in accordance with the rule *Invert – Cover*. In such a mapping, inverters can appear only between gates marked by the same numbers (*0-0* or *1-1*). The circuit in Fig. 51 with NOR and NAMD gates is the result of mapping of the circuit with AND and OR gates in Fig. 50.

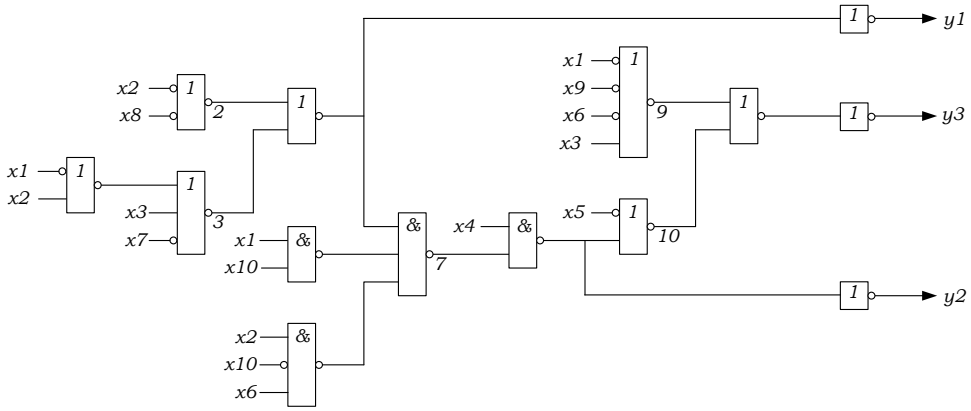


Figure 51. Example 1 with *NOR* and *NAND* gates (the first version)

Since we do not have violations in the marking of the circuit in Fig. 50, there are no inverters between gates in Fig. 51. However, we have three inverters at the outputs because y_1, y_2 and y_3 are the outputs of gates marked by *1* (do you remember the rule *Cover – Invert*?).

In Fig. 52, we used the marking beginning from *0* for the same circuit with AND – OR gates. The corresponding logic circuit with NAND and NOR gates is shown in Fig. 53.

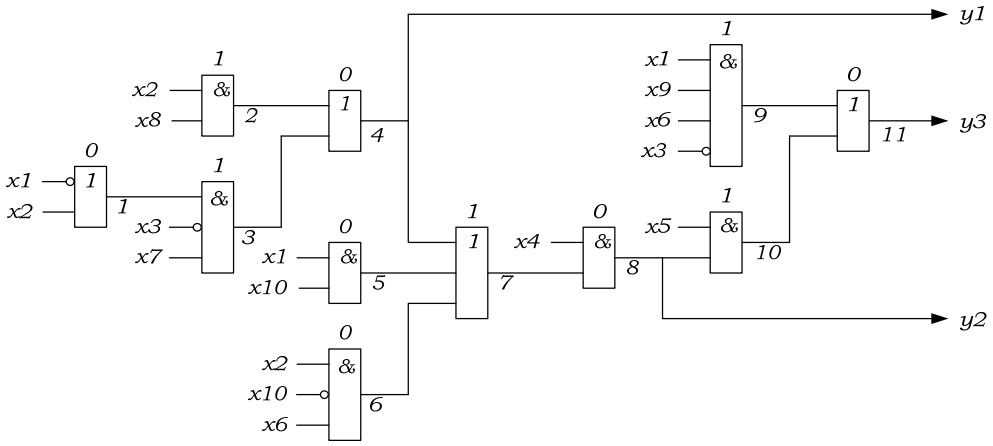


Figure 52. Example 1 with *AND* and *OR* gates – the second version of marking

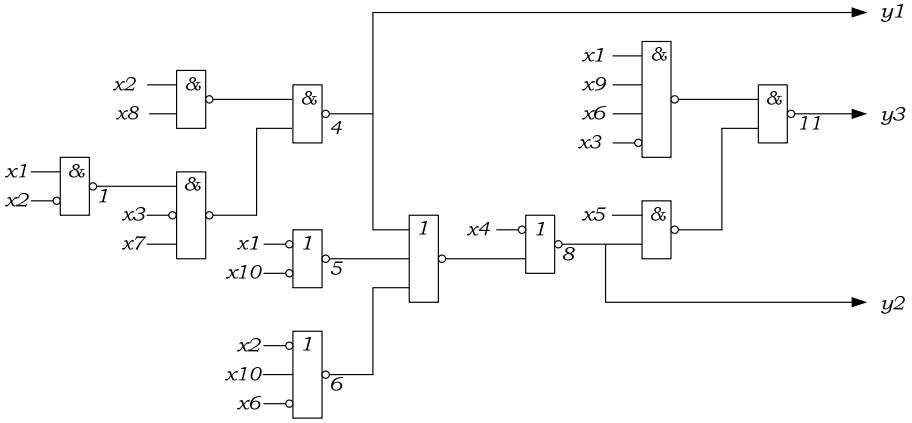
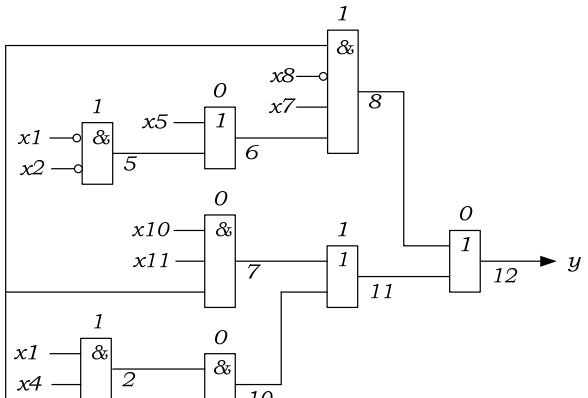


Figure 53. Example 1 with NOR and NAND gates (the second version)

In the circuit in Fig. 54 (Example 2), we could not avoid a violation in marking (two connected gates AND_4 andAND_7 are marked by 0) so we have an inverter in the circuit with NOR and NAND gates in Fig. 55.



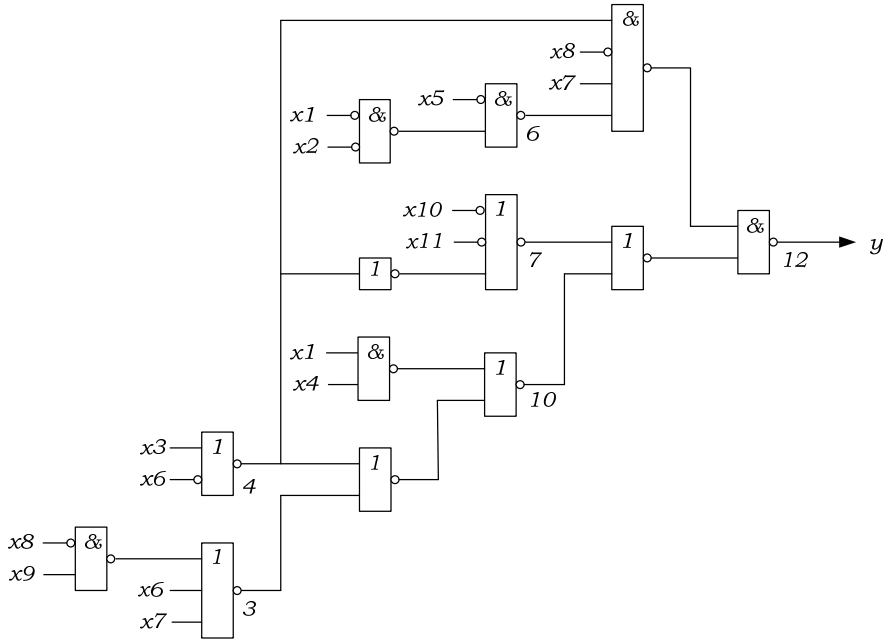


Figure 55. Example 2 with NOR and NAND gates

Chapter 2 Abstract Automata

In the first Chapter, we presented the elements of Boolean algebra as a tool for description of *combinational circuits*. In such circuits, the output at a cirtain time depends only on the inputs at the same time and does not depend on what was at the inputs of these circuits at the previous time.

This chapter deals with models for description of *sequential circuits* whose behavior depends not only on their present inputs, but, generally, on a prehistory, including past inputs. In the first section, we will introduce such a model – abstract automaton or a finite state machine (FSM) and will discuss the main automaton models – *Mealy, Moore* and their combined model. Then we will talk about the transformations between Mealy and Moore models and their minimization.

2.1 Behavior of abstract automaton

Let us consider the unit with one input and one output (Fig. 1) working at discrete times $t = 0, 1, 2 \ldots$ We suppose that at each point in time t input z_f (a letter of input alphabet $Z = \{z_1, \ldots, z_F\}$) appears at the unique input of the unit.

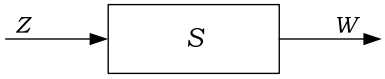


Figure 1. Unit S

Output symbol w_g (a letter of output alphabet $W = \{w_1, \ldots, w_G\}$) appears as a response to the input z_f at the same point in time t at the unique output of the unit. Assume, for example, that $Z = \{z_1, z_2, z_3\}$ and $W = \{ w_1, w_2, w_3, w_4\}$ and the output word $\omega(t) = w_1w_3w_4w_2w_1w_3$ is the reply to the input word $\xi(t) = z_2z_1z_1z_3z_2z_2$ (Fig. 2).

t	0	1	2	3	4	5
$\xi(t)$	z_2	z_1	z_1	z_3	z_2	z_2
$\omega(t)$	w_1	w_3	w_4	w_2	w_1	w_3

Figure 2. Input-output sequence for unit S

As seen from Fig. 2, the responses of unit S to the same inputs are sometimes different. Really, at times *1* and *2* we have the same input z_1 but different outputs w_3 and w_4, at times *0* and *5* we have the same input z_2 but different outputs w_1 and w_3 etc. Thus at any time *t*, the signal at the output of our unit depends not only on the input signal but also on the prehistory; i.e., it depends on the input sequence applied to the unit before this time *t*. Hence, unit S is not a combinational but a sequential circuit and we can not use Boolean algebra to describe its behavior; we need another model for such a behavior. For this, we will introduce *abstract automaton*, or simply *automaton*. We use here the term 'abstract' to emphasize that at this level we are dealing with idealized models and put aside the real properties of input and output signals; we consider them only as letters of some alphabets.

2.2 Mealy and Moore automata

Abstract automaton has a set of states *A*. In each point in time, the automaton is in some state a_m from this set of states ($a_m \in A$). When signal z_f appears at the input, the automaton produces signal w_g at the output according to its output function λ and transits into the next state a_s according to its transition function δ. Thus, to present an automaton we must present three sets – the *set of inputs Z*, the *set of outputs W*,

the *set of states A* and two functions – the *transition function* δ and the *output function* λ.

Mealy automaton and Moore automaton are the two more popular models now. In Mealy automaton, the next state depends on the current state and the current input, the current output depends on the current state and the current input as well:

$$a(t + 1) = \delta(a(t), z(t));$$
$$w(t) = \lambda\ (a(t), z(t)).$$

In Moore automaton, the next state depends on the current state and the current input, but the current output depends only on the current state:

$$a(t + 1) = \delta(a(t), z(t));$$
$$w(t) = \lambda\ (a(t)).$$

2.3 Automaton representation

2.3.1 Mealy automaton. We use two modes for automata representation – a *table form* and a *state diagram* (graph) form. Table 1 and Table 2 present Mealy automaton S_1 in the table form. Columns of these tables correspond to states, and the rows correspond to inputs. Mealy automaton S_1, thus described, has three states and two inputs. The next state $a_s = \delta(a_m,z_f)$ is written at the intersection of column a_m and row z_f in the transition table (Table 1), output $w_g = \lambda(a_m,z_f)$ is written at the intersection of column a_m and row z_f in the output table (Table 2). So, if automaton S_1 is in state a_2 and its input is equal to z_1, the output signal equals to w_2 (see intersection of column a_2 and row z_1 in Table 2) and the next state equals to a_1 (see the intersection of the same column and the same row in Table 1).

Table 1. $a_s = \delta(a_m, z_f)$

z_f	a_1	a_2	a_3
z_1	a_2	a_1	a_1
z_2	a_3	a_3	a_3

Table 2. $w_g = \lambda\ (a_m, z_f)$

z_f	a_1	a_2	a_3
z_1	w_3	w_2	w_1
z_2	w_1	w_3	w_1

Sometimes, the transition and the output tables are combined into one table (Table 3). In this table, the next state a_s and the output w_g are written together (a_s/w_g) at the intersection of the state column a_m and the input row z_f.

Table 3. The same Mealy automaton S_1

z_f	a_1	a_2	a_3
z_1	a_2/w_3	a_1/w_2	a_1/w_1
z_2	a_3/w_1	a_3/w_3	a_3/w_1

Thus, in the automaton S_1: $A = \{a_1, a_2, a_3\}$, $Z = \{z_1, z_2\}$, $W= \{w_1, w_2, w_3\}$ and the transition function δ and the output function λ, defined at the pairs (a_m, z_f), are presented in Table 1 and Table 2, or in combined Table 3. If from a practical point of view it is necessary to emphasize an initial state – the state of an automaton at the initial point of time, we will call it a_1 and write it in the left column of such tables.

The *state diagram* or the *graph* of the same Mealy automaton S_1 is shown in Fig. 3. In such a diagram, vertices correspond to the automaton states and arcs correspond to the automaton transitions. If there is a transition from state a_m to state a_s in Mealy automaton, then in the state diagram, there is an arc directed from vertex a_m to vertex

a_s. The input z_f, initiating this transition, and the output $w_g = \lambda(a_m,z_f)$ are written at this arc.

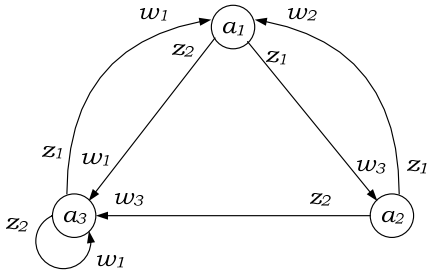


Figure 3. The state diagram of Mealy automaton S_1

2.3.2 Moore automaton. Table 4 is an example of Moore automaton presented in the table form. As the expressions for transition functions of Mealy and Moore automata are identical, the next state $a_s = \delta(a_m,z_f)$ is also written at the intersection of the column a_m and the row z_f in the transition table of Moore automaton. But the output of Moore automaton depends only on the current state $(w_g = \lambda(a_m))$, so we should not construct a special output table for Moore automaton – it is sufficient to write the output w_g over the corresponding state a_m. So, if Moore automaton S_2 is in the state a_4 and its input is equal to z_1, the next state is equal to a_1 (see the intersection of the column a_4 and row z_1 in Table 4). The output signal is equal to w_1 all the time while the automaton is in the state a_1 (see the output signal over the state a_1).

Table 4. Moore automaton S_2

	w_1	w_2	w_3	w_1	w_3
z_f	a_1	a_2	a_3	a_4	a_5
z_1	a_3	a_3	a_2	a_1	a_1
z_2	a_4	a_4	a_5	a_4	a_4

Thus, in the automaton S_2: $A = \{a_1, a_2, a_3, a_4, a_5\}$, $Z = \{z_1, z_2\}$, $W = \{w_1, w_2, w_3\}$, the transition function δ and the output function λ are presented in Table. 4. Sometimes, Moore automaton table like this is called a *marked transition table,* because output marks each corresponding state of this table.

The state diagram of Moore automaton S_2 is shown in Fig. 4. As in Mealy automaton, the vertices and the arcs correspond to the states and the transitions in Moore automaton state diagram. Since the outputs in such an automaton depend only on the states, each output is written near the corresponding state (vertex). The transformations from the table representation of the Mealy and Moore automata to the state diagrams and vice versa are evident.

2.3.3 Incomplete automaton. Mealy automaton is called *complete or completely specified*, if its transition function δ and output function λ are defined for each pair (a_m,z_f). Moore automaton is called *complete or completely specified*, if its transition function δ is defined for each pair (a_m,z_f) and its output function λ is defined for each state a_m. Automata S_1 and S_2 are complete. An automaton is incomplete or incompletely specified, if it is not complete. The example of incomplete Mealy

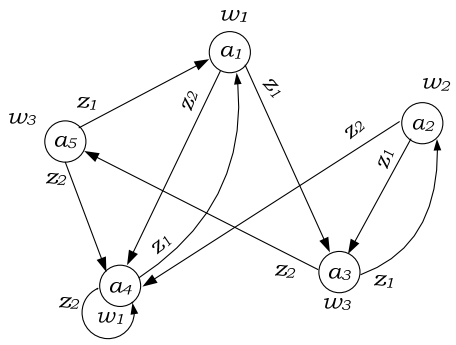


Figure 4. Moore automaton S_2

automaton S_3 is presented in Tables 5-6 and in Fig. 5. As seen from this example, if a transition is not defined for a pair (a_m,z_f) in automaton tables, in the automaton graph there is no arc going out from state a_m with input z_f written on this arc. Thus, the following transitions are lacking in the state diagram in Fig. 5:

1. From the state a_1 with the input z_1;
2. From the state a_3 with the input z_2;
3. From the state a_4 with the input z_2.

On the other hand, in Fig. 5 we have the arc from the state a_2 with the input z_3 because this transition is defined although the output signal is not defined at the pair (a_2,z_3).

Table 5. $a_s = \delta(a_m, z_f)$

z_f	a_1	a_2	a_3	a_4
z_1	-	a_4	a_3	a_1
z_2	a_2	a_2	-	-
z_3	a_3	a_3	a_2	a_3

Table 6. $w_g = \lambda(a_m, z_f)$

z_f	a_1	a_2	a_3	a_4
z_1	-	w_1	w_4	w_2
z_2	w_1	w_3	-	-
z_3	w_2	-	w_5	w_4

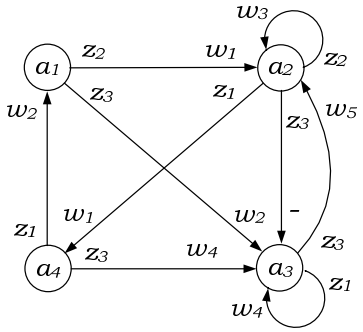


Figure 5. The incomplete Mealy automaton S_3

2.4 Automata responses to input sequences

To expand our understanding of automaton representation let us conduct two experiments – one with Mealy automaton S_1 and one with Moore automaton S_2.

2.4.1 Mealy automaton. Consider the behavior of Mealy automaton S_1 (Tables 1-2 or Fig. 3) for the given state a_1 and the input sequence $\xi(t) = z_2\ z_1\ z_1\ z_1\ z_2\ z_2$. The first column a_1 in these tables corresponds to the beginning of our experiments. The first letter in the input sequence ξ equals to z_2. From the output table of automaton S_1 (Table 2) we find that $w_1 = \lambda(a_1, z_2)$ and we write w_1 in the first column of Fig. 6. To find the next state we turn to the transition table of S_1 (Table 1) and find that $a_3 = \delta(a_1, z_2)$. So, we write a_3 in the second column of Fig. 6. Repeating the same for the state a_3 and the second input z_1 (second column) we get $w_1 = \lambda(a_3, z_1)$, $a_1 = \delta(a_3, z_1)$ and so on. As a result, we got the output sequence (output word) $\omega(t)$ with the same length as in $\xi(t)$. Note, that at the last step we got the next state a_3 written in the last column of Fig. 6.

state sequence a(t)	a_1	a_3	a_1	a_2	a_1	a_3	a_3
input sequence ξ(t)	z_2	z_1	z_1	z_1	z_2	z_2	
output sequence ω(t)	w_1	w_1	w_3	w_2	w_1	w_1	

Figure 6. Experiment with Mealy automaton S_1

The output sequence $\omega(t)$ is a *response of Mealy automaton* S_1 in the state a_1 to the input sequence $\xi(t)$:

$$\omega(t) = \lambda(a_1, \xi(t)).$$

It is evident that for the incomplete automaton the response to some input sequence in some state may be undefined. This can happen in two cases:

1. For some state a_p and input z_f from the input sequence $\xi(t)$, the output function $\lambda(a_p, z_f)$ is not specified;

2. For some state a_p and input z_f from the input sequence $\xi(t)$, the transition function $\delta(a_p, z_f)$ is not specified.

For example, the response of the incomplete Mealy automaton S_3, presented in Tables 5-6 or Fig. 5, is

$$\lambda(a_2, z_1, z_1, z_2, z_1, z_3, z_3) = w_1, w_2, w_1, w_1, w_4, w_5.$$

However, the responses $\lambda(a_2, z_1, z_1, z_2, z_3, z_3, z_3)$ and $\lambda(a_2, z_1, z_3, z_3, z_1, z_2, z_1)$ are not defined since, in the first case, $\lambda(a_2, z_3)$ is not defined:

a(t)	a_2	a_4	a_1	a_2	a_3	a_2	a_3
ξ(t)	z_1	z_1	z_2	z_3	z_3	z_3	
ω(t)	w_1	w_2	w_1	-	w_5	-	

In the second case, $\delta(a_4, z_2)$ is not defined:

a(t)	a_2	a_4	a_3	a_2	a_4	-
ξ(t)	z_1	z_3	z_3	z_1	z_2	z_1
ω(t)	w_1	w_4	w_5	w_1	-	

2.4.2 Moore automaton. The response of Moore automaton S_2 (Table 4 or Fig.4) in the state a_1 to the same input sequence $\xi(t) = z_2\ z_1\ z_1\ z_1\ z_2\ z_2$ (we used it for Mealy automaton S_1) may be defined in a similar way (Fig. 7):

state sequence a(t)	a_1	a_4	a_1	a_3	a_2	a_4	a_4
input sequence ξ(t)	z_2	z_1	z_1	z_1	z_2	z_2	
output sequence ω(t)	w_1	w_1	w_1	w_3	w_2	w_1	w_1

Figure 7. Experiment with Moore automaton S_2

As seen from Fig. 7, the length of the output sequence $\omega(t)$ is equal to seven, whereas the length of the input sequence $\xi(t)$ is equal to six. Note, however, that the first output w_1 in $\omega(t)$ does not depend on the input sequence $\xi(t)$. Really, w_1 does not depend on the first input z_2 in this sequence, w_1 is defined only by the first state a_1. Therefore, we do not include the first output symbol w_1 in Moore automaton response. We call the output sequence $\omega(t) = \lambda(a_1, \xi(t))$, shifted right one place, a *response of Moore automaton* S_2 to the input sequence $\xi(t)$ in the state a_1. This response is underlined in Fig. 7.

2.5 Transformations between Mealy and Moore models

Two automata S and S' are *equivalent* if their responses to any input sequence in their initial states are equal. Let us return to the experiments with Mealy automaton S_1 (Fig. 6) and Moore automaton S_2 (Fig. 7). Their underlined responses to the input sequence $\xi(t) = z_2\ z_1\ z_1\ z_1\ z_2\ z_2$ are equal. Does this mean that these automata are equivalent? Of course, not, it is possible that there exists such an input sequence $\xi'(t)$, to which the responses of these automata in their initial states would be different. However, from this example, the question arises: is it possible to construct Mealy automaton that would be equivalent to the given Moore automaton? And vice versa, is it possible to construct Moore automaton that would be equivalent to the given Mealy automaton? We will show that there are the positive answers to these questions. Now we will consider the transformations between these automata models.

2.5.1 Moore to Mealy. To transform a graph of Moore automaton to the graph of Mealy automaton, it is sufficient to carry the output (w_g in our example) written near the state of Moore automaton to all the arcs incoming to the same state of Mealy automaton (Fig. 8).

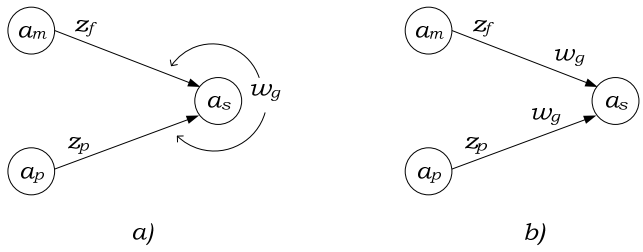


Figure 8. Subgraphs of Moore (a) and Mealy (b) automata

Automaton Mealy S_4, thus constructed for Moore automaton S_2 (Fig. 4) is presented in Fig. 9.

In a general case, let us have Moore automaton S_A with the set of states $A_A = \{ a_1, \dots , a_M\}$, the set of inputs $Z_A = \{ z_1, \dots , z_F\}$, the set of outputs $W_A = \{ w_1, \dots , w_G\}$, the transition function δ_A and the output function λ_A. Mealy automaton S_B that is equivalent to S_A, has the same set of states ($A_B = A_A = \{ a_1, \dots , a_M\}$), the same set of inputs $Z_B = Z_A = \{ z_1, \dots , z_F\}$, the same set of outputs $W_B = W_A = \{ w_1, \dots , w_G\}$ and the same transition function $\delta_B = \delta_A$ but their output functions are different.

Let Moore automaton S_A transit from the state a_m to the state a_s with the input signal z_f and its output in the state a_s equal to w_g:

$$\delta_A (a_m, z_f) = a_s;\ \ \lambda_A(a_s) = w_g.$$

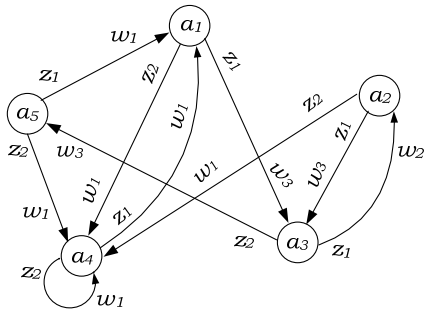


Figure 9. Automaton Mealy S_4, equivalent to Moore automaton S_2

Then, in Mealy automaton, the same transition takes place and the output at this transition is equal to w_g:

$$\delta_B (a_m,z_f) = a_s;\ \ \lambda_B(a_m,z_f) = w_g.$$

It corresponds to the carrying of the output, written near the state of Moore automaton, to all arcs incoming to the same state of Mealy automaton.

If we use a table for representation of Moore automaton S_A (Table 7) then the transition and output functions of Mealy automaton S_B, equivalent to S_A, can be constructed in the following way. The transition function of S_B (Table 8) coincides with the transition function of Moore automatonS_A. To construct the output function of S_B (Table 9) we replace the state a_s in Table 8 by the output signal w_g that marks the state a_s in the marked transition table (Table 7) of Moore automatonS_A.

Table 7. Transition table of S_2

	w_1	w_2	w_3	w_1	w_3
z_f	a_1	a_2	a_3	a_4	a_5
z_1	a_3	a_3	a_2	a_1	a_1
z_2	a_4	a_4	a_5	a_4	a_4

Table 8. Transition table of S_4

z_f	a_1	a_2	a_3	a_4	a_5
z_1	a_3	a_3	a_2	a_1	a_1
z_2	a_4	a_4	a_5	a_4	a_4

Table 9. Output table of S_4

z_f	a_1	a_2	a_3	a_4	a_5
z_1	w_3	w_3	w_2	w_1	w_1
z_2	w_1	w_1	w_3	w_1	w_1

From the method for the construction of Mealy automaton S_B, just considered, it is evident that this automaton is equivalent to Moore automaton S_A. Really, if some input signal z_f appears at the input of Moore automaton S_A in the state a_m, then this automaton transits to the state $a_s = \delta(a_m,z_f)$ and the output signal $w_g = \lambda_A(a_s)$ is generated while the automaton S_A is in this state a_s. But Mealy automaton S_B also transits from the state a_m to the same state $a_s = \delta_B(a_m,z_f)$ ($\delta_B = \delta_A$) with the same output signal $w_g = \lambda_B(a_m,z_f)$. Thus, for an input sequence with the length equal to one (for one input signal), the corresponding responses in any state a_m of the automata S_A and S_B will coincide. By mathematical induction, it is easy to show that any input sequence with the length equal to n will produce the same response in the corresponding states of the Moore and Mealy automata S_A and S_B.

2.5.2 Mealy to Moore. Let we have Mealy automaton S_B with the set of states $A_B = \{ a_1, \dots , a_M\}$, the set of inputs $Z_B = \{ z_1, \dots , z_F\}$, the set of outputs $W_B = \{ w_1, \dots , w_G\}$, the transition function δ_B and the output function λ_B. Moore automatonS_A that is

equivalent to S_B, has the same set of inputs $Z_A = Z_B = \{ z_1, \dots , z_F\}$ and the same set of outputs $W_A = W_B = \{ w_1, \dots , w_G\}$. We will illustrate the construction of the set of states of Moore automaton by Fig. 10. In this figure, there are four transitions into the state a_s of Mealy automaton with three different outputs w_p, w_q, and w_r.

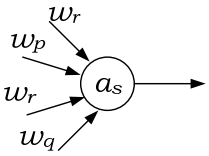


Figure 10. Four transitions into the state of Mealy automaton

Each such state of Mealy automaton generates as many states of Moore automaton, as many *different* outputs are at the transitions into this state. We present such states of Moore automaton as pairs (state, output) in Mealy automaton, the state a_s in Fig.10 generates three states

$$A_s = \{(a_s, w_r), (a_s, w_p), (a_s, w_q)\}.$$

The whole set of states A_A of Moore automaton S_A is the union of states generated by all states of Mealy automaton S_B:

$$A_A = \bigcup_{S=1}^{M} A_s .$$

The output function λ_A of Moore automaton S_A is defined very simple: for each state which is the pair (a_s, w_p), the output is equal to w_p – to the second component of this pair.

Before we define the transition function of Moore automaton, we will appeal to an example. As an example of transformation from the Mealy model to the Moore model, we use Mealy automaton S_1 (we repeat this automaton here in Fig. 11). In this automaton:

$$A_B = \{a_1, a_2, a_3\}; \qquad Z_B = \{z_1, z_2\};$$
$$W_B = \{w_1, w_2, w_3\}.$$

The transition function δ_B and the output function λ_B of S_1 are defined in Fig. 11.

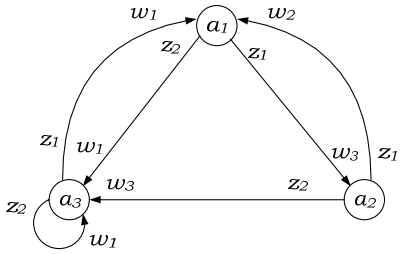


Figure 11. The state diagram of Mealy automaton S_1

We construct Moore automaton S_5. In this automaton:

$$Z_A = \{z_1, z_2\}; \qquad W_A = \{w_1, w_2, w_3\}.$$

Three states of Mealy automaton S_1 generate the following states of Moore automaton S_5 (we renamed pairs by $b_1, \ldots, b_5$):

$$A_1 = \{(a_1, w_1), (a_1, w_2)\} = \{b_1, b_2\};$$
$$A_2 = \{(a_2, w_3)\} = \{ b_3\};$$
$$A_3 = \{(a_3, w_1), (a_3, w_3)\} = \{b_4, b_5\}.$$

Thus, each state of Moore automaton S_5 is a pair (state, output) of Mealy automaton S_1. In our example:

$$b_1 = (a_1, w_1); \quad b_2 = (a_1, w_2); \quad b_3 = (a_2, w_3);$$
$$b_4 = (a_3, w_1); \quad b_5 = (a_3, w_3).$$

The set of states of S_5 is equal to

$$A_A = A_1 \cup A_2 \cup A_3 = \{b_1, b_2, b_3, b_4, b_5\}.$$

To distinguish these states of Moore automaton S_5 from the states of Mealy automaton S_1, we denoted them $b_1, \ldots, b_5$. We can now even draw the states of the automaton S_5 with output signals (Fig.12). It remains only to define the transitions between these states – to define the transition function of the automaton S_5.

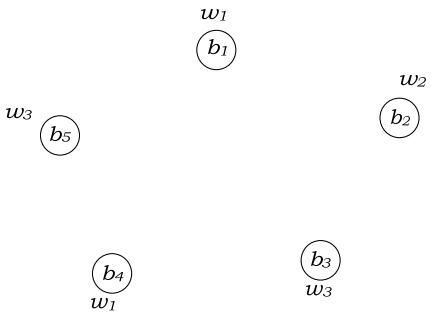


Figure 12. The states and output function of Moore automaton S_5

Fig. 13 illustrates the definition of function δ_A of Moore automaton S_A. If there is a transition from a_m to a_s with an input z_f and an output w_k in Mealy automaton S_B, then there should be the transitions from all the states A_m, generated by the state a_m of Mealy automaton, to the state (a_s, w_k) with the same input z_f in Moore automaton S_A.

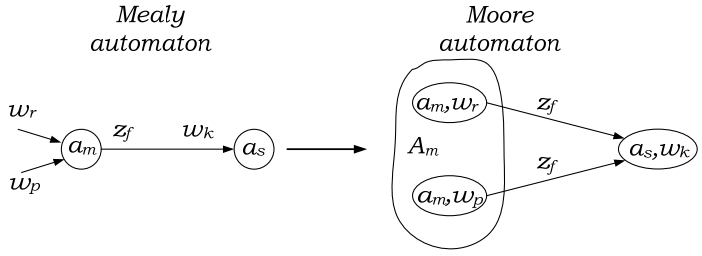


Figure 13. Definition of function δ_A of Moore automaton S_A

Let us return to our example. To illustrate, how to design a graph of Moore automaton S_5 we will take one of the transitions of Mealy automaton S_1 and construct the corresponding transitions in Moore automaton S_5 (Fig. 14). In the Mealy automaton S_1, there is a transition from the state a_1 to the state a_2 with the input z_1 and the

output w_3. Then, in Moore automaton S_5, there should be transitions from all the states of $A_1 = \{b_1, b_2\}$, generated by a_1, with the same input z_f to the state $(a_2, w_3) = b_3$ (it is the state generated by the state a_2 and the output w_3 at the discussed transition).

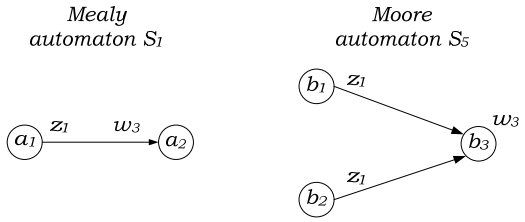


Figure 14. One transition in S_1 and the corresponding transitions in S_5

Continue in the same way with all other transitions of Mealy automaton S_1 we will get the state diagram of Moore automaton S_5 (Fig. 15).

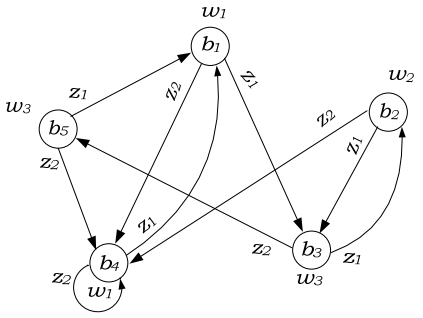


Figure 15. The state diagram of Moore automaton S_5

Let us discuss the case of an incomplete Mealy automaton, where the transition function δ_B is specified at the pair (a_m, z_f) ($a_s = \delta(a_m, z_f)$), but the output function λ_B is not specified at this transition (see, for example, the transition $a_3 = \delta(a_2, z_3)$ in the automaton S_3 in Fig. 5). Then the set A_s, generated by a_s, contains the pair $(a_s, -)$ with unspecified second component. In our example in Fig. 5, the state a_3 of the automaton S_3 generates the set $A_3 = \{(a_3, w_2), (a_3, w_4), (a_3, -)\}$. We leave you the transformation of this Mealy automaton S_3 to Moore automaton as an exercise.

Suppose that we would like to transform Moore automaton S_5 (Fig. 15) to some Mealy automaton S_6. As in such transformation (Moore model → Mealy model), the number of states is not changed, Mealy automaton S_6 will have five states. As a result, we get the chain presented in Fig. 16. Here we ran into the situation of two equivalent automata S_1 and S_6 of the same Mealy type having a different number of states. Thus, we came to the problem of the state minimization which we will discuss in the next section.

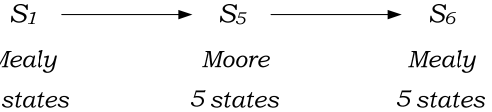


Figure 16. Equivalent Mealy automata S_1 and S_6 with the different number of states

2.6 State minimization

2.6.1 Equivalent automata. Two states a_m and a_s are said to be *equivalent* ($a_m \equiv a_s$), if the responses to *any input sequence* in these states coincide, i.e. $\lambda(a_m, \xi) = \lambda(a_s, \xi)$ for any input sequence ξ. If two states are not equivalent, they are *distinguishable*.

Two states a_m and a_s are said to be *k-equivalent* ($a_m \overset{k}{\equiv} a_s$), if their responses to any input sequence ξ_k of *the length k* in these states coincide, i.e. $\lambda(a_m, \xi_k) = \lambda(a_s, \xi_k)$. If two states are not *k*-equivalent, they are *k-distinguishable*.

Two automata S and S' of the same type (Mealy or Moore) are *equivalent (S≡ S'),* if for each state a_m of the automaton *S* there exists a state a_s' of the automaton *S',* equivalent to a_m, and, vice a versa, for each state a_s' of the automaton *S',* there exists a state a_m of the automaton *S*, equivalent to a_s'.

An automaton *S* is *minimal*, if there are no equivalent states in this automaton *S* (from $a_m \equiv a_s$ it follows that $a_m = a_s$ for the minimal automaton). We will consider here a method of the state minimization of complete automata. The main idea of this method is illustrated by Fig.17 and consists in:

1. Partition of a set of states into disjoint blocks of equivalent states;
2. Replacement of each such block with one state.

The minimal automaton, thus constructed, has exactly as many states as the number of blocks in this partition. In Fig.17, we have five equivalent blocks containing 16 dots (states in non-minimal automaton). The minimal automaton will have only five states.

The equivalent and *k*-equivalent relations, just introduced, allow us to find partitions π and π_k of the state set *A* with the blocks of equivalent and *k*-equivalent states. Having used the partition π we can find redundant states in the set *A*.

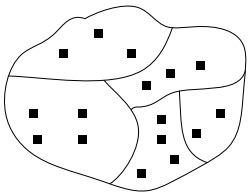


Figure 17. Partition of a set of states into equivalent classes

Let, for example, states a_m and a_s be equivalent. This means that these states are indistinguishable regarding their responses to any input sequence and it is not significant, whether the automaton is in the state a_m or in the state a_s. Consequently, one of these states can be removed from the set *A*. If each equivalent block in the partition π contains one state, the set *A* is *nonreducible*.

2.6.2 Minimization of Mealy automaton. The algorithm for state minimization of Mealy automaton consists of the following steps:

1. Find sequential partitions $\pi_1, \pi_2, \ldots, \pi_k, \pi_{k+1}$ of the state set *A* into blocks one, two, … , *k, (k+1)*-equivalent states until $\pi_k = \pi_{k+1}$ at *(k + 1)* step. It is easy to

show, that $\pi_k = \pi$ in this case, i.e. k-equivalent states are equivalent and k is not more than $(M - 1)$, where M is the number of states in set A.

2. Take one state from each equivalent block and form a state set A_{min} of the minimal automaton S_{min} that is equivalent to the automaton S.
3. Define the functions δ_{min} and λ_{min} of the automaton S. For this, delete the columns with the states not included in A_{min}, from the transition and output tables of the automaton S. Replace the states not included in A_{min} by equivalent ones from A_{min} in these tables.
4. Take one of the states, equivalent to a_1 as an initial state a_{1min} of automaton S_{min}.

As an example, let us consider the state minimization for Mealy automaton S_7 presented in Tables 10 and 11. If we combine states with equal columns in Table 11 we will get the partition of the set of states into blocks of 1-equivalent states:

$$\pi_1 = \overline{a_1, a_2, a_5, a_6}; \overline{a_3, a_4} = \{B_1, B_2\}.$$

Table 10. Transition table of S_7

z_f	a_1	a_2	a_3	a_4	a_5	a_6
z_1	a_3	a_4	a_3	a_4	a_5	a_6
z_2	a_5	a_6	a_5	a_6	a_1	a_2

Table 11. Output table of S_7

z_f	a_1	a_2	a_3	a_4	a_5	a_6
z_1	w_1	w_1	w_1	w_1	w_1	w_1
z_2	w_1	w_1	w_2	w_2	w_1	w_1

Indeed, two states a_m and a_s are 1-equivalent, if, in these states, an automaton has the same responses to any input sequence of the length one (i.e. columns a_m and a_s must be equal in the output table of this automaton).

Construct a table for the partition π_1 (Table 12) replacing the states in the columns of Table 10 by their 1-equivalent blocks. Obviously, two 1-equivalent states are 2-equivalent, if they transit to 1-equivalent states with equal inputs.

Table 12. Partition π_1

	B_1				B_2	
z_f	a_1	a_2	a_5	a_6	a_3	a_4
z_1	B_2	B_2	B_1	B_1	B_2	B_2
z_2	B_1	B_1	B_1	B_1	B_1	B_1

From Table 12 we get the partition π_2 (Table 13), combining equal columns in each block in Table 12:

$$\pi_2 = \overline{a_1, a_2}; \overline{a_5, a_6}; \overline{a_3, a_4} = \{C_1, C_2, C_3\}.$$

Table 13. Partition π_2

	C_1		C_2		C_3	
z_f	a_1	a_2	a_5	a_6	a_3	a_4
z_1	C_3	C_3	C_2	C_2	C_3	C_3
z_2	C_2	C_2	C_1	C_1	C_2	C_2

In exactly the same way, we construct the partition π_3

$$\pi_3 = \overline{a_1, a_2}; \overline{a_5, a_6}; \overline{a_3, a_4} = \{D_1, D_2, D_3\}$$

which is equal to π_2. Thus, π_2 is the partition of the state set A of Mealy automaton S_7 into blocks of equivalent states.

To construct a minimal automaton S_{min} (Tables 14 and 15) we take any state from each block of the partition π_2 to form the state set A_{min} of this automaton. Let, for example, $A_{min} = \{a_1, a_4, a_5\}$. After this, we remove the columns with states a_2, a_3, a_6 not included in A_{min}, from the transition and output tables of automaton S_7. Inside these tables, we replace states not included in A_{min} by equivalent ones from A_{min}. For example, we replace a_3 by a_4 at the intersection of column a_1 and row z_1 and a_6 by a_5 at the intersection of column a_4 and row z_2.

Table 14. Transition table of S_{min}

z_f	a_1	a_4	a_5
z_1	a_4	a_4	a_5
z_2	a_5	a_5	a_1

Table 15. Output table of S_{min}

z_f	a_1	a_4	a_5
z_1	w_1	w_1	w_1
z_2	w_1	w_2	w_1

2.6.3 Minimization of Moore automaton. To minimize Moore automaton, in the first step, we should find the partition of the state set into 0-equivalent blocks. Two states a_m and a_s of Moore automaton are said to be 0-equivalent ($a_m \overset{0}{\equiv} a_s$), if they are marked by equal outputs. Two 0-equivalent states are 1-equivalent, if they transit to 0-equivalent states under equal inputs. All the next k-equivalent blocks for Moore automaton can be constructed in exactly the same way as for Mealy automaton. As a result of minimization of Moore automaton S_8 in Table 16 with 12 states, we get the minimal Moore automaton S_9 with 4 states (Table 17). We give here only a sequence of partitions without the corresponding tables.

Table 16. Moore automaton S_8

	w_1	w_1	w_3	w_3	w_3	w_2	w_3	w_1	w_2	w_2	w_2	w_2
z_f	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
z_1	a_{10}	a_{12}	a_5	a_7	a_3	a_7	a_3	a_{10}	a_7	a_1	a_5	a_2
z_2	a_5	a_7	a_6	a_{11}	a_9	a_{11}	a_6	a_4	a_6	a_8	a_9	a_8

$$\pi_0 = \overline{a_1, a_2, a_8}; \overline{a_3, a_4, a_5, a_7}; \overline{a_6, a_9, a_{10}, a_{11}, a_{12}} = \{B_1, B_2, B_3\};$$
$$\pi_1 = \overline{a_1, a_2, a_8}; \overline{a_3, a_4, a_5, a_7}; \overline{a_6, a_9, a_{11}}; \overline{a_{10}, a_{12}} = \{C_1, C_2, C_3, C_4\};$$
$$\pi_2 = \overline{a_1, a_2, a_8}; \overline{a_3, a_4, a_5, a_7}; \overline{a_6, a_9, a_{11}}; \overline{a_{10}, a_{12}} = \{D_1, D_2, D_3, D_4\}.$$
$$\pi_2 = \pi_1.$$

Table 17. Minimal Moore automaton S_9

	w_1	w_3	w_2	w_2
z_f	a_1	a_3	a_6	a_{10}
z_1	a_{10}	a_3	a_3	a_1
z_2	a_3	a_6	a_6	a_1

2.6.4 Minimization of combined automaton. In some applications, it is interesting to use the automaton that combines the properties of Mealy and Moore automata. We call it Combined automaton (C-automaton). We can describe the behavior of C-automaton as follows:

$$a(t+1) = \delta(a(t), z(t));$$
$$w(t) = \lambda_1(a(t), z(t));$$
$$u(t) = \lambda_2(a(t)).$$

Thus, C-automaton has two output functions – one as in the Mealy model and the second one – as in the Moore model. It is possible to think about this in this way: the output $u_h = \lambda_2(a_m)$ is generated every time when automaton is in the state a_m, whereas the output $w_g = \lambda_1(a_m, z_f)$ is generated in the state a_m when the input z_f is present. For C-automata representation, it is also possible to use tables and state diagrams. To present C-automaton in a tabular form we use a transition table and an output table. The transition table of C-automaton S_{10} (Table 18) is similar to the transition table in the Mealy model, while, in the output table (Table 19), states are marked by the outputs from the set of outputs U.

Table 18. Transition table of C-automaton S_{10}

z_f	a_1	a_2	a_3	a_4	a_5
z_1	a_5	a_3	a_2	a_5	a_1
z_2	a_3	a_2	a_2	a_2	a_4

Table 19. Output table of C-automaton S_{10}

	u_1	u_2	u_1	u_1	u_3
z_f	a_1	a_2	a_3	a_4	a_5
z_1	w_1	w_2	w_1	w_1	w_2
z_2	w_2	w_1	w_2	w_2	w_1

In the state diagram (Fig. 18), the outputs from the set W are written on the arcs, the outputs from the set U are written near the corresponding states. Of course, it is possible to transform C-automaton to Mealy automaton or to Moore automaton just as we have transformed the Mealy model to the Moore model and vice versa.

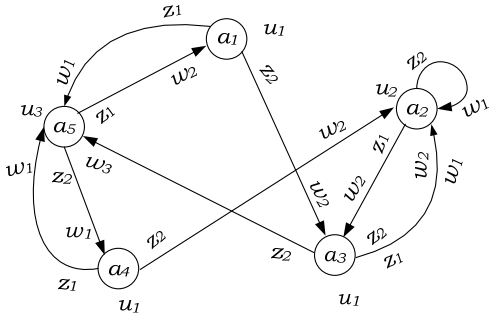


Figure 18. State diagram (graph) of C-automaton S_{10}

For the purpose of state minimization of a complete C-automaton, we can use the algorithm for Mealy automaton minimization from the previous section, if we assume that two states a_m and a_s of C-automaton are 1-equivalent, if they are marked by the equal outputs and have the equal columns in the output table. As an example we provide the state minimization of C-automaton S_{11} (Tables 20-21). The minimization process, presented in Tables 22-23, corresponds to the following sequence of partitions:

$$\pi_1 = \overline{a_1, a_2, a_5, a_7, a_8}; \overline{a_3, a_4, a_6, a_9, a_{11}}; \overline{a_{10}, a_{12}} = \{B_1, B_2, B_3\};$$
$$\pi_2 = \overline{a_1, a_2}; \overline{a_5, a_7}; \overline{a_8}; \overline{a_3, a_4, a_6, a_9, a_{11}}; \overline{a_{10}, a_{12}} = \{C_1, C_2, C_3, C_4, C_5\};$$
$$\pi_3 = \overline{a_1, a_2}; \overline{a_5, a_7}; \overline{a_8}; \overline{a_3, a_4, a_6, a_9, a_{11}}; \overline{a_{10}, a_{12}} = \{D_1, D_2, D_3, D_4, D_5\}.$$
$$\pi_3 = \pi_2.$$

Table 20. Transition table of nonminimal C-automaton S_{11}

z_f	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
z_1	a_{10}	a_{12}	a_5	a_5	a_3	a_5	a_9	a_{10}	a_7	a_2	a_7	a_1
z_2	a_5	a_7	a_9	a_4	a_6	a_6	a_{11}	a_4	a_9	a_8	a_{11}	a_8

Table 21. Output table of nonminimal C-automaton S_{11}

	u_1	u_1	u_2	u_2	u_1	u_2	u_1	u_1	u_2	u_3	u_2	u_3
z_f	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
z_1	w_1	w_1	w_2	w_2	w_1	w_2	w_1	w_1	w_2	w_2	w_2	w_2
z_2	w_2	w_2	w_1	w_1	w_2	w_1	w_2	w_2	w_1	w_1	w_1	w_1

Table 22. Partition π_1

	B_1					B_2					B_3	
z_f	a_1	a_2	a_5	a_7	a_8	a_3	a_4	a_6	a_9	a_{11}	a_{10}	a_{12}
z_1	B_3	B_3	B_2	B_2	B_3	B_1	B_1	B_1	B_1	B_1	B_1	B_1
z_2	B_1	B_1	B_2	B_2	B_2	B_2	B_2	B_2	B_2	B_2	B_1	B_1

Table 23. Partition π_2

	C_1		C_2		C_3	C_4					C_5	
z_f	a_1	a_2	a_5	a_7	a_8	a_3	a_4	a_6	a_9	a_{11}	a_{10}	a_{12}
z_1	C_5	C_5	C_4	C_4	C_5	C_2	C_2	C_2	C_2	C_2	C_1	C_1
z_2	C_2	C_2	C_4	C_4	C_4	C_4	C_4	C_4	C_4	C_4	C_3	C_3

The minimal C-automaton S_{12} is presented in Tables 24-25.

Table 24. Transition table of the minimal C-automaton S_{12}

z_f	a_1	a_3	a_5	a_8	a_{10}
z_1	a_{10}	a_5	a_3	a_{10}	a_1
z_2	a_5	a_3	a_3	a_3	a_8

Table 25. Output table of the minimal C-automaton S_{12}

	u_1	u_2	u_1	u_1	u_3
z_f	a_1	a_3	a_5	a_8	a_{10}
z_1	w_1	w_2	w_1	w_1	w_2
z_2	w_2	w_1	w_2	w_2	w_1

Chapter 3 Structure Automata

In the previous Chapter, we were considering abstract automaton as a 'black box' with one input and one output (Fig. 2.1) which transforms input sequences (words of the input alphabet *Z*) into output sequences (words of the output alphabet *W*). We were not interested in the contents of this black box. In this Chapter, we will concentrate on the interior of a black box and examine how to realize the behavior, described at the level of abstract automaton, by means of hardware components.

3.1 Synthesis of Mealy automaton

3.1.1 Structure automaton. We can look at structure automaton as a follow-up detailing of abstract automaton. Unlike abstract automaton, structure automaton (Fig. 1) has L inputs and N outputs. The signals zero or one can appear at each input x_l ($l=1, \ldots, L$) and at each output y_n ($n=1, \ldots, N$) of structure automaton. Thus, the input of structure automaton is a binary vector with L components, each of which is equal to zero or one. Each output of structure automaton is a vector with N components, each of which is also equal to zero or one.

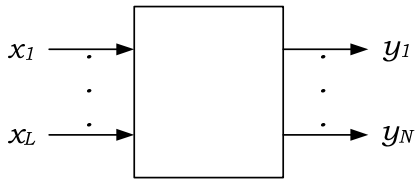


Figure 1. Structure automaton

Fig. 2 presents a basic structure for Mealy automaton with two parts – Logic (combinational circuit) and Memory. Memory contains *memory elements* – Moore automata with two states (zero and one). Usually, *flip-flops* are used as memory elements in structure automaton.

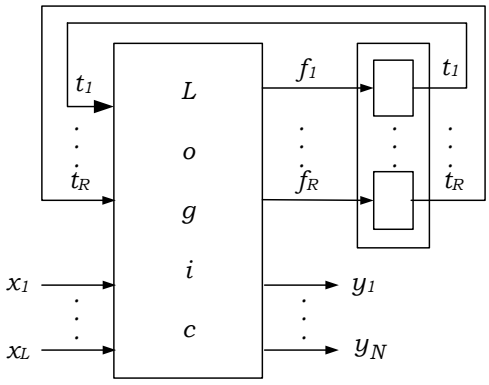


Figure 2. The structure of structure automaton

As the first example in this chapter, we will use an abstract Mealy automaton S_1 in Tables 1 and 2 (the transition and output functions). To transform this abstract automaton into the corresponding structure automaton we should encode its each input z_f, each output w_g and each state a_m by binary vectors. Since abstract automaton S_1 has three inputs z_1, z_2, z_3, six outputs $w_1,\ldots,w_6$ and five states $a_1,\ldots,a_5$, the corresponding structure automaton will have two binary inputs x_1, x_2, three

binary outputs y_1, y_2, y_3 and three binary memory elements t_1, t_2, t_3. The basic structure for this automaton is shown in Fig. 3.

Table 1. Transition table of automaton S_1

z_f	a_1	a_2	a_3	a_4	a_5
z_1	a_2	a_3	-	a_4	a_1
z_2	a_3	a_1	a_1	a_5	-
z_3	a_2	-	a_4	-	a_4

Table 2. Output table of automaton S_1

z_f	a_1	a_2	a_3	a_4	a_5
z_1	w_1	w_6	-	w_4	w_5
z_2	w_3	w_1	w_5	w_2	-
z_3	w_4	-	w_3	-	w_1

Before we discuss how encoding affects the complexity of the automaton logic circuit we will use so-called *trivial encoding* in which the code (binary number) is equal to the decimal number of the encoded object. For example, for state a_1 we will use code *001*, for state a_2 – code *010* etc. The corresponding tables for input, output and state encoding (assignment) are presented in Tables 3 – 5.

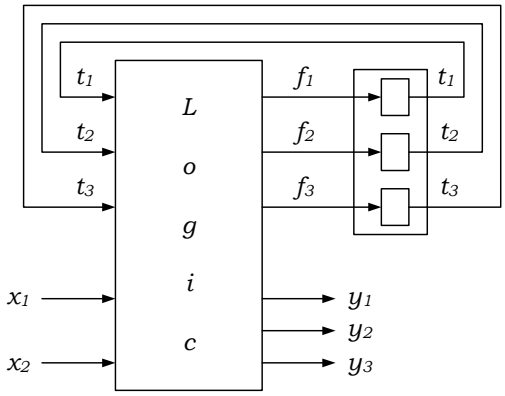


Figure 3. The structure of the automaton in our example

Table 3. Input encoding

z_f	x_1x_2
z_1	*0 1*
z_2	*1 0*
z_3	*1 1*

Table 4. Output encoding

w_g	$y_1y_2y_3$
w_1	*0 0 1*
w_2	*0 1 0*
w_3	*0 1 1*
w_4	*1 0 0*
w_5	*1 0 1*
w_6	*1 1 0*

Table 5. State encoding

a_m	$t_1t_2t_3$
a_1	*0 0 1*
a_2	*0 1 0*
a_3	*0 1 1*
a_4	*1 0 0*
a_5	*1 0 1*

As the memory element, we will use Moore automaton with two states. Its transition table is shown in Table 6. We suppose that the output of this automaton is equal to its state – when the automaton is in the state *0*, the output is equal to *0*, when it is in the state *1*, the output is equal to *1*, so it is not necessary to mark states by their outputs.

Table 6. Transition table of the memory element

	t	
f	*0*	*1*
0	*0*	*1*
1	*1*	*0*

3.1.2 Execution of structure automaton. Now we return to the structure of the automaton S_1 in Fig. 3 to discuss how it works. Let abstract automaton S_1 transit from state a_1 to state a_2 with input signal z_3 (see Table 1). w_4 is the output signal at this transition (Table 2). When abstract automaton S_1 is in the state a_1, structure automaton S_1 is in the state *001* (Table 5). The input vector *11*, corresponding to z_3 (Table 3), appears at the inputs of structure automaton (see Fig. 4). The output vector *100*, corresponding to w_4 (Table 4), is generated at the outputs y_1, y_2, y_3 of circuit *Logic*.

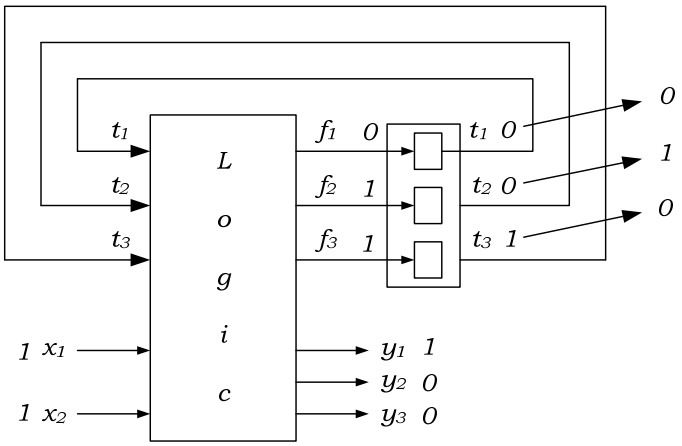


Figure 4. The structure of the automaton at the transition from a_1 to a_2 with input z_3

To transfer automaton S_1 from state a_1 (code *001*) to state a_2 (code *010*) we must transfer the first memory element t_1 from state *0* to state *0*, the second t_2 – from state *0* to state *1* and the third t_3 – from state *1* to state *0*. To implement these transitions of memory elements, we should supply the corresponding signals to their inputs. To determine these inputs we must use the transition table of the memory element (Table 6). According to this table: for transition of the first memory element from *0* to *0* its input should be equal to *0*, for transitions of the second memory element from *0* to *1* and the third – from *1* to *0*, their inputs should be equal to *1*. Thus, vector *011* should appear at the inputs f_1, f_2, f_3 of the memory elements (Fig. 4). The functions f_1, f_2, f_3 are called *input memory functions* or excitation memory functions.

Thus, after choosing memory elements and encoding inputs, outputs and states the problem of the logic synthesis for the automaton S_1 with a basic structure is reduced to the synthesis of the combinational circuit, which realizes the following functions:

$$y_n = y_n(t_1, t_2, t_3, x_1, x_2);\ n = 1, \ldots,3;$$

$$f_r = f_r(t_1, t_2, t_3, x_1, x_2);\ r = 1, \ldots,3.$$

3.1.3 Automaton structure table. The structure table of automaton S_1 is shown in Table 7. This table has the following columns: a_m and a_s are the current and the next states; $t_1t_2t_3$ and $t_{1n}t_{2n}t_{3n}$ are codes of a_m and a_s; z_f and w_g are the input and output signals; x_1x_2 contains binary input signals; $y_1y_2y_3$ and $f_1f_2f_3$ contain the values of outputs of *Logic* circuit at Fig. 4.

Table 7. The structure table of the automaton S_1

a_m	$t_1t_2t_3$	a_s	$t_{1n}t_{2n}t_{3n}$	z_f	x_1x_2	w_g	$y_1y_2y_3$	$f_1f_2f_3$
a_1	001	a_2	010	z_1	01	w_1	001	011
	001	a_3	011	z_2	10	w_3	011	010
	001	a_2	010	z_3	11	w_4	100	011
a_2	010	a_3	011	z_1	01	w_6	110	001
	010	a_1	001	z_2	10	w_1	001	011
	010	-	-	z_3	11	-	-	-
a_3	011	-	-	z_1	01	-	-	-
	011	a_1	001	z_2	10	w_5	101	010
	011	a_4	100	z_3	11	w_3	011	111
a_4	100	a_4	100	z_1	01	w_4	100	000
	100	a_5	101	z_2	10	w_2	010	001
	100	-	-	z_3	11	-	-	-
a_5	101	a_1	001	z_1	01	w_5	101	100
	101	-	-	z_2	10	-	-	-
	101	a_4	100	z_3	11	w_1	001	001

To fill in the last column of Table 7 let us look at the transition from a_1 to a_2 at the first row of this table. This transition involves three transitions of memory elements: the first memory element t_1 from state *0* to state *0*, the second t_2 – from state *0* to state *1* and the third t_3 – from state *1* to state *0*. Let us use Table 6 to find what input causes the transition of the first memory element from the state $t_1=0$ to the state $t_1=0$. Looking at the first column and the first row of this table we see that such input $f_1=0$. In exactly the same way, we will find that to transfer the second memory element from $t_2=0$ to $t_2=1$, its input should be equal to *1* (the first column and the second row of Table 6). To transfer the third memory element from $t_3=1$ to $t_3=0$, its input should also be equal to *1* (the second column and the second row of Table 6). Thus, we write *011* in the column $f_1f_2f_3$ in the first row of Table 7. It now should be evident how to fill in the other entries of this column.

3.1.4 Logic circuit synthesis. An automaton structure table may be considered as the truth table for functions y_1, y_2, y_3 and f_1, f_2, f_3 with variables t_1, t_2, t_3, x_1, x_2 (see the gray columns in this table). Thus, from this table we can derive the covers for y_1, y_2, y_3, f_1, f_2, f_3 as the set of input vectors where these functions are equal to one.

$y_1 =$

t_1	t_2	t_3	x_1	x_2
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1

$y_2 =$

t_1	t_2	t_3	x_1	x_2
0	0	1	1	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0

$f_1 =$

t_1	t_2	t_3	x_1	x_2
0	1	1	1	1
1	0	1	0	1

$f_2 =$

t_1	t_2	t_3	x_1	x_2
0	0	1	0	1
0	0	1	1	0
0	0	1	1	1
0	1	0	1	0
0	1	1	1	0
0	1	1	1	1

$y_3 =$

t_1	t_2	t_3	x_1	x_2
0	0	1	0	1
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
0	1	1	1	1
1	0	1	0	1
1	0	1	1	1

$f_3 =$

t_1	t_2	t_3	x_1	x_2
0	0	1	0	1
0	0	1	1	1
0	1	0	0	1
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1

Prior to minimization of these functions, we can define three kinds of don't care in our example:

1. Codes *000, 110* and *111* are not used for state assignment, therefore functions $y_1, y_2, y_3, f_1, f_2, f_3$ are not specified for the cubes

t_1	t_2	t_3	x_1	x_2
0	0	0	x	x
1	1	0	x	x
1	1	1	x	x

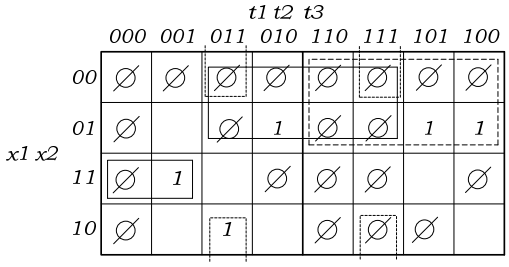
2. Code *00* is not used for encoding of input signals, therefore functions $y_1, y_2, y_3, f_1, f_2, f_3$ are not specified for the cube

t_1	t_2	t_3	x_1	x_2
x	x	x	0	0

3. The transition and output functions of the abstract automaton (see Tables 1 and 2) are not completely defined. As a result of this, structure automaton is not completely defined either – see dashes "-" in Table 7 in columns $y_1, y_2, y_3, f_1, f_2, f_3$. Thus, these functions are not specified for the cubes

t_1	t_2	t_3	x_1	x_2
0	1	0	1	1
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0

Karnaugh maps and minimized covers for functions $y_1, y_2, y_3, f_1, f_2, f_3$ are shown in Fig. 5 – Fig10.

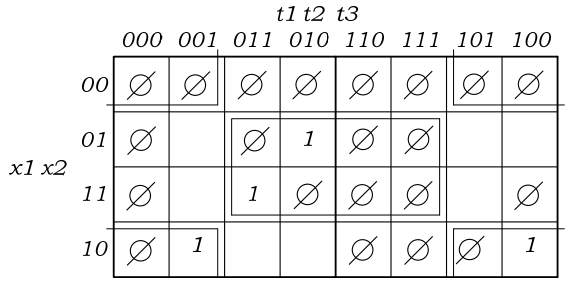


$y_1 =$

t_1	t_2	t_3	x_1	x_2
x	1	x	0	x
x	1	1	x	0
1	x	x	0	x
0	0	x	1	1

Figure 5.

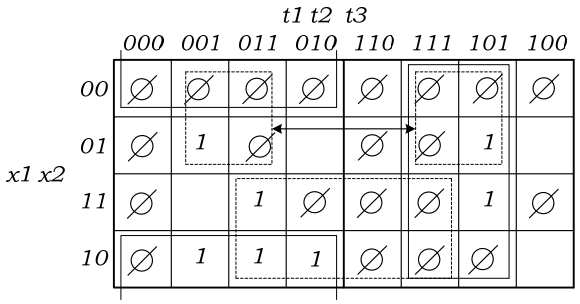
Function y_1



$y_2 =$

t_1	t_2	t_3	x_1	x_2
x	1	x	x	1
x	0	x	x	0

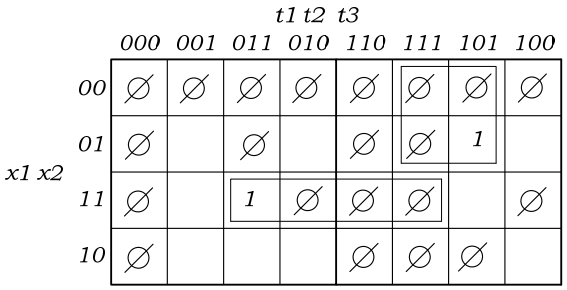
Figure 6. Function y_2



$y_3 =$

t_1	t_2	t_3	x_1	x_2
0	x	x	x	0
x	x	1	0	x
x	1	x	1	x
1	x	1	x	x

Figure 7. Function y_3

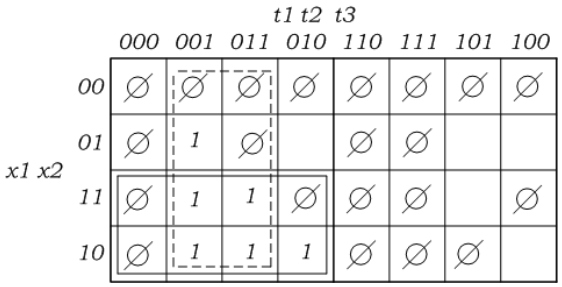


$f_1 =$

t_1	t_2	t_3	x_1	x_2
x	1	x	1	1
1	x	1	0	x

Figure 8. Function f_1

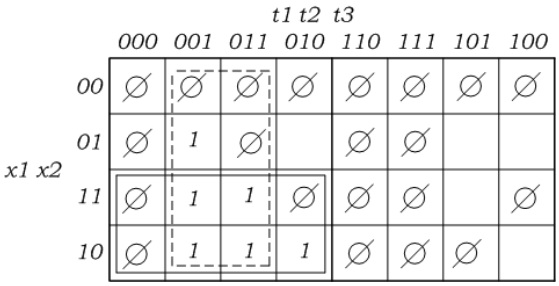
The logic circuit of automaton S_1, constructed by using minimized covers for functions y_1, y_2, y_3, f_1, f_2, f_3, is shown in Fig. 11. It is evident that this circuit corresponds to the structure in Fig. 4. Indeed, this logic circuit consists of two parts – the combinational circuit with inputs t_1, t_2, t_3, x_1, x_2 and outputs y_1, y_2, y_3, f_1, f_2, f_3 and the memory, containing three memory elements with inputs f_1, f_2, f_3 and outputs t_1, t_2, t_3.



$f_2 =$

t_1	t_2	t_3	x_1	x_2
0	x	x	1	x
0	x	1	x	x

Figure 9. Function f_2



$$f_3 = \begin{vmatrix} t_1 & t_2 & t_3 & x_1 & x_2 \\ 0 & x & x & x & 1 \\ x & 1 & 0 & x & x \\ 1 & x & x & 1 & x \end{vmatrix}$$

Figure 10. Function f_3

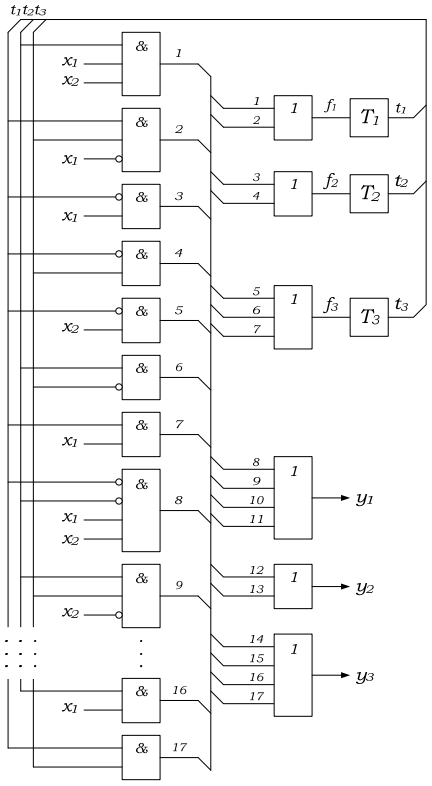


Figure 11. Logic circuit of automaton S_1

Consider the circuit in Fig. 11. The subsequent minimization of this circuit is possible through the use of methods for two-level minimization, but the main effect in the automaton logic circuit minimization may be obtained through various techniques of factorization and decomposition. Such techniques for the logic design of automata with a large number of inputs and outputs will be discussed in the next chapters.

Here we only note that in the circuit in Fig. 11, the considerable success in the two-level minimization was obtained by using the same terms in different output and input memory functions.

3.1.5 Output signal assignment. Now we will discuss the possibility for OR gate minimization in two level circuit in Fig. 11. The number of inputs into OR gates for outputs y_1, y_2, y_3 is equal to the number of 'ones' in the columns y_1, y_2, y_3 in Table 7. This number of 'ones' depends on the number of 'ones' in codes for output signals of abstract automaton (see Table 4) and on $p(w_g)$ – the number of appearances of each output in the column w_g in Table 7. We tabulate this information in the first two columns of Table 8.

Next two columns of this table contain codes for each w_g $(g = 1, \ldots,6)$ in a trivial encoding which we have been using until now, and $c(w_g)$ – the number of 'ones' which each output produces in the column last but one of Table 7. Really, if code of $w_1 = 001$ (it contains one 'one') and w_1 appears three times in Table 7 ($p(w_1) = 3$), the number of 'ones' $c(w_1)$ in the columns y_1, y_2, y_3 only due to w_1 is equal to 3. Exactly in the same way, output w_2 produces only one 'one' in Table 7 (one appearance and one 'one' in the code), output w_3 produces four 'ones' (two appearances and two 'ones' in the code) etc.

The algorithm for the optimal output encoding is very simple. First of all, place the outputs in the order of decreasing $p(w_g)$. Use zero code for encoding of the output with $p(w_g) = max$. In our example, $p(w_1) = max$ and we encode w_1 by 000 – see column $y_1y_2y_3$ in 'Optimized encoding' in Table 8. Then, we assign all possible codes with one component, equal to one, to the outputs with the next values of $p(w_g)$ – w_3, w_4 and w_5 in our example. In the next steps we use codes with two 'ones', after this – with three 'ones', etc. As a result, we get the encoding in the column $y_1y_2y_3$ in the 'Optimized encoding' in Table 8. The sum of numbers in the last column in this table is equal to 10; it is much lower than the sum of numbers in the column $c(w_g)$ in the 'Trivial encoding'.

Table 8. Optimized output encoding

w_g	$p(w_g)$	*Trivial encoding*		*Optimized encoding*	
		$y_1y_2y_3$	$c(w_g)$	$y_1y_2y_3$	$c(w_g)$
w_1	3	0 0 1	3	0 0 0	0
w_2	1	0 1 0	1	0 1 1	2
w_3	2	0 1 1	4	0 0 1	2
w_4	2	1 0 0	2	0 1 0	2
w_5	2	1 0 1	4	1 0 0	2
w_6	1	1 1 0	2	1 0 1	2

3.1.6 Logic synthesis with D flip-flops. *D flip-flop* (Fig. 12) is the most simple and the most frequently used memory element. The transition table of this flip-flop is presented in Table 9. The name *D* flip-flop results from word *'Delay'* – as seen from Table 9, the next state of D flip-flop is equal to the previous input (a next state is a *delayed* input).

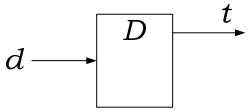


Figure 12. *D* flip-flop

Table 9. Transition table of *D* flip-flop

	t	
d	*0*	*1*
0	*0*	*0*
1	*1*	*1*

Table 10 contains the structure table of the automaton S_1 with D flip-flops. Here we have used the optimized output assignment from Table 8 and a trivial state assignment in which a binary code for each state is equal to the number of this state. To fill out the last column of Table 10 we do not need to apply to the transition table of D flip-flop (Table 9); we should simply copy the column $t_{1n}t_{2n}t_{3n}$ into the last column. Really, in the column $d_1d_2d_3$, we write inputs to the memory elements and these inputs, according to Table 9, are equal to the next states of the memory elements written in the column $t_{1n}t_{2n}t_{3n}$.

Table 10. The structure table of automaton S_1 with D flip-flops

a_m	$t_1t_2t_3$	a_s	$t_{1n}t_{2n}t_{3n}$	z_f	x_1x_2	w_g	$y_1y_2y_3$	$d_1d_2d_3$
a_1	001	a_2	010	z_1	01	w_1	000	010
	001	a_3	011	z_2	10	w_3	001	011
	001	a_2	010	z_3	11	w_4	010	010
a_2	010	a_3	011	z_1	01	w_6	101	011
	010	a_1	001	z_2	10	w_1	000	001
	010	-	-	z_3	11	-	-	-
a_3	011	-	-	z_1	01	-	-	-
	011	a_1	001	z_2	10	w_5	100	001
	011	a_4	100	z_3	11	w_3	001	100
a_4	100	a_4	100	z_1	01	w_4	010	100
	100	a_5	101	z_2	10	w_2	011	101
	100	-	-	z_3	11	-	-	-
a_5	101	a_1	001	z_1	01	w_5	100	001
	101	-	-	z_2	10	-	-	-
	101	a_4	100	z_3	11	w_1	000	100

As before for output functions y_1, y_2, y_3, we can minimize the number of inputs into OR gates for input memory functions d_1, d_2, d_3. For this, we should minimize the number of 'ones' in the column $d_1d_2d_3$. Since this column is equal to the column $t_{1n}t_{2n}t_{3n}$ it is sufficient to minimize that in the column $t_{1n}t_{2n}t_{3n}$. Thus, for the state assignment, we can use the algorithm for the output assignment from the previous section. Let $p(a_s)$ – the number of appearances of each state in the column a_s in Table 10 and $c(a_s)$ – the number of ones which each state produces in the column $t_{1n}t_{2n}t_{3n}$ (column $d_1d_2d_3$). We will insert this information into the first two columns of Table 11.

Table 11. Optimized state asignment

a_s	$p(a_s)$	*Trivial encoding*		*Optimized encoding*	
		$t_1t_2t_3$	$c(a_s)$	$t_1t_2t_3$	$c(a_s)$
a_1	3	0 0 1	3	0 0 0	0
a_2	2	0 1 0	2	0 1 0	2
a_3	2	0 1 1	4	1 0 0	2
a_4	3	1 0 0	3	0 0 1	3
a_5	1	1 0 1	2	0 1 1	2

First of all, place the states in the order of decreasing $p(a_s)$. Use zero code for encoding of output with $p(a_s) = max$. Here we have two states – a_1 and a_4 with the same weights equal to three. We can use code *000* for any of them, for example for a_1 – see column $t_1t_2t_3$ in the 'Optimized encoding' in Table 11. Then, we assign all possible codes with one component equal to one to the states with the next values of $p(a_s)$ – a_4, a_2 and a_3 in our example. In the next steps we use codes with two 'ones', after this – with three

'ones', etc. As a result, we get the encoding in the column $t_1t_2t_3$ in the 'Optimized encoding' in Table 11. The sum of numbers in the last column in this table is equal to 9, it is much lower than the sum of numbers in the column $c(a_s)$ in the 'Trivial encoding'.

This state assignment is used in Table 12. For such an assignment, we will get the new don't cares:

1. Codes *101, 110* and *111* are not used for the state assignment, therefore functions $y_1, y_2, y_3, d_1, d_2, d_3$ are not specified for the cubes

t_1	t_2	t_3	x_1	x_2
1	0	1	x	x
1	1	0	x	x
1	1	1	x	x

2. Code *00* is not used for encoding of input signals, therefore functions $y_1, y_2, y_3, d_1, d_2, d_3$ are not specified for the cube

t_1	t_2	t_3	x_1	x_2
x	x	x	0	0

3. Functions $y_1, y_2, y_3, d_1, d_2, d_3$ are not specified for the cubes corresponding to the rows with dashes "-" in the columns for these functions

t_1	t_2	t_3	x_1	x_2
0	1	0	1	1
1	0	0	0	1
0	0	1	1	1
0	1	1	1	0

Table 12. The structure table of automaton S_1 with optimized state assignment

a_m	$t_1t_2t_3$	a_s	$t_{1n}t_{2n}t_{3n}$	z_f	x_1x_2	w_g	$y_1y_2y_3$	$d_1d_2d_3$
a_1	000	a_2	010	z_1	01	w_1	000	010
	000	a_3	100	z_2	10	w_3	001	100
	000	a_2	010	z_3	11	w_4	010	010
a_2	010	a_3	100	z_1	01	w_6	101	100
	010	a_1	000	z_2	10	w_1	000	000
	010	-	-	z_3	11	-	-	-
a_3	100	-	-	z_1	01	-	-	-
	100	a_1	000	z_2	10	w_5	100	000
	100	a_4	001	z_3	11	w_3	001	001
a_4	001	a_4	001	z_1	01	w_4	010	001
	001	a_5	011	z_2	10	w_2	011	011
	001	-	-	z_3	11	-	-	-
a_5	011	a_1	000	z_1	01	w_5	100	000
	011	-	-	z_2	10	-	-	-
	011	a_4	001	z_3	11	w_1	000	001

Immediately from Table 12 we derive the covers for $y_1, y_2, y_3, d_1, d_2, d_3$:

$y_1 =$

t_1	t_2	t_3	x_1	x_2
0	1	0	0	1
1	0	0	1	0
0	1	1	0	1

$d_1 =$

t_1	t_2	t_3	x_1	x_2
0	0	0	1	0
0	1	0	0	1

$y_2 =$

t_1	t_2	t_3	x_1	x_2
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0

$d_2 =$

t_1	t_2	t_3	x_1	x_2
0	0	0	0	1
0	0	0	1	1
0	0	1	1	0

$y_3 =$

t_1	t_2	t_3	x_1	x_2
0	0	0	1	0
0	1	0	0	1
1	0	0	1	1
0	0	1	1	0

$d_3 =$

t_1	t_2	t_3	x_1	x_2
1	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	1	1	1

M$_2$, y_3, f_1, ; (check them!). The logic circuit of Mealy automaton S_1 with D flip-flops and optimized output and state encoding is presented in Fig. 19.

$y_1 =$

t_1	t_2	t_3	x_1	x_2
x	1	x	0	x
1	x	x	x	0

Figure 13. Function y_1

$d_1 =$

t_1	t_2	t_3	x_1	x_2
x	1	0	0	x
0	0	0	x	0

Figure 16. Function d_1

$y_2 =$

t_1	t_2	t_3	x_1	x_2
x	0	1	x	x
0	0	x	1	1

Figure 14. Function y_2

$d_2 =$

t_1	t_2	t_3	x_1	x_2
x	x	1	x	0
0	0	0	x	1

Figure 17. Function d_2

$y_3 =$

t_1	t_2	t_3	x_1	x_2
0	0	x	x	0
1	x	x	x	1
x	1	0	0	x

Figure 15. Function y_3

$d_3 =$

t_1	t_2	t_3	x_1	x_2
x	0	1	x	x
x	x	1	1	x
1	x	x	x	1

Figure 18. Function d_3

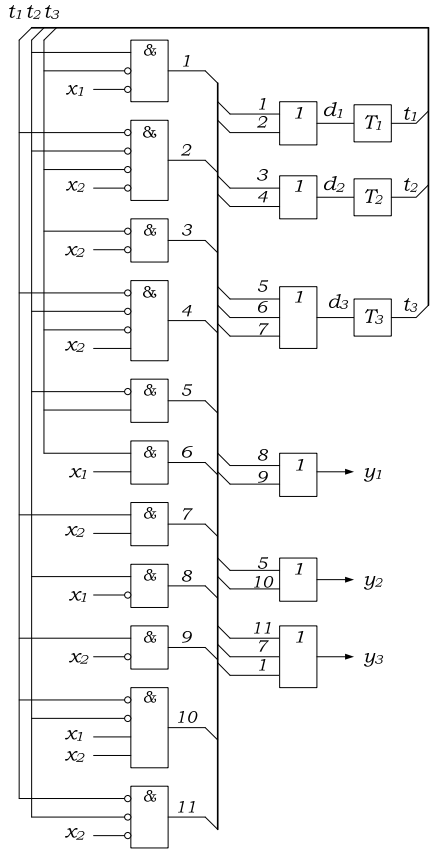


Figure 19. Logic circuit of Mealy automaton S_1 with D flip-flops

3.2 Synthesis of Moore automaton

Fig. 20 presents a basic structure for Moore automaton with three parts – two combinational circuits (Logic1 and Logic2) and Memory. As before in a Mealy model, the outputs of Logic1 are the input memory functions which depend on a current state $t_1, \ldots, t_R$ and input $x_1, \ldots, x_L$. The outputs $y_1, \ldots, y_N$ are the outputs of Logic2, because they depend only on the current state $t_1, \ldots, t_R$. As an example, we will use abstract Moore automaton S_2 in Table 13.

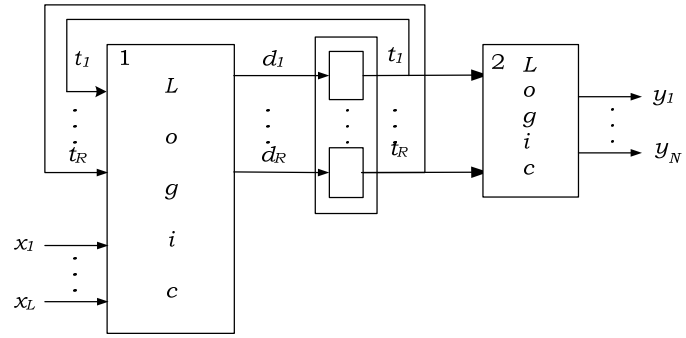


Figure 20. The structure of Moore automaton (general form)

Table 13. Moore automaton S_2

	w_1	--	w_2	w_3	w_2
	a_1	a_2	a_3	a_4	a_5
z_1	a_5	a_2	a_1	a_2	a_5
z_2	--	--	a_4	a_5	a_2
z_3	a_3	a_3	--	--	a_4

Table 14 is the structure table of the automaton S_2. First, we insert abstract automaton in this table by filling columns a_m, w_g, a_s and z_f from Table 13. Note, that we write w_g in the left part of Table 14 after codes for current states, because the output of Moore automaton depends only on the current state.

Table 14. The structure table of Moore automaton S_2

a_m	$t_1t_2t_3$	w_g	y_1y_2	a_s	$t_{1n}t_{2n}t_{3n}$	z_f	x_1x_2	$d_1d_2d_3$
a_1	101	w_1	01	a_5	001	z_1	00	001
	101			-	-	z_2	01	-
	101			a_3	010	z_3	10	010
a_2	000	-	-	a_2	000	z_1	00	000
	000			-	-	z_2	01	-
	000			a_3	010	z_3	10	010
a_3	010	w_2	00	a_1	101	z_1	00	101
	010			a_4	100	z_2	01	100
	010			-	-	z_3	10	-
a_4	100	w_3	10	a_2	000	z_1	00	000
	100			a_5	001	z_2	01	001
	100			-	-	z_3	10	-
a_5	001	w_2	00	a_5	001	z_1	00	001
	001			a_2	000	z_2	01	000
	001			a_4	100	z_3	10	100

To construct whole table, we must encode each input z_f, each output w_g and each state a_m by binary vectors (Tables 15 – 17). To minimize the number of repetitions of 'ones' in the columns for output and input memory functions, we use the number of repetitions of outputs $p(w_g)$ and the number of repetitions of next states $p(a_s)$ in the corresponding columns of Table 14. Finally, we rewrite column $t_{1n}t_{2n}t_{3n}$ into column $d_1d_2d_3$, since these columns are equal when we use D flip-flops as memory elements.

Table 15. Input encoding

z_f	x_1x_2
z_1	0 0
z_2	0 1
z_3	1 0

Table 16. Output encoding

w_g	$p(w_g)$	y_1y_2
w_1	1	0 1
w_2	2	0 0
w_3	1	1 0

Table 17. State assignment

a_s	$p(a_s)$	$t_1t_2t_3$
a_1	1	1 0 1
a_2	3	0 0 0
a_3	2	0 1 0
a_4	2	1 0 0
a_5	3	0 0 1

As abstract automaton S_2 has three inputs z_1, z_2, z_3, three outputs $w_1,\ldots,w_3$ and five states $a_1,\ldots,a_5$; the corresponding structure automaton has two binary inputs x_1, x_2, two binary outputs y_1, y_2 and three binary memory elements t_1, t_2, t_3. The basic structure for this automaton is shown in Fig. 21.

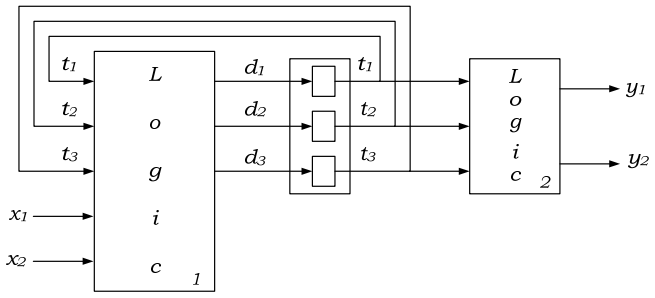


Figure 21. The structure of Moore automaton S_2

Don't cares for input memory functions d_1, d_2, d_3:

1. Codes *011, 110* and *111* are not used for state assignment, therefore the functions d_1, d_2, d_3 are not specified for the cubes

t_1	t_2	t_3	x_1	x_2
0	1	1	x	x
1	1	0	x	x
1	1	1	x	x

2. Code *11* is not used for encoding of input signals, therefore the functions d_1, d_2, d_3 are not specified for the cube

t_1	t_2	t_3	x_1	x_2
x	x	x	1	1

3. Functions d_1, d_2, d_3 are not specified for the cubes corresponding to the rows with dashes "-" in the columns for these functions

t_1	t_2	t_3	x_1	x_2
1	0	1	0	1
0	0	0	0	1
0	1	0	1	0
1	0	0	1	0

Don't care for y_1, y_2:

1. Codes *011, 110* and *111* are not used for state assignment, therefore the functions y_1, y_2 are not specified for the cubes

t1	t2	t3
0	1	1
1	1	0
1	1	1

2. Output of the automaton S_2 is not specified in the state a_2 so y_1, y_2 are not specified for the cube corresponding to the code of this state

t_1	t_2	t_3
0	0	0

Immediately from Table 14 we derive the covers for d_1, d_2, d_3, y_1, y_2:

$d_1 =$

t_1	t_2	t_3	x_1	x_2
0	1	0	0	0
0	1	0	0	1
0	0	1	1	0

$d_2 =$

t_1	t_2	t_3	x_1	x_2
1	0	1	1	0
0	0	0	1	0

$d_3 =$

t_1	t_2	t_3	x_1	x_2
1	0	1	0	0
0	1	0	0	0
1	0	0	0	1
0	0	1	0	0

$y_1 =$

t_1	t_2	t_3
1	0	0

$y_2 =$

t_1	t_2	t_3
1	0	1

Minimized covers for functions y_1, y_2, d_1, d_2, d_3 are shown in Fig. 22 – Fig.26 (here we gave only Karnaugh maps for y_1, y_2). Logic circuit of this automaton with D flip-flops and optimized output and state encoding is presented in Fig. 27.

$d_1 =$

t_1	t_2	t_3	x_1	x_2
x	1	x	x	x
0	x	1	1	x

Figure 22. Function d_1

$d_2 =$

t_1	t_2	t_3	x_1	x_2
1	x	x	1	x
x	x	0	1	x

Figure 23. Function d_2

$d_3 =$

t_1	t_2	t_3	x_1	x_2
x	x	1	0	0
x	1	x	x	0
1	x	x	x	1

Figure 24. Function d_3

t_3 \ $t_1 t_2$	00	01	11	10
0	∅		∅	1
1		∅	∅	

$y_1 =$

t_1	t_2	t_3
1	x	0

Figure 25. Function y_1

t_3 \	00	01	11	10
0	∅		∅	
1		∅	∅	1

$y_2 =$

t_1	t_2	t_3
1	x	1

Figure 26. Function y_2

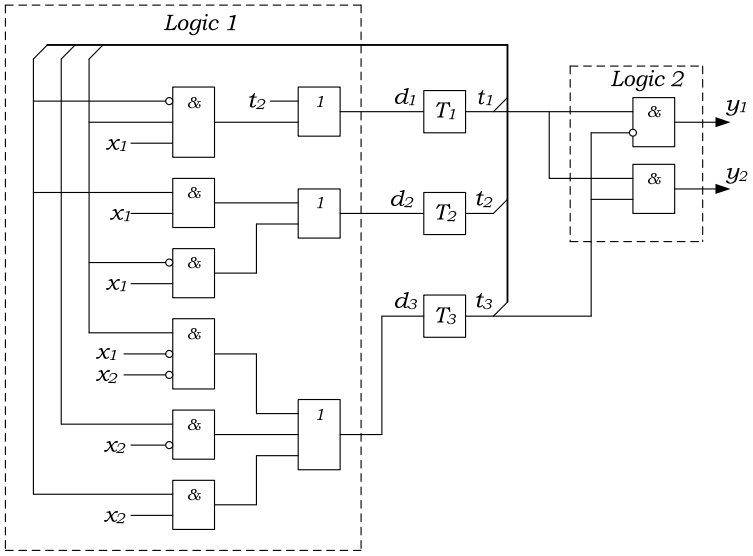


Figure 27. Logic circuit of Moore automaton S_2

3.3 Synthesis of Combined automaton

In the previous Chapter, we have introduced a *combined automaton model*. Such an automaton has the properties of Mealy and Moore automata. We call it Combined automaton (*C-automaton*). This automaton has two output functions – the first as in the Mealy model and the second – as in the Moore model. The transition table of C-automaton S_3 (Table 18) is similar to the transition table for the Mealy model, while, in the output table (Table 19), states are marked by the outputs from the set of outputs U.

Table 18. $S_3 : a_s = \delta(a_m , z_f)$

z_f	a_1	a_2	a_3	a_4	a_5
z_1	-	a_5	a_3	a_2	-
z_2	a_2	-	a_4	a_3	a_2
z_3	a_3	a_1	-	a_5	a_3

Table 19. $S_3 : w_g = \lambda_1(a_m , z_f)$; $u_f = \lambda_2 (a_m)$

	u_1	-	u_2	u_3	u_2
z_f	a_1	a_2	a_3	a_4	a_5
z_1	-	w_2	w_4	w_1	-
z_2	w_2	-	w_2	w_4	w_6
z_3	w_3	w_4	-	w_5	w_1

Without detailed comments, we will discuss the synthesis of this C-automaton.

Table 20. The structure table of combined automaton S_3

a_m	$t_1t_2t_3$	u_p	r_1r_2	a_s	$t_{1n}t_{2n}t_{3n}$	z_f	x_1x_2	w_g	$y_1y_2y_3$	$d_1d_2d_3$
a_1	100	u_1	01	-	-	z_1	00	-	-	-
	100			a_2	001	z_2	01	w_2	000	001
	100			a_3	000	z_3	10	w_3	100	000

a_2	001 001 001	-	-	a_5 - a_1	010 - 100	z_1 z_2 z_3	00 01 10	w_2 - w_4	000 - 001	010 - 100
a_3	000 000 000	u_2	00	a_3 a_4 -	000 011 -	z_1 z_2 z_3	00 01 10	w_4 w_2 -	001 000 -	000 011 -
a_4	011 011 011	u_3	10	a_2 a_3 a_5	001 000 010	z_1 z_2 z_3	00 01 10	w_1 w_4 w_5	010 001 011	001 000 010
a_5	010 010 010	u_2	00	- a_2 a_3	- 001 000	z_1 z_2 z_3	00 01 10	- w_6 w_1	- 101 010	- 001 000

Table 21. State assignment

a_m	$p(a_s)$	$t_1t_2t_3$
a_1	1	1 0 0
a_2	3	0 0 1
a_3	4	0 0 0
a_4	1	0 1 1
a_5	2	0 1 0

Table 22. Output w_g encoding

w_g	$p(w_g)$	$y_1y_2y_3$
w_1	2	0 1 0
w_2	3	0 0 0
w_3	1	1 0 0
w_4	3	0 0 1
w_5	1	0 1 1
w_6	1	1 0 1

Table 23. Output u_p encoding

u_p	$p(u_p)$	r_1r_2
u_1	1	0 1
u_2	2	0 0
u_3	1	1 0

Table 24. Input encoding

z_f	x_1x_2
z_1	0 0
z_2	01
z_3	1 0

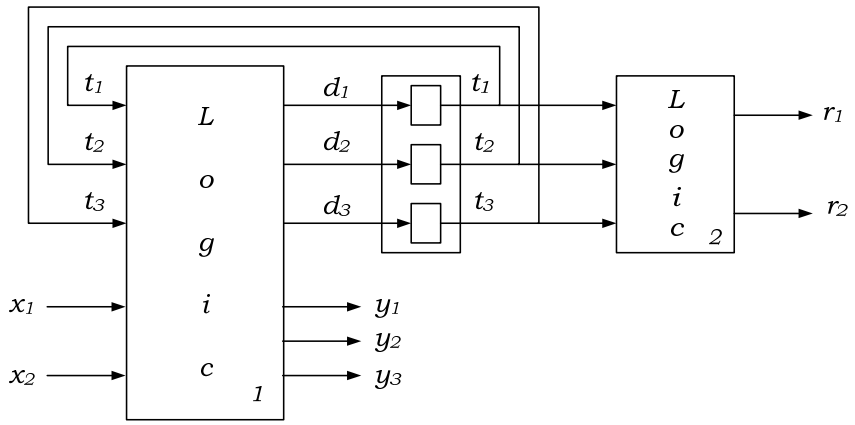


Figure 28. The structure of Combined automaton S_3

Don't cares for y1, y2, y3, d1, d2, d3

1. Codes *101, 110* and *111* are not used for state assignment, therefore functions $y_1, y_2, y_3, d_1, d_2, d_3$ are not specified for the cubes

t_1 t_2 t_3 x_1 x_2

1	*0*	*1*	*x*	*x*
1	*1*	*0*	*x*	*x*
1	*1*	*1*	*x*	*x*

2. Code *11* is not used for input encoding, functions y_1, y_2, y_3, d_1, d_2, d_3 are not specified for the cube

t_1	t_2	t_3	x_1	x_2
x	*x*	*x*	*1*	*1*

3. Functions y_1, y_2, y_3, d_1, d_2, d_3 are not specified for the cubes corresponding to the rows with dashes "-" in the columns for these functions

t_1	t_2	t_3	x_1	x_2
1	*0*	*0*	*0*	*0*
0	*0*	*1*	*0*	*1*
0	*0*	*0*	*1*	*0*
0	*1*	*0*	*0*	*0*

Don't cares for r_1, r_2

1. Codes *101, 110* and *111* are not used for state assignment, therefore the functions r_1, r_2 are not specified for the cubes

t_1	t_2	t_3
1	*0*	*1*
1	*1*	*0*
1	*1*	*1*

2. Output of automaton S_3 is not specified on state a_2 so r_1, r_2 are not specified for the cube corresponding to the code of this state

t_1	t_2	t_3
0	*0*	*1*

Initial covers for y_1, y_2, y_3, d_1, d_2, d_3

$y_1 =$

t_1	t_2	t_3	x_1	x_2
1	*0*	*0*	*1*	*0*
0	*1*	*0*	*0*	*1*

$d_1 =$

t_1	t_2	t_3	x_1	x_2
0	*0*	*1*	*1*	*0*

$y_2 =$

t_1	t_2	t_3	x_1	x_2
0	*1*	*1*	*0*	*0*
0	*1*	*1*	*1*	*0*
0	*1*	*0*	*1*	*0*

$d_2 =$

t_1	t_2	t_3	x_1	x_2
0	*0*	*1*	*0*	*0*
0	*0*	*0*	*0*	*1*
0	*1*	*1*	*1*	*0*

$y_3 =$

t_1	t_2	t_3	x_1	x_2
0	*0*	*1*	*1*	*0*
0	*0*	*0*	*0*	*0*
0	*1*	*1*	*0*	*1*

$d_3 =$

t_1	t_2	t_3	x_1
1	*0*	*0*	*0*
0	*0*	*0*	*0*
0	*1*	*1*	*0*
0	*1*	*0*	*0*

0	1	1	1	0
0	1	0	0	1

Initial covers for r_1, r_2

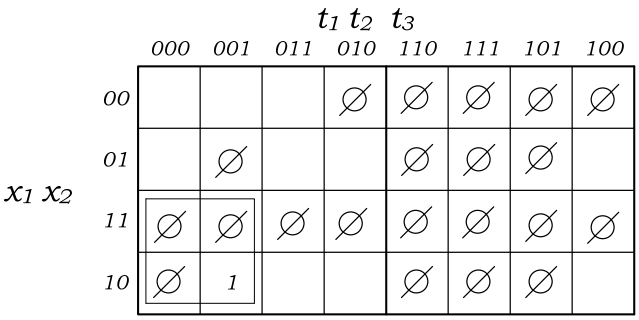
$r_1 =$

t_1	t_2	t_3
0	1	1

$r_2 =$

t_1	t_2	t_3
1	0	0

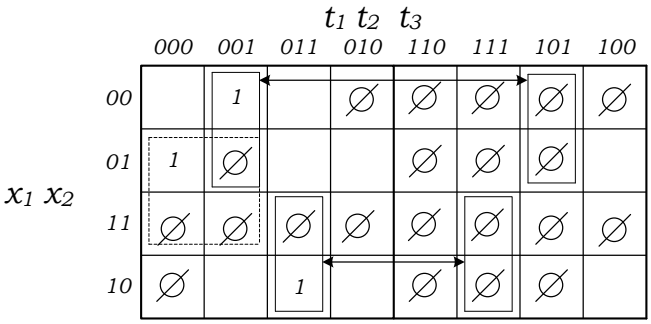
Finding minimized covers



$d_1 =$

t_1	t_2	t_3	x_1	x_2
0	0	x	1	x

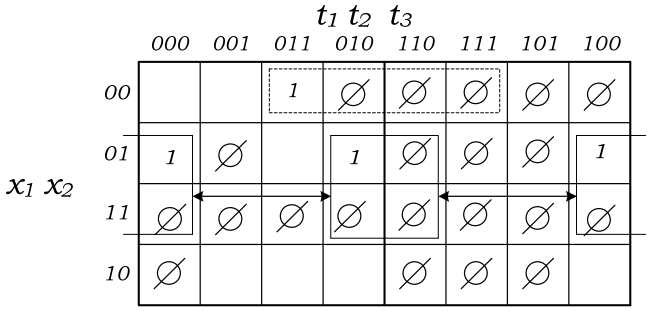
Figure 29. Function d_1



$d_2 =$

t_1	t_2	t_3	x_1	x_2
x	0	1	0	x
0	0	x	x	1
x	1	1	1	x

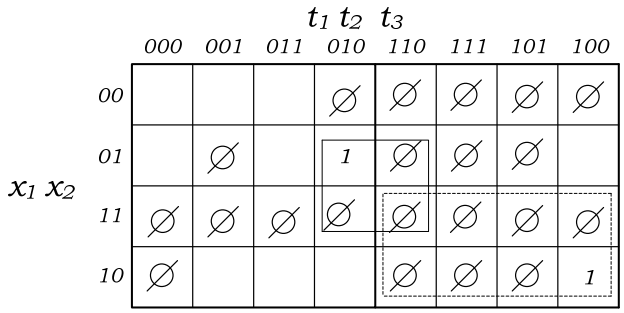
Figure 30. Function d_2



$d_3 =$

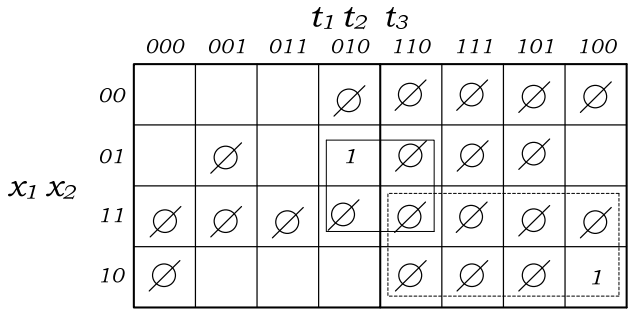
t_1	t_2	t_3	x_1	x_2
x	x	0	x	1
x	1	x	0	0

Figure 31. Function d_3



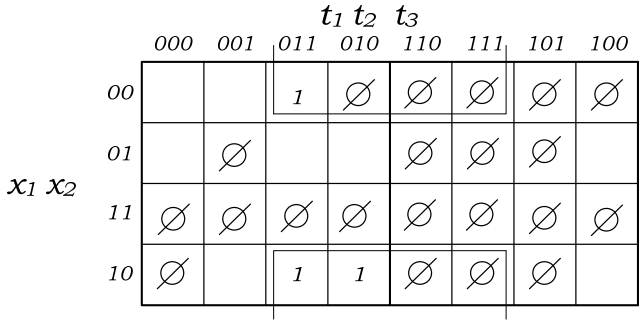
$$y_1 = \begin{array}{|ccccc|} t_1 & t_2 & t_3 & x_1 & x_2 \\ 1 & x & x & 1 & x \\ x & 1 & 0 & x & 1 \end{array}$$

Figure 32. Function y_1



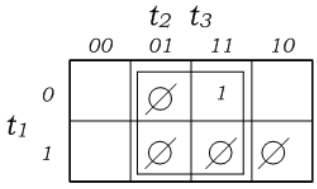
$$y_2 = \begin{array}{|ccccc|} t_1 & t_2 & t_3 & x_1 & x_2 \\ x & 1 & x & x & 0 \end{array}$$

Figure 33. Function y_2



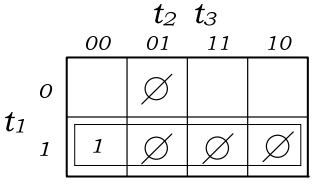
$$y_3 = \begin{array}{|ccccc|} t_1 & t_2 & t_3 & x_1 & x_2 \\ x & x & 1 & 1 & x \\ x & 1 & x & x & 1 \\ x & x & 0 & 0 & 0 \end{array}$$

Figure 34. Function y_3



$$r_1 = \begin{array}{|ccc|} t_1 & t_2 & t_3 \\ x & x & 1 \end{array}$$

Figure 35. Function r_1



$$r_2 = \begin{array}{|ccc|} t_1 & t_2 & t_3 \\ 1 & x & x \end{array}$$

Figure 36. Function r_2

Logic circuit

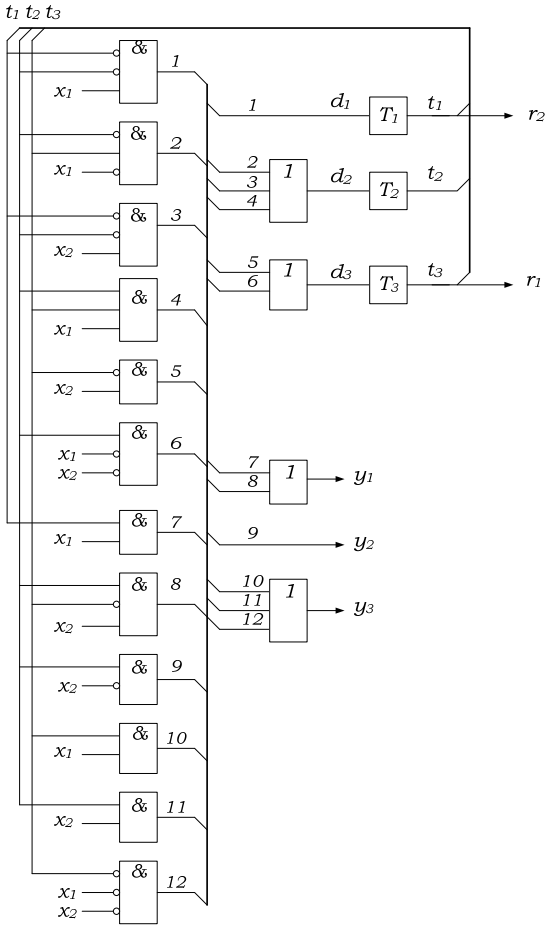


Figure 37. Logic circuit of Combined automaton S_3 with D flip-flops

Chapter 4 Algorithmic State Machines and Finite State Machines

In this Chapter, we will introduce Algorithmic state machines and consider their use for description of the behavior of control units. Next, we will use algorithmic state machines to design Finite State Machines (FSM) with hardly any constraints on the number of inputs, outputs and states.

4.1 Flowcharts and Algorithmic state machines

4.1.1 Example of ASM. An Algorithmic state machine (ASM) is the directed connected graph containing an initial vertex (Begin), a final vertex (End) and a finite set of operator and conditional vertices (Fig. 1). The final, operator and conditional vertices have only one input, the initial vertex has no input. Initial and operator vertices have only one output, a conditional vertex has two outputs marked by "1" and "0". A final vertex has no outputs.

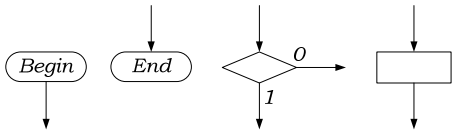


Figure 1. Vertices of Algorithmic state machine

As the first example, let us consider a very simple Traffic Light Controller (TLC) presented in the flowchart in Fig. 2. This controller is at the intersection of a main road and a secondary road. Immediately after vertex *Begin* we have a waiting vertex (one of the outputs of this vertex is connected to its input) with a logical condition *Start*. It means that the controller begins to work only when signal *Start = 1*. At this time, cars can move along the main road for two minutes. For that, the traffic light at the main road is green, the traffic light at the secondary road is red and the special timer that counts seconds is set to zero *(main_grn := 1; sec_red := 1; t := 0).*

Although our TLC is very simple it is also a little smart – it can recognize an ambulance on the road. When an ambulance is on the road the signal *amb* is equal to one (*amb = 1*), when there is no ambulance on the road this signal is equal to zero (*amb = 0*). First we will discuss the case when there are no ambulances on the road.

Thus, when *amb = 0* and *t = 120 sec* TLC transits into some intermediate state to allow cars to finish driving along the main road: *main_yel := 1; sec_red := 1; t := 0.* TLC is in this state only for three seconds (*t = 3 sec*), after which cars can move along the secondary road for 30 seconds: *main_red := 1; sec_grn := 1; t := 0.*

Thirty seconds later, if there are no ambulances on the road (*amb = 0; t = 30 sec*), there is one more intermediate state. Now cars must finish driving along the secondary road: *main_red := 1; sec_yel := 1; t := 0.* After three seconds, if, once again, there are no ambulances on the road, the process reaches vertex End, or, that is the same, it returns to the beginning vertex *Begin*.

When there is ambulance on the road (*amb = 1*) outputs of conditional vertices with logical condition *amb*, marked by “1” bring us to the intermediate state to let cars to finish their driving: *main_yel := 1; sec_yel := 1; t := 0.* One more logical condition *dmain* tells us where the ambulance is – whether it is on the main road or on the secondary one. If it is on the main road (*dmain = 1*), after three seconds the traffic

light will be green on the main road, otherwise (*dmain = 0*) the traffic light will be green on the secondary road.

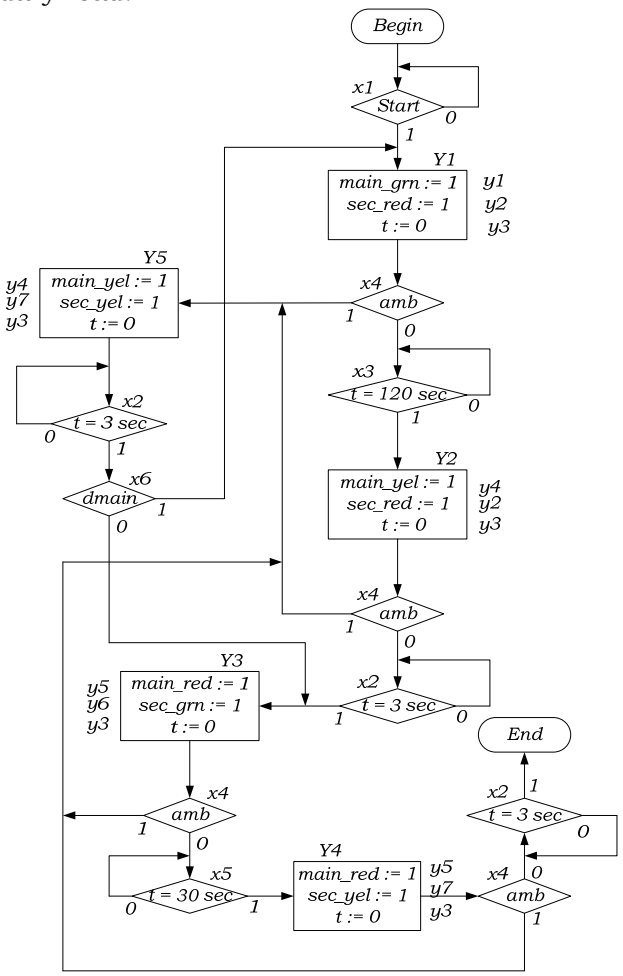


Figure 2. A simple Traffic Light Controller

In the flowchart, a logical condition is written in each conditional vertex. It is possible to write the same logical condition in different conditional vertices. A microinstruction (an operator), containing one, two, three or more microoperations, is written in each operator vertex of the flowchart. It is possible to write the same operator in different operator vertices.

If we replace logical conditions by $x_1, x_2, \dots, x_L$, microoperations by $y_1, y_2, \dots, y_N$ and operators by $Y_1, Y_2, \dots, Y_T$ we will get Algorithmic State Machine (ASM). ASM for the flowchart in Fig. 2 is shown in Fig. 3.

ASM vertices are connected in such a way that:

1. Inputs and outputs of the vertices are connected by arcs directed from an output to an input, each output is connected with only one input;
2. Each input is connected with at least one output;
3. Each vertex is located on at least one of the paths from vertex “Begin” to vertex “End”. Hereinafter we will not consider ASMs with subgraphs,

containing an infinite cycle. An example of such a subgraph with an infinite loop between vertices with Y_1 and Y_3 is shown in Fig. 4. The dots in this ASM between vertex *"Begin"* and the conditional vertex with x_1 and between this vertex and vertex *"End"* mean that ASM has other vertices on the path from vertex *"Begin"* to vertex *"End"*. The vertices in the loop are not on the path from *"Begin"* to *"End"*.

4. One of the outputs of a conditional vertex can be connected with its input. We will call such conditional vertices the "*waiting vertices*", since they simulate the waiting process in the system behavior description.

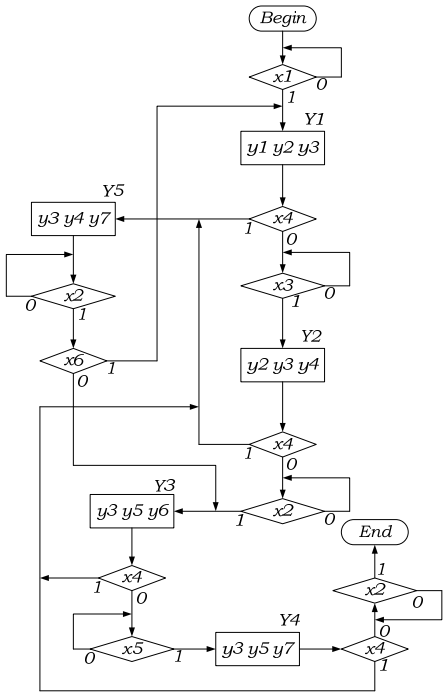


Figure 3. ASM for the flowchart in Fig. 2

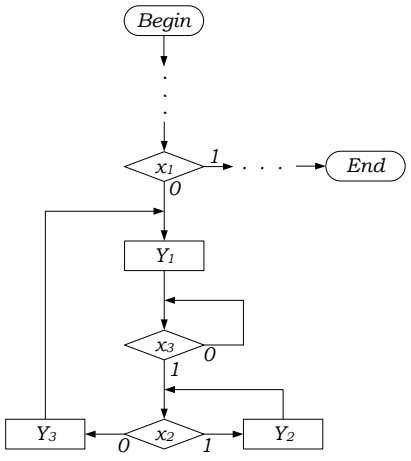


Figure 4. Subgraph with an infinite loop

One more example of ASM G_1 with logical conditions $X = \{x_1, \ldots, x_7\}$ and microoperations $Y = \{y_1, \ldots, y_{10}\}$ is shown in Fig. 5. This ASM has eight operators $Y_1, \ldots, Y_8$, they are written near operator vertices.

4.1.2 Transition functions. Let us discuss the paths between the vertex "Begin", the vertex "End" and operator vertices passing only through conditional vertices. We will write such paths as follows:

$$Y_i \tilde{x}_{i1} \ldots \tilde{x}_{iR} Y_j \qquad (1)$$

In such a path, $\tilde{x}_{ir}$ is equal to x_{ir} if the path proceeds from the conditional vertex with x_{ir} via output *'1'*, and $\tilde{x}_{ir}$ is equal to x'_{ir} if the path proceeds from the conditional vertex with x_{ir} via output *'0'*. For example, we have the following paths from Y_b (vertex *Begin*) in ASM G_1:

$Y_b\ x'_1\ Y_2;$
$Y_b\ x_1 x_2 x'_3\ Y_6;$
$Y_b\ x_1 x'_2\ Y_1;$
$Y_b\ x_1 x_2 x_3\ Y_5.$

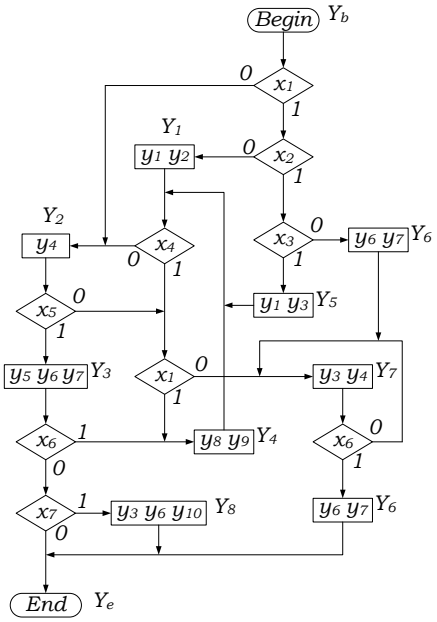


Figure 5. ASM G_1

Let us match a product of variables in the path (1) from operator vertex Y_i to operator vertex Y_j

$$\alpha_{ij} = \tilde{x}_{i1} \ldots \tilde{x}_{iR}$$

with this path from Y_i to Y_j. For example, for ASM G_1 in Fig. 5

$a_{17} = x_4\, x'_1;$ $\quad a_{12} = x'_4;$ $\quad a_{14} = x_4\, x_1.$

If there exist H paths between Y_i and Y_j through the conditional vertices, then

$$a_{ij} = a^1_{ij} + a^2_{ij} + \ldots + a^H_{ij}$$

where a^h_{ij} ($h = 1, \ldots, H$) is the product for the h-th path. Let us call a_{ij} a transition function from operator (microinstruction) Y_i to operator (microinstruction) Y_j.

Note that for the path Y_6Y_7 (operator Y_7 follows operator Y_6 immediately without conditional vertices) $a_{67} = 1$, as the product of an empty set of variables is equal to one.

4.1.3 Value of ASM at the sequence of vectors. Denote all possible L-component vectors of the logical conditions $x_1, \ldots, x_L$ by $\Delta_1, \ldots, \Delta_2{}^L$ and define the execution of an ASM on any given sequence of vectors $\Delta_1, \ldots, \Delta_{mq}$ beginning from the initial operator Y_b. We will demonstrate this procedure by means of ASM G_1 in Fig. 5 and the sequence (2) containing eight vectors $\Delta_1, \ldots, \Delta_8$:

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
Δ_1	=	1	0	1	0	1	1	1
Δ_2	=	0	1	1	0	1	0	0
Δ_3	=	1	0	1	0	0	1	0
Δ_4	=	0	1	0	0	0	0	1
Δ_5	=	1	1	0	1	1	1	0
Δ_6	=	1	1	0	0	1	0	1
Δ_7	=	0	1	1	1	0	0	0
Δ_8	=	0	1	0	1	0	0	1

(2)

ASM G_1 in Fig. 5 contains logical variables $x_1,\ldots,x_7$ and operators $Y_b, Y_1, \ldots, Y_8, Ye$. Now let us find the sequence of operators which would be implemented, if we consecutively, beginning from Y_b, give variables the values from these vectors. We suppose that the values of logical conditions can be changed only during an execution of operators.

Step 1. Write the initial operator

$$Y_b.$$

Step 2. Let logical variables $x_1,\ldots,x_7$ take their values from vector Δ_1. From the set of the transition functions $a_{b1},\ldots, a_{b8}, a_{be}$ we choose such a function a_{bt} that $a_{bt}(\Delta_1) = 1$. In our example for the operator Y_b, the following transition functions are not identically equal to zero:

$$a_{b5} = x_1 x_2 x_3; \qquad a_{b6} = x_1 x_2 x'_3; \qquad a_{b1} = x_1 x'_2; \qquad a_{b2} = x'_1.$$

We will call such functions *non-trivial transition functions* to distinguish them from the trivial functions, which are identically equal to zero. Function a_{ij} is *trivial* if there is no path from operator Y_i to operator Y_j. In the example at this step, we choose the function a_{b1}, since only a_{b1} is equal to one on the first vector Δ_1:

$$a_{b1}(\Delta_1) = 1.$$

Write Y_1 to the right of Y_b:

$$Y_b Y_1.$$

Step 3. Let $x_1,\ldots,x_7$ take their values from vector Δ_2. From the set of the transition functions $a_{11},\ldots, a_{18}, a_{1e}$ we choose non-trivial functions

$$a_{14} = x_4 x_1; \qquad a_{17} = x_4 x'_1; \qquad a_{12} = x'_4$$

and among them – the only function $a_{12}(\Delta_2) = 1$. Write Y_2 to the right of $Y_b Y_1$:

$$Y_b Y_1 Y_2.$$

The computational process for the given sequence of vectors may reach its end in two cases:

1. The final vertex "End" is reached. In this case, the last operator is Y_e. The number of operators in the operator row (without Y_b and Y_e) is less or equal (if we reached the final vertex with the last vector) to the number of vectors;
2. The vectors are exhausted but we have not yet reached the final vertex. In this case, the number of operators in the operator row is equal to the number of vectors.

In our example, we reached the final vertex "End" at the seventh vector

$$\Delta_7 = 0\ 1\ 1\ 1\ 0\ 0\ 0$$

and we get the row

$$Y_b\ Y_1\ Y_2\ Y_4\ Y_2\ Y_3\ Y_8\ Y_e. \tag{3}$$

The operator row thus obtained is *the value of the ASM G_1 for the given sequence of vectors (2).*

4.2 Synthesis of Mealy FSM

We will use Algorithmic state machines to describe the behavior of digital systems, mainly of their control units. But if we must construct a logic circuit of the control unit we should use a Finite state machine (FSM). We will consider methods of synthesis of FSM Mealy, Moore and their combined model implementing a given ASM, with hardly any constrains on the number of inputs, outputs and states.

4.2.1 Construction of a marked ASM. As an example we will use ASM G_1 in Fig. 6. A Mealy FSM implementing given ASM may be constructed in two stages:

Stage1. Construction of a marked ASM;

Stage 2. Construction of a state diagram (state graph).

At the first stage, the inputs of vertices following operator vertices are marked by symbols $a_1, a_2, \ldots, a_M$ as follows:

1. Symbol a_1 marks the input of the vertex following the initial vertex *"Begin"* and the input of the final vertex *"End"*;
2. Symbols $a_2, \ldots, a_M$ mark the inputs of all vertices following operator vertices;
3. Vertex inputs are marked only once;
4. Inputs of different vertices, except the final one, are marked by different symbols.

Marked ASM G_1 in Fig. 6 is a result of the first step. Symbols $a_1, \ldots, a_6$ are used to mark this ASM. Note, that we mark the inputs not only of conditional vertices but

of operator vertices as well (see mark a_3 at the input of the vertex with operator Y_7). It is important that each marked vertex follows an operator vertex.

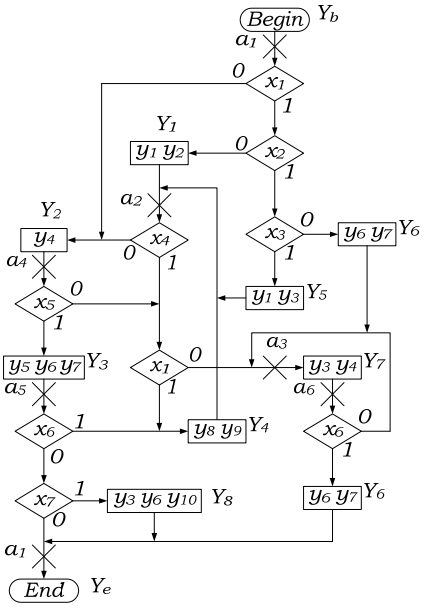


Figure 6. ASM G_1 marked for the Mealy FSM synthesis

4.2.2 Transition Paths. At the second stage, we will consider the following paths in the marked ASM:

$$a_m \tilde{x}_{m1} \ldots \tilde{x}_{mR_m} Y_g a_s \qquad (P1)$$

$$a_m \tilde{x}_{m1} \ldots \tilde{x}_{mR_m} a_1 \qquad (P2)$$

We call these paths *transition paths*. Thus, the path *P1* proceeds from a_m to a_s ($a_m = a_s$ is also allowed) and contains only one operator vertex at the end of this path. The path *P2* proceeds from a_m only to a_1 without operator vertex. Here, $\tilde{x}_{mr} = x_{mr}$, if on the transition path we leave the conditional vertex with x_{mr} via output '1' and $\tilde{x}_{mr} = x'_{mr}$ if we leave it via output '0'. If $R_m = 0$ on the path *P1,* two operator vertices follow one after another and this path turns into

$$a_m Y_g a_s .$$

There are sixteen transition paths in the marked ASM G_2 in Fig. 6:

$a_1\ x_1x_2x_3\ Y_5\ a_2$	$a_2\ x_4x_1\ Y_4\ a_2$	$a_4\ x_5\ Y_3\ a_5$	$a_5\ x'_6x_7\ Y_8\ a_1$
$a_1\ x_1x_2x'_3\ Y_6\ a_3$	$a_2\ x_4x'_1\ Y_7\ a_6$	$a_4\ x'_5x_1\ Y_4\ a_2$	$a_5\ x'_6x'_7\ a_1$
$a_1\ x_1x'_2\ Y_1\ a_2$	$a_2\ x'_4\ Y_2\ a_4$	$a_4\ x'_5x'_1\ Y_7\ a_6$	$a_6\ x_6\ Y_6\ a_1$
$a_1\ x'_1\ Y_2\ a_4$	$a_3\ Y_7\ a_6$	$a_5\ x_6\ Y_4\ a_2$	$a_6\ x'_6\ Y_7\ a_6$

Note, that the path $a_2\ x_4x'_1\ a_3$ doesn't correspond to the transition path *P1* (the operator vertex is absent on the path) and to transition path *P2* (it isn't a path to a_1). Thus, it isn't a transition path and we should go on to get the path $a_2\ x_4x'_1\ Y_7\ a_6$. For the same reason, paths $a_4\ x'_5x'_1\ a_3$ and $a_6\ x'_6\ a_3$ are not the transition paths either.

4.2.3 Graph of FSM. Next we construct a graph (state diagram) of FSM Mealy with states (marks) $a_1, \ldots, a_M$, obtained at the first stage. We have six such states $a_1, \ldots, a_6$ in our example. Thus, the FSM graph contains as many states as the number of marks we get at the previous stage. Now we should define transitions between these states.

FSM has a transition from state a_m to state a_s with input $X(a_m, a_s)$ and output Y_g (see the upper subgraph in Fig. 7) if, in ASM, there is transition path *P1*

$$a_m \tilde{x}_{m1} \ldots \tilde{x}_{mR_m} Y_g a_s .$$

Here $X(a_m, a_s)$ is the product of logical conditions written in this path:

$$X(a_m, a_s) = \tilde{x}_{m1} \ldots \tilde{x}_{mRm} .$$

In exactly the same way, for the path $a_m Y_g a_s$ we have a transition from state a_m to state a_s with input $X(a_m, a_s) = 1$ and output Y_g, as the product of an empty set of variables is equal to zero. If, for a certain r $(r = 1, \ldots, R_m)$, symbol x_{mr} (or x'_{mr}) occurs several times on the transition path, all symbols x_{mr} (x'_{mr}) but one are deleted; if for a certain r $(r = 1, \ldots, R_m)$, both symbols x_{mr} and x'_{mr} occur on the transition path, this path is removed. In such a case $X(a_m, a_s) = 0$.

For the second transition path *P2*, FSM transits from state a_m to the initial state a_1 with input $X(a_m, a_1)$ and output Y_0 (see the lower subgraph in Fig. 7). Y_0 is the operator containing an empty set of microoperations.

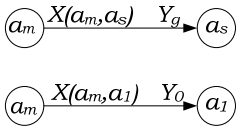


Figure 7. Subgraphs for transition paths *P1* and *P2*

As a result, we obtain a Mealy FSM with as many states as the number of marks we used to mark the ASM in Fig. 6. The state diagram of the Mealy FSM is shown in Fig. 8.

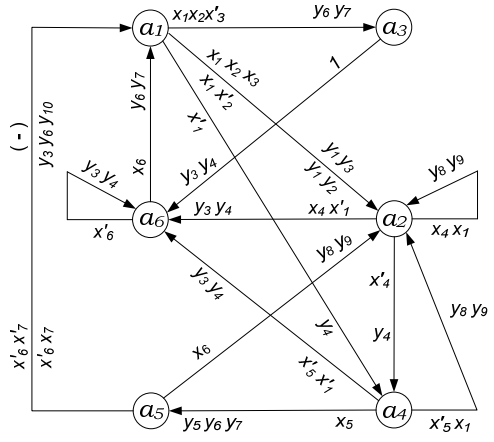


Figure 8. The state diagram of the Mealy FSM

4.2.4 How not to loose transition paths. Sometimes, if ASM contains many conditional vertices, it is difficult not to loose one or several transition paths. Here we give a very simple algorithm to resolve this problem. This algorithm has only two steps.

1. Find the first transition path leaving each conditional vertex through output '*1*'. For subgraph of ASM in Fig. 9 we will get the following first path from state a_2:

$$a_2\ x_1\ x_2\ x_5\ Y_6\ a_3.$$

2. Invert the *last non-inverted* variable in the previous path, return to ASM and continue the path (if it is possible) leaving each conditional vertex through output '*1*'. To construct the second path, we should invert variable x_5. We cannot continue because we reached an operator vertex:

$$a_2\ x_1\ x_2\ x'_5\ Y_2\ a_3.$$

We should construct paths in the same manner until all variables in a transition path will be inverted. For our example, we will get the following paths:

$a_2\ x_1\ x'_2\ x_5\ x_6\ Y_3\ a_4;$
$a_2\ x_1\ x'_2\ x_5\ x'_6\ x_7\ x_4\ Y_5\ a_5;$
$a_2\ x_1\ x'_2\ x_5\ x'_6\ x_7\ x'_4\ Y_7\ a_4;$
$a_2\ x_1\ x'_2\ x_5\ x'_6\ x'_7\ Y_5\ a_5;$

$a_2\ x_1\ x'_2\ x'_5\ Y_2\ a_3;$
$a_2\ x'_1\ x_3\ x_7\ x_4\ Y_5\ a_5;$
$a_2\ x'_1\ x_3\ x_7\ x'_4\ Y_7\ a_4;$
$a_2\ x'_1\ x_3\ x'_7\ Y_5\ a_5;$

$a_2\ x'_1\ x'_3\ x_6\ Y_3\ a_4;$
$a_2\ x'_1\ x'_3\ x'_6\ x_7\ x_4\ Y_5\ a_5;$
$a_2\ x'_1\ x'_3\ x'_6\ x_7\ x'_4\ Y_7\ a_4;$
$a_2\ x'_1\ x'_3\ x'_6\ x'_7\ Y_5\ a_5.$

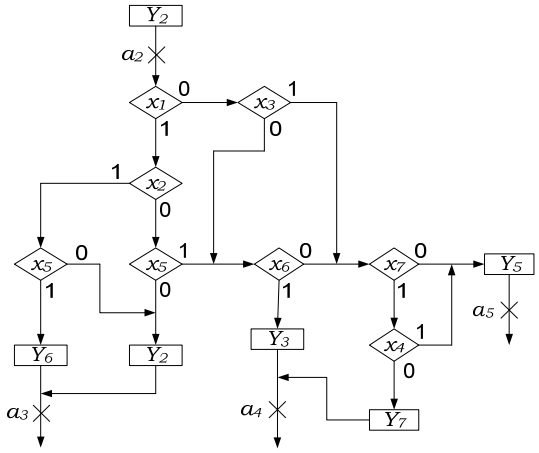


Figure 9. Subgraph of ASM

4.2.5 Transition tables of Mealy FSM. The graph of Mealy FSM in Fig. 8 has only 6 states and 16 arcs. Practically, however, we must construct FSMs with tens of states and more than one-two hundreds of transitions. In such a case, it is difficult to use a graph, so we will present it as a table. Table 1 for the same Mealy FSM has five columns:

- a_m – a current state;
- a_s – a next state;
- $X(a_m,a_s)$ – an input signal;
- $Y(a_m,a_s)$ – an output signal;

- H – a number of line.

Actually, immediately from ASM, we should write transition paths, one after another, into the transition table. In Table 1, $\sim x_t$ is used instead of x'_t for the inversion of x_t.

Now we will discuss what kind of FSM we have received. Our ASM G_1 in Fig. 6 which we used to construct FSM S_1 in Table 1, has seven logical conditions and ten microoperations. FSM S_1 has seven binary inputs in the column $X(a_m,a_s)$ and ten binary outputs in the column $Y(a_m,a_s)$. The input signal of this FSM (Fig. 10) is the 7-component vector, the output signal of this FSM is the 10-component vector.

Table 1. Direct transition table of Mealy FSM S_1

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a_1	a_2	$x_1x_2x_3$	y_1y_3	1
	a_3	$x_1x_2\sim x_3$	y_6y_7	2
	a_2	$x_1\sim x_2$	y_1y_2	3
	a_4	$\sim x_1$	y_4	4
a_2	a_2	x_4x_1	y_8y_9	5
	a_6	$x_4\sim x_1$	y_3y_4	6
	a_4	$\sim x_4$	y_4	7
a_3	a_6	1	y_3y_4	8
a_4	a_5	x_5	$y_5y_6y_7$	9
	a_2	$\sim x_5x_1$	y_8y_9	10
	a_6	$\sim x_5\sim x_1$	y_3y_4	11
a_5	a_2	x_6	y_8y_9	12
	a_1	$\sim x_6x_7$	$y_3y_6y_{10}$	13
	a_1	$\sim x_6\sim x_7$	-	14
a_6	a_1	x_6	y_6y_7	15
	a_6	$\sim x_6$	y_3y_4	16

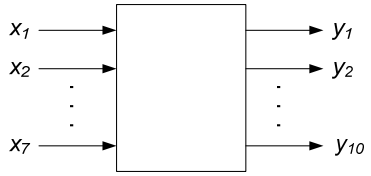


Figure 10. FSM as a black box

Let us take one of the rows from Table 1, for example row 3, and look at the behavior of FSM presented in this row. Our FSM transits from state a_1 into state a_2 when the product $x_1\,x'_2 = 1$. It is clear that such a transition takes place for any input vector in which the first component is equal to 1, the second component is equal to 0. The values of other components are not important. Thus, we can say that the third row of Table 1 presents transitions from a_1 with any vector which is covered by cube $10xxxxx$. In other words, this row presents not one but $2^5 = 32$ transitions. In exactly the same way, the first and the second row present 16 transitions, the fourth row – 64 transitions and the eighth row – 128 transitions.

Two microoperations y_1, y_2, written in the third row of the output column, mean that two components y_1 and y_2 are equal to 1 and others are equal to 0 ($y_1 = y_2 = 1$; $y_3 = y_4 = \ldots = y_{10} = 0$) in the output vector. I remind you that if the operator, written in the operator vertex of some ASM, contains microoperations y_m, y_n, only these microoperations are equal to 1 and other microoperations are equal to 0 during implementation of this operator.

Let us compare Table 1 with a classical FSM representation in Table 3.7 from Chapter 3. If we would like to present our FSM with six states a_1, …, a_6 and seven inputs x_1, …, x_7 in the classical table, this table will have about $6x2^7$ rows, because each row of this table describes only one FSM transition. In our Table 1 from this Chapter, we have only 16 rows because each row of such table presents lot of transitions.

The specific feature of such FSM is the multiplicity of inputs in the column $X(a_m,a_s)$, maybe several tens or even hundreds, but each product in one row contains only few variables from the whole set of input variables – as a rule, not more than 8 – 10 variables. It means that each time the values of the output variables depend only on the values of a small number of the input variables. Really, if, for example, FSM has 30 input variables, the total number of input vectors is equal to 2^{30}, and if each time the values of the output variables depended on the values of all the input variables, no designer could either describe or construct such an FSM.

Let us briefly discuss the correspondence between FSM S_1 (Table 1) and ASM G_1 (Fig. 6) which we used to construct FSM S_1. In Section 4.1.3 we got the value of ASM G_1

$$Y_b\, Y_1\, Y_2\, Y_4\, Y_2\, Y_3\, Y_8\, Y_e$$

for some random sequence of vectors (2) of logical conditions:

		x_1	x_2	x_3	x_4	x_5	x_6	x_7
Δ_1	=	1	0	1	0	1	1	1
Δ_2	=	0	1	1	0	1	0	0
Δ_3	=	1	0	1	0	0	1	0
Δ_4	=	0	1	0	0	0	0	1
Δ_5	=	1	1	0	1	1	1	0
Δ_6	=	1	1	0	0	1	0	1
Δ_7	=	0	1	1	1	0	0	0
Δ_8	=	0	1	0	1	0	0	1

Now we will find the response of FSM S_1 in the initial state a_1 to the same sequence of input vectors:

State sequence	a_1	a_2	a_4	a_2	a_4	a_5	a_1
Input sequence	Δ_1	Δ_2	Δ_3	Δ_4	Δ_5	Δ_6	
Response	y_1y_2	y_4	y_8y_9	y_4	$y_5y_6y_7$	$y_3y_6y_{10}$	(4)
Microinstructions	Y_1	Y_2	Y_4	Y_2	Y_3	Y_8	

Let FSM be in the initial state a_1 with the first vector $\Delta_1 = 1010111$ at its input. To determine the next state and the output we should find such a row in the array of transitions from a_1 (Table 1) that the product $X(a_m,a_s)$, written in this row, be equal to one at input vector Δ_1. Since $x_1x'_2(\Delta_1) = 1$ (the third row), FSM S_1 produces output signal $y_1y_2 = Y_1$ and transits into state a_2. Similarly, we find that $x'_4(\Delta_2)$ is equal to one at one of transitions from state a_2 and FSM transits to the state a_4 with the output signal $y_4 = Y_2$ (see row 7 in Table 1) etc. As a result, we get the response of FSM S_1 in the initial state a_1 to the input sequence Δ_1, …, Δ_6 in the fourth row of sequence (4).

As seen from this row, the FSM response is equal to the value of ASM G_1 for the same input sequence. Note, that we consider here only the FSM response until its return to the initial state a_1 and this response $Y_1\, Y_2\, Y_4\, Y_2\, Y_3\, Y_8$ corresponds to the value of ASM G_1 between the operator Y_b (vertex "Begin") and the operator Y_e (vertex "End").

Let us define *FSM S as implementing ASM G* if the response of this FSM in the state a_1 to any input sequence (until its return to the state a_1) is equal to the value of ASM G for the same input sequence. From the considered method of synthesis of Mealy FSM S_1 from ASM G_1 it follows that this FSM S_1 implements ASM G_1.

4.2.6 Synthesis of Mealy FSM logic circuit. As in Chapter 3, we will construct a Mealy FSM logic circuit with the structure presented in Fig. 11. To design this circuit we will use an FSM structure table (Table 2). This table was constructed from the direct transition table (Table 1) by adding three additional columns:

- $K(a_m)$ – a code of the current state;
- $K(a_s)$ – a code of the next state;
- $D(a_m,a_s)$ – an input memory function.

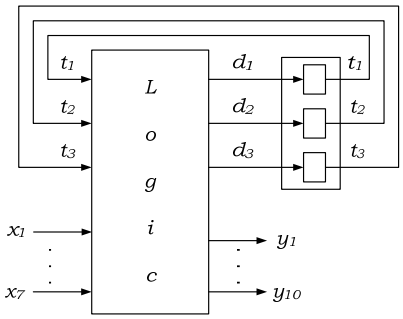


Figure 11. The structure for the Mealy FSM logic circuit

Table 2. Structure table of FSM S_1

a_m	$K(a_m)$	a_s	$K(a_s)$	$X(a_m,a_s)$	$Y(a_m,a_s)$	$D(a_m,a_s)$	H
a_1	001	a_2	000	$x_1x_2x_3$	y_1y_3	–	1
		a_3	101	$x_1x_2\sim x_3$	y_6y_7	d_1d_3	2
		a_2	000	$x_1\sim x_2$	y_1y_2	–	3
		a_4	010	$\sim x_1$	y_4	d_2	4
a_2	000	a_2	000	x_4x_1	y_8y_9	–	5
		a_6	100	$x_4\sim x_1$	y_3y_4	d_1	6
		a_4	010	$\sim x_4$	y_4	d_2	7
a_3	101	a_6	100	1	y_3y_4	d_1	8
a_4	010	a_5	110	x_5	$y_5y_6y_7$	d_1d_2	9
		a_2	000	$\sim x_5x_1$	y_8y_9	–	10
		a_6	100	$\sim x_5\sim x_1$	y_3y_4	d_1	11
a_5	110	a_2	000	x_6	y_8y_9	–	12
		a_1	001	$\sim x_6x_7$	$y_3y_6y_{10}$	d_3	13
		a_1	001	$\sim x_6\sim x_7$	-	d_3	14
a_6	100	a_1	001	x_6	y_6y_7	d_3	15
		a_6	100	$\sim x_6$	y_3y_4	d_1	16

To encode FSM states we constructed Table 3 where $p(a_s)$ is the number of appearances of each state in the next state column a_s in Table 2. The algorithm for state assignment is absolutely the same as in Chapter 3. First, we use the zero code for state a_2 with max $p(a_2) = 5$. Then codes with one '1' are used for states a_6, a_1, a_4 with the next max appearances and, finally, two codes with two 'ones' are used for the left states a_3 and a_5.

To fill column $D(a_m,a_s)$ it is sufficient to write there column $K(a_s)$ because the input of D flip-flop is equal to its next state. However, here we use the same notation as in column $Y(a_m,a_s)$ and write d_r in the column $D(a_m,a_s)$ if d_r is equal to 1 at the corresponding transition (a_m,a_s) – equal to 1 in column $K(a_s)$. After that, the shaded part of Table 2 is something like a truth table with input variables $t_1, t_2, t_3, x_1, \ldots, x_7$ in the columns $K(a_m)$ and $X(a_m,a_s)$ and output variables (functions) $y_1, \ldots, y_{10}, d_1, d_2, d_3$ in the columns $Y(a_m,a_s)$ and $D(a_m,a_s)$.

Table 3. State assignment

a_s	$p(a_s)$	$t_1\ t_2\ t_3$
a_1	3	0 0 1
a_2	5	0 0 0
a_3	1	1 0 1
a_4	2	0 1 0
a_5	1	1 1 0
a_6	4	1 0 0

Let A_m be a product, corresponding to the state code $K(a_m)$, and X_h be the product of input variables, written in the column $X(a_m,a_s)$ in the h row. For example, from the column $K(a_m)$: $K(a_1) = 001$, then $A_1 = t'_1t'_2t_3$; $K(a_2) = 000$, then $A_2 = t'_1t'_2t'_3$; $K(a_3) = 101$, then $A_3 = t_1t'_2t_3$ etc. Immediately from the column $X(a_m,a_s)$ we get:

$$X_1 = x_1x_2x_3;\ X_2 = x_1x_2x'_3;\ X_6 = x_4x'_1;\ X_8 = 1;\ X_{16} = x'_6.$$

We call the term

$$e_h = A_m X_h$$

the product corresponding to the h row of the FSM structure table if a_m is the current state in this row. For example, from Table 2, we get:

$$\begin{aligned} e_1 &= t'_1t'_2t_3\ x_1x_2x_3; \\ e_2 &= t'_1t'_2t_3\ x_1x_2x'_3; \\ e_6 &= t'_1t'_2t'_3\ x_4x'_1; \\ e_8 &= t_1t'_2t_3; \\ e_{16} &= t_1t'_2t'_3\ x'_6. \end{aligned}$$

Let $H(y_n)$ is the set of rows with y_n in the column $Y(a_m,a_s)$. Then, as in the truth table:

$$y_n = \sum_{h \in H(y_n)} e_h.$$

For example, y_6 is written in rows 2, 9, 13, 15 in the column $Y(a_m,a_s)$. Then

$$y_6 = e_2 + e_9 + e_{13} + e_{15} = t'_1t'_2t_3x_1x_2x'_3 + t'_1t_2t'_3\ x_5 + t_1t_2t'_3\ x'_6x_7 + t_1t'_2t'_3\ x_6.$$

In exactly the same way, if $H(d_r)$ is the set of rows with d_r in the column $D(a_m,a_s)$, then

$$d_r = \sum_{h \in H(d_r)} e_h.$$

For example, d_2 is written in rows 4, 7, 9 in the column $D(a_m,a_s)$. Then

$$d_2 = e_4 + e_7 + e_9 = t'_1t'_2t_3\ x'_1 + t'_1t'_2t'_3\ x'_4 + t'_1t_2t'_3\ x_5.$$

Thus, immediately from Table 1 we can get expressions for outputs of circuit *"Logic"* in Fig. 11:

$$y_1 = e_1 + e_3 = t'_1t'_2t_3\, x_1x_2x_3 + t'_1t'_2t_3\, x_1x'_2;$$
$$y_2 = e_3 = t'_1t'_2t_3\, x_1x'_2;$$

$$y_3 = e_1 + e_6 + e_8 + e_{11} + e_{13} + e_{16} = t'_1t'_2t_3\, x_1x_2x_3 + t'_1t'_2t'_3\, x_4x'_1 + t_1t'_2t_3 +$$
$$+ t'_1t_2t'_3\, x'_5x'_1 + t_1t_2t'_3\, x'_6x_7 + t_1t'_2t'_3\, x'_6;$$
$$\ldots$$
$$y_{10} = e_{13} = t_1t_2t'_3\, x'_6x_7;$$

$$d_1 = e_2 + e_6 + e_8 + e_9 + e_{11} + e_{16} = t'_1t'_2t_3\, x_1x_2x'_3 + t'_1t'_2t'_3\, x_4x'_1 + t_1t'_2t_3 +$$
$$+ t'_1t_2t'_3\, x_5 + t'_1t_2t'_3\, x'_5x'_1 + t_1t'_2t'_3\, x'_6;$$
$$d_2 = e_4 + e_7 + e_9 = t'_1t'_2t_3\, x'_1 + t'_1t'_2t'_3\, x'_4 + t'_1t_2t'_3\, x_5;$$
$$d_3 = e_2 + e_{13} + e_{14} + e_{15} = t'_1t'_2t_3\, x_1x_2x'_3 + t_1t_2t'_3\, x'_6x_7 + t_1t_2t'_3\, x'_6x'_7 + t_1t'_2t'_3\, x_6.$$

How many *different products* are there in these expressions? The answer is very simple – only sixteen, because we have 16 rows in Table 2 and only one product corresponds to one row. Thus, we should not write any expressions but can design the logic circuit immediately from the structure table. For that, it is sufficient to construct H AND-gates, one for each row, and $N+R$ OR-gates, one for each output variable y_n ($n = 1, \ldots, 10$ in our example) and one for each input memory function d_r ($r = 1, 2, 3$ in our example). The logic circuit of Mealy FSM is shown in Fig. 12. We have constructed 16 AND-gates, as there are 16 rows in its structure table. The number of OR-gates in this circuit is less than the number of input memory functions and output functions. Really, if y_n or d_r (y_2 and y_{10} in our example) are written only in one row of the structure table, it is not necessary to construct OR-gate for such y_n or d_r, we can get these signals from the corresponding AND-gates. Moreover, we have constructed one OR-gate for y_8 and y_9 since these outputs are always together in the structure table of Mealy FSM S_1.

4.2.7 ASM with waiting vertices. In this section, we will show that the algorithm for FSM synthesis does not change if ASM contains waiting vertices. In a waiting vertex, one of its outputs is connected with its input (see the ASM subgraph in Fig. 13). Let us find all transition paths from the state a_8. The first two are trivial – see the first two rows in Table 4.

To find the next path we should invert the variable x_7. The output *'0'* for x_7 brings us to the input of this conditional vertex. So, the next paths will be:

$$a_8 \sim x_7\, x_7\, x_{12}\, (y_{11})\, a_{13};$$
$$a_8 \sim x_7\, x_7\, \sim x_{12}\, (y_{23},\, y_{29})\, a_{17}.$$

The products of input variables for both of these paths are equal to zero ($x'_7\, x_7 = 0$), so FSM cannot transit from the state a_8 to any other state when $x_7 = 0$. If FSM cannot transit into any other state, it remains in the same state a_8 or, we can say, it transits from a_8 to a_8 with $X(a_8, a_8) = x'_7$. No output variables are equal to *'1'* at this transition, so we have '–' in the column $Y(a_m, a_s)$ in the third row.

The next example (Fig.14) presents a general case. The only difference from the previous example – the waiting vertex is in the middle of the path. After the third path

in Table 5 we should invert variable x_{11} and again return to the input of the conditional vertex with x_{11}. We can construct the following transitions paths:

$$a_{10}\ {\sim}x_4\ {\sim}x_{11}\ x_{11}\ x_{27}\ (y_{33})\ a_{22};$$
$$a_{10}\ {\sim}x_4\ {\sim}x_{11}\ x_{11}\ {\sim}x_{27}\ (y_7,\ y_{31})\ a_{17}.$$

The products for both of these paths are equal to zero. So, when $x_4 = 0$, we reached a waiting vertex with condition x_{11}. If $x_{11} = 0$ (return to the input), FSM transits from state a_{10} to state a_{10} (remains in this state) with input $x'_4\,x'_{11}$ and each output variable is equal to zero (the forth row in Table 5).

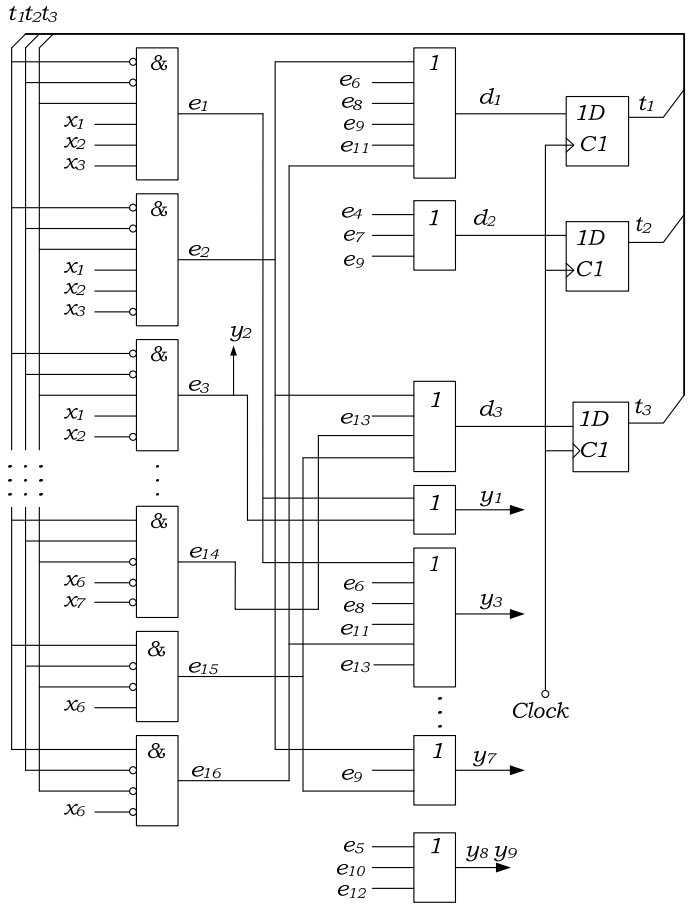


Figure 12. The logic circuit for Mealy FSM S_1

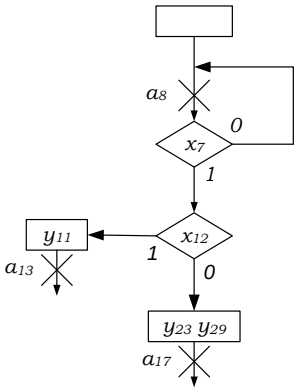


Table 4. Transitions for subgraph G_1

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
		. . .		
a_8	a_{13} a_{17} a_8	x_7x_{12} $x_7{\sim}x_{12}$ ${\sim}x_7$	y_{11} $y_{23}y_{29}$ –	

Figure 13. Subgraph G_1 with waiting vertex

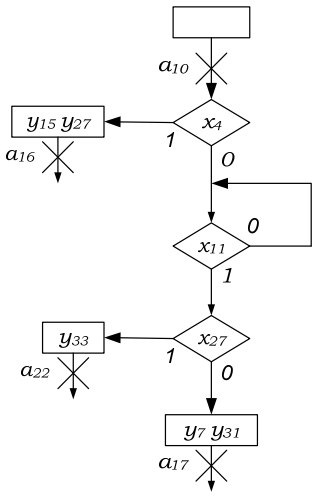


Table 5. Transitions for subgraph G_2

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
		. . .		
a_{10}	a_{16}	x_4	$y_{15}y_{27}$	
	a_{22}	$\sim x_4 x_{11} x_{27}$	y_{33}	
	a_{17}	$\sim x_4 x_{11} \sim x_{27}$	$y_7 y_{31}$	
	a_{10}	$\sim x_4 \sim x_{11}$	–	

Figure 14. Subgraph G_2 with a waiting vertex

4.3 Synthesis of Moore FSM

As an example, we will use ASM G_1 in Fig. 15. A Moore FSM, implementing given ASM, can be constructed in two stages:

Stage 1. Construction of a marked ASM;
Stage 2. Construction of an FSM transition table.

At the first stage, the vertices "Begin", "End" and operator vertices are marked by symbols $a_1, a_2, \ldots, a_M$ as follows:

1. Vertices "Begin" and "End" are marked by the same symbol a_1;
2. Operator vertices are marked by different symbols $a_2, \ldots, a_M$;
3. All operator vertices should be marked.

Thus, while synthesizing a Moore FSM, symbols of states do not mark inputs of vertices following the operator vertices (as in the Mealy FSM) but operator vertices themselves. The number of marks is $T+1$, where T is the number of operator vertices in the marked ASM. In our example (Fig. 15), we need marks $a_1, \ldots, a_{10}$ for ASM G_1.

We will find the following transition paths in the marked ASM:

$$a_m \tilde{x}_{m1} \ldots \tilde{x}_{mR_m} a_s .$$

Thus, the transition path is the path between two operator vertices, containing R_m conditional vertices. Here, as above in the case of Mealy FSM, $\tilde{x}_{mr} = x_{mr}$, if in the transition path, we leave the conditional vertex with x_{mr} via output '1' and $\tilde{x}_{mr} = x'_{mr}$ if we leave the vertex with x_{mr} via output '0'. If $R_m = 0$ in such a path, there are no conditional vertices between two operator vertices, and this path turns into $a_m a_s$.

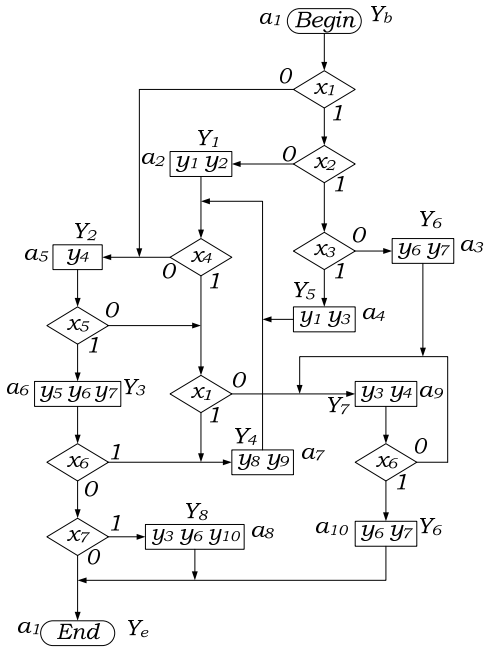


Figure 15. ASM G_1 marked for the Moore FSM synthesis

At the second stage we construct a transition table (or the state diagram) of the Moore FSM with states (marks) $a_1, \ldots, a_M$, obtained at the first stage. We have ten such states $a_1, \ldots, a_{10}$ in our example. Thus, the FSM contains as many states as the number of marks we get at the previous stage. Now we should define transitions between these states.

Thus, a Moore FSM has a transition from state a_m to state a_s with input $X(a_m, a_s)$ (see the upper subgraph in Fig. 16) if, in ASM, there is a transition path $a_m \tilde{x}_{m1} \ldots \tilde{x}_{mR_m} a_s$. Here $X(a_m, a_s)$ is a product of logical conditions written in this path: $X(a_m, a_s) = \tilde{x}_{m1} \ldots \tilde{x}_{mR_m}$. In exactly the same way, for the path $a_m a_s$ (see the lower subgraph in Fig. 16) we have a transition from state a_m to state a_s with input $X(a_m, a_s) = 1$, because the product of an empty set of variables is equal to zero. If a_m marks the operator vertex with operator Y_t, then $\lambda(a_m) = Y_t$, i.e. we identify the operator Y_t written in the operator vertex with this state a_m.

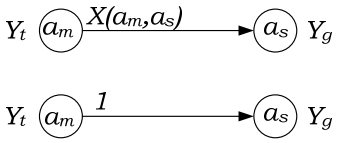


Figure 16. Subgraphs to illustrate transitions in the Moore FSM

The transition table for Moore FSM S_2, thus constructed, is presented in Table 6. The outputs are written in column $Y(a_m)$ immediately after the column with the current states. To design the logic circuit for this FSM we will use the structure presented in Fig. 17. It consists of two logic blocks (*Logic1* and *Logic2*) and memory block with four *D* flip-flops. *Logic1* implements input memory functions, depending on flip-flop outputs $t_1, \ldots, t_4$ (feedback) and input variables $x_1, \ldots, x_7$. *Logic2* implements output functions, depending only on flip-flop outputs $t_1, \ldots, t_4$.

Table 6. The transition table of Moore FSM S_2

a_m	$Y(a_m)$	a_s	$X(a_m, a_s)$	h
a_1	–	a_4	$x_1x_2x_3$	1
		a_3	$x_1x_2\sim x_3$	2
		a_2	$x_1\sim x_2$	3
		a_5	$\sim x_1$	4
a_2	y_1y_2	a_7	x_4x_1	5
		a_9	$x_4\sim x_1$	6
		a_5	$\sim x_4$	7
a_3	y_6y_7	a_9	1	8
a_4	y_1y_3	a_7	x_4x_1	9
		a_9	$x_4\sim x_1$	10
		a_5	$\sim x_4$	11
a_5	y_4	a_6	x_5	12
		a_7	$\sim x_5x_1$	13
		a_9	$\sim x_5\sim x_1$	14
a_6	$y_5y_6y_7$	a_7	x_6	15
		a_8	$\sim x_6x_7$	16
		a_1	$\sim x_6\sim x_7$	17
a_7	y_8y_9	a_7	x_4x_1	18
		a_9	$x_4\sim x_1$	19
		a_5	$\sim x_4$	20
a_8	$y_3y_6y_{10}$	a_1	1	21
a_9	y_3y_4	a_{10}	x_6	22
		a_9	$\sim x_6$	23
a_{10}	y_6y_7	a_1	1	24

To encode FSM states we constructed Table 7 where $p(a_s)$, as before, is the number of appearances of each state in the next state column a_s in Table 6. The algorithm for state assignment is absolutely the same as in the case of Mealy FSM. First, we use the zero code for state a_9 with max $p(a_9) = 6$. Then codes with one '1' are used for states a_7, a_5, a_1 and a_2 with the next max appearences and, finally, five codes with two 'ones' are used for the left five states a_3, a_4, a_6, a_8 and a_{10}.

Table 8 is the structure table of the Moore FSM S_2. Its logic circuit is constructed in Fig. 18. In this circuit, A_m is a product of state variables for the state a_m ($m = 1, \ldots, 10$). As above we construct one AND-gate for one row of the structure table, but we need not construct the gates for rows 6, 8, 10, 14, 19 and 23, as all input memory functions are equal to zero in these rows (see the column $D(a_m,a_s)$ in Table 8). Neither

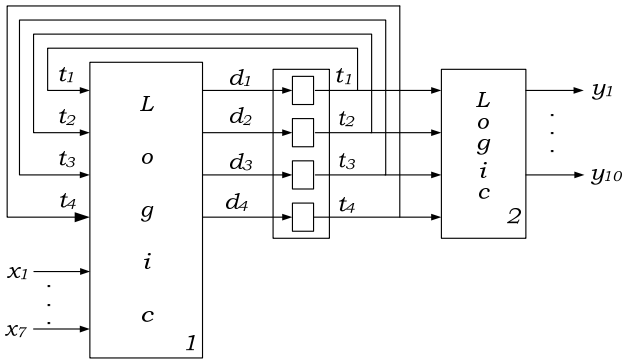


Table 7. State assignment

a_s	$p(a_s)$	$t_1t_2t_3t_4$
a_1	3	0100
a_2	1	0010
a_3	1	1001
a_4	1	0110
a_5	4	0001
a_6	1	1100
a_7	5	1000

Table 8. The structure table of the Moore FSM S_2

a_m	$Y(a_m)$	$K(a_m)$	a_s	$K(a_s)$	$X(a_m,a_s)$	$D(a_m,a_s)$	h
a_1	–	0100	a_4	0110	$x_1x_2x_3$	d_2d_3	1
			a_3	1001	$x_1x_2\sim x_3$	d_1d_4	2
			a_2	0010	$x_1\sim x_2$	d_3	3
			a_5	0001	$\sim x_1$	d_4	4
a_2	y_1y_2	0010	a_7	1000	x_4x_1	d_1	5
			a_9	0000	$x_4\sim x_1$	–	6
			a_5	0001	$\sim x_4$	d_4	7
a_3	y_6y_7	1001	a_9	0000	1	–	8
a_4	y_1y_3	0110	a_7	1000	x_4x_1	d_1	9
			a_9	0000	$x_4\sim x_1$	–	10
			a_5	0001	$\sim x_4$	d_4	11
a_5	y_4	0001	a_6	1100	x_5	d_1d_2	12
			a_7	1000	$\sim x_5x_1$	d_1	13
			a_9	0000	$\sim x_5\sim x_1$	–	14
a_6	$y_5y_6y_7$	1100	a_7	1000	x_6	d_1	15
			a_8	0100	$\sim x_6x_7$	d_2	16
			a_1	0011	$\sim x_6\sim x_7$	d_3d_4	17
a_7	y_8y_9	1000	a_7	1000	x_4x_1	d_1	18
			a_9	0000	$x_4\sim x_1$	–	19
			a_5	0001	$\sim x_4$	d_4	20
a_8	$y_3y_6y_{10}$	0011	a_1	0100	1	d_2	21
a_9	y_3y_4	0000	a_{10}	0101	x_6	d_2d_4	22
			a_9	0000	$\sim x_6$	–	23
a_{10}	y_6y_7	0101	a_1	0100	1	d_2	24

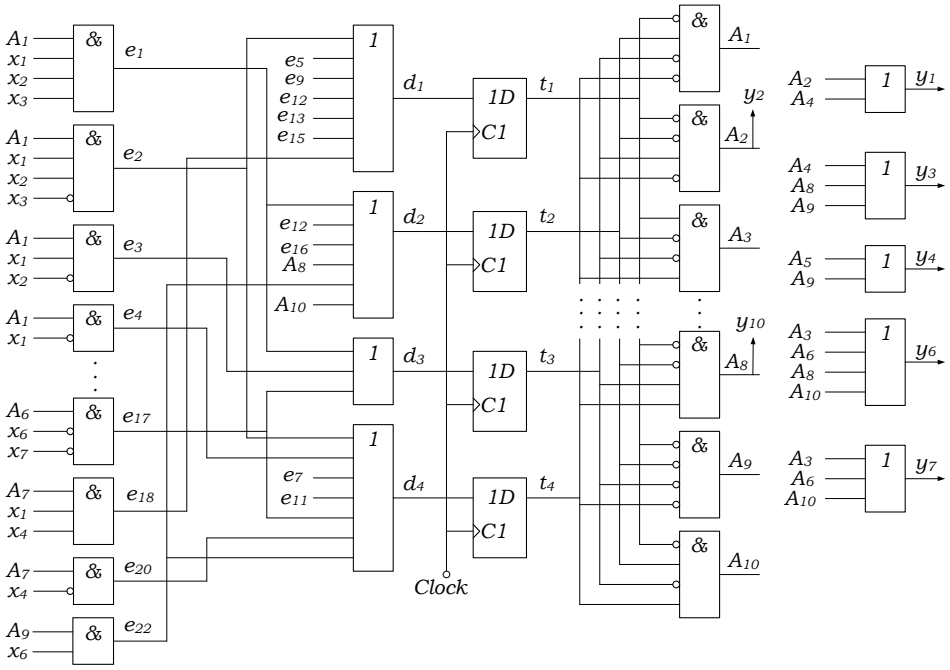


Figure 18. The logical circuit for the Moore FSM S_2

do we construct the gates for rows 21 and 24, since there are no input variables in the corresponding terms e_{21} and e_{24}: $e_{21} = A_8$ and $e_{24} = A_{10}$ and we use A_8 and A_{10} directly as inputs in OR-gate for d_2.

4.4. Synthesis of Combined FSM model

In this book we will use two kinds of transition tables – direct and reverse. In a *direct table* (Table 9), transitions are ordered according to the current state (the first column in this table) – first we write all transitions *from* the state a_1, then *from* the state a_2, etc. In a *reverse table* (Table 10), transitions are ordered according to the next state (the second column in this table) – first we write all transitions *to* the state a_1, then *to* the state a_2, etc.

Table 9. Direct transition table of Mealy FSM S_3

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	h
a1	a2	x6	y8y9	1
a1	a5	~x6*x7	y6	2
a1	a5	~x6*~x7	y3y6y10	3
a2	a2	x4*x1	y1y2	4
a2	a6	x4*~x1	y3y4	5
a2	a4	~x4	y4	6
a3	a6	1	y3y5	7
a4	a1	x5	--	8
a4	a2	~x5*x1	y8y9	9
a4	a6	~x5*~x1	y3y4	10
a5	a2	x1*x2*x3	y1y3	11
a5	a3	x1*x2*~x3	y1y4	12
a5	a2	x1*~x2	y1y2	13
a5	a4	~x1	y4	14
a6	a5	x6	y6y7	15

a6	a6	~x6	y3y5	16

Now we will discuss the transformation of Mealy FSM into Combined FSM and synthesis of its logic circuit. I remind here that Combined FSM has two kinds of output signals:

1. Signals depending on the current state and the current input (as in the Mealy model);
2. Signals depending only on the current state (as in the Moore model);

As an example, we use the transition table of Mealy FSM in Table 9. Our first step is to construct a reverse table for this FSM (Table 10).

Fig. 19,a illustrates all transitions into state a_5 of Mealy FSM from Table 10. Here we have three transitions with different outputs but all of them contain the same output variable y_6. So, we can identify this output variable y_6 with the state a_5 as a Moore signal (see Fig. 19,b).

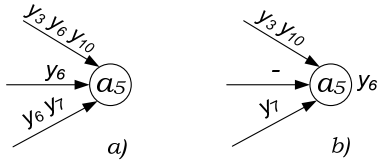


Figure 19. Transformation from Mealy FSM to Combined FSM

Table 10. Reverse transition table of Mealy FSM S_3

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a4	a1	x5	--	1
a2	a2	x4*x1	y1y2	2
a1	a2	x6	y8y9	3
a4	a2	~x5*x1	y8y9	4
a5	a2	x1*x2*x3	y1y3	5
a5	a2	x1*~x2	y1y2	6
a5	a3	x1*x2*~x3	y1y4	7
a2	a4	~x4	y4	8
a5	a4	~x1	y4	9
a1	a5	~x6*x7	y6	10
a1	a5	~x6*~x7	y3y6y10	11
a6	a5	x6	y6y7	12
a4	a6	~x5*~x1	y3y4	13
a2	a6	x4*~x1	y3y4	14
a3	a6	1	y3y5	15
a6	a6	~x6	y3y5	16

After this, the transformation of Mealy FSM into Combined model is trivial. Let us return to the reverse Table 10 and begin to construct the reverse transition table of Combined FSM S_4 (Table 11 In Table10, we look at the transitions to each state, beginning from transitions to state a_1. Let Y_s be the set of output variables at the transitions into state a_s ($Y_5 = \{y_3, y_6, y_7, y_{10}\}$ in Fig19,a or in Table 10) and Y_s^{Moore} be the subset of common output variables at all transitions into a_s ($Y_5^{Moore} = \{y_6\}$ in Fig19,a or in Table 10). Then, in Table 11, we delete Y_s^{Moore} from the column $Y(a_m,a_s)$ at each row with transition to a_s and write Y_s^{Moore} next to a_s in the column $Y(a_s)$. In our example:

$Y_1^{Moore} = Y_2^{Moore} = \varnothing$; $Y_3^{Moore} = \{ y_1, y_4\}$; $Y_4^{Moore} = \{ y_4\}$;
$Y_5^{Moore} = \{ y_6\}$; $Y_6^{Moore} = \{ y_3\}$.

Table 11. Reverse transition table of Combined FSM S_4

a_m	a_s	$Y(a_s)$	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a4	a1	--	x5	--	1
a1	a2	--	x6	y8y9	2
a2	a2	--	x4*x1	y1y2	3
a4	a2	--	~x5*x1	y8y9	4
a5	a2	--	x1*x2*x3	y1y3	5
a5	a2	--	x1*~x2	y1y2	6
a5	a3	y1y4	x1*x2*~x3	--	7
a2	a4	y4	~x4	--	8
a5	a4	y4	~x1	--	9
a1	a5	y6	~x6*~x7	y3y10	10
a1	a5	y6	~x6*x7	--	11
a6	a5	y6	x6	y7	12
a2	a6	y3	x4*~x1	y4	13
a3	a6	y3	1	y5	14
a4	a6	y3	~x5*~x1	y4	15
a6	a6	y3	~x6	y5	16

Now we consider the design of the logic circuit of Combined FSM. For this, let us return to the Mealy FSM S_1 with direct transition Table 1. Its reverse transition table is presented in Table 12. Immediately from this table we construct the direct transition table of Combined FSM S_1 (Table 13). To construct the logic circuit for this FSM we should encode the states and construct FSM structure table. But before state assignment we will make one more step.

Table 12. Reverse transition table of Mealy FSM S_1

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a_5	a_1	$\sim x_6x_7$	$y_3y_6y_{10}$	1
a_5		$\sim x_6\sim x_7$	-	2
a_6		x_6	y_6y_7	3
a_1	a_2	$x_1x_2x_3$	y_1y_3	4
a_1		$x_1\sim x_2$	y_1y_2	5
a_2		x_4x_1	y_8y_9	6
a_4		$\sim x_5x_1$	y_8y_9	7
a_5		x_6	y_8y_9	8
a_1	a_3	$x_1x_2\sim x_3$	y_6y_7	9
a_1	a_4	$\sim x_1$	y_4	10
a_2		$\sim x_4$	y_4	11
a_4	a_5	x_5	$y_5y_6y_7$	12
a_2	a_6	$x_4\sim x_1$	y_3y_4	13
a_3		1	y_3y_4	14
a_4		$\sim x_5\sim x_1$	y_3y_4	15
a_6		$\sim x_6$	y_3y_4	16

Unlike the transition table of the Mealy FSM, Table 13 contains many empty entries in the column $Y(a_m,a_s)$. It means that all output variables are equal to zero in these

rows. If, after state assignment, we get an empty entry in column $D(a_m,a_s)$ for such a row, we shouldn't construct a product for this row, because all output variables and input memory functions are equal to zero in this row. Now we will try to maximize the number of such rows in the structure table of S_5.

Table 13. Direct transition table of Combined FSM S_5

a_m	$Y(a_m)$	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a_1	--	a_2	$x_1x_2x_3$	y_1y_3	1
	--	a_3	$x_1x_2\sim x_3$	-	2
	--	a_2	$x_1\sim x_2$	y_1y_2	3
	--	a_4	$\sim x_1$	-	4
a_2	--	a_2	x_4x_1	y_8y_9	5
	--	a_6	$x_4\sim x_1$	-	6
	--	a_4	$\sim x_4$	-	7
a_3	y_6y_7	a_6	1	-	8
a_4	y_4	a_5	x_5	-	9
		a_2	$\sim x_5x_1$	y_8y_9	10
		a_6	$\sim x_5\sim x_1$	--	11
a_5	$y_5y_6y_7$	a_2	x_6	y_8y_9	12
		a_1	$\sim x_6x_7$	$y_3y_6y_{10}$	13
		a_1	$\sim x_6\sim x_7$	-	14
a_6	y_3y_4	a_1	x_6	y_6y_7	15
		a_6	$\sim x_6$	-	16

Table 13 contains one row with empty entry in the column $Y(a_m,a_s)$ for the next states a_1 (row 14) and a_3 (row 2), two rows for a_4 (rows 4 and 7), one row for a_5 (row 9) and four rows for a_6 (rows 6, 8, 11 and 16). This information is presented in the first two columns of Table 14, $z(a_s)$ is the number of empty entries in column $Y(a_m,a_s)$ for the next state a_s in Table 13. So, if we use zero code for states a_1 or a_3 or a_5, we shouldn't construct a product for one row ($z(a_1) = z(a_3) = z(a_5) = 1$), if we use zero code for state a_4 – for two rows ($z(a_4) = 2$); but if we use zero code for state a_6, we will construct four product less ($z(a_6) = 4$). Thus, we use code *000* for state a_6 with max $z(a_s)$. State assignment for other states is presented in Table 15. We have used here the same algorithm as we have used previously for Mealy and Moore models.

Table 14. Next states with zero outputs

a_s	$z(a_s)$	$t_1\,t_2\,t_3$
a_1	1	
a_3	1	
a_4	2	
a_5	1	
a_6	4	0 0 0

Table 15. State assignment

a_s	$p(a_s)$	$t_1\,t_2\,t_3$
a_1	3	0 1 0
a_2	5	0 0 1
a_3	1	1 0 1
a_4	2	1 0 0
a_5	1	1 1 0
a_6	4	0 0 0

Table 16. Structure table of Combined FSM S_5

a_m	$Y(a_m)$	$K(a_m)$	a_s	$K(a_s)$	$X(a_m,a_s)$	$Y(a_m,a_s)$	$D(a_m,a_s)$	H
a_1	--	010	a_2	001	$x_1x_2x_3$	y_1y_3	d_3	1
	--		a_3	101	$x_1x_2\sim x_3$	-	d_1d_3	2
	--		a_2	001	$x_1\sim x_2$	y_1y_2	d_3	3
	--		a_4	100	$\sim x_1$	-	d_1	4

a_2	--	001	a_2	001	x_4x_1	y_8y_9	d_3	5
	--		a_6	000	$x_4\sim x_1$	-	-	6
	--		a_4	100	$\sim x_4$	-	d_1	7
a_3	y_6y_7	101	a_6	000	1	-	-	8
a_4	y_4	100	a_5	110	x_5	-	d_1d_2	9
			a_2	001	$\sim x_5x_1$	y_8y_9	d_3	10
			a_6	000	$\sim x_5\sim x_1$	-	-	11
a_5	$y_5y_6y_7$	110	a_2	001	x_6	y_8y_9	d_3	12
			a_1	010	$\sim x_6x_7$	$y_3y_6y_{10}$	d_2	13
			a_1	010	$\sim x_6\sim x_7$	-	d_2	14
a_6	y_3y_4	000	a_1	010	x_6	y_6y_7	d_2	15
			a_6	000	$\sim x_6$	-	-	16

Table 16 is the structure table of Combined FSM S_5. We have three kinds of output variables here:

1. *Only Mealy signals:* y_1, y_2, y_8, y_9, y_{10}. They are written in column $Y(a_m,a_s)$ and are not written in column $Y(a_m)$ in Table 16;
2. *Only Moore signals:* y_4, y_5. They are written in the column $Y(a_m)$ and are not written in column $Y(a_m,a_s)$ in Table 16;
3. *Combined signals:* (both Mealy and Moore type) y_3, y_6, y_7. They are written in both columns $Y(a_m,a_s)$ and $Y(a_m)$ in Table 16.

The logic circuit of FSM S_5 is constructed in Fig. 20. In this circuit, A_m is a product of state variables for the state a_m ($m = 1, \ldots, 6$). The left part of this circuit, exactly as in the synthesis of the Mealy FSM logic circuit, implements input memory functions d_1, d_2, d_3 and *Mealy signals* y_1, y_2, y_8, y_9, y_{10}. As above, we construct one AND-gate for one row of the structure table, but we need not construct the gates for rows 6, 8, 11, 16 because all output variables and input memory functions are equal to zero in these rows in the columns $Y(a_m,a_s)$ and $D(a_m,a_s)$ in Table 16. As in the Mealy case, we do not construct OR gates for y_2 and y_{10} since they appear only once in the column $Y(a_m,a_s)$.

Moore signals y_4, y_5 are constructed as in the synthesis of Moore FSM logic circuit. Signal y_4 appears twice near the states a_4 and a_6 in the column $Y(a_m)$, so $y_4 = A_4 + A_6$ and we construct OR gate for this signal. Output signal y_5 appears only once in the column $Y(a_m)$ for the state a_5, so we get it straight from A_5: $y_5 = A_5$.

Combined signal y_6 is written in rows 13 and 15 in the column $Y(a_m,a_s)$ and near the states a_3 and a_5 in the column $Y(a_m)$, so

$$y_6 = e_{13} + e_{15} + A_3 + A_5.$$

Exactly in the same way

$$y_3 = e_1 + e_{13} + A_6; \quad y_7 = e_{15} + A_3 + A_5.$$

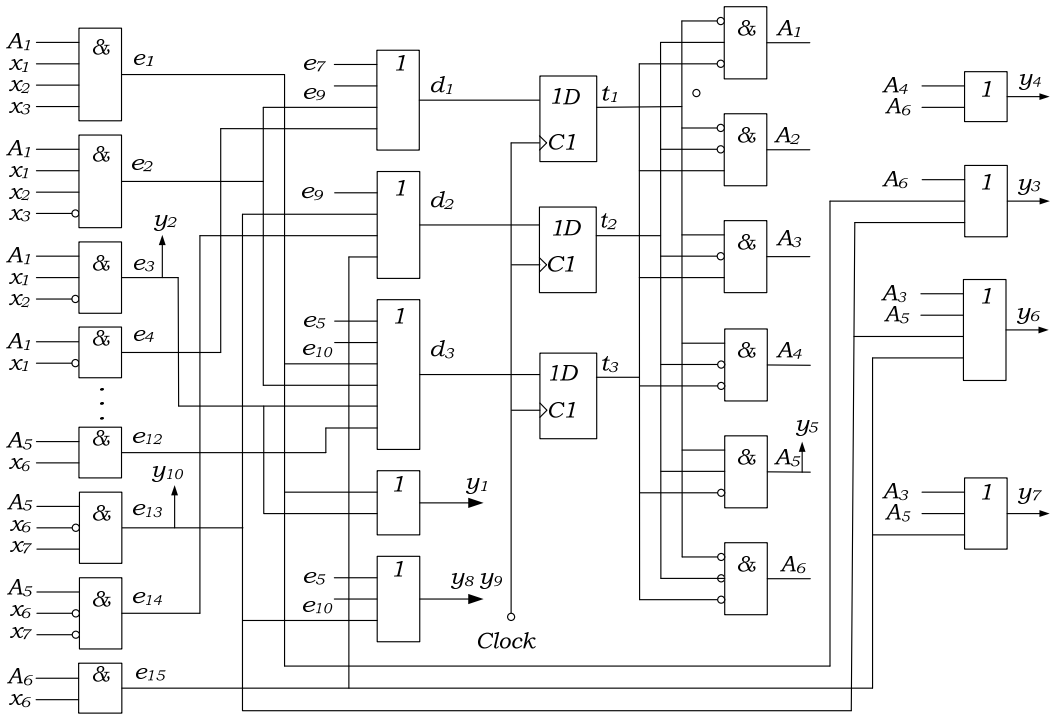


Figure 20. Logic circuit of Combined FSM S_5

4.5. FSM decomposition

In this section, we will discuss a very simple model for FSM decomposition. As an example, we use Mealy FSM S_6 (Table 17) and a partition π on the set of its states:

$$\pi = \{A_1, A_2, A_3\};$$
$$A_1 = \{a_2, a_3, a_9\};\ A_2 = \{a_4, a_7, a_8\};\ A_3 = \{a_1, a_5, a_6\}.$$

The number of component FSMs in the FSM network is equal to the number of blocks in partition π. Thus, in our example, we have three component FSMs S^1, S^2, S^3.

Let B^m is the set of states in the component FSM S^m. B^m contains the corresponding block of the partition π plus one additional state b_m. So, in our example:

S^1 has the set of states $B^1 = \{a_2, a_3, a_9, b_1\};$
S^2 has the set of states $B^2 = \{a_4, a_7, a_8, b_2\};$
S^3 has the set of states $B^3 = \{a_1, a_5, a_6, b_3\}.$

Table 17. Mealy FSM S_6

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a1	a3	x1*x2*x3	y1y2	1
a1	a6	x1*x2*~x3	y2y12	2
a1	a1	x1*~x2	y1y2	3
a1	a5	~x1	y1y2y12	4

```
a2  a2  x6          --        5
a2  a3  ~x6         y3y5      6
a3  a3  x10         y3y5      7
a3  a9  ~x10*x4     y10y15    8
a3  a8  ~x10*~x4    y5y8y9    9
a4  a6  x7          y13       10
a4  a4  ~x7*x9      y13y18    11
a4  a8  ~x7*~x9     y13y14    12
a5  a6  x1          y16y17    13
a5  a5  ~x1         y7y11     14
a6  a1  x5          y1y2      15
a6  a1  ~x5         y16y17    16
a7  a2  x8          y14y18    17
a7  a4  ~x8         y13y18    18
a8  a7  x9          y4y6      19
a8  a4  ~x9         y6        20
a9  a9  x11*x6      y10y15    21
a9  a2  x11*~x6     y5y8y9    22
a9  a3  ~x11        y3y8y9    23
```

To construct a transition table for each component FSM we should define the transitions between the states of these FSMs. For this, each transition between two states a_i and a_j of Mealy FSM S_6 from Table 17 should be implemented one after another as one or two transitions in component FSMs. There are two possible cases:

1. In Mealy FSM S_6, there is a transition between a_i and a_j (Fig. 21, left) and both of these states are in the same component FSM S^m. In such a case, we will have the same transition in this component FSM S^m (Fig. 21, right). It means that we must rewrite the corresponding row from the table of FSM S_6 into the table of component FSM S^m.

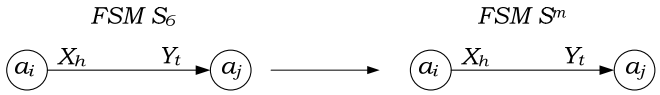


Figure 21. Two states a_i and a_j are in the same component FSM

2. Two states a_i and a_j are in different component FSMs (Fig. 22). Let a_i be in the component FSM S^m ($a_i \in B^m$) and a_j be in the component FSM S^p ($a_j \in B^p$). In such a case, one transition of FSM S_6 should be presented as two transitions – one in the component FSM S^m and one in the component FSM S^p:
   - FSM S^m transits from a_i into its additional state b_m with the same input X_h. At its output, we have the same output variables from set Y_t plus one additional output variable z_j, where index j is the index of state a_j in the component FSM S^p.
   - FSM S^p is in its additional state b_p. It transits from this state into state a_j with input signal z_j, that is an additional output variable in the component FSM S^m. The output at this transition is Y_0 – the signal with all output variables being equal to zero.

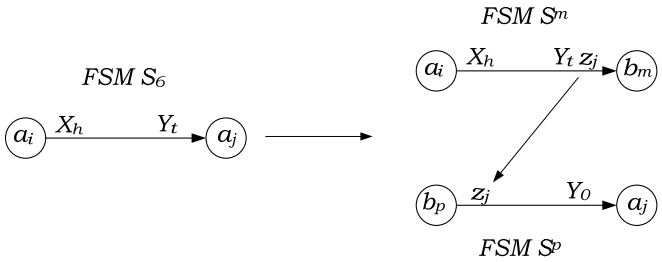


Figure 22. Two states a_i and a_j are in the different component FSMs

Thus, the procedure for FSM decomposition is reduced to:

a) Copying the row

$$a_i \quad a_j \quad X(a_i, a_j) \quad Y(a_i, a_j)$$

from the table of the decomposed FSM S to the table of the component FSM S^m if both states a_i and a_j are the states of S^m;

b) Replacing the row

$$a_i \quad a_j \quad X(a_i, a_j) \quad Y(a_i, a_j)$$

in the table of the decomposed FSM S by the row

$$a_i \quad b_m \quad X(a_i, a_j) \quad Y(a_i, a_j)\ z_j$$

in the table of the component FSM S^m, and by the row

$$b_p \quad a_j \quad z_j \quad --$$

in the table of the component FSM S^p, if a_i is the state of S^m and a_j is the state of S^p.

As a result of decomposition of FSM S_6, we obtain the network with three component FSMs in Fig. 23. Their transition tables are presented in Tables 18 – 20.

Now we will illustrate some examples of transitions for cases (a) and (b):

- In FSM S_6, there is a transition from state *a2* to state *a3* with input *~x6* and output *y3y5* (row 6 in Table 17). As both these states *a2* and *a3* are in the same component FSM S_1, in this FSM there is a transition from *a2* to *a3* with the same input *~x6* and the same output *y3y5* (row 2 in Table 18). Exactly in the same way, we rewrite row 12 of Table 17 into row 3 of Table 19 and row 2 of Table 17 into row 2 of Table 20 because the current states and the next states are in the same component FSMs.

- In FSM S_6, there is a transition from state *a3* to state *a8* with input *~x10*~x4* and the output *y5y8y9* (row 9 in Table 17). Since *a3* is the state of component FSM S^1 and *a8* is the state of another component FSM S^2, in FSM S^1 there is a transition from *a3* to *b1* with the same input *~x10*~x4* and output *y5y8y9z8* (row 5 in Table 18). The last output *z8* is the input of FSM S^2 that wakes this FSM up and transits it from state *b2* to state *a8* (row 8 in Table 19). Similarly, we convert row 1 of Table 17 into two rows – the first in Table 20 and the tenth

in Table 18 etc. Note that we add the last row in each FSM table to remain component FSMs in the state b_m when each z_j is equal to zero.

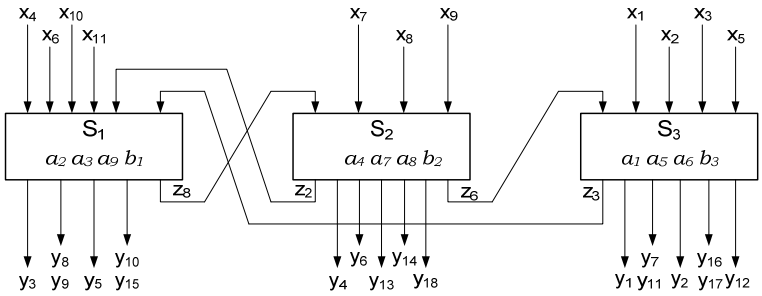


Figure 23. Network with three component FSMs

Table 18. Component FSM S^1

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a2	a2	x6	--	1
a2	a3	~x6	y3y5	2
a3	a3	x10	y3y5	3
a3	a9	~x10*x4	y10y15	4
a3	b1	~x10*~x4	y5y8y9z8	5
a9	a9	x11*x6	y10y15	6
a9	a2	x11*~x6	y5y8y9	7
a9	a3	~x11	y3y8y9	8
b1	a2	z2	--	9
b1	a3	z3	--	10
b1	b1	~z2*~z3	--	11

Table 19. Component FSM S^2

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a4	b2	x7	y13z6	1
a4	a4	~x7*x9	y13y18	2
a4	a8	~x7*~x9	y13y14	3
a7	b2	x8	y14y18z2	4
a7	a4	~x8	y13y18	5
a8	a7	x9	y4y6	6
a8	a4	~x9	y6	7
b2	a8	z8	--	8
b2	b2	~z8	--	9

Let us discuss how this network works. Let *a1* be an initial state in FSM S_6. After decomposition, state a_1 is in FSM S^3, so, at the beginning, just FSM S^3 is in state *a1*. Other FSMs are in states *b1* and *b2* correspondingly. It is possible to say that they "are sleeping" in these states. FSM S^3 transits from the state to the state until *x1*x2*x3 = 1* in state *a1* (see row 1 in Table 20). Only at this transition FSM S^3 produces output signal *z3* and transits into state *b3* (sleeping state). This signal *z3* is the input signal of FSM S^1. It wakes FSM S^1 up and transits it from the sleeping state *b1* to state *a3* (see row 10 in Table 18). Now FSM S^1 transits from the state to the state until, in state *a3,* it transits into state *b1* with input signal *~x10*~x4 = 1* and wakes FSM S^2 up by signal *z8* (see row 5 in Table 18 and row 8 in Table 19).

Table 20. Component FSM S^3

a_m	a_s	$X(a_m,a_s)$	$Y(a_m,a_s)$	H
a1	b3	x1*x2*x3	y1y2z3	1
a1	a6	x1*x2*~x3	y2y12	2
a1	a1	x1*~x2	y1y2	3
a1	a5	~x1	y1y2y12	4
a5	a6	x1	y16y17	5
a5	a5	~x1	y7y11	6
a6	a1	x5	y1y2	7
a6	a1	~x5	y16y17	8
b3	a6	z6	--	9
b3	b3	~z6	--	10

Unlike FSMs S^1 and S^3, the component FSM S^2 has two possibilities to wake other component FSMs up – in state *a4* with input signal *x7 = 1* (row 1 of Table 19) and in state *a7* with input signal *x8 = 1* (row 4 in the same Table), etc. Thus, each time all component FSMs, except one, are in the states of type b_m and only one of them is in the state of type a_i.

Chapter 5 Multilevel and Multioutput Synthesis

In this Chapter, we will concentrate on the multilevel minimization of logic circuits. Several simple and straightforward methods for obtaining circuit structure with more than two levels will be considered. In these methods, we will present four procedures – factoring, term decomposition, full inclusion and equal gates removal. At the end of the Chapter we will show how to construct optimized multilevel and multioutput circuits of Finite State Machines using only these four procedures.

5.1 Factoring

5.1.1 Two factoring structures. The first example of *factoring* is presented in Fig. 1. The left part of this figure implements the function

$$f_1 = x_1x_2x'_3x_4 + x_1x_2x'_5 + x_1x_2x_3x'_4. \qquad (1)$$

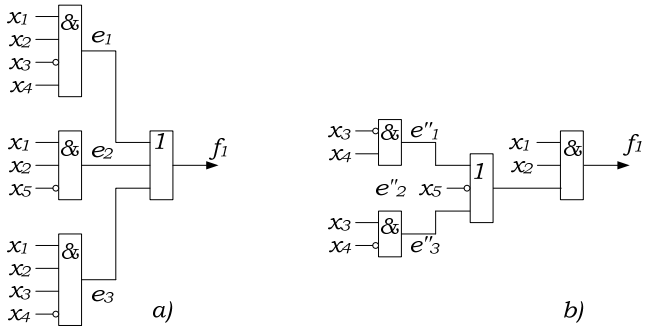


Figure 1. Factoring from all terms

All AND-gates of this circuit have the common input x_1x_2, so we can factor this common term (we call it a *factor*) in function (1):

$$f_1 = x_1x_2\,(x'_3x_4 + x'_5 + x_3x'_4) \qquad (2)$$

The corresponding logic circuit is constructed in Fig. 1,b. In this circuit, e''_1, e''_2 and e''_3 contain inputs remained after deleting factor x_1x_2 from e_1, e_2 and e_3, and if there remains only one letter (x'_5 in our example), it will be an exact input into OR-gate.

Let us suppose again that the cost of a gate is equal to the number of its inputs, and that the cost of logic circuit is the sum of the costs of gates – the total number of inputs into all gates. If C_1 and C_2 are the costs of circuits before and after factoring then $C_1 - C_2$ is a minimization or a *gain of factoring*. We can evaluate the gain of factoring for the common term z by the formula

$$w(z) = m(n - 1) - 1 + r. \qquad (3)$$

Here m is the number of letters in factor z, n is the number of gates in factoring and r is the number of gates in which only one letter is left after factoring. In our example $w(z) = 2(3 - 1) - 1 + 1 = 4$. Really, if we count C_1 and C_2 in Fig.1, $C_1 - C_2 = 4$.

One more example of factoring is presented in Fig. 2. Unlike the previous example, here we can factor the common term $x_1x_3x'_4$ only from two AND-gates, not from all of them:

$$f_2 = x_1x_2x_3x'_4x_5 + x_4x_6x'_7 + x_1x'_2x_3x'_4x'_5 + x'_1x'_2;$$
$$f_2 = x_1x_3x'_4\,(x_2x_5 + x'_2x'_5) + x_4x_6x'_7 + x'_1x'_2.$$

The result of factoring is shown in Fig. 2,b. On the right, we have OR-gate with three inputs – two of them from all AND-gates that do not take part in factoring (e_2, e_4) and the third one – from the output of the factoring structure for e_1, e_3 similar to Fig. 1,b.

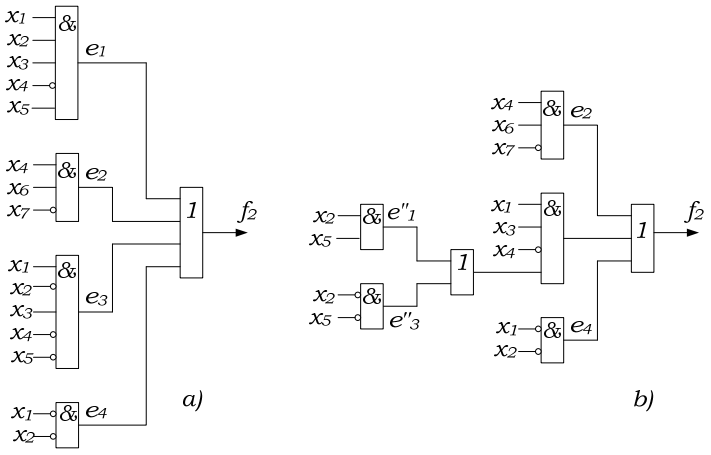


Figure 2. Factoring not from all terms

Again, we can evaluate the gain of factoring for the common term z by the formula

$$w(z) = m(n - 1) - 2 + r. \qquad (4)$$

Here m, n, and r are the same as in expression (3). See if you can understand why "-2" is used in this formula instead of "*-1*".

We discussed here two structures for factoring – structure one in Fig. 1,b (factoring from all AND-gates) and structure two in Fig. 2,b (factoring from some of AND-gates). The duality of Boolean functions permits us to use factoring not only for the sum-of-products, but for the product-of-sum as well (see Fig.3 and Fig. 4).

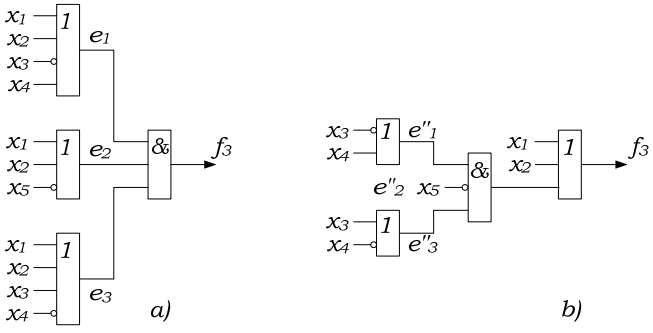


Figure 3. The first factoring structure for the product-of-sums

5.1.2 More than one factor. In the previous examples we have only one possible factor for factoring. Now we will discuss a case with several probable factors. As an example let us use a two-level logic circuit corresponding to Boolean function $f = e_1 + e_2 + e_3 + e_4 + e_5$ with the products:

$e_1 = x_1x_2x_3x_4x_5x_6x_7x_{11}$;
$e_2 = x_1x_2x_3x_8$;
$e_3 = x_1x_2x_5x_6x_{10}x_{11}x_{12}$;
$e_4 = x_5x_6x_9$;
$e_5 = x_1x_2x_5x_6x_{10}x_{12}x_{13}$.

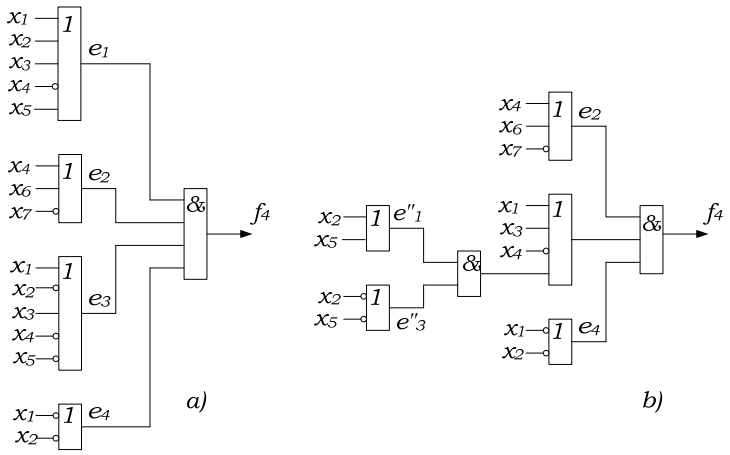


Figure 4. The second factoring structure for the product-of-sums

Let $e_i \cap e_j$ be the intersection between the products e_i and e_j (the common letters in these products). Our first step is to form all possible intersections between each pair of products in f. To do this, we construct Table 1. The first column of this table contains products $e_1, \ldots, e_5$. Intersections between all pairs of products are in the next columns, for example, $e_1 \cap e_2$ is in the column e_1 in the second row, $e_1 \cap e_3$ – in the column e_1 in the third row etc.

Table 1. Possible factors at the first step

$e_1 = x_1x_2x_3x_4x_5x_6x_7x_{11}$	e_1				
$e_2 = x_1x_2x_3x_8$	$x_1x_2x_3$	e_2			
$e_3 = x_1x_2x_5x_6x_{10}x_{11}x_{12}$	$x_1x_2x_5x_6x_{11}$	x_1x_2	e_3		
$e_4 = x_5x_6x_9$	x_5x_6	-	x_5x_6	e_4	
$e_5 = x_1x_2x_5x_6x_{10}x_{12}x_{13}$	$x_1x_2x_5x_6$	x_1x_2	$x_1x_2x_5x_6x_{10}x_{12}$	x_5x_6	

To find all possible factors, thus constructed, we should extract all different intersections from Table 1. There are six such factors $z_1, \ldots, z_6$ in this table. In this step, do not pay attention at the information in the parenthesis after each factor in expression (5):

$$
\begin{array}{ll}
z_1 = x_1x_2x_3\ (e_1, e_2^*); & w(z_1) = 2; \\
z_2 = x_1x_2x_5x_6x_{11}\ (e_1, e_3); & w(z_2) = 3; \\
z_3 = x_5x_6\ (e_1, e_3, e_4^*, e_5); & w(z_3) = 5; \\
z_4 = x_1x_2x_5x_6\ (e_1, e_3, e_5); & w(z_4) = 6; \\
z_5 = x_1x_2\ (e_1, e_2, e_3, e_5); & w(z_5) = 4; \\
z_6 = x_1x_2x_5x_6x_{10}x_{12}\ (e_3^*, e_5^*); & w(z_6) = 6.
\end{array}
\qquad (5)
$$

We will use formulas (3) and (4) to evaluate the gain of each factor. To do this we should find m, n and r for each factor. Here m is the number of letters in the factor, n is the number of gates in factoring and r is the number of gates in which only one letter is left after factoring of this factor. m is trivial – for z_1, m is equal to 3; for z_2, m is equal to 5 etc. To find n, we should intersect each z_t ($t = 1, \ldots, 6$) with each e_i ($i = 1, \ldots, 5$). If z_t is contained in e_i, then z_t is the factor of e_i and we write e_i in the parenthesis after z_t. Thus, for z_1, z_2 and z_6, n is equal to 2, for z_3 and z_5, n is equal to 4 etc.

While performing such intersections, it is possible to find r as well. For example, when we intersect z_1 with e_2 we see that $z_1 \in e_2$ and only one letter is left after factoring z_1 from e_2, because z_1 has three letters but e_2 has four. The symbol * next to e_2 in the line for z_1 means that only one letter is left. We have the same for z_3 (e_4^*) and z_6 (e_3^*, e_5^*). When we have m, n and r for each factor, the evaluation is trivial. $w(z_t)$ for each z_t is presented in the second column of (5).

In the first step of factoring, we use a factor with a maximal gain. If we have several such factors (two in our example – z_4 and z_6) it is possible to implement one of the following strategies:

1. Take the first of such factors (the simplest strategy);
2. Take the factor with maximal length from these factors;
3. Take the factor contained in the maximum number of gates;
4. Move one step forward for each such factor and select factor after the second evaluation step etc.

We will use the first trivial strategy and select z_4 with

$$w(z_4) = 6 = max.$$

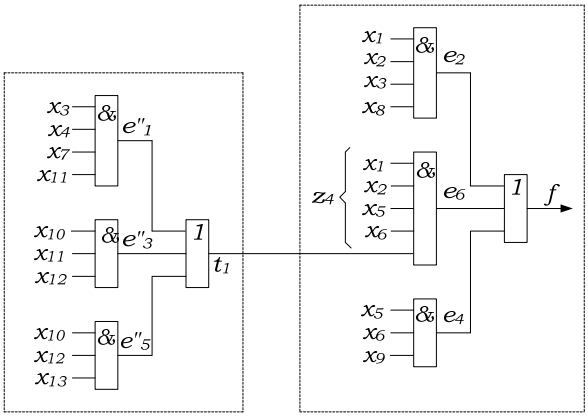


Figure 5. The circuit after the first step of factoring

The circuit after factoring of z_4 is shown in Fig. 5. It implements two functions presented as sum-of-product:

1. Function f is the sum-of-products with three AND-gates, one of them contains the factor z_4, and two others – the products that do not take part in factoring;
2. Function t_1 is the sum-of-products with three AND-gates, each of them corresponds to one of the products that took part in factoring. These ANDs have inputs remaining after factoring of z_4.

Fig. 6 presents the factoring process. The first box in this figure contains the set of products e_1, …, e_5, the second one – the partitioning of this set into two subsets after the first step. Thus, we must continue the factoring separately for two functions presented as sum-of-products – function f containing products e_2, e_4, e_6 and function t_1 containing products e''_1, e''_3, e''_5. A similar partition will be at each next step so the process of factoring converges very fast.

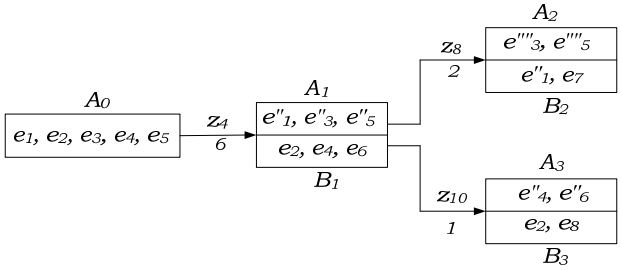


Figure 6. Steps of factoring

The subsequent steps of factoring for functions t_1 and f are presented in Tables 2 and 3. The factoring process comes to the end when there are no factors with the gain greater than zero. The final circuit after factoring is presented in Fig. 7. The total cost reduction is equal to

$$w(z_4) + w(z_8) + w(z_{10}) = 9.$$

Table 2. Factoring of function t_1

$e''_1 = x_3x_4x_7x_{11}$	e''_1	
$e''_3 = x_{10}x_{11}x_{12}$	x_{11}	e''_3
$e''_5 = x_{10}x_{12}x_{13}$	-	$x_{10}x_{12}$

$z_7 = x_{11}\ (e''_1, e''_3);$ $\quad w(z_7) = -1;$

$z_8 = x_{10}x_{12}\ (e''_3{}^*, e''_5{}^*);$ $\quad w(z_8) = 2.$

$$w(z_8) = 2 = max.$$

Table 3. Factoring of function f

$e_2 = x_1x_2x_3x_8$	e_2	
$e_4 = x_5x_6x_9$	-	e_4
$e_6 = x_1x_2x_5x_6t_1$	x_1x_2	x_5x_6

$z_9 = x_1x_2\ (e_2, e_6);$ $\quad w(z_9) = 0;$

$z_{10} = x_5x_6\ (e_4{}^*, e_6);$ $\quad w(z_{10}) = 1.$

$$w(z_{10}) = 1 = max.$$

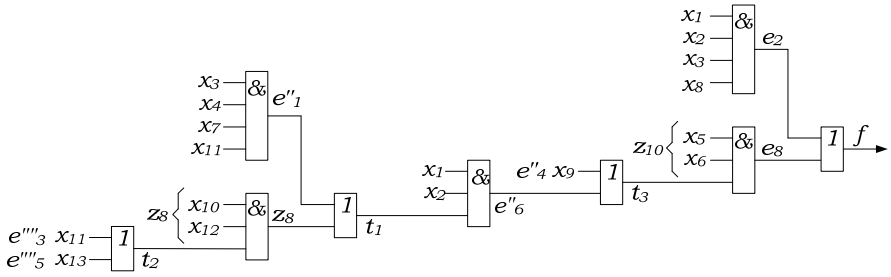


Figure 7. The circuit after factoring

5.2 Term Decomposition

5.2.1 Simple example. The first example of *term decomposition* is presented in Fig. 8. Left part of this figure contains three separate AND-gates implementing three functions g_1, g_2 and g_3. All gates of this circuit have the common inputs x_4, x'_5, x'_6, so we can construct additional AND-gate z with these inputs and replace inputs x_4, x'_5, x'_6 of initial gates with the output of gate z (Fig. 8,b).

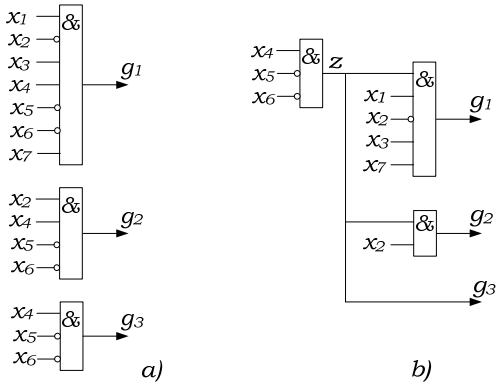


Figure 8. Simple term decomposition

If C_1 and C_2 are the costs of circuits before and after term decomposition then $C_1 - C_2$ is a minimization or a *gain of term decomposition*. We can evaluate the gain of term decomposition for the common term z by the formula

$$w(z) = m(n - 1) - n + r. \qquad (6)$$

Here m is the number of letters in the common term z, n is the number of gates in term decomposition and r is the number of functions (initial AND-gates) equal to the common term. In our example $w(z) = 3(3 - 1) - 3 + 1 = 4$. Really, if we count C_1 and C_2 in Fig. 8, $C_1 - C_2 = 4$.

5.2.2 More than one common term. In the previous example, we had only one possible term for term decomposition. Now we will discuss the case with several probable common terms. As an example let us use a circuit in Fig. 9 that corresponds to the following products:

$g_1 = x_1x_2x_3x_4x_5x_6x_7x_{11};$
$g_2 = x_1x_2x_3x_8;$
$g_3 = x_1x_2x_5x_6x_{10}x_{11}x_{12};$
$g_4 = x_5x_6x_9;$
$g_5 = x_1x_2x_5x_6x_{10}x_{12}.$

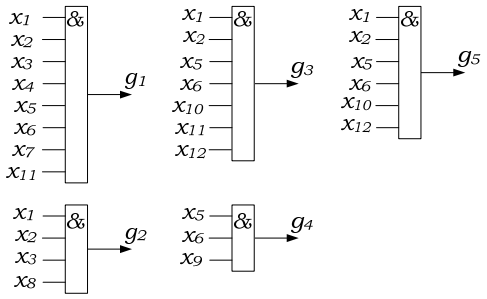


Figure 9. Logic circuit before term decomposition

As in factoring, the algorithm of term decomposition consists of several steps. The first step is to form intersections between each pair of products to find all possible common terms containing two or more variables (see Table 4). We will use formula (6) to evaluate the gain of each common term. To do this we should find m, n and r for each term: m is trivial – it is the number of letters in the common term. For z_1, m is

equal to 3; for z_2, m is equal to 5 etc. It is clear that the common term with one variable makes no sense in term decomposition.

Table 4. Possible common terms at the first step

$g_1 = x_1x_2x_3x_4x_5x_6x_7x_{11}$	g_1			
$g_2 = x_1x_2x_3x_8$	$x_1x_2x_3$	g_2		
$g_3 = x_1x_2x_5x_6x_{10}x_{11}x_{12}$	$x_1x_2x_5x_6x_{11}$	x_1x_2	g_3	
$g_4 = x_5x_6x_9$	x_5x_6	-	x_5x_6	g_4
$g_5 = x_1x_2x_5x_6x_{10}x_{12}$	$x_1x_2x_5x_6$	x_1x_2	$x_1x_2x_5x_6x_{10}x_{12}$	x_5x_6

$z_1 = x_1x_2x_3\ (g_1, g_2);$ $w(z_1) = 1;$
$z_2 = x_1x_2x_5x_6x_{11}\ (g_1, g_3);$ $w(z_2) = 3;$
$z_3 = x_5x_6\ (g_1\ g_2, g_4, g_5);$ $w(z_3) = 2;$ (7)
$z_4 = x_1x_2x_5x_6\ (g_1, g_3, g_5);$ $w(z_4) = 5;$
$z_5 = x_1x_2\ (g_1, g_2, g_3, g_5);$ $w(z_5) = 2;$
$z_6 = x_1x_2x_5x_6x_{10}x_{12}\ (g_3, g_5^*);$ $w(z_6) = 5.$

To find n, we should intersect each z_t ($t = 1, \ldots, 6$) with each g_i ($i = 1, \ldots, 5$). If $z_t \in g_i$, then z_t is the common term for g_i and we write g_i in the parenthesis after z_t. While performing such an

intersection it is possible to find r as well. For example, when we intersect z_6 with g_5 we see that $z_6 = g_5$. The symbol * near g_5 in the line for z_6 means that the product g_5 is equal to the common term z_6. When we have m, n and r for each common term, the evaluation is trivial – $w(z_t)$ for each z_t is presented in the second column of (7).

In the first step of term decomposition, we use a common term with the maximal gain. If we have several such terms (two in our example – z_4 and z_6), as in factoring, it is possible to implement the following several strategies:

1. Take the first of such common terms (the simplest strategy);
2. Take the common term with maximal length from these common terms;
3. Take the common term contained in the maximum number of gates;
4. Move one step forward for each such common term and select common term after the second step evaluation etc.

We will use the first strategy and select z_4 with

$$w(z_4) = 5 = max.$$

The circuit after decomposition of z_4 is shown in Fig. 10.

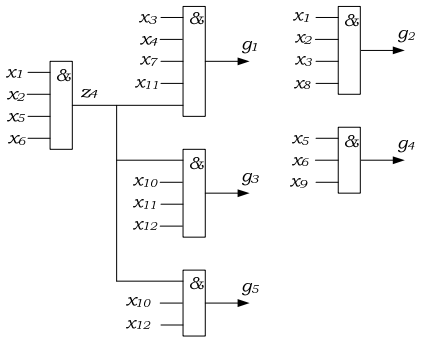


Figure 10. Logic circuit after the first step of term decomposition

Unlike factoring, where we had a partition of products into two subsets after each step, there is no partition of initial products is here. Moreover, the common term taking part in term decomposition should be added to the set of products and will be used at the next step together with other products. Only the product equal to the common term should be excluded from the list of products in the next step of term decomposition.

The next step of term decomposition is presented in Table 5. The process comes to the end when there are no factors with the gain greater than zero. The final circuit after term decomposition is shown in Fig. 11, the whole process is illustrated by Fig. 12.

Table 5. The second (final) step of term decomposition

$g''_1 = x_3x_4x_7x_{11}z_4$	g''_1				
$g_2 = x_1x_2x_3x_8$	-	g_2			
$g''_3 = x_{10}x_{11}x_{12}z_4$	$x_{11}z_4$	-	g''_3		
$g_4 = x_5x_6x_9$	-	-	-	g_4	
$g''_5 = x_{10}x_{12}z_4$	-	-	$x_{10}x_{12}z_4$	-	g''_5
$z_4 = x_1x_2x_5x_6$		x_1x_2	-	x_5x_6	-

$z_7 = x_{11}z_4\ (g''_1, g''_3);$ $w(z_1) = 0;$
$z_8 = x_1x_2\ (g_2, z_4);$ $w(z_2) = 0;$
$z_9 = x_{10}x_{12}z_4\ (g''_3, g''_5*);$ $w(z_9) = 2;$
$z_{10} = x_5x_6\ (g_4, z_4);$ $w(z_{10}) = 0.$

$$w(z_9) = 2 = max.$$

The total cost reduction is equal to

$$w(z_4) + w(z_9) = 7.$$

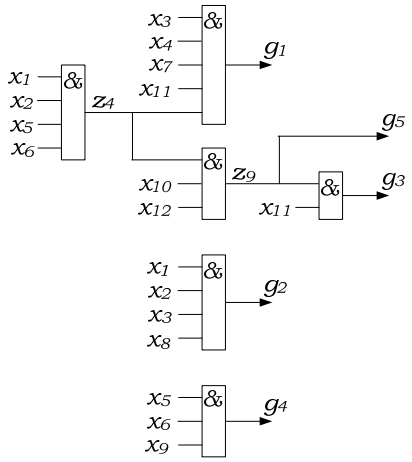


Figure 11. The circuit after t-decomposition

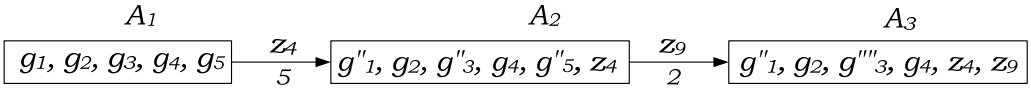


Figure 12. Steps of t-decomposition

5.2.3 Term decomposition for OR gates. Term decomposition can be applied to OR gates as well. We will give the next example without any comments and you can fulfill each step on your own (Fig. 13).

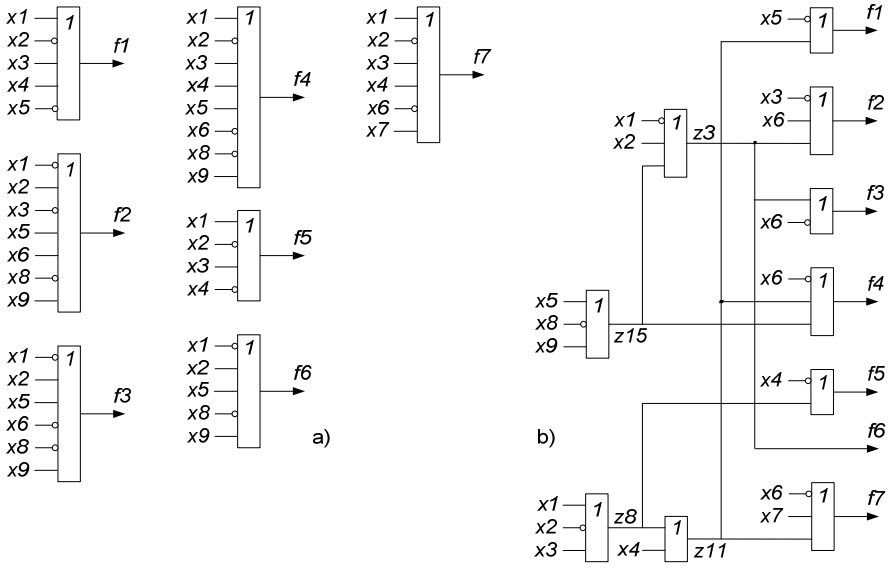


Figure 13. Term decomposition for OR gates

5.3 Gate inclusion

Let us define gate *m* as *included* in gate *n*, or gate *n* as *covering* gate *m*, if they have the same type (both AND or both OR) and the set of inputs of gate *m* is a subset of the set of inputs of gate *n*. The simplest case of gate inclusion is presented in Fig 14,a. In this case, we can replace inputs of gate *n*, equal to the inputs of gate *m* (x_1 and x_2 in our example), with the output of gate *m* (Fig. 14,b).

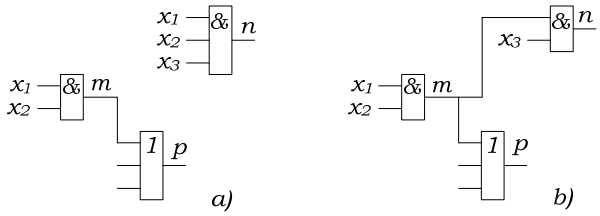


Figure 14. Gate inclusion

5.4 Removal of equal gates

Let us define as gates *m* and *n equal*, if they have the same type (both AND or both OR) and the set of inputs of gate *m* is equal to the set of inputs of gate *n*. The circuit in Fig. 15,a contains four equal two-input AND-gates. In this case, we should

1. Remove all equal gates, except one (gates *l*, *m* and *n* in our example);
2. Connect inputs of gates (*t*, *p* and *q*) formerly connected to the outputs of removed gates, with the output of the remained gate (gate *k* in our example).

The last two procedures – gate inclusion and removal equal gates are covered by term decomposition. Really, in the first step of term decomposition – pair intersection, we can find equal gates and gates included into other gates. However, term decomposition has two problems: (1) the large number of gates taking part in this procedure; (2) multiple comparisons demand a lot of intersections between sets of inputs. It is more simple and faster to check gate inclusion and remove equal gates before term decomposition. Moreover, after these two procedures, only gates with three and more inputs remain for term decomposition (see if you can understand why it is so).

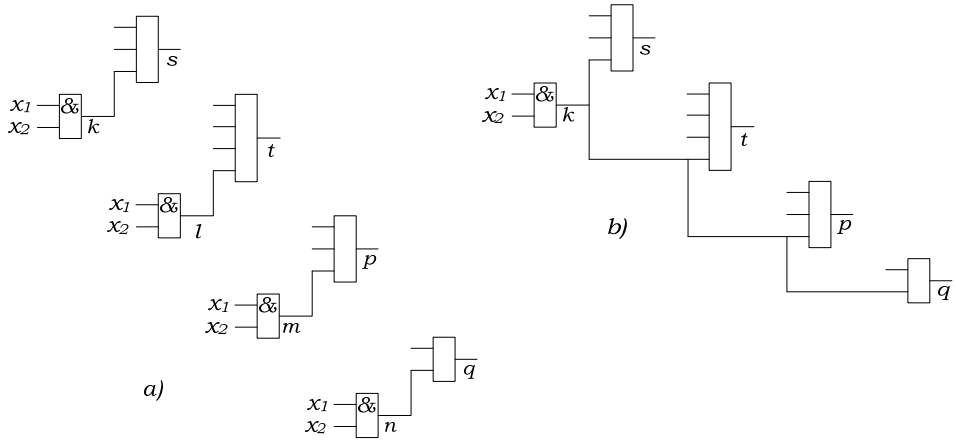


Figure 15. Removal three equal AND-gates

5.5 Multilevel and multioutput circuits for Finite State Machines

In Section 4.2.6 of Chapter 4, we considered a very simple method for synthesis of the two level FSM logic circuit from its structure table. Recall that we have used the term

$$e_h = A_m X_h$$

in accordance with the h row of such a table ($h = 1, \ldots, H$). Here A_m is a product of state variables corresponding to the current state a_m written in the h row, X_h is a product of input variables written in the same row, and H is the number of rows in the structure table. Then we constructed H AND-gates corresponding to terms $e_1, \ldots, e_H$. If the output variable y_n appears only once, for example, in row i of the structure table, we obtain the output y_n at the output of AND-gate number i. If the output variable y_n is written in several rows, for example, in rows $p_1, \ldots, p_T$ of the structure table, we construct OR-gate with T inputs and connect these inputs with the outputs of AND-gates $p_1, \ldots, p_T$. The output y_n is obtained at the output of this OR-gate. In exactly the same way, we construct OR-gate for each input memory function which occurs more than once in the column $D(a_m,a_s)$ of the structure table. The logic circuit of FSM thus constructed contains not more than H AND-gates and not more than $(N + R)$ OR-gates where N and R are the numbers of output variables and input memory functions in the FSM structure table.

In this section, we will use the reverse structure table. Recall that in such a table all transitions are ordered according the next state – first we write all transitions *to* state a_1, then *to* state a_2 , etc. As an example we will consider the logic synthesis of FSM S, Table 6 is its reverse structure table. As in four previous sections, we assume that the circuit cost is equal to the sum of inputs of its gates.

Table 6. The reverse structure table of FSM *S*

a1	001	a1	001	x8*x7	y7y9y14y15	d3	1
a1	001	a1	001	x8*~x7*x1*x9*x5	y13	d3	2
a1	001	a1	001	~x8*x1*x9*x5	y13	d3	3
a3	011	a1	001	x9*x5	y13	d3	4
a4	000	a1	001	x4*~x9*x3	y2y10y12	d3	5
a5	010	a1	001	x4	--	d3	6
a1	001	a2	100	x8*~x7*~x1	y1y2y3	d1	7
a1	001	a2	100	~x8*~x1	y1y2y3	d1	8
a2	100	a2	100	~x2	--	d1	9
a2	100	a3	011	x2	y4	d2d3	10
a4	000	a3	011	x4*~x9*~x3	y5y6	d2d3	11
a4	000	a3	011	x4*x9	y5y6	d2d3	12
a1	001	a4	000	x8*~x7*x1*~x9*x3*~x6	y7y8y9	--	13
a1	001	a4	000	~x8*x1*x9*~x5	y7y8y9	--	14
a3	011	a4	000	x9*~x5	y7y8y9	--	15
a3	011	a4	000	~x9*x3*~x6	y7y8y9	--	16
a3	011	a4	000	~x9*~x3	y7y8y9	--	17
a1	001	a4	000	~x8*x1*~x9*x3*~x6	y7y8y9	--	18
a1	001	a4	000	~x8*x1*~x9*~x3	y7y8y9	--	19
a1	001	a4	000	x8*~x7*x1*~x9*~x3	y7y8y9	--	20
a4	000	a4	000	~x4	--	--	21
a1	001	a4	000	x8*~x7*x1*x9*~x5	y7y8y9	--	22
a1	001	a5	010	x8*~x7*x1*~x9*x3*x6	y10y11y12	d2	23
a3	011	a5	010	~x9*x3*x6	y10y11y12	d2	24
a1	001	a5	010	~x8*x1*~x9*x3*x6	y10y11y12	d2	25
a5	010	a5	010	~x4	--	d2	26

The structure table is divided into M arrays, each of which corresponds to the set of transitions into one state. For FSM in Table 6, M is equal to five. In several initial steps, we will separately design logic circuits for transitions into each state. Moreover, even then we will construct circuits separately for each subset of output signals.

A design of the logic circuit consists of the following steps:

Step 1. Divide each array of transitions to the state a_s (s = 1, …, M) into as many subarrays, as the number of different microinstructions (the subsets of output variables) in the column $Y(a_m, a_s)$ within this array. For example, in Table 6, transitions into state a_1 have four microinstructions:

$$\begin{array}{l} y7, y9, y14, y15; \\ y13; \\ y2, y10, y12; \\ \varnothing . \end{array} \qquad (8)$$

We should include the empty microinstruction, corresponding to row 6, in this list because not all of input memory functions are equal to zero at this transition (*d3 = 1*) and we must construct AND-gate for this row.

Thus, in our example we have four such subarrays containing

1. Row 1 with outputs *y7, y9, y14, y15*;
2. Rows 2, 3, 4 with output *y13;*
3. Row 5 with outputs *y2, y10, y12;*
4. Row 6 without output signals.

Step 2. For each subarray corresponding to one of microinstruction in (8), construct as many AND-gates as the number of rows in this subarray of the structure table. These gates implement products $A_m X(a_m, a_s)$, corresponding to each row. In our example for the transitions into *a1* we have six such AND-gates (see Fig.16,a).

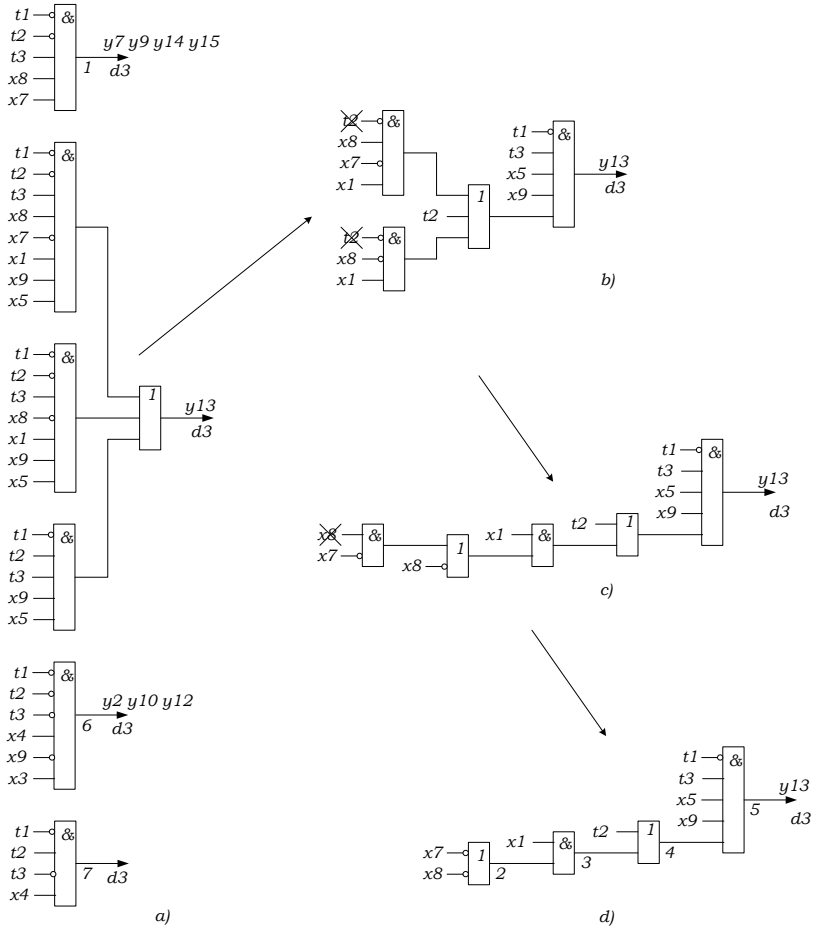


Figure 16. Logic circuit for transitions into state a_1

Step 3. If some subarray contains more than one row, connect the outputs of corresponding AND-gates, constructed at step 2 for the subarray, with OR-gate to form the signals of microoperations (output variables) and input memory functions written in the rows of this subarray (rows 2, 3, 4 for y13 – Fig. 16,a).

Step 4. Factor the logic circuits constructed in point 3 using the algorithm described in Section 5.1 'Factoring'. Let us do this for functions *y13, d3.* Table 7 contains the first step of this factoring. We made all pair intersections between products corresponding to rows with *y13, d3* and found two possible factors z_1 and z_2. We factor z_2 with max gain (see Fig. 16,b).

It is possible to make one more simplification in the circuit in Fig. 16,b. OR-gate has input t_2 and AND-gates connected with this OR-gate have inputs t'_2. According Boolean algebra $A + A'B = A + B$, so we can delete inputs t'_2 from AND-gates. To make such minimization we do not have to write any formulas. If some OR-gate has some input p (p') we should check all AND-gates connected with this OR-gate and remove inputs p' (p) from these AND-gates.

Table 7. The first step of factoring for y_{13} and d_3

$e_1 = t'_1t'_2t_3x_8x'_7x_1x_9x_5$	e_1	
$e_2 = t'_1t'_2t_3x'_8x_1x_9x_5$	$t'_1t'_2t_3x_1x_9x_5$	e_2
$e_3 = t'_1t_2t_3x_9x_5$	$t'_1t_3x_9x_5$	$t'_1t_3x_9x_5$

$z_1 = t'_1t'_2t_3x_1x_9x_5$ (e_1, e_2^*); $w(z_1) = 6(2 - 1) - 2 + 1 = 5$;
$z_2 = t'_1t_3x_9x_5$ (e_1, e_2, e_3^*); $w(z_2) = 4(3 - 1) - 1 + 1 = 8$;

$w(z_2) = 8 = max.$

The last step of factoring is shown in Table 8 and Fig. 16,c. After removing input x_8 from AND-gate with two inputs we must remove this AND-gate as well, and transfer input x'_7 into the OR-gate. The final step of factoring is presented in Fig. 16,d.

Table 8. The second steps of factoring for y_{13}, d_3

$e''_1 = x_8x'_7x_1$	e''_1
$e''_2 = x'_8x_1$	x_1

$z_3 = x_1$ $(e''_1, e''_2{}^*)$; $w(z_3) = 1(2 - 1) - 1 + 1 = 1$;

$w(z_3) = 1 = max.$

Logic circuit after factoring for transitions into state a_1 contains seven gates – we numbered gates after the last step. Each step of circuit factoring for transitions into states a_2 and a_3 is presented in Fig.17 and Fig. 18. Logic circuits for transitions into states a_4 and a_5 without intermediate steps are shown in Fig. 19 and Fig. 20. We have left the design of these last circuits to our readers as exercise to be done on their own. At last, we bring all these circuits together in Fig. 21.

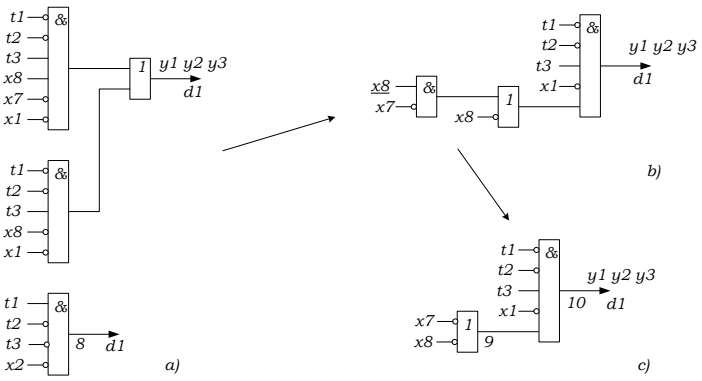


Figure 17. Logic circuit for transitions into state a_2

<u>*Step 5. Delete equal gates in the logic circuit thus constructed.*</u> If we look at the circuit after factoring in Fig. 21 we will find that it contains some equal gates. For example, six two-input OR-gates OR_2, OR_9, OR_{14}, OR_{16}, OR_{18} and OR_{27} are equal because they have the same type and the same inputs. However, AND-gates $AND_3(x_1,2)$ and $AND_{28}(x_1,27)$ are not equal because they have inputs from different gates. To find that they are also equal we must determine that OR_2 and OR_{27} are equal and change the input 27 by input *2* in the description of AND_{28}. Therefore, to find that two gates are equal in a multilevel circuit we should find that their preceding gates are equal etc. For this reason we should rank the gates in the circuit.

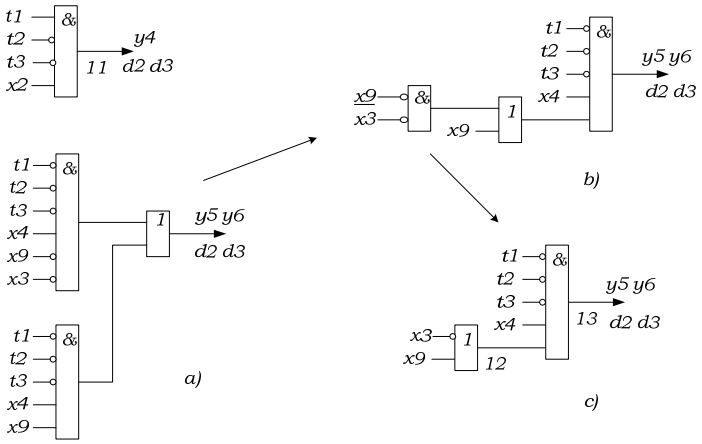


Figure 18. Logic circuit for transitions into state a_3

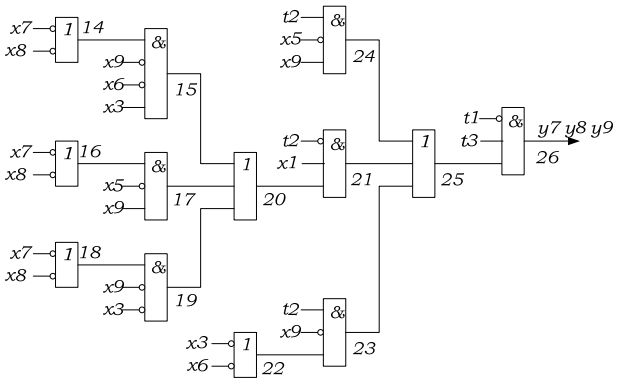


Figure 19. Logic circuit for transitions into state a_4

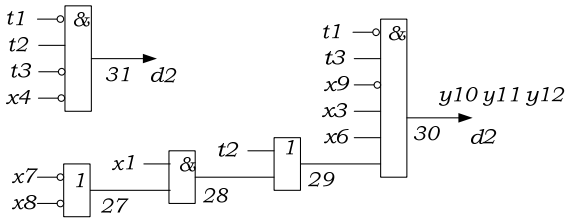


Figure 20. Logic circuit for transitions into state a_5

Gates containing only inputs $t_1, \ldots, t_R$ (the outputs of the memory elements, in our example $R = 3$) and input variables $x_1, \ldots, x_L$ (in our example $L = 9$) are referred to as gates of the first rank. The gates with inputs $t_1, \ldots, t_R$, input variables and the output of at least one gate of the first rank are referred to as gates of the second rank etc. Thus, the i-rank gate can have inputs $t_1, \ldots, t_R$, input variables and the inputs from outputs of gates with the rank less than $(i - 1)$ and at least one input from the gate with rank $(i - 1)$. The results of ranking for the circuit in Fig. 21 are presented in Table 9 and Fig.22. In this figure, the rank of gate is written above the gate.

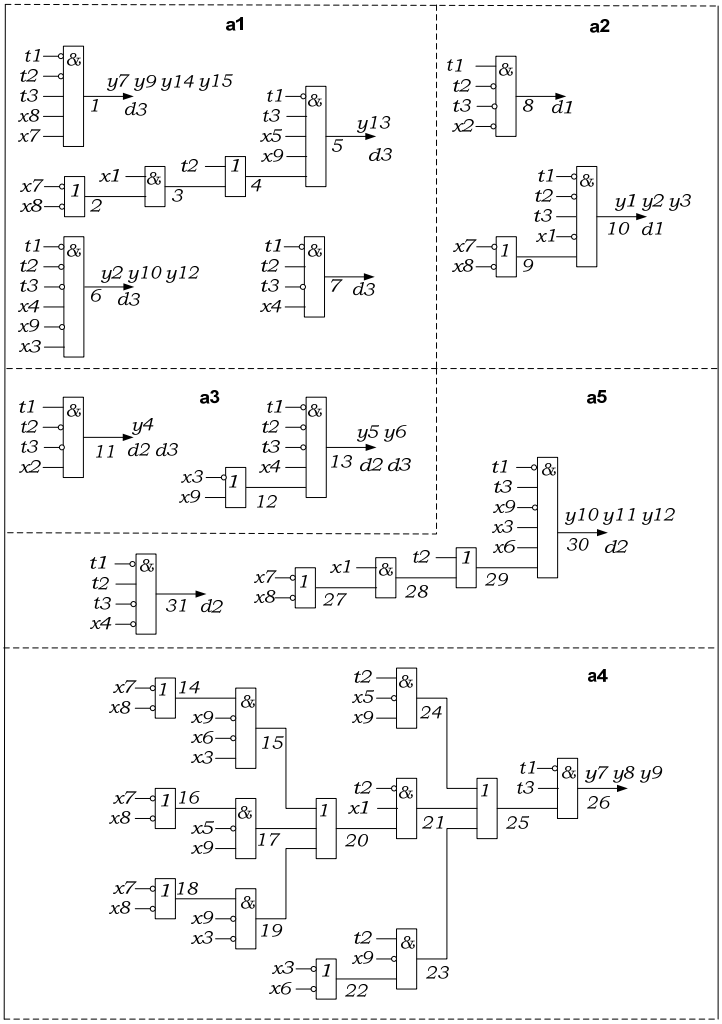


Figure 21. Logic circuit after factoring

Table 9. Ranks of gates

Rank	AND-gates	OR-gates
1	1, 6, 7, 8, 11, 24, 31	2 , 9, 12, 14, 16, 18, 22, 27
2	3, 10, 13, 15, 17, 19, 23, 28	
3		4, 20, 29
4	5, 21, 30	
5		25
6	26	

It is evident that equal gates can only be of the same rank. The following steps should be used to find and delete equal gates:

1. Find equal gates with rank i ($i = 1, 2, 3, ...$) beginning from rank 1, separately for AND-gates and OR-gates. In our example, we have the following set of equal first-rank gates:
$$OR_2 = OR_9 = OR_{14} = OR_{16} = OR_{18} = OR_{27}.$$
2. Remove all gates except the first one from each such set. Thus, after the first step we removed five gates OR_9, OR_{14}, OR_{16}, OR_{18}, OR_{27}. Replace the inputs from the gates thus removed with the number of the first (not removed) gate from the corresponding sets.
3. Repeat steps (1) – (3) for the elements of the $(i+1)$-*th* rank. We get equal AND-gates AND_3 and AND_{28} of the second rank and equal OR-gates OR_4 and OR_{29} of the third one.

The circuit after removal equal gates is shown in Fig. 23.

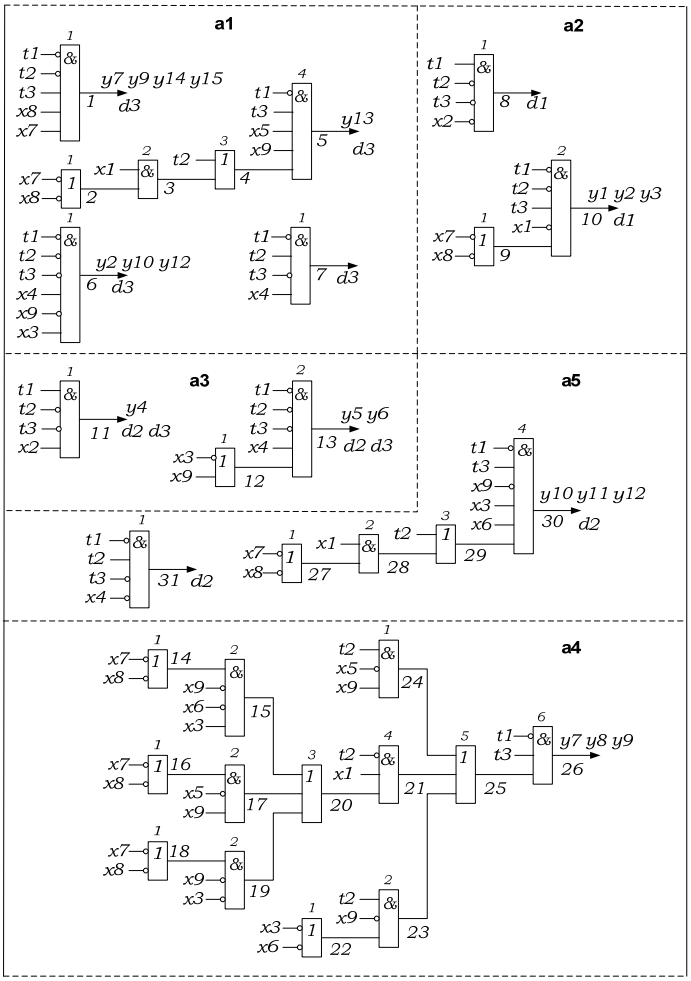


Figure 22. Ranking after factoring

Step 6. Repeat factoring and removing equal gates until the circuit cannot be change any longer.
Look at the circuit implementing the transitions into a_4 in Fig. 23. We drew the part of this circuit containing gates AND_{15}, AND_{17}, AND_{19} and OR_{20} in Fig. 24,a. After the removal of the equal gates, logic elements AND_{15}, AND_{17}, and AND_{19} got the same inputs from OR_2 instead of different inputs from OR_{14}, OR_{16}, and OR_{18}. Thereby, we got new possibilities for repeated factoring – see sequential steps of factoring in Fig. 24,b,c.

The circuit after the second factoring is shown in Fig. 25. Once again, after factoring we find the equal gates: $OR_{22} = OR_{32}$ and we remove the last one (Fig. 26). Thus, we
should repeat factoring and removing equal gates as long as we get these procedures are impossible for the circuit.

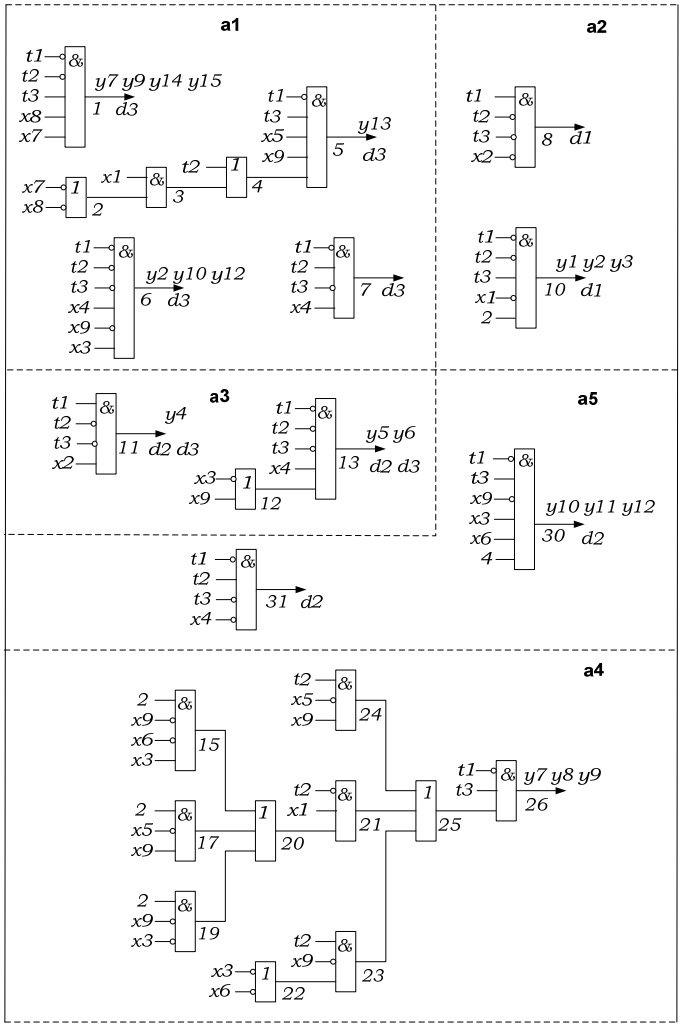


Figure 23. Logic circuit after removal equal gates

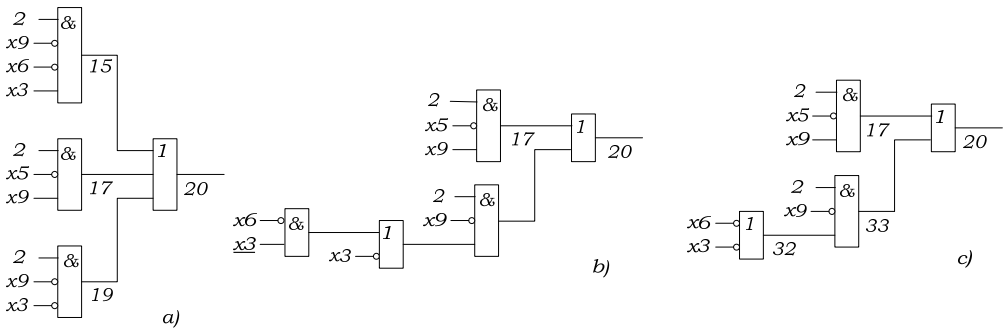


Figure 24. Repeated factoring

Step 7. Find the inclusion of gates into other gates. Unfortunately, we do not have such cases in our rather simple example.

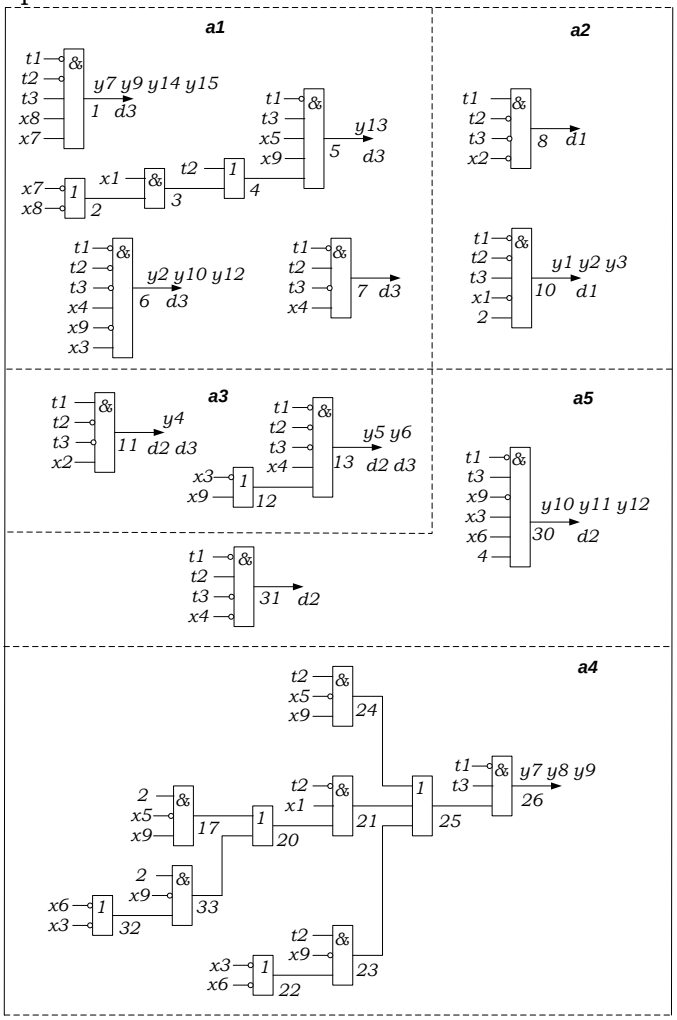


Figure 25. Circuit after the second factoring

Step 8. Make term decomposition for AND-gates. Fig. 27,a contains AND-gates with three and more inputs which we have selected from the circuit in Fig. 26 for term decomposition. It is evident that term decomposition makes it possible to find equal gates and inclusion of some gates into other ones as well. However, if a circuit contains many gates, term decomposition takes a lot of time and it is faster to implement steps 5 – 7 before term decomposition.

Let us demonstrate that the term decomposition problem may be divided into several independent subproblems. For this purpose, we define such a relation ω on the set of AND-gates that two gates AND_i and AND_j are in this relation iff they have not less than two common inputs.

Construct the graph G_ω of this relation (Fig. 28 for the circuit in Fig. 27,a). The vertices of this graph are the gates in Fig. 27. We connect two vertices by edge if the corresponding gates have two or more common inputs. From the definition of the relation ω, it is evident that there can be no common factors for the gates from various subgraphs of G_ω. Thus, the problem of term decomposition is divided into as many subproblems as the number of unconnected components in the graph G_ω. Even in our simple example, G_ω contains five components and there are only 11 vertices (gates) in the largest component. For a complex FSM, the graph G_ω contains a large number of components, since:

1. There is a large number of input variables in a complex FSM and there are not so many input variables in each row of its structure table (in each term corresponding to each row);
2. x_i and x'_i are different inputs of gates;
3. The number of gates and the number of inputs in each gate are decreased, as a result of steps 4 – 7 (factoring, removal equal gates and inclusion of gates into other ones).

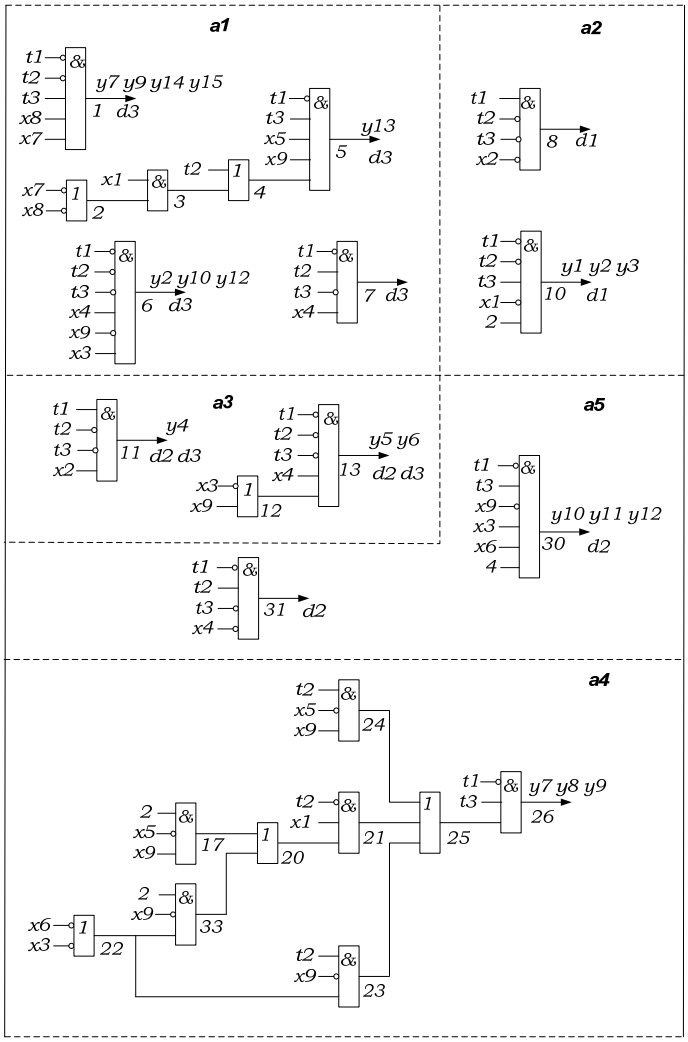


Figure 26. Circuit after the second removal equal gates

The result of term decomposition in our example is shown in Fig. 27,b. Fig. 29 contains the total circuit after this step.

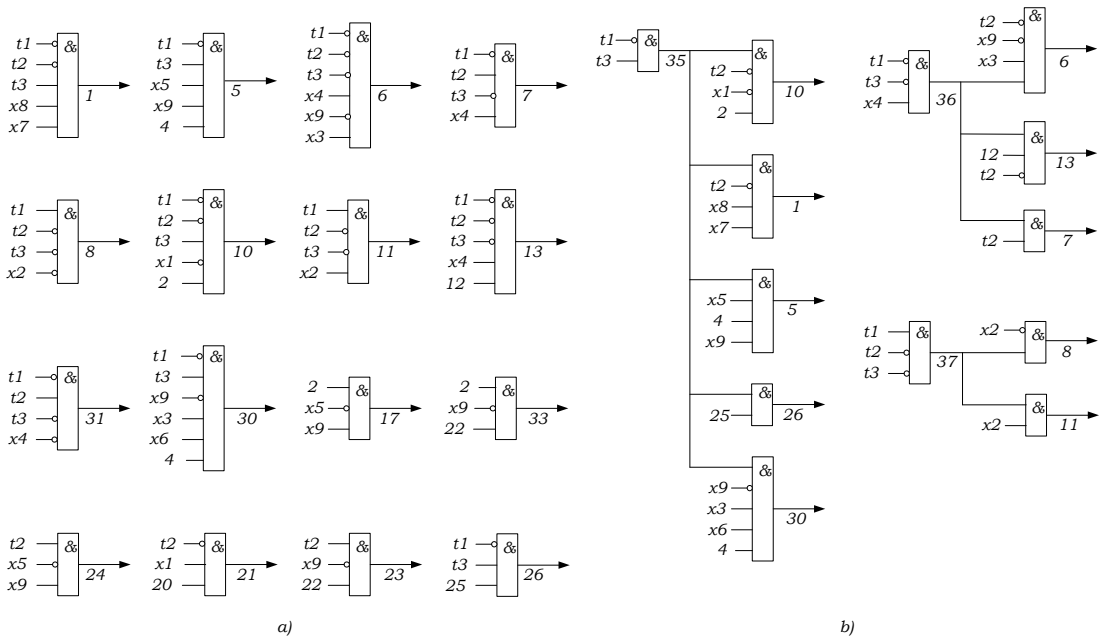


Figure 27. Term decomposition in our example

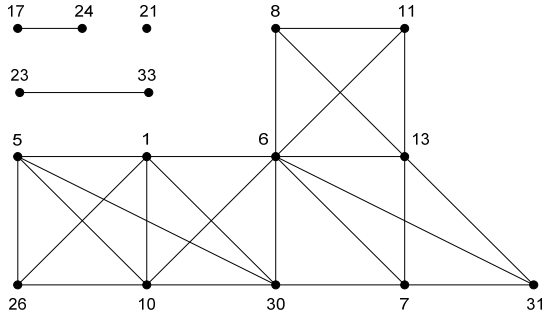


Figure 28. The graph G_ω of relation ω

Step 9. Construct OR-gate for each output variable y_n (n = 1, …, 15 in our example) and for each input memory function d_r (r = 1, …, 3 in our example) which occur more than once in the circuit after step 8. If we look at Fig. 29 we will find that several outputs appear more than once in this circuit. For example, y_2 is written at the outputs of gates AND_6 and AND_{10}, d_2 is written at the outputs AND_{11}, AND_{13}, AND_{30} and AND_{31}. It is evident that FSM has only one output y_2, so, first – the circuit in Fig. 29 is not the final circuit and, second – output y_2 will be equal to one when the output of AND_6 or the output of AND_{10} are equal to one. Thus, for y_2 and for each output that appears more than once in Fig. 29, we should construct OR-gate with the inputs connected to the outputs of gates where these signals are written.

To formalize this process we constructed Table 10 where each row contains the list of gates for each output. Now we can immediately construct OR-gates for the outputs which occur more than once in the logic circuit (Fig. 30,a).

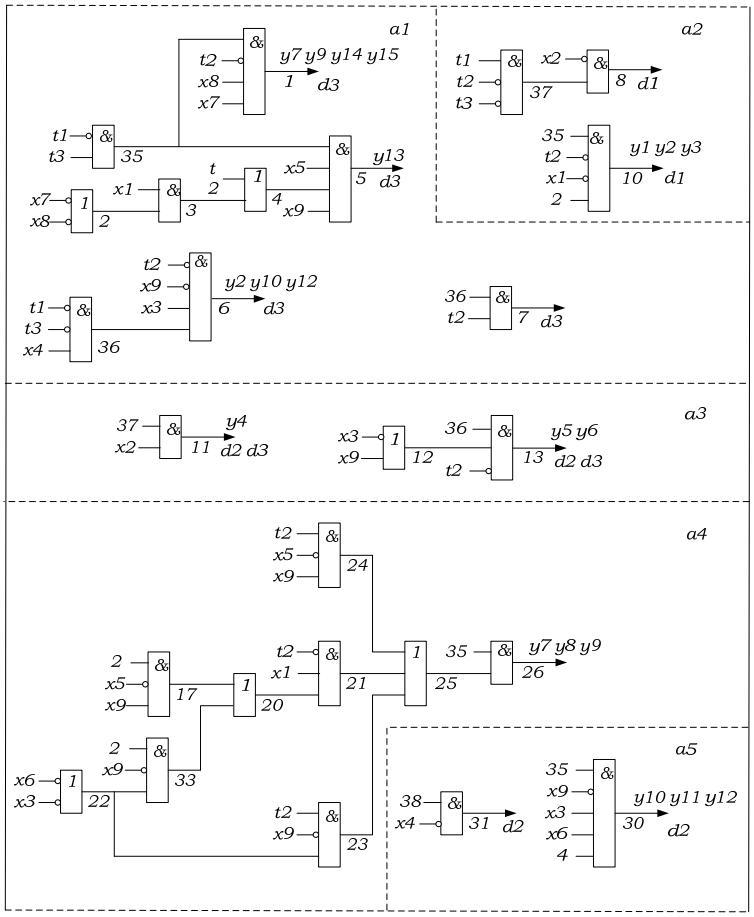


Figure 29. Logic circuit after term decomposition

Table 10. Gates for outputs

Outputs	*Gates*	*Outputs*	*Gates*	*Outputs*	*Gates*
y1	*e10*	*y7*	*e1 e26*	*y13*	*e5*
y2	*e6 e10*	*y8*	*e26*	*y14*	*e1*
y3	*e10*	*y9*	*e1 e26*	*y15*	*e1*
y4	*e11*	*y10*	*e6 e30*	*d1*	*e8 e10*
y5	*e13*	*y11*	*e30*	*d2*	*e11 e13 e30 e31*
y6	*e13*	*y12*	*e6 e30*	*d3*	*e1 e5 e6 e7 e11 e13*

Step 10. Find equal OR-gates among the gates constructed at step 9. Leave only one gate in each set of equal gates. The logic circuit after removal of equal gates is shown in Fig. 30,b.

Step 11. Find the inclusion of OR-gates into other gates among the gates constructed at steps 10. Unfortunately, we do not have any such cases in our rather simple example.

Step 12. Make term decomposition for all OR-gates. Just as at Step 8 for AND-gates, we consider here only the gates with not less than three inputs since the minimal number of inputs in a common term in term decomposition is equal to two (after removal equal gates and full inclusion). Similar to step 8, we should construct the graph of the relation ω for OR-gates. The problem of term decomposition for OR-gates is divided into as many subproblems as the number unconnected components in the graph of ω. In our rather simple example, we have only two OR-gates with more than two inputs (see *d2* and *d3* in Fig. 30,b).

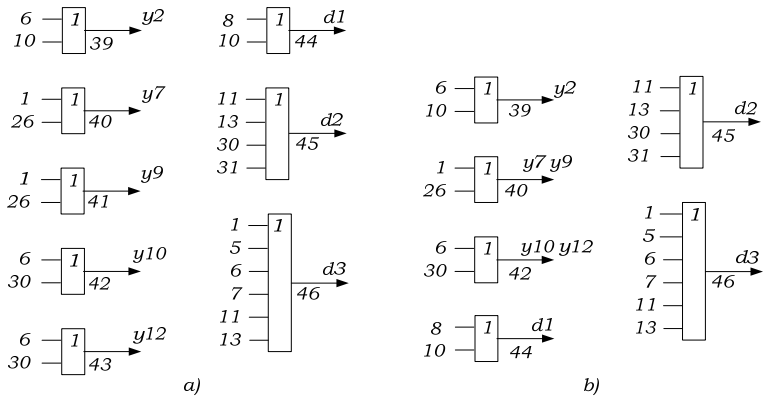


Figure 30. OR-gates before (a) and after (b) removal equal gates

The final logic circuit is shown in Fig. 31. We have placed the circuit from Fig. 30,b at the bottom of Fig. 31. Of course, we should remove appearances of the outputs y_2, y_7, y_9, y_{10}, y_{12} and input memory functions d_1, d_2, d_3 from other parts of the logic circuit. Thus, only the outputs that have one entry in the column "*Gates*" of Table 10 will be in the part of the circuit that is above the "*OR for outputs and input memory functions*" in Fig. 31.

Step 13. Relax and drink your coffee.

Really, the reason, that we cannot demonstrate gate inclusion and term decomposition for OR-gates, can be explained not only by the simplicity of our example, but also a very effective optimization at the previous steps that allows to decrease the number of inputs in the most gates of our circuit. To overcome some dissatisfaction of our last steps, let us discuss one more example, from the synthesis another FSM, presented in Table 11 and Fig. 32 with OR-gates for output variables and input memory functions which occur more than once in the circuit after Step 9.

Table 11. Gates for outputs in one more example

Outputs	*Gates*	*Outputs*	*Gates*
y1	*e5 e31 e36 e39*	*y8*	*e4 e44*
y2	*e3 e5 e36*	*y9*	*e16 e30*
y3	*e39 e56 e60*	*y10*	*e3 e16 e43*
y4	*e4 e31 e36 e42*	*d1*	*e56 e60*
y5	*e3 e16 e43*	*d2*	*e3 e4 e31 e36 e39 e44*
y6	*e16 e30*	*d3*	*e3 e4 e31 e42 e43*
y7	*e16 e44 e56*	*d4*	*e3 e4 e5 e42 e43 e44 e56*

Step 10a. From Table 11 or Fig. 32 we immediately get that $OR_{75} = OR_{68}$ and $OR_{76} = OR_{67}$. We remove OR_{75} and OR_{76} and get y_9 together with y_6 from OR_{68} and y_{10} together with y_5 from OR_{67} (Fig. 33).

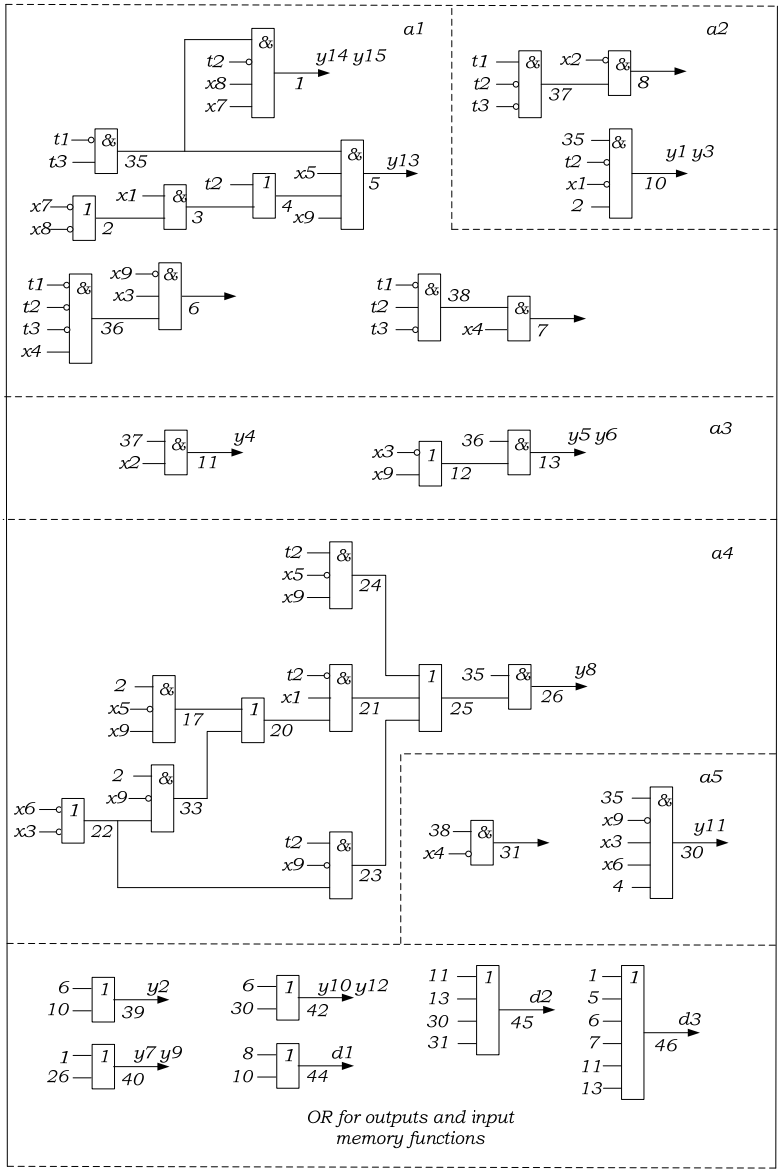


Figure 31. The final logic circuit

Step 11a. We checked full inclusion for OR gates and found that $OR_{70} \subset OR_{72}$, $OR_{70} \subset OR_{74}$ and $OR_{71} \subset OR_{65}$. The circuit after this step is presented in Fig. 34.

Step 12a. For term decomposition, we constructed the graph of relation ω for OR-gates with three and more inputs (Fig. 35). The problem of term decomposition for OR-gates is divided into as many subproblems as the number of unconnected components in the graph of ω. In our example we have two subgraphs and one of them is nontrivial.

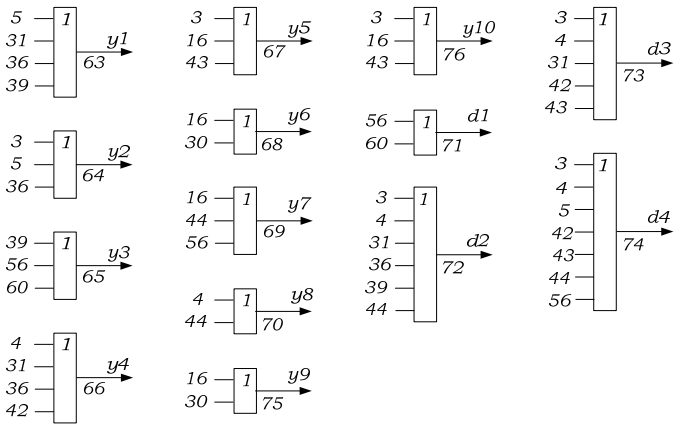


Figure 32. OR-gates for y_n and d_r in one more example

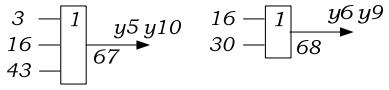


Figure 33. OR-gate transformation after removal of equal gates

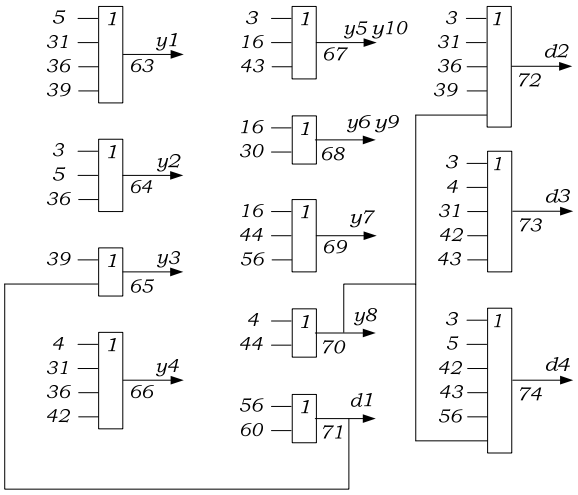


Figure 34. OR-gates after full inclusion

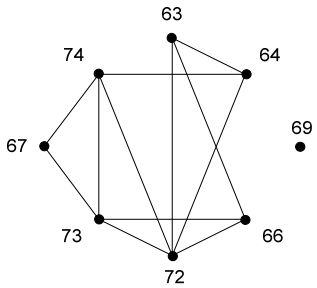


Figure 35. The graph of relation ω for OR-gates

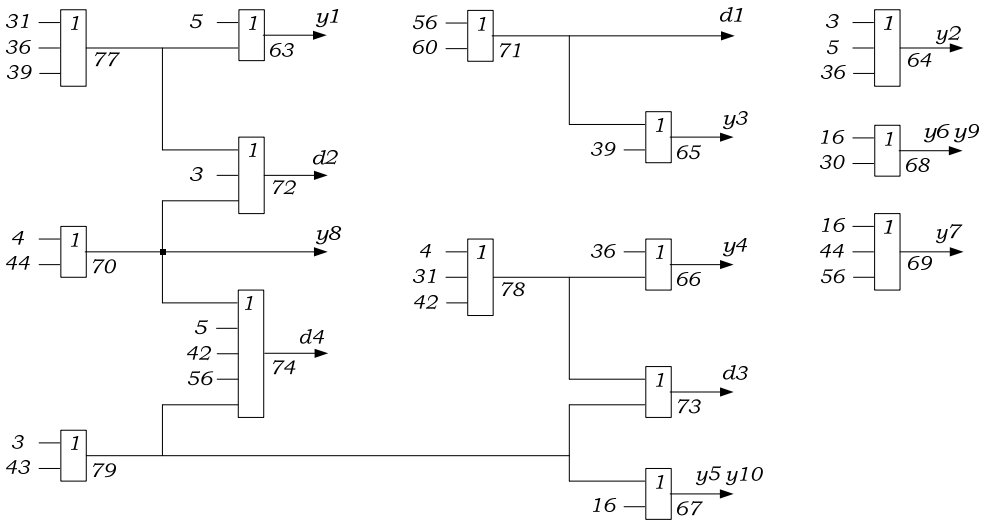


Figure 36. Logic circuit after term decomposition

The logic circuit after term decomposition is presented in Fig. 36. Its cost is 10 inputs lower than in the initial circuit in Fig. 32.