Applying Finite State Machine Theory and Object-Oriented Programming to the Logical Synthesis of Control Devices

Valery Sklyarov

Resumo **- Este Artigo descreve algumas técnicas aplicadas no desenvolvimento de unidades de controlo baseadas em dispositivos de lógica programável (PLD). Combinam a teoria de maquinas de estados finitas com a programação orientada a objectos, permitindo a introdução de novas facilidades, tais como: extensibilidade, flexibilidade e reutilização, durante o desenvolvimento. Estas técnicas recorrem o mais possível a estruturas predefinidas.**

Abstract **- This paper discusses some approaches to the design of control units based on programmable logic devices (PLD). They combine finite state machine theory with object oriented programming to allow control units to be designed with new facilities, such as: extensibility, flexibility and reuse. The techniques use predefined frames and basic schemes (templates) as far as possible.**

I. INTRODUCTION

There are many kinds of devices that can be decomposed into a **datapath** (execution unit) and **control units**. The datapath is composed of storage units (such as registers, counters, etc.) and combinational (or functional) units. A control unit performs a set of *instructions* by generating the appropriate sequence of *micro instructions* that depends on intermediate *logical conditions* or intermediate states of the datapath. Each instruction describes what operations must be applied to which operands stored in the datapath (or in external memory). In real applications, the control unit is often the **most complex** part of the design [1].

Consider the following problem: for a given set of instructions $\square=\{\square_1,...,\square_k\}$ and constraints $\nearrow=\{\nearrow_1,...,\nearrow_p\}$, design the control unit which will perform $\square$ and satisfy the set of conditions, $\nearrow$. There are many methods of logical synthesis that can be applied to solve this general problem [1-4]. Suppose it is also necessary to broaden our problem. We want to provide *extensibility*, *flexibility* and *reuse*. This implies accommodating the following additional requirements:

allow the set of instructions $\square$ to be extended after the control unit has been designed and produced;

allow the instructions in the set $\square$ to be changed after the control unit has been designed and produced;

enable previously designed components of the control unit to be used for future applications without redesigning it;

enable exception handling.

Let us discuss an approach that can be used to solve the extended problem. This approach is based on the results of earlier work, especially [3,4], and some of the ideas of object-oriented programming.

II. BASIC STRUCTURE OF THE CONTROL UNIT AND THE DESCRIPTION OF ITS BEHAVIOUR

The main objective of the design process can be expressed as follows: it is necessary to transform the given description of $\square$ to a scheme built from given functional elements. In order to describe the behaviour of the control unit we will use **graph-schemes** [3,4] that on the one hand are similar to algorithmic state machine notation [1] and on the other hand have some distinctions. Various kinds of graph-schemes allow you to describe *sequential* [3] and *parallel* [5] devices, the *duration of clock pulses* for synchronisation [6], the *hierarchical ordering* of operations to be performed [7], etc. They provide good separation of the control unit's **interface** from its **implementation**. Figure 1 shows an example of a graph-scheme which describes the behaviour of a control unit with 8 inputs and 11 outputs (see figure 2). Each node of the graph-scheme contains either a **micro instruction** Y_j (a set of micro operations, that are executed at the same time assigned to the node) or a **macro instruction** Z_i (a set of macro operations). Each macro operation can also be described using graph scheme notation [4,7].

In order to transform a given description into a scheme, it is necessary to apply some methods of logic synthesis that generally lead to the design of an **arbitrary** scheme. The approach considered is based on the use of so-called **predefined structures** containing programmable (or reprogrammable) components [3,4] (like PLA, PALs, GALs etc.) . For example, figure 3 shows a one-level PLD-based predefined structure (here $X = X^1\cup...\cup X^T$ - is the set of inputs, $Y = Y^1\cup...\cup Y^T$ - is the set of outputs, $\tau_1,...,\tau_R$, $D_1,...,D_R$ - are internal variables). Other predefined structures were considered in [3,4].

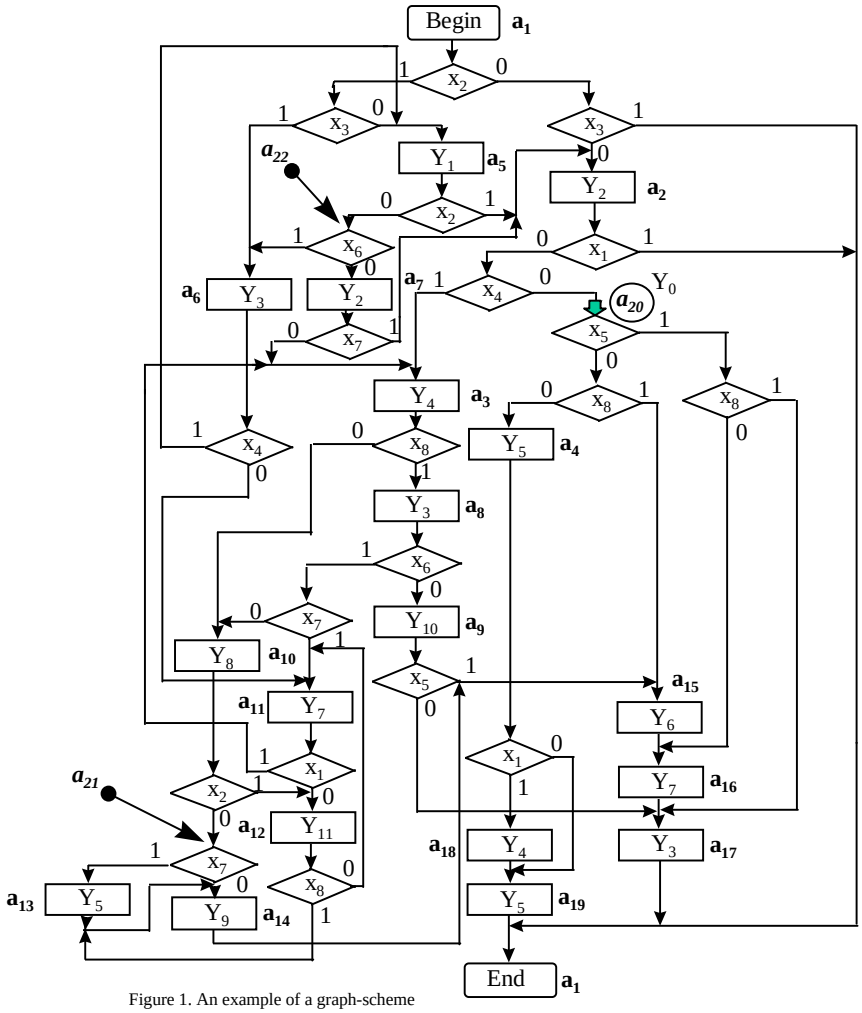


Figure 1. An example of a graph-scheme

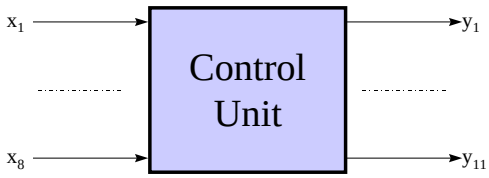


Figure 2. Control unit (external view)

This approach has several advantages. All groups of future connections are known and the complexity of the scheme only depends on the total number of components (PLAs, for instance). In practice, the final level of complexity can be assessed at the first stage of the design. The behaviour of the scheme (functional capabilities) can be extended by adding new components without reprogramming any of the components used (any addition does not change the structure). The behaviour of the scheme can be changed by replacing some components without reprogramming of the rest. Finally we can provide **extensibility**, **flexibility** and **reuse.**

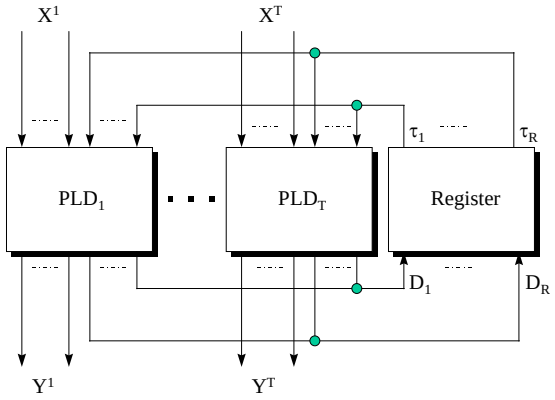


Figure 3. One-level PLD-based frame (PLD - is Programmable Logic Device like PLA, PAL, GAL, ROM, etc.).

III. BASIC STEPS OF LOGICAL SYNTHESIS

The approach being considered is based on the following steps of logic synthesis.

Step 1. Converting a graph-scheme to special **structural tables** that are used in a similar way to state transition diagrams of a finite state machine. The objective is to

satisfy the given constraints. Consider, for example, how this step is carried out for the structure shown in figure 3 For given graph-scheme Γ and PLA(n,m,q) (where n, m and q are the number of inputs, the number of outputs and the number of products PLA respectively) it is necessary to build a set of structural tables [3] such that: they describe the total behaviour represented by Γ, the number of tables is minimal; each table contains information for programming a corresponding PLA (it means that either all or part of the constraints have been satisfied). Let us consider the main ideas of the method which can be used to solve this problem [8]. Suppose we want to design a **Moore machine** and it is necessary to distribute input variables among PLAs (see figure 3). Consider a **graph G_I. Vertices** of G_I correspond to logical conditions (input variables). **Edges** of G_I express relationships between logical conditions. A relationship exists between two logical conditions if they must be included at least in one common product (in this case the weight of the edge is equal to 1). If they are included in more than one product the weight of edge is not equal to 1 (usually greater then 1). In order to solve our task G_I must be cut into a minimal number of independent subgraphs such that: a) each subgraph satisfies given constraints for the PLA, b) the weight of edges to be deleted is minimal. When we delete an edge we *add new state* (or states, dependently on the real weight of the edge). Suppose we have a transition from a_m to a_s and $X(a_m, a_s)$ is a product which forces this transition. In certain circumstances the approach considered enables us to replace this transition with two following transitions: $X(a_m, a_i)$ and $X(a_i, a_s)$, where a_i is a new (intermediate) state (so we split the first transition using an intermediate state). Very often it leads to better results. Figure 4 demonstrates how to build G_I for an arbitrary fragment of a graph-scheme. There are three edges with fractional weighting numbers. The value 1/3 for the numbers has been calculated because in figure 4,a the entry point of the rhombus that contains x_3 is connected with three other rhombi (which contain x_2, x_6 and x_7 respectively). All formal rules for calculating weights were considered in [4,8]. The total weight of the edge linked x_i with x_j in G_I is a sum of values calculated from the graph-scheme. In this case it is necessary to consider all pairs of rhombus that contain x_i (x_j) and x_j (x_i) and there is a directed line from output of x_i (x_j) to input of x_j (x_i).

The *sequence of actions* which are being performed is the following:

- **marking the given graph scheme** with labels $\mathbf{a_1,...,a_M}$. In order to do this we can use known methods [3,4]. For instance, if we are designing the Moore machine we are marking “Begin” and “End” nodes of the graph-scheme with a_1 and all other rectangular nodes with $a_2,...,a_M$. Any two different rectangular nodes must be marked with different tokens. After marking, the tokens $a_1,...,a_M$ are considered as the states of the Moore machine. Using this method for the graph scheme shown in figure 1, we have obtained the states $\mathbf{a_1,...,a_{19}}$;
- **building the graph G_I**. Figure 5,a shows the graph G_I to be built for the given graph-scheme (see figure 1);

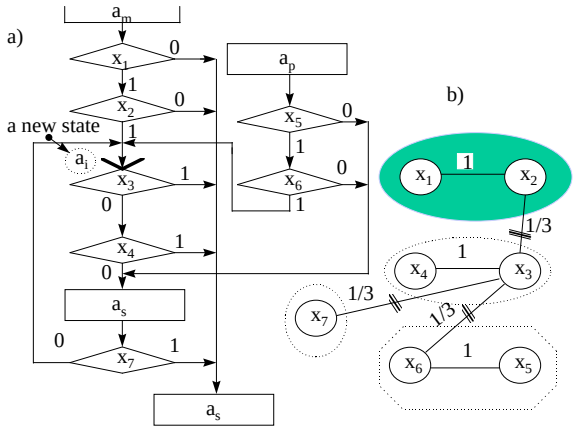


Figure 4. A fragment of a graph-scheme (a) and the graph G_I (b)

- **splitting transitions** if necessary. Consider PLA(9,8,25). In this case all constraints have been satisfied (R=]$\log_2$M[=5, n-R=9-5=4, our graph G_I has two non connected subgraphs, each subgraph contains not greater than 4 vertices). Therefore it is not necessary to delete some edges and to split transitions. Suppose we want to use another PLA(7,8,25). As a result it is necessary to cut G_I into sub graphs containing not greater than two vertices each (see figure 5,b). Now we have 4 sub graphs and it is necessary to add three extra states (a_{20},a_{21},a_{22}) which are also shown in figure 1. For future explanations and examples we will use PLA(9,8,25) and figure 5,a;

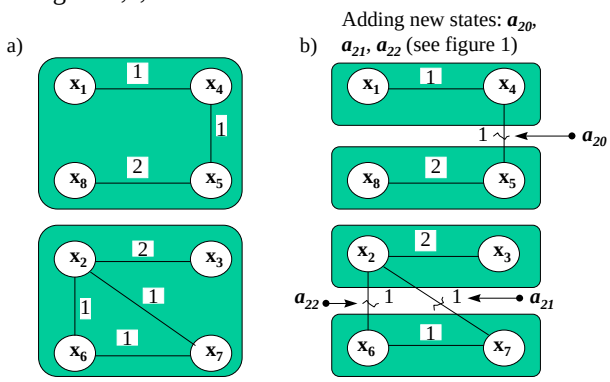


Figure 5. The graph G_I built for the graph-scheme in figure 1 in case of using PLA with 9 inputs (a) and with 7 inputs (b).

- **building structural subtables**. The number of subtables is equal to T+1, where T is the number of subgraphs in G_I (in our example we have 3 subtables). Each subtable has 7 columns which are the following: $\mathbf{a_m}$ - is an initial state, **Y** - are active outputs (for Moore machines the column **Y** can be combined with the column $\mathbf{a_m}$), $\mathbf{K(a_m)}$ is the code of

a_m, $\mathbf{a_s}$ is the next state, **K(a_s)** - is the code of a_s, **X(a_m,a_s)** - is a product of inputs that forces a corresponding transition, **F(a_m,a_s)** are active lines connected to a register (see figure 3). After the step 1 only columns $\mathbf{a_m}$, **Y**, $\mathbf{a_s}$, **X(a_m,a_s)** will be filled. Tables 1-3 containts three structural subtables $\mathbf{W_1}$, $\mathbf{W_2}$, $\mathbf{W_0}$ for our example. Subtables W_1 (table 1) and W_2 (table 2) will be later used for programming PLA_1 and PLA_2 respectively. Subtable W_0 (table 3) includes just **unconditional transitions** which will be later distributed between PLA_1 and PLA_2.

Table 1. W_1

a_m, Y	K(a_m)	a_s	K(a_s)	X(a_m, a_s)	F(a_m, a_s)
a_1 $Y_0=\Upsilon$	00000	a_1	00000	$\bar{x}_2 x_3$	-
		a_2	0100-	$\bar{x}_2 \bar{x}_3$	D_2
		a_5	10000	$x_2 \bar{x}_3$	D_1
		a_6	1100-	$x_2 x_3$	D_1D_2
a_5 (Y_1)	10000	a_2	0100-	x_2	D_2 D_5
		a_6	1100-	$\bar{x}_2 x_6$	D_1D_2 D_5
		a_7	10001	$\bar{x}_2 \bar{x}_6$	D_1 D_5
a_7 (Y_2)	10001	a_2	0100-	x_7	D_2 D_5
		a_3	00001	$\bar{x}_7$	D_5
a_8 (Y_3)	01100	a_9	00010	$\bar{x}_6$	D_4
		a_{10}	01010	$x_6 \bar{x}_7$	D_2 D_4
		a_{11}	10010	$x_6 x_7$	D_1 D_4
a_{10} (Y_8)	01010	a_{12}	00011	x_2	D_4D_5
		a_{13}	01011	$\bar{x}_2 x_7$	D_2 D_4D_5
		a_{14}	10011	$\bar{x}_2 \bar{x}_7$	D_1 D_4D_5

Step 2. **State encoding** (assignment). The objective is to reduce the *functional dependency* of outputs on inputs. This will allow us to reduce the number of output lines used in programmable components, and to simplify the scheme of the control unit. The method is based on the use of special tables that on the one hand look like the Karnaugh maps, but on the other hand are quite different (they were introduced in [9]). Consider all transitions **A(a_m)** from the state $\mathbf{a_m}$. For any structural table the following expression is true:

$$\bigvee_{a_s \in A(a_m)} X(a_m, a_s) \equiv 1$$

Consider all codes K(a_m,a_s) for which $a_s \notin A(a_m)$. If for given a_m bit k in all codes K(a_m,a_s) has values either 0 and don't care (-), or 1 and don't care, then k does not depend on input variables from the set X [9]. It leads us to the following method of coding. For each structural subtable W_t ($0<t\le T$) it is necessary to find a bifurcation $\pi_H^t = \{H_{t1}, H_{t2}\}$, for which

$H_{t1} \cup H_{t2} = H = \{h_1,...,h_R\}$, $H_{t1} \cap H_{t2} = \varnothing$, and $h_1,...,h_R$ - correspond to bits 1,...,R of codes from column K(a_s), $H_{11} \cup ... \cup H_{T1} = H$, for all $t \in \{1,...,T\}$, $|H_{t1}| = \max$. If $h_r \in H_{t2}$ then h_r satisfies the requirement considered above.

Figure 6 gives a simple example of state encoding, where $\pi_H^1 = \{\{h_1,h_2\},\{h_3,h_4\}\}$, $\pi_H^2 = \{\{h_4\},\{h_1,h_2,h_3\}\}$, $\pi_H^3 = \{\{h_3\},\{h_1,h_2,h_4\}\}$, $|\{h_3,h_4\}|=2$, $|\{h_1,h_2,h_3\}|=3$ $|\{h_1,h_2,h_4\}|=3$, $\{h_1,h_2\} \cup \{h_4\} \cup \{h_3\} = \{h_1,h_2,h_3,h_4\} = H$

Table 2. W_2

a_m, Y	K(a_m)	a_s	K(a_s)	X(a_m, a_s)	F(a_m, a_s)
a_2 (Y_2)	0100-	a_1	00000	x_1	-
		a_3	00001	$\bar{x}_1 x_4$	D_5
		a_4	00100	$\bar{x}_1 \bar{x}_4 \bar{x}_5 \bar{x}_8$	D_3
		a_{15}	00101	$\bar{x}_1 \bar{x}_4 \bar{x}_5 x_8$	D_3D_5
		a_{16}	00110	$\bar{x}_1 \bar{x}_4 x_5 \bar{x}_8$	D_3D_4
		a_{17}	00111	$\bar{x}_1 \bar{x}_4 x_5 x_8$	$D_3D_4D_5$
a_3 (Y_4)	00001	a_8	01100	x_8	D_2 D_3
		a_{10}	01010	$\bar{x}_8$	D_2 D_4
a_4 (Y_5)	00100	a_{18}	11010	x_1	D_1D_2 D_4
		a_{19}	11011	$\bar{x}_1$	$D_1D_2D_4D_5$
a_6 (Y_3)	1100-	a_5	10000	x_4	D1
		a_{11}	10010	$\bar{x}_4$	D1 D4
a_9 (Y_{10})	00010	a_{15}	00101	x_5	D_3D_5
		a_{17}	00111	$\bar{x}_5$	$D_3D_4D_5$
a_{11} (Y_7)	10010	a_3	00001	x_1	D_5
		a_{12}	00011	$\bar{x}_1$	D_4D_5
a_{12} (Y_{11})	00011	a_7	10001	x_8	D_1 D_5
		a_{14}	10011	$\bar{x}_8$	D_1 D_4D_5

Table 3. W_0

a_m, Y	K(a_m)	a_s	K(a_s)	X(a_m, a_s)	F(a_m, a_s)
a_{13}(Y_5)	01011	a_{14}	10011	1	$D_1D_4D_5$
a_{14}(Y_9)	10011	a_{15}	00101	1	D_3D_5
a_{15}(Y_6)	00101	a_{16}	00110	1	D_3D_4
a_{16}(Y_7)	00110	a_{17}	00111	1	$D_3D_4D_5$
a_{17}(Y_3)	00111	a_1	00000	1	-
a_{18}(Y_4)	11010	a_{19}	11011	1	$D_1D_2D_4D_5$
a_{19}(Y_5)	11011	a_1	00000	1	-

Finally the codes to be assigned enable for each $\mathbf{a_m}$ select a proper transition to $\mathbf{a_s}$ using only bits **K(a_m,a_s)** from $\mathbf{H_{t1}}$. In order to perform state assignment, according to the rules considered above, we can use spesial **tables μ_t** [9], shown in figure 6 (table μ_t is used for the structural table W_t). Rows of each table μ_t correspond to various possible

codes of H_{t1}; columns of each table μ_t correspond to various possible codes of H_{t2}; all states of W_t from each set $A(a_m)$ must be accommodated in the same column of μ_t. Figure 7 demonstrates states encoding for our example. All subsets *$A(a_m)$ are highlighted* either with grey shading or with a double arrow header line pointing to corresponding states. The resulting codes are placed in columns *$K(a_m)$*, *$K(a_s)$* of our tables (see tables 1-3). The tables μ_1 and μ_2 have been filled with the states according to the algorithm [4,9]. The bits of codes that depend only on states and don't depend on inputs are *highlighted* in the columns *$K(a_s)$*. The columns $F(a_m,a_s)$ have been filled with characters $D_1,...,D_5$ on the assumption that the register in figure 3 is being composed of **D flip-flops**. The values of the components $D_1,...,D_5$ that depend only on states are *highlighted* in the columns *$F(a_m,a_s)$*;

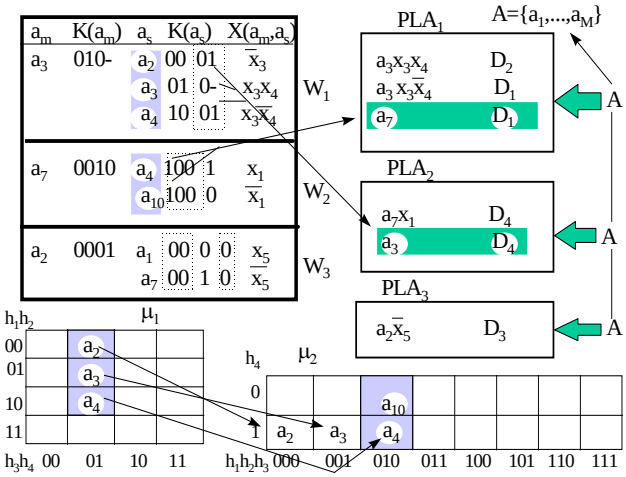
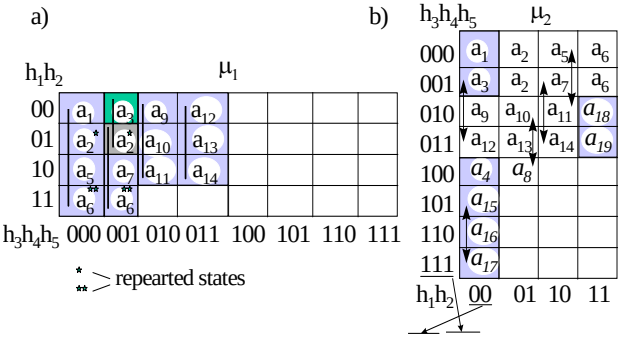


Figure 6. State encoding and PLAs programming



The code of the state a_{17} is: 00111.

Figure 7. Encoding tables: a) for the first structural table; b) for the second structural table

Finally the **PLA_1** has outputs **D_1,D_2** and the **PLA_2** - outputs **D_3,D_4,D_5**. We have occupied 2+3=5 outputs and 6+5=11 outputs are still free and can be used for assignment variables from the set Y. Fragments marked with 1 in figure 8 show the implementation of non highlighted functions D_1,D_2 from the table W_1 for the PLA_1 and non highlighted functions D_3,D_4,D_5 from the table W_2 for the PLA_2.The highlighted functions D_1,D_2 from W_2 and W_0 (tables 2,3) are implemented in the PLA_1 according to the following table:

state	state code	D_1D_2
a_{13}	01011	1 0
a_{18}	11010	1 1
a_3	00001	0 1
a_4	00100	1 1
a_6	1100-	1 0
a_{12}	00011	1 0

The highlighted functions D_3,D_4,D_5 from W_1 and W_0 (tables 1,3) are implemented in the PLA_2 according to the following table:

state	state code	$D_3D_4D_5$
a_{13}	01011	0 1 1
a_{14}	10011	1 0 1
a_{15}	00101	1 1 0
a_{16}	00110	1 1 1
a_{18}	11010	0 1 1
a_5	10000	0 0 1
a_7	10001	0 0 1
a_8	01100	0 1 0
a_{10}	01010	0 1 1

Now we can minimise $D_1,...,D_5$:

state code	D_1D_2	state code	$D_3D_4D_5$
0-011	1 0	0101-	0 1 1
11010	1 1	10011	1 0 1
00001	0 1	00101	1 1 0
00100	1 1	00110	1 1 1
1100-	1 0	11010	0 1 1
		1000-	0 0 1
		01100	0 0 1

Fragments marked with 2 in figure 8 show the implementation of the highlighted functions D_1D_2 from the tables W_2, W_0 for the PLA_1 and the highlighted functions $D_3D_4D_5$ from the tables W_1, W_0 for the PLA_2.

Step 3. **Combinational logic optimization and designing the final scheme**. The main ideas are based on a special *decomposition* aimed at using predefined frames in spite of constructing schemes with an arbitrary structure. Dependent on the particular structure to be selected, the following methods can be applied: distribution of variables (usually output variables) among components; adding new components and reorganizing the previous distribution; boolean function minimisation.

Suppose that our *micro instructions* are the following: $\mathbf{Y_0}=\varnothing$, $\mathbf{Y_1}=\{y_4,y_7,y_9,y_{11}\}$, $\mathbf{Y_2}=\{y_1,y_3,y_7\}$, $\mathbf{Y_3}=\{y_4\}$, $\mathbf{Y_4}=\{y_5,y_8\}$, $\mathbf{Y_5}=\{y_1,y_9\}$, $\mathbf{Y_6}=\{y_{10}\}$, $\mathbf{Y_7}=\{y_5,y_6\}$, $\mathbf{Y_8}=\{y_3,y_7,y_{11}\}$, $\mathbf{Y_9}=\{y_2,y_5,y_6,y_8\}$, $\mathbf{Y_{10}}=\{y_2,y_5,y_{10}\}$, $\mathbf{Y_{11}}=\{y_2\}$. Let us attempt to distribute 11 output variables $y_1,...,y_{11}$ among 11 free outputs of the PLA_1 and the PLA_2. We can use for these purposes the method suggested in [3]. Consider such sets of states Q_t, t=1,...,T which can be recognised on the outputs of our PLAs after their preliminary programming (see fragments 1 and 2 in figure 8). These sets are given below in the form *[state(s)]code_of_the_state(s)* for the PLA_1 and the PLA_2 respectively:

$\mathbf{Q_1}$={{[$a_{12} \vee a_{13}$]0-011, [a_{18}]11010, [a_3]00001, [a_4]00100, [a_6]1100-, [a_5]10000};

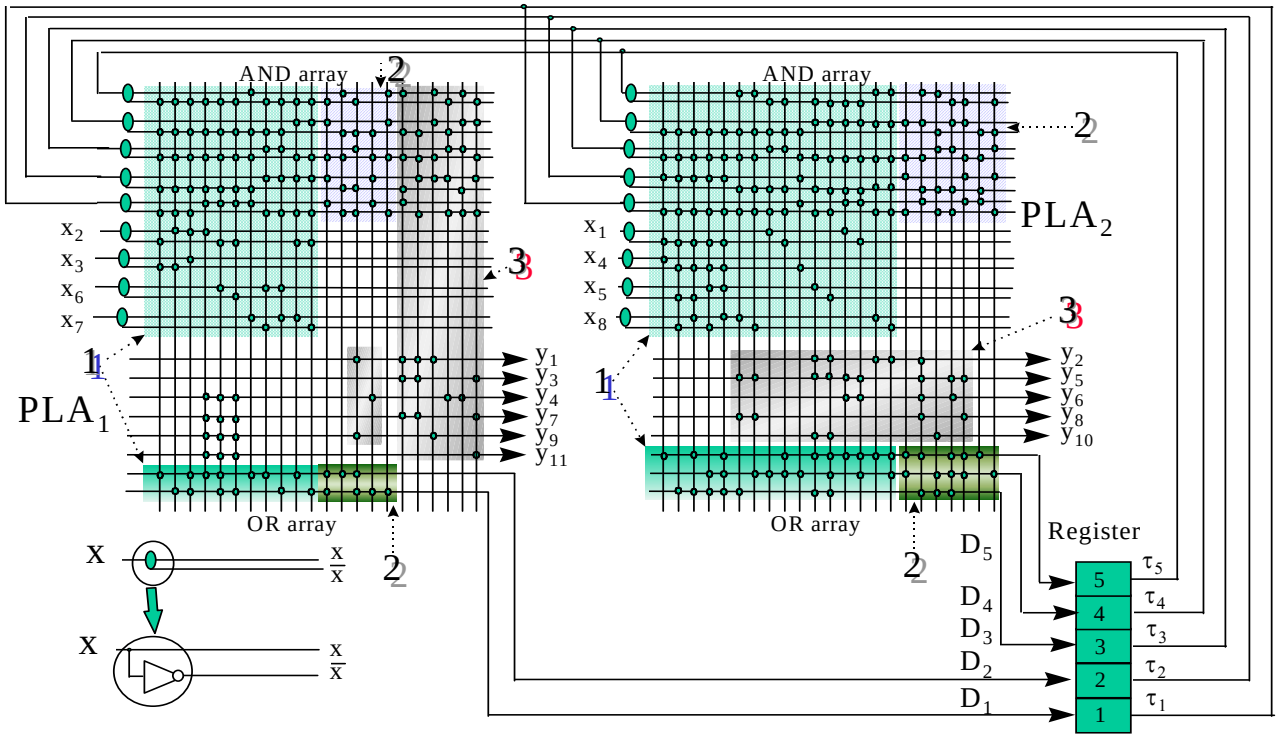


Figure 8. Final scheme of the control unit, described in figure 1

$\mathbf{Q_2}$={{[$a_{13} \vee a_{10}$]0101-, [a_{14}]10011, [a_{15}]00101, [a_{16}]00110, [a_{18}]11010, [$a_5 \vee a_7$]1000-, [a_8]01100, [a_3]00001, [a_4]00100, [a_9]00010, [a_{11}]10010, [a_{12}]00011}.

For each set $\mathbf{Q_t}$ consider the set $\mathbf{E_t}$ that contains elements written in the form: *output_variable(the set of states in which the output_variable has an active value)*. If the state(s) in E_t belong(s) to one element of Q_t then we strike out it (them). The sets E_1 and E_2, built for our example, are the following:

$\mathbf{E_1}$={$\mathbf{y_1}$(a_2,~~a_4~~,a_7,a_{13},a_{19}), $\mathbf{y_2}$(a_9,a_{12},a_{14}}, $\mathbf{y_3}${a_2,a_7,a_{10}), $\mathbf{y_4}$(~~a_5~~,~~a_6~~,a_8,a_{17}), $\mathbf{y_5}$(~~a_3~~,a_9,a_{11},a_{14},a_{16},~~a_{18}~~), $\mathbf{y_6}$(a_{11},a_{14},a_{16}), $\mathbf{y_7}$(a_2,~~a_5~~,a_7,a_{10}), $\mathbf{y_8}$(~~a_3~~,a_{14},~~a_{18}~~), $\mathbf{y_9}$(~~a_4~~,~~a_5~~,a_{13},a_{19}), $\mathbf{y_{10}}$(a_9,a_{15}), $\mathbf{y_{11}}$(~~a_5~~,a_{10})};

E_2={$\mathbf{y_1}$(a_2,~~a_4~~,a_7,a_{13},a_{19}), $\mathbf{y_2}$(~~a_9~~,~~a_{12}~~,~~a_{14}~~}, $\mathbf{y_3}${a_2,a_7,a_{10}), $\mathbf{y_4}$(a_5,a_6,~~a_8~~,a_{17}), $\mathbf{y_5}$(~~a_3~~,~~a_9~~,~~a_{11}~~,~~a_{14}~~,~~a_{16}~~,~~a_{18}~~), $\mathbf{y_6}$(~~a_{11}~~,~~a_{14}~~,~~a_{16}~~), $\mathbf{y_7}$(a_2,a_5,a_7,a_{10}), $\mathbf{y_8}$(~~a_3~~,~~a_{14}~~,~~a_{18}~~), $\mathbf{y_9}$(~~a_4~~,a_5,a_{13},a_{19}), $\mathbf{y_{10}}$(~~a_9~~,~~a_{15}~~), $\mathbf{y_{11}}$(a_5,a_{10})};

If all states in E_t for y_m are struck out, then y_m can be assigned to an output of the PLA_t [3] without the use of any new products. When we apply this rule, the variables $\mathbf{y_2}$, $\mathbf{y_5}$, $\mathbf{y_6}$, $\mathbf{y_8}$, $\mathbf{y_{10}}$ will be assigned to outputs of the PLA_2 (see the right fragment 3 in figure 8). After that all outputs of the PLA_2 are occupied.

Micro operations $\mathbf{y_1}$, $\mathbf{y_3}$, $\mathbf{y_4}$, $\mathbf{y_7}$, $\mathbf{y_9}$, $\mathbf{y_{11}}$ to be left can be considered as boolean functions of the variables τ_1,...,τ_5 which can be expressed after minimisation in the form of the following matrixes:

	$\mathbf{y_1}$ $\mathbf{y_3}$ $\mathbf{y_4}$ $\mathbf{y_7}$ $\mathbf{y_9}$ $\mathbf{y_{11}}$
10001	1 1 0 1 0 0
0100-	1 1 0 1 0 0
-1011	1 0 0 0 1 0
01010	0 1 0 1 0 1
01100	0 0 1 0 0 0
00111	0 0 1 0 0 0

These matrixes can be directly realised in PLA_1 (see the left fragment 3 in figure 8). If some constraints for outputs and (or) products of PLAs are not satisfied we can apply other methods, considered, for example in [3,4]. They are aimed at adding new components and reorganising the previous assignment.

IV. SYNCHRONISATION PROBLEM

Let's return to Step 1 (see section III). Suppose we want to use another PLA(7,8,25). In this case the graph G_l, has to be cut into 4 sub graphs, shown in figure 5,b. As a result we are adding *three extra states* (a_{20}, a_{21}, a_{22}) in figure 1 and we are splitting some previous transitions. Consider, for example, the transition from a_5 to a_7 (see figure 9). Before splitting we had just one transition ($a_5 \rightarrow a_7$) and after splitting we have two transitions ($a_5 \rightarrow a_{22} \rightarrow a_7$). The final duration of the transition from a_5 to a_7 has been *increased twice*. If it causes a problem we should change duration of the second transition ($a_{22} \rightarrow a_7$). The main idea is the following. Consider the set of states $A=A_r \cup A_e$, where A_r is the set of original states and A_e is the set of extra states. For our example we have: A_r ={a_1,...,a_{19}}, A_e ={a_{20},...,a_{22}}. Let us divide all transitions

into two groups. The first group is composed of **synchronous transitions** from $a_m \in A_r$. The second group is composed of **asynchronous transitions** from $a_i \in A_e$. Asynchronous transitions are performed immediately after setting the register (see figure 3) in any state $a_i \in A_e$ and independently of clocks. The combinational part of the finite state machine has special output (indicator) which *indicates* setting the register in extra states. The signal from the **indicator** is used either to force an asynchronous transition or to decrease the duration of the clock for the next transition [6] (see figure 10). The last idea is also used for changing frequency of clocks if we want to *change duration of various macro instructions*. We can use for these purposes different engineering decisions (considered, for instance, in [6]) or self-synchronous approach.

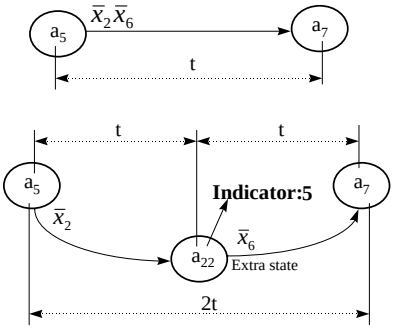


Figure 9. Transitions splitting problem

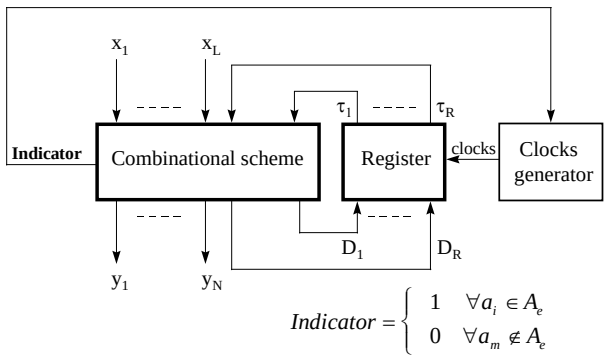


Figure 10. Changing clocks duration or making asynchronous transitions

V. MULTILEVEL SCHEMES BASED ON PREDEFINED FRAMES

All **multilevel schemes** are based on the *one-level scheme*, shown in figure 3. The combinational part in figure 3 can be built from various reprogrammable elements, such as PLAs, PALs, ROMs, GALs and other PLDs. The *common structure* of the multilevel scheme is given in figure 11. Two non filled blocks of the structure (combinational scheme and register) are taken from figure 3 with all the necessary connections. There are also three additional blocks that are a **Coder (C)**, a **Selector (S)** and a **Decoder (D)**. These blocks can be either included onto the final structure or not, depending on the approach which we are going to use for logical synthesis [4,10].

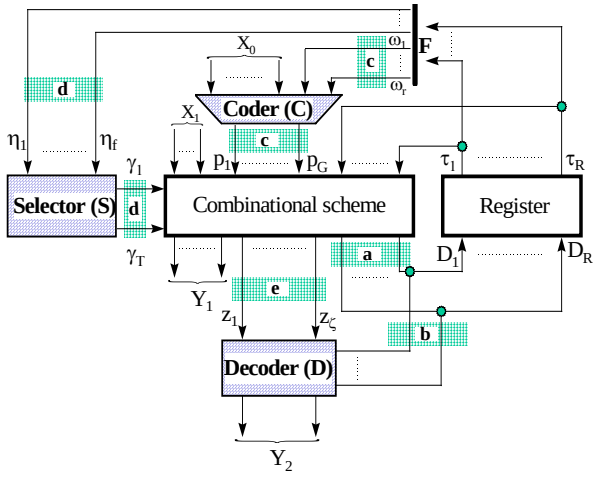


Figure 11. Common structure of the multilevel scheme

The **first approach** is related to *micro instruction encoding* [4]. In this case we will use only one additional block D which can be constructed from either PLD [4] or standard decoders [11]. The connections b, c, d will be eliminated from figure 11 and the final scheme will be composed of the one-level sub scheme and the block D with the connections e. The process of synthesis can be separated into the following steps.

Step 1. Building sub tables $W_0,W_1,...,W_T$ (see section III, step 1).

Step 2. State encoding (see section III, step 2).

Step 3. Micro instruction encoding. The characters $Y_1,...,Y_M$ are assigned codes containing $\zeta=]\log_2 M[$ bits, where $]a[$ is the nearest integer greater than or equal to a. The combinational scheme generates components $z_1,...,z_\zeta$ for the codes. The block D converts the codes to the corresponding values of micro operations. Some of the micro operations can be either assigned or not on outputs of the combinational scheme (see subset Y_1 in figure 11). Finally $Y_1 \cup Y_2 = Y = \{y_1,...,y_M\}$. In order to perform micro instruction encoding we can apply the same method that has been already considered for the state encoding (see section III, step 2).

Step 4. Distributing micro operations between subsets Y_1 and Y_2 (if necessary). For these purposes we can use the approach to be considered in the section III (see section III, step 3).

If we intend to build D from standard decoders, it is worth-while to minimise the total number of decoders. The corresponding task can be transformed into building and *colouring* the special graph which reflects the relationships between various micro operations [11].

The **second approach** is aimed at *structural table lines encoding* [10]. In this case we will also use only block D which can be based on PLD (ROM in particular). The connections a, c, d will be deleted from figure 11 and the final scheme will be composed of the one-level sub

scheme and the block D with the connections b and e. The process of synthesis can be separated into the following steps.

Step 1. Building sub-tables $W_0,W_1,...,W_T$ (see section III, step 1).

Step 2. Structural sub-table lines encoding. Each line of such sub-tables is assigned a token $e_i \in \{e_1,...,e_I\}$, where *I* is the total number of lines in all structural sub-tables $W_0,W_1,...,W_T$ (see the column E_i of the tables 4-6 below). The characters $e_1,...,e_I$ are assigned codes containing $\zeta=]\log_2 I[$ bits. The combinational scheme generates components $z_1,...,z_\zeta$ for the codes. The block D converts the codes to the corresponding values of $D_1,...,D_R$ and micro operations from the set Y_2. As before, some of micro operations can either be assigned or not on outputs of the combinational scheme (see subset Y_1 in figure 11). In order to perform line encoding, we can use the same method that has been already considered for state encoding (see section III, step 2). For instance, the results of line encoding for tables 4-6 is demonstrated in figure 12.

Step 3. Distributing micro operations between subsets Y_1 and Y_2 (if necessary). For these purposes we can use the approach to be considered in section III (see section III, step 3).

Table 4. W_1

a_m	$K(a_m)$	a_s	$K(a_s)$	E_i	$X(a_m, a_s)$	$P(a_m, a_s)$
a_1	0000-	a_1	00**00-**	e_1	$\bar{x}_2 x_3$	01
		a_2	01**0-0**	e_2	$\bar{x}_2 \bar{x}_3$	00
		a_5	10-**00**	e_3	$x_2 \bar{x}_3$	10
		a_6	11**0-0**	e_4	$x_2 x_3$	11
a_5	10-00	a_2	01**0-0**	e_5	x_2	10
		a_6	11**0-0**	e_6	$\bar{x}_2 x_6$	01
		a_7	10**01-**	e_7	$\bar{x}_2 \bar{x}_6$	00
a_7	1001-	a_2	01**0-0**	e_8	x_7	01
		a_3	00**010**	e_9	$\bar{x}_7$	00
a_8	00101	a_9	00**100**	e_{10}	$\bar{x}_6$	00
		a_{10}	01**10-**	e_{11}	$x_6 \bar{x}_7$	10
		a_{11}	11**100**	e_{12}	$x_6 x_7$	11
a_{10}	0110-	a_{12}	01**011**	e_{13}	x_2	10
		a_{13}	00**011**	e_{14}	$\bar{x}_2 x_7$	01
		a_{14}	11**011**	e_{15}	$\bar{x}_2 \bar{x}_7$	00

Table 5. W_2

a_m	$K(a_m)$	a_s	$K(a_s)$	E_i	$X(a_m, a_s)$	$P(a_m, a_s)$
a_2	010-0	a_1	00**00-**	e_{16}	x_1	10
		a_3	10**001**	e_{17}	$\bar{x}_1 x_4$	01
		a_{20}	11**001**	e_{18}	$\bar{x}_1 \bar{x}_4$	00
a_{20}	11001	a_4	00**110**	e_{19}	$\bar{x}_5 \bar{x}_8$	00
		a_{15}	01**110**	e_{20}	$\bar{x}_5 x_8$	01
		a_{16}	10**110**	e_{21}	$x_5 \bar{x}_8$	10
		a_{17}	11**110**	e_{22}	$x_5 x_8$	11
a_3	***10001 00010***	a_8	00**101**	e_{23}	x_8	01
		a_{10}	01**10-**	e_{24}	$\bar{x}_8$	00
a_4	00110	a_{18}	10**101**	e_{25}	x_1	01
		a_{19}	11**101**	e_{26}	$\bar{x}_1$	00
a_6	110-0	a_5	10-**00**	e_{27}	x_4	01
		a_{11}	11**100**	e_{28}	$\bar{x}_4$	00
a_9	00100	a_{15}	01**110**	e_{29}	x_5	01
		a_{17}	11**110**	e_{30}	$\bar{x}_5$	00
a_{11}	11100	a_3	10**001**	e_{31}	x_1	01
		a_{12}	01**0-1**	e_{32}	$\bar{x}_1$	00
a_{12}	010-1	a_7	10**01-**	e_{33}	x_8	01
		a_{14}	11**011**	e_{34}	$\bar{x}_8$	00

Table 6. W_0

a_m	$K(a_m)$	a_s	$K(a_s)$	E_i	$X(a_m, a_s)$
a_{13}	00011	a_{14}	**11011**	e_{35}	1
a_{14}	11011	a_{15}	**01110**	e_{36}	1
a_{15}	01110	a_{16}	**10110**	e_{37}	1
a_{16}	10110	a_{17}	**11110**	e_{38}	1
a_{17}	11110	a_1	**0000-**	e_{39}	1
a_{18}	10101	a_{19}	**11101**	e_{40}	1
a_{19}	11101	a_1	**0000-**	e_{41}	1

It should be mentioned that block D is especially useful for **Mealy machines**, because the values $z_1,...,z_\xi$ depend on both states and input variables. These variables can be directly used to generate micro operations on outputs of block D in Mealy machines.

The **third approach** enables us to perform *replacement of input variables from the set* $X=\{x_1,...,x_N\}$. In this case we will use only one additional block C which can be constructed from either PLD [4] or standard multiplexers [10]. The connections b, d, e will be eliminated from figure 11 and the final scheme will be composed of the one-level sub scheme and C with the connections c. The process of synthesis can be separated into the following steps.

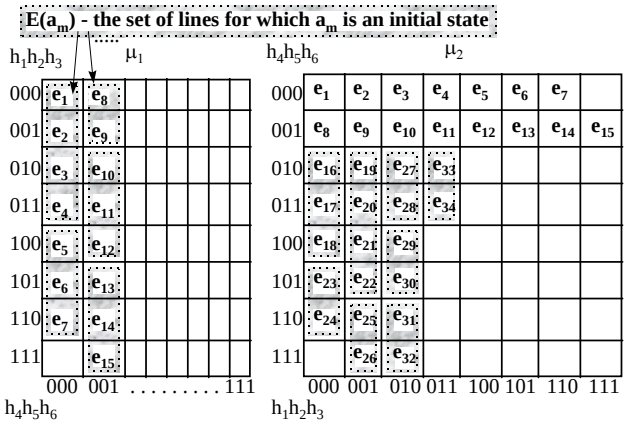


Figure 12. Structural sub table lines encoding

Step 1. Building structural table for the given graph-scheme. You can use for these purposes any known method (see, for instance, [3]).

Step 2. Replacing input variables with new variables from the set P={p_1,...,p_G}, $G =]\max_{a_m \in A}|\log_2 A(a_m)|[$. The tables 4,5 above demonstrate such replacement (see the column **P(a_m,a_s)**). As a result we have obtained the following boolean functions:

$$p_1 = a_1x_2 \vee a_5x_2 \vee a_8x_6 \vee a_{10}x_2 \vee a_2x_1 \vee a_{20}x_5;$$

$$p_2 = a_1x_3 \vee a_5\bar{x}_2x_6 \vee a_7x_7 \vee a_8x_6x_7 \vee a_{10}\bar{x}_2x_7 \vee a_2\bar{x}_1x_4 \vee a_{20}x_8 \vee a_3x_8 \vee a_4x_1 \vee a_6x_4 \vee a_9x_5 \vee a_{11}x_1 \vee a_{12}x_8;$$

Step 3. State encoding. The objective is minimising the functions from the set **P** (the functions **p_1,...,p_G**). The particular methods of encoding were considered in [4].

Step 4. Distributing micro operations between elements of the first level (see the section III, step 3). The methods of this step for Mealy machines were considered in [3,4].

The **fourth approach** is related to using of *mutually exclusive elements* in the scheme of the first level [10]. In this case we will use one additional block S which can be constructed from either PLD or standard decoders. The connections a, b, c, e will be eliminated from figure 11 and the final scheme will be composed of the one-level sub-scheme and S with the connections d. Consider non-intersecting subsets A_1,...,A_T and $A_1 \cup ... \cup A_T = A$. Let us perform separate state encoding in each *subset* A_t. Usually the length of code is less than in case of state encoding in the total set A. Consider the set $\gamma = \{\gamma_1,...,\gamma_T\}$ of variables such that $\gamma_t=1$ if and only if the control unit is in the state $a_m \in A_t$, and $\gamma_t=0$ in the opposite case (t=1,...,T). As a result, element$_t$ knows a real state from the set A by analysing both the state from A_t and γ_t (see figure 11). In this case the number of inputs for each element$_t$ of the one-level scheme is equal to $]\log_2|A_t|[+1$ and often less than R (especially for complex control units). The methods of synthesis of such schemes were introduced in [4,10]. The element F in figure 11 allows us to reduce the total number of lines (r<R, f<R). For some schemes this element denotes just special connections, for instance, part ϑ_1 of lines τ_1,...,τ_R are connected to the combinational scheme and another part ϑ_2 to either the Coder or the Selector. Depending on a particular scheme we can provide either $\vartheta_1 \cap \vartheta_2 \neq \varnothing$ or $\vartheta_1 \cap \vartheta_2 = \varnothing$.

The PLD can also include **internal memory** (internal register). They are called programmable logic sequencers. For such kinds of devices we can also consider one-level and multilevel structures. The corresponding methods of synthesis, based on using predefined frames, were suggested in [12].

VI. TEMPLATES (PREDEFINED BASIC SCHEMES)

The idea of **basic schemes** for control units were introduced in [13] and later developed in [3,4,14]. They contain elements with *changeable functions* (like PLD) which are initially *undefined*. All external connections of elements are *fixed* and they can not be changed. Basically each particular scheme can be considered as a **template** for, generally speaking, an infinite number of different applications. The *customising* of the base scheme (implementing, for instance, a graph-scheme that describes a particular algorithm of control) is carried out by programming (reprogramming) its elements with changeable functions.

In order to construct the basic scheme, it is necessary to estimate all the likely constraints for future applications. In other words we should define a **class** of applications and the constraints for the class. These constraints are the following [4]: the maximum number of input variables L_{max}; the maximum number of output variables N_{max}, the maximum number of states M_{max}, the maximum number of flip-flops in the register R_{max}, the maximum number of lines in the total structural table, etc. Consider, for example, the *one-level basic scheme* suggested in [13] (see the figure 13).

The combinational part of the scheme is composed of PLAs and ROMs (the ROMs are used just in order to generate values of output variables and functions D_{r+1},...,D_R). The number of PLAs and ROMs is calculated by evaluating the given constraints. The value r determines the maximum number of transitions σ from one state ($\sigma \leq 2^r$). In order to fix this value we can also test our constraints. The possible superfluity is eliminated by installing just the elements used in a particular scheme and erasing all extra components that are not required. This is possible without changing connections between elements. All assertions related to the scheme in figure 13 were proved in [4,13]. So we just demonstrate how to apply known methods in order to implement given graph-scheme (see figure 1) in the basic scheme (see figure 13) with the following parameters: T=2, Q=1, r=2, R=5, PLA(10,5,25), ROM(5,10), where for the ROM were given the number of inputs or address size (5) and the number of outputs (10).

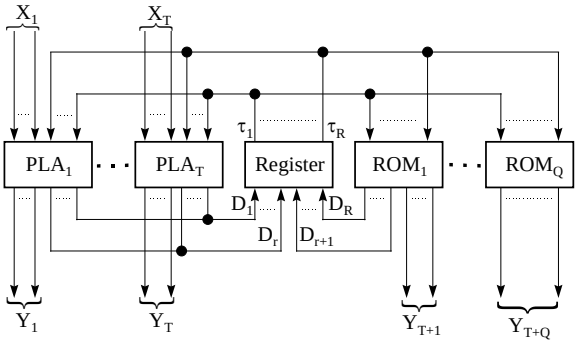


Figure 13. One-level basic scheme

The methods, considered in [4,13], include the following sequence of steps.

Step 1. Building sub tables $W_0,W_1,...,W_T$.

Step 2. State encoding.

Step 3. Distributing output variables and designing the final scheme.

In order to build sub tables $W_0,W_1,...,W_T$ we can use the basic approach considered in Section III with trivial modifications [4,13]. Because r=2 it is not possible to perform six transitions from the state a_2 (see table 2). So we have to split these transitions using the rules [4]. As a result a new state $\mathbf{a_{20}}$ has been added.

The step 2 has some distinctions which are explained below in detail. Consider the graph $\mathbf{G_\xi}$ which reflects the following relationships:

$$(a_m \xi a_s) \Leftrightarrow A(a_m) \cap A(a_s) \neq \varnothing.$$

Vertices of G_ξ correspond to states from the set A. Two vertices a_m and a_s are connected with an edge if and only if $\mathbf{(a_m \xi a_s)}$. Each vertex a_m has been added with the set $\mathbf{A(a_m)}$ and all vertices for which $|A(a_m)|<2$ have been eliminated. Each edge has been assigned the set $\mathbf{A(a_m,a_s)}$ which is determined as follows: $A(a_m,a_s)=A(a_m)\cap A(a_s)$. The final graph G_ξ is shown in figure 14.

Let us build a new compressed graph $\tilde{G}_\xi$ which contains *joined vertices of the* G_ξ. The vertices a_m, a_s,...,a_k can be joined if and only if $|A(a_m)\cup A(a_s)\cup...\cup A(a_k)|\le 2^r$. If a_m, a_s,...,a_k are joined, the new common vertex corresponds to the set $A(a_m)\cup A(a_s)\cup...\cup A(a_k)$. Each edge of the $\tilde{G}_\xi$ connected vertices v_m and v_s has the *weight* $\rho(v_m,v_s)$ which is calculated as follows:

$$\rho(v_m,v_s) = \sum_{a_k \in A(v_m,v_s)} |A(a_k)|,$$

where $A(v_m,v_s)$ is the set of common states in vertices v_m and v_s.

The final graph $\tilde{G}_\xi$ is shown in figure 15. Let us mark the number of vertices of $\tilde{G}_\xi$ with δ. Our task can be solved if $\mathbf{\delta \le 2^{R-r}}$. In the most circumstances we can satisfy this constraint. In [4,13] was proved that the problem of state encoding can be transformed into the **mapping** of the graph $\tilde{G}_\xi$ onto an R-r dimensional **cube** $\mathbf{C_\mu}$ (see figure 16,a). If we are able to solve this task we can directly fill the single encoding table μ (see figure 16,b).

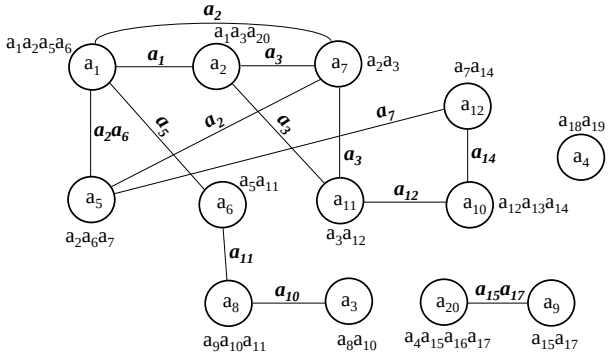


Figure 14. Graph G_ξ

Some states, such as a_1, a_2, a_{12}, a_{10}, a_7, a_5, a_6, are repeated in the table μ twice. They are located in *neighbouring corners* of C_μ and therefore, they are assigned the codes having don't care components. Just one (underlined) state a_3 is located in the *diagonal* of C_μ and a_3 was assigned two different binary codes that are 10001 and 00010. This is allowed, but all transitions from the state a_3 are repeated twice (from two states 10001 and 00010). That is why when we map the $\tilde{G}_\xi$ onto the C_μ we aspire to *minimise* the total weight of diagonal (non neighbouring) edges [4].

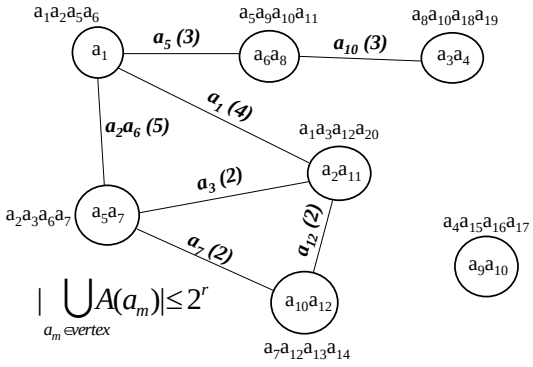


Figure 15. Compressed graph $\tilde{G}_\xi$

The results of state encoding are shown in the columns $K(a_m)$, $K(a_s)$ of the tables 4-6. All bits whose values don't depend on input variables are highlighted with a bold font.

The step 3 can be performed using the basic approach considered in Section III. Finally our scheme consists of two PLAs and one ROM. The PLAs can be directly programmed using the tables 4-6 and explanations given in Section III. Figure 17 demonstrates the ROM to be programmed just for the outputs D_3, D_4, D_5. Consider some examples of programming. For all transitions from the state $\mathbf{a_5}$ we must set $\mathbf{D_4=1}$, $\mathbf{D_3=D_5=0}$. Because the third bit of a_5 has don't care value we are using two addresses

of the ROM 10**0**00 and 10**1**00. For both addresses we are programming outputs as 010. For all transitions from the state $\mathbf{a_3}$ we must set $\mathbf{D_3=D_5=1}$, $\mathbf{D_4=0}$. Because the state a_3 has two codes (each of them indicates the same state a_3) we are using two addresses of the ROM 10001 and 00010. For both addresses we are programming outputs as 101. The same approach has been used to implement the transition from the set a_3 in the PLA_2.

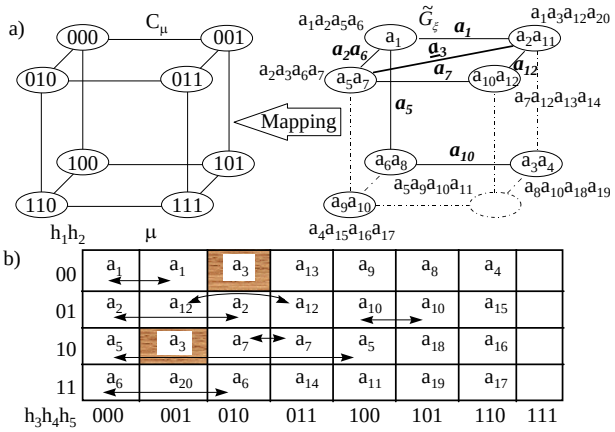


Figure 16. Mapping of the graph $\tilde{G}_\xi$ onto the cube C_μ (a) and table μ (b)

All predefined frames, considered in Section V, can also be investigated as a foundation for *multilevel basic schemes*. Some of them were suggested in [3,4]. Let's examine, for example, the basic scheme shown in figure 18 and based on the one-level scheme and the block C (see figure 11).

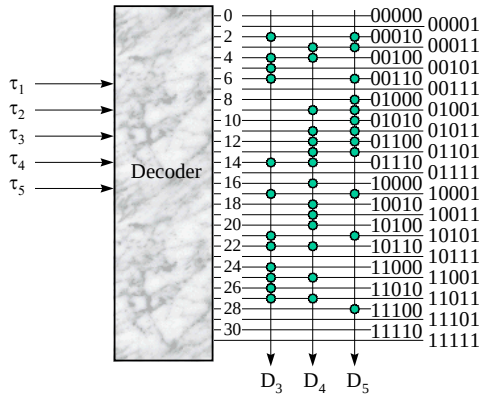


Figure 17. The ROM of the basic scheme to be programmed for D_3-D_5

The PLAs (PLA_1,...,PLA_T) are being considered as a PLA(z,q), where $z=n+m$ - the total number of external pins which are either inputs or outputs. The ROMs (ROM_0,ROM_1,...,ROM_Q) are being considered as a ROM(n,m), where n - is the number of inputs (an address size) and m - is the number of outputs. They are used in order to generate values of the functions D_1,...,D_R and output variables from the set Y. The basic function of the PLAs are a replacement of variables from the set X with new variables from the set P and |X|>>|P|.

The input lines of ROM_0 are p_1,...,p_G,τ_1,...,τ_R. Some of the lines p_1,...,p_G can be logically connected to the lines τ_1,...,τ_R providing the function OR. Such connections are admissible if and only if there is no ambiguity between various transitions. The obvious way to prevent ambiguity is the following. Consider a vector with the elements τ_1,...,τ_r,c_{r+1},...,c_R, p_{R-r+1},...,p_G, where c_{r+1},...,c_R are common variables. Suppose we are taking into account all transitions from a state $\tilde{a}$ having the code $\tilde{\tau}_1 ... \tilde{\tau}_r \tilde{\tau}_{r+1} ... \tilde{\tau}_R$ and $\tilde{\tau}_{r+1}=...=\tilde{\tau}_R=0$. If the vector $\tilde{\tau}_1 ... \tilde{\tau}_r \tilde{\tau}_{r+1} ... \tilde{\tau}_R$ can be unambiguously identified (recognised) by examining just the components $\tilde{\tau}_1 ... \tilde{\tau}_r$ then all transitions from the $\tilde{a}$ can also be unambiguously identified by examining the vectors τ_1,...,τ_r,c_{r+1},...,c_R,p_{R-r+1},...,p_G. Finally it follows that all vectors coming to the inputs of ROM_0 that provide different transitions, must be *orthogonal*. The problem of searching for *orthogonal vectors* that satisfy the requirement mentioned above is not so difficult, and can be resolved when we are carrying out the *state encoding*.

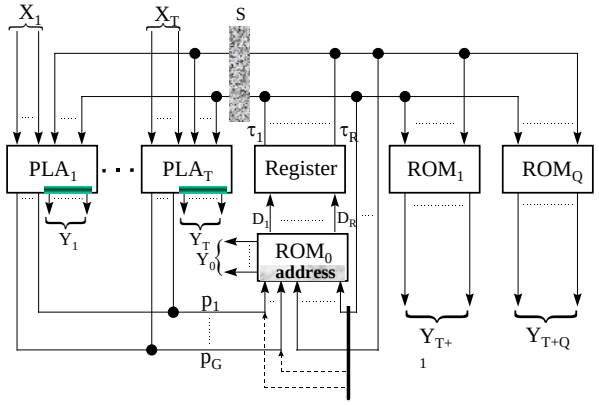


Figure 18. Two-levels predefined (basic) scheme

The scheme in figure 18 can be used for many applications (for implementing many different algorithms of logical control). Therefore we have to define its basic parameters (see the beginning of this section). For instance, in order to define the number of new variables G we should estimate the maximum number of transitions from any one state in future algorithms (graph-schemes).

Suppose we have already built the basic scheme with the structure shown in figure 18, that has the following predefined parameters: R=5, G=2, PLA(15,20), ROM(6,16), the pins τ_5 and p_1 are logically connected. The synthesis of the control unit for the given graph-scheme (see figure 1) can be separated into the following steps.

Step 1. Marking the given graph-scheme for designing either the *Moore machine* (see the section III) or the *Mealy machine* [3]. Building the structural table.

Step 2. Replacing variables from the set X with variables from the set P (see the section V).

Step 3. State encoding. The objective is to optimise (usually to minimise) the functions p_1,...,p_G and to satisfy the requirements considered above at this section.

Step 4. Micro operation assignment and synthesis of the final scheme.

Suppose we have already built the structural table and performed input variable replacement (see tables 4-6). For our current task we can consider the three tables as a single one (because we can use just one PLA). The results of state encoding, which satisfy all the necessary requirements (see the step 3), are shown in the **Karnaugh map** in figure 19. The final scheme is composed of two PLDs. The first one is the PLA and the second one is the ROM. Figures 20, 21 demonstrate the results of PLD programming (see figure 20 for the PLA and figure 21 for the ROM). The states, from which only unconditional transitions are being performed have been underlined in the right part of figure 21. Two neighbouring horizontal lines of ROM with addresses XXXXX0 and XXXXX1 will be programmed identically for such states.

The functions $\mathbf{p_1}$ and $\mathbf{p_2}$ are the following (they were taken from the section V and minimised):

$$p_1 = \overline{\tau}_3\overline{\tau}_4\overline{\tau}_5x_2 \vee a_8x_6 \vee a_2x_1 \vee a_{20}x_5;$$

$$p_2 = \tau_3\overline{\tau}_4\overline{\tau}_5x_8 \vee \overline{\tau}_1\tau_2\overline{\tau}_3\tau_4x_1 \vee a_1x_3 \vee a_5\overline{x}_2x_6 \vee a_7x_7 \vee a_8x_6x_7 \vee a_{10}\overline{x}_2x_7 \vee a_2\overline{x}_1x_4 \vee a_6x_4 \vee a_9x_5;$$

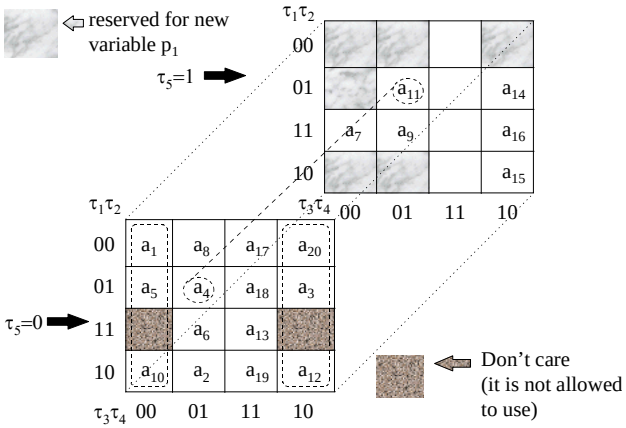


Figure 19. The Karnaugh map for states encoding

VII. EXCEPTION HANDLING

Exceptions indicate something unusual or unexpected in an execution unit. They are caused either by errors or by something that requires an immediate assistance. Exceptions are detected in an execution unit during run-time and are indicated by special variables from the set X. If an indicator is in active state, the control unit immediately interrupts the executing of the control algorithm and handles the respective exception. After the exception has been handled, the control unit continues the execution of the algorithm from the interrupted point. If the control unit has no idea how to cope with an exception it indicates an unrecoverable error requiring external assistance.

A scheme which supports an exception handling mechanism is shown in figure 22. The memory of the scheme has a *multilevel structure* (it may be based on a stack [15], for instance). If any exception has taken place, the memory is switched to the level which is responsible for exception handling. After an exception is being handled the memory will be switched back to the interrupted point of the main algorithm. These actions are similar to *push* and *pop* operations with a **stack**.

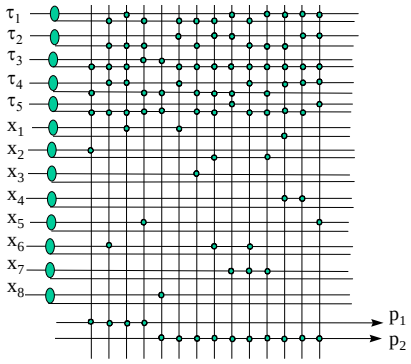


Figure 20. The programming of PLA for the two-levels basic scheme (see figure 18)

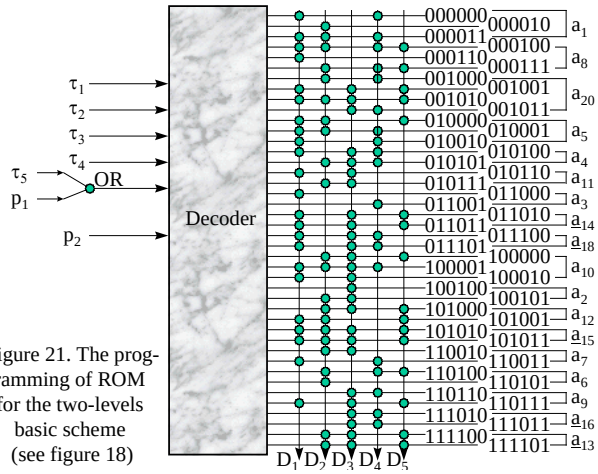


Figure 21. The programming of ROM for the two-levels basic scheme (see figure 18)

In order to design a control unit based on stack memory we can invoke the general approach suggested in [4,7,15]. It provides a way of explicitly separating an exception handling algorithm (graph-scheme) from an ordinary algorithm (graph-scheme).

IIX. INHERITANCE AND PROTECTION

There is only one known basic way of dealing with complexity: “Divide and conquer”. This famous idea can be applied in a variety of ways. A complex hardware unit in general has a **hierarchical structure** of control and can be seen in different levels of abstraction, such as *micro operations level*, *macro operations level*, etc. For example, for the computer we can distinguish micro operations, assembly language instructions, operating system service functions (application programming interface), etc. As a result we are representing a process of controlling in a different *hierarchical levels*. In each

particular level we can distinguish between the outside view and the inside view of the control part (generally speaking this part can be considered as either a hardware unit or a software component). The **interface** of a control part provides its outside view and therefore emphasises the abstraction while hiding its structure. By contrast, the **implementation** of a control part is its inside view, which encompasses the secrets of its behaviour. The interface can be divided into *accessible, partially accessible* and *non accessible* parts (compare this with public, protected and private declarations in object-oriented programming). In general they respectively denote the following: "can be directly used in any level"; "has some predefined restrictions for using in various levels"; "can be used only at the same level".

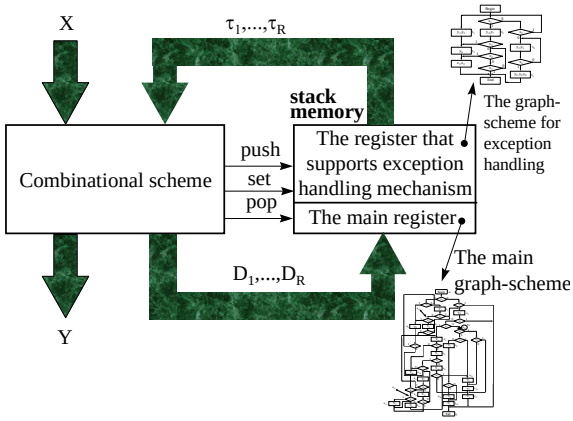


Figure 22. Providing exception handling mechanism

When we are building a hierarchy we are dealing with **inheritance** which can be considered as a basic way to represent multilevel abstractions. The purpose of inheritance is to provide a commonality of representation and a calling interface.

The approach to be considered can be applied to multilevel hierarchical digital control units. Their behaviour can be described by **hierarchical graph-schemes** [7]. However it is necessary to solve some new problems, which are the following:

- how to provide support for *inheritance* making it possible for upper levels to share the structure and/or behaviour in one or more lower levels (denoting single inheritance and multiple inheritance respectively);
- how to provide *protection* for various operations from unauthorised access;
- how to provide *logical synthesis* of hierarchical systems at a hardware level;
- how to construct *derived instructions* that allow us to add new facilities to existing instructions without redesigning the control unit or with minimal efforts.

This approach directly invokes the basic ideas of object-oriented programming [16,17] and can be based on: using *predefined frames* and *templates*, considered above; *hierarchical descriptions* of control algorithms [4,5,7] and general ideas of papers [18,19]. The paper [18] has attempted to provide possible changes in designed control units based on PLDs. The objective is to supply all changes in existing PLDs that have been already programmed. The paper [19] combines using microprocessors in the upper level of control and PLD based control units in the lower level.

IX. RUN-TIME SUPPORT

Let us return back to Section I where the following problem was presented: for a given set of instructions $\square=\{\square_1,...,\square_k\}$ and constraints $\nearrow=\{\nearrow_1,...,\nearrow_p\}$, design the control unit which will perform $\square$ and satisfy the set of conditions, $\nearrow$. Consider multilevel description of $\square$ which is the following $\square=\square^0\cup\square^1\cup...\cup\square^h$ where the set $\square^0$ includes instructions of the level 0, $\square^1$ includes instructions of the level 1,..., $\square^h$ includes instructions of the level h (see figure 23).

Let us look at instruction $\square_i\in\square^j$ (j>0). The $\square_i$ have been described by a graph-scheme Γ_i of the level j. The graph-scheme incorporates micro operations, logical conditions and macro operations. Each macro operation has been described by a graph-scheme of lower levels. So we can say that Γ_i **encapsulates** input and output variables (see figure 23) and complex operations (macro operations) which can be viewed as control functions (compare it with encapsulation in object-oriented programming). Finally encapsulation allows us to separate the purpose of an instruction from its implementation. In other words we want to focus on what the instructions do instead of on how to implement them.

The macro operations can be either *fixed* or *non fixed* in the control unit (see figure 23). In the first case the control unit has been completely designed. In the second case it incorporates additional components which can be programmed during run-time and can be loaded with converted graph-schemes for implementing new instructions from the set $\square$. Generally speaking the set $\square$ can even be extended after the control unit has been designed and produced. This idea is similar to **run time support** in object-oriented programming (early binding and late binding in particular). It leads us to **virtual instructions** definition which is closely related to **virtual states** of the finite automata. Such states are not fixed and can be changed during execution time (compare this with virtual functions in object-oriented programming). It should be mentioned that the basic schemes considered above are mainly based on such PLDs as **ROM** that can be directly replaced with **RAM** which can be loaded and reloaded during run time.

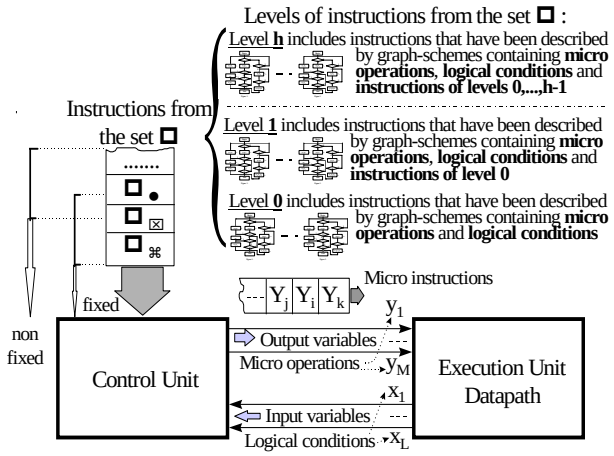


Figure 23. Multilevel description of the set □

Currently PLDs are manufactured by many famous companies, such as Intel, AMD, Monolithic Memories, etc. The comparison of various PLDs was given in [20], where you can also find different details. Programming of PLDs is achieved using various memory technologies such as fuses, EPROM cells, EEPROM cells or Static RAM cells [20]. There are many development systems running on a personal computer that enable you to obtain a customised silicon chip in a short period of time. Many benefits give us erasable PLDs that use EPROM cells as logic control elements which can be erased with ultraviolet light and reprogrammed. In addition, they offer several very significant benefits [20]. The basic architecture of PLDs (see, for example, [20, p. 1-6]) is based on PLAs and has been developed in many different chips (Intel 85C060, AMD 22V10-15, Lattice GAL 22V10-15, etc.). Their specifications and comparisons are given in a variety of catalogues (see, for instance, [20]). All these chips can be used as static components of the schemes considered above. In order to provide run-time support we must replace some of static components with dynamic components such as RAM. It is also worth-while to develop customised silicon chips which contain static and dynamic reprogrammable components and incorporate a control-oriented architecture. The chip delivers the necessary speed and can be used for embedded digital control systems in various areas such as industrial automation, robotics, etc.

X. CONCLUSION

The approach involves finite state machine theory and some ideas from object-oriented programming, as follows:

using graph-schemes to provide better separation of the control unit **interface** from its **implementation**. In particular they provide support for *hierarchical ordering*. Different macro operations represent various levels of abstraction. This approach forces us to search for commonality among branches of the graph-scheme and positively influences future design steps. On the other hand the graph-scheme can also be viewed as the *encapsulation* of *data* (input and output variables) and *functions* (macro operations);

using predefined frames to design **reusable parts**. Generally speaking, reuse denotes the ability of a device to be used again. Sometimes we want to add functionality or to change behaviour. In the approach we are considering, we don't need to start the design process again from the beginning. The new scheme inherits the invariable part of the previous interface and just adds (or replaces) the existing part that is different in the new context [18]. This is an analogous to the *inheritance relationship* between classes in object-oriented programming. Another concept is related to *polymorphism* (to virtual states in particular). Consideration of **virtual states** (programming the output codes of an internal register) simplifies many different problems of logic synthesis (states encoding in particular), and enables you to create *universal predefined structures* for a variety of applications;

using **predefined schemes** (or templates). This idea was initially considered in [13] and can be formulated as follows. For a given (generally speaking infinite) set of graph-schemes, it is necessary to build a scheme that is based on programmable (or reprogrammable) components, and that can be used to implement a given behaviour just by programming its components (you cannot change either the structure or the connections in the scheme) The set of graph schemes can be introduced via various constraints (input constraints, output constraints and functional complexity). In order to find a good solution you can delete some components from the final scheme without changing its structure (and connections). This enables us to deal with superfluous components. The approach is based on the methods of finite state machine theory [3,4] and the results of the two points discussed previously. The use of predefined templates makes it possible to simplify many different problems of logic synthesis and related applications;

run-time support based on reprogramming of the component matrix in a scheme after the control unit has been designed (during execution of control);

exception handling mechanism which makes it possible to explicitly separate an ordinary graph-scheme and an exception handling graph-scheme.

As follows from the previous discussion, we have attempted to combine the results of two well known and closely related areas which are *finite state machine theory* and *object-oriented programming*. As a result, the approach considered can be used to design various control units and provide them with new facilities. Object-oriented technology also has a positive effect on other phases of the hardware life cycle, such as maintenance and improvement. You can make one modification to an ancestor macro instruction and affect all of its descendants. Without inheritance, you would need to make the same change to many relative instructions.

XI ACKNOWLEDGEMENTS

Many thanks to ***Ivor Horton*** for his help with this and previous articles. I wish also to thank many people of ***Electronics and Telecommunications Department*** for their encouragement, especially Professor ***Antonio Ferrari***.

REFERENCES

[1] Randy H. Katz. Contemporary Logic Design. The Benjamin/Cummings Publishing Company, Inc., 1994, 681 p.

[2] Giovanni De Micheli. Synthesis and Optimization of Digital Circuits. McGraw-Hill, Inc., 1994, 579 p.

[3] Samary Baranov, Valery Sklyarov. Digital Devices Based on Programmable Matrix LSI. Moscow, Radio and Communications, 1986, 272 p.

[4] Valery Sklyarov Synthesis of Finite State Machines Based on Matrix LSI. Minsk, Science and Technique, 1984, 287 p.

[5] Valery Sklyarov Parallel Graph-Schemes and Finite State Machines Synthesis. Latvian Academy of Science, Automatics and Computers, Riga, 1987, N 5, p. 68-76.

[6] Victor Kirpichnikov, Valery Sklyarov Description and Synthesis of Control Devices. USSR Academy of Science, Technical Cybernetics, Moscow, 1979, N 1, p. 127-137.

[7] Valery Sklyarov Hierarchical Graph-Schemes. Latvian Academy of Science, Automatics and Computers, Riga, 1984, N 2, p. 82-87.

[8] Valery Sklyarov Synthesis of Control Units Based on PLAs. Latvian Academy of Science, Automatics and Computers, Riga, 1982, N 4, p. 28-35.

[9] Valery Sklyarov State encoding for Finite State Machines Based on PLAs. Latvian Academy of Science, Automatics and Computers, Riga, 1982, N 4.

[10] Valery Sklyarov Synthesis of Control Units Based on Programmable Logic Devices. USSR Academy of Science, Technical Cybernetics, Moscow, 1983, N 5, p 59-69.

[11] Valery Sklyarov Using Decoders in Control Units. University News. Instruments, Leningrad (St.Petersburg), 1982, N 12, p. 27-31.

[12] Valery Sklyarov Synthesis of Control Units Based on Programmable Logic Devices with Memory. Ukraine Academy of Science, Cybernetics, Kiev, 1984, p. 57-64.

[13] .Valery Sklyarov Regularly Structured Finite State Machines. Control Systems and Machines. Kiev, 1984, N 2, p. 23-28.

[14] Valery Sklyarov Control Devices Based on Predefined Frames (basic schemes) with Changeable Structure. Latvian Academy of Science, Automatics and Computers, Riga, 1985, N 2, p. 70-78

[15] Valery Sklyarov Synthesis of Control Units Based on Stack Memory. University News. Instruments, Leningrad (St.Petersburg), 1984, N 4, p 41-45.

[16] Bjarne Stroustrup The C++ Programming Language. Second Edition, Addison Wesley Publishing Company, 1994, 691 p.

[17] Grady Booch Object-Oriented Analysis and Design. Second Edition. The Benjamin/Cummings Publishing Company, Inc., 1994, 589 p.

[18] Valery Sklyarov Providing Modifications of Designed Control Units Based on PLAs. Control Systems and Machines. Kiev, 1983, N 6, p. 8-12.

[19] Valery Sklyarov Control of Technological Equipment Using Microprocessors. Automatics and Telemechanics, Moscow, 1985, N 1, p. 118-121.

[20] Programmable Logic. Intel Corporation, 1991.