AN UNIFIED APPROACH TO DESIGNS IMPLEMENTED WITH MULTIPLEXERS AND TO THE DECOMPOSITION OF BOOLEAN FUNCTIONS

Marek A. Perkowski James E. Brown
Department of Electrical Engineering
Portland State University
P.O. Box 751
Portland, Oregon 97207
tel. (503) 464-3806 x 23.

Abstract

This paper presents a simple, unified, graphical approach to designs implemented with multiplexers and to the decomposition of Boolean functions. In doing so, designs composed of both completely and incompletely specified Boolean functions are presented and illustrated using the numerical method as well as an applicable computer program.

1. INTRODUCTION

One of the most general approaches to solve engineering and science optimization problems is to apply the principle of decomposition. A problem of greater size is decomposed into a number of problems smaller in size that can be solved separately and with less effort.

The principle of decomposition has also been applied in logic design with respect to both Boolean functions and Finite State Machines [1]. There are several types of single-output Boolean function decompositions of which the best known is a simple disjoint decomposition. An example of such a decomposition is shown in the block diagram in Fig. 1.

The function $f1$, with its inputs a, b, c, and d, is decomposed to two subfunctions:

- subfunction 2, denoted by $\varphi(c,\ d)$, and having inputs c, d,
- subfunction 1 with inputs a, b as well as φ.

This approach allows us to minimize one two-input function and one three-input function instead of having to minimize a four variable function.

Each of subfunctions can possibly be further decomposed. Therefore, a multi-level, tree-like implementation structure is created. Since many Boolean minimization computer programs, especially those that attempt to find optimal solutions, are only practical for solving problems of limited size, and which may even restrict the number of inputs, this approach not only yields results lower in cost but can also produce optimal solutions more quickly. This may even be the only way to get any results at all from such programs. Understanding the principles of Boolean decomposition is, therefore, important for everybody who is interested in minimization of Boolean functions. Although minimization with TTL-logic implemented gates like NANDs or NORs is no more of practical importance, the designers who use Programmable Logic Devices (PLD) or the designers of VLSI chips are faced with increasing complexity of problems related to automatic design of logic, especially of logic realizing control units. Many sophisticated programs have been written to perform this tasks for two-level realizations (Programmable Logic Arrays - PLA, Programmable Array Logic - PAL) and are used both at the universities (like Espresso from Berkeley VLSI Design Tools) and in industry. Due to the extensive area and to the slow speed involved when Boolean functions are realized using PLAs, however, there has recently emerged a renewed interest in the multi-level realizations of Boolean functions (Brayton [7], Sangiovanni-Vincentelli - U.C. Berkeley, Hachtel - University of Colorado). The use of Boolean decomposition should also be considered when selecting one of these methods of realization. Designs implemented with multiplexers are closely related and find applications both in VLSI design and in the board-level design of logic functions using MSI chips. Such designs minimize the silicon area or the chip count while simplifying the design process.

Today, an increasing number of graduating engineers are employed at the entry-level as CAD tool developers, or users, which leads us to feel that the undergraduate curricula should include even more CAD material. We have, therefore, enhanced our logic design classes with some elementary topics related to the logical and to the physical CAD, especially to the problems related to designs implemented with PLAs, PALs, PLDs and EPROMs, as well as VLSI design, both custom and with standard cell li-

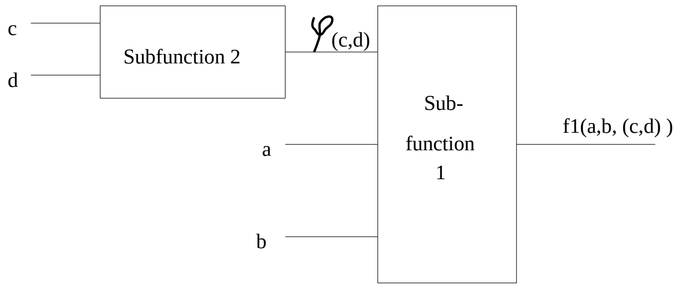

Figure 1:

braries.

Since most undergraduate logic design textbooks regard the decomposition of Boolean functions as an advanced topic, it is seldom mentioned in them. Graduate Switching Circuits textbooks like, for instance, the well known book by Kohavi [1], however, present decomposition in such a manner that is difficult to teach and very time consuming to apply in practical design. On the other hand, designs implemented with multiplexers are taught in undergraduate courses but to our knowledge no general, simple and systematical, graphical or analytical method has yet been presented.

In this paper, we will present a new, unified approach to the decomposition of Boolean functions and to designs implemented with multiplexers. The three following variants will be discussed in more detail:

1. *The graphical variant* using Karnaugh maps is very easy to teach and is successfully used by students for maps of up to six variables. It is analogous to the most familiar concepts of elementary logic design. The authors used this method for practical problems of strongly unspecified functions with up to 10 variables prior to writing the computer program mentioned below.

2. *The analytical variant* using the well-known *calculus of arrays of cubes* (such calculus is presented for instance in [3], [4] and [5]). Teaching of this material can be used to master cube calculus and present the ways to implement logic CAD algorithms. We teach this material in a graduate class.

3. *The computer program* makes decomposition and designs implemented with multiplexers for up to 15 input variables possible. It is programmed in C and implementations for Vax and personal computers are available. Students can copy a PC diskette with the source code and design examples.

2. GRAPHICAL METHODS

A. DESIGNS REALIZED WITH MULTIPLEXERS

Let us first describe how Karnaugh maps (K-maps for short) can be used to implement a single-output function

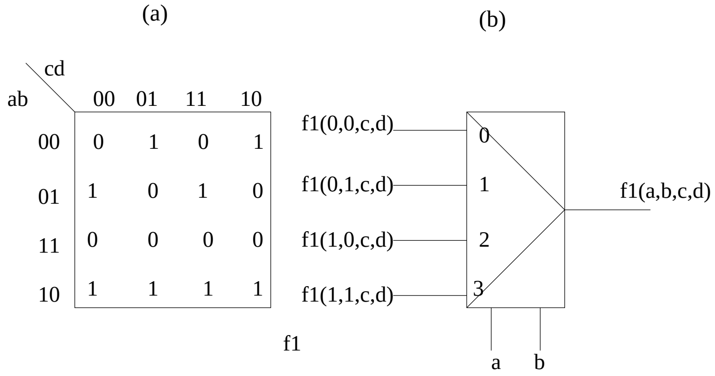

Figure 2:

using multiplexers.

Example 1.

Let us assume that we want to realize the four-input function $f1(a, b, c, d)$, shown in Fig. 2a, using a 4 x 1 multiplexer with two select lines (2 bit address).

Let as further assume that inputs a, b were selected as address inputs of the multiplexer. We will denote the *set of address variables* is {a, b} or say that variables a and b are the *address variables.* Then, for a = 0 and b = 0 a function described in row $\overline{a}\ \overline{b}$ of the K-map should be realized as the function $f1(a, b, c, d)$. We will denote the function from row $\overline{a}\ \overline{b}$ as $f1(0,\ 0,\ c,\ d)$. The other notation used will be $f1_0(c,\ d)$. To realize function $f1(a,\ b,\ c,\ d)$ we then provide the data input number 0 (which denotes [a,b] = [0,0]) in the multiplexer with the function $f1$ $(0,\ 0,\ c,\ d)$. Similarly, the function from row 2, a = 0, b = 1, will be provided at data input 1, the function from row 4, a = 1, b = 0, will be provided at data input 2 of the multiplexer, and the function from row 3 ([a,b] = [1,1]), will be provided at data input 3. This is shown in Figure 2b.

Now our task is to find the realizations of the functions:

$$f1(0,\ 0,\ c,\ d) = f1_0(c, d),$$
$$f1(0,\ 1,\ c,\ d) = f1_1(c, d),$$
$$f1(1,\ 0,\ c,\ d) = f1_2(c, d),$$
$$f1(1,\ 1,\ c,\ d) = f1_3(c, d).$$

These functions are specified by the corresponding loops in the K-map $\overline{a}\ \overline{b}$, $\overline{a}\ b$, $a\ \overline{b}$, and $a\ b$, respectively. We will denote these as *loop functions.* For better manipulation, the inexperienced user can rewrite each loop in the form of a separate K-map. As we see, all of them are functions of the variables c and d only as shown in the K-maps in Fig. 3.

Before gaining experience in the manipulation of such maps, we can transfer these functions to K-maps of a more familiar form, as shown in Fig. 4. After solving a few examples, however, the student gets enough proficiency so to omit this step and operate directly on the initial submaps of the K-map.

The data input functions $f1_0$, $f1_1$, $f1_2$, $f1_3$ will now be realized using any of the well-known Boolean minimiza-

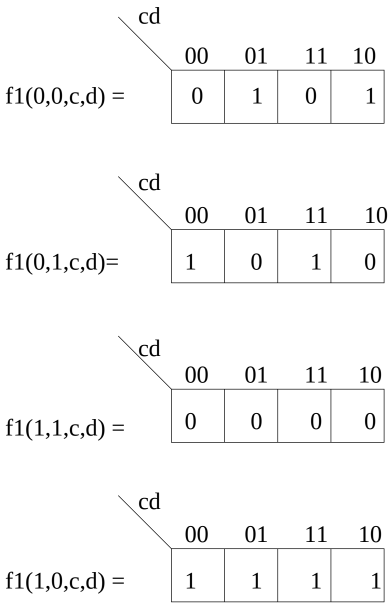

Figure 3:

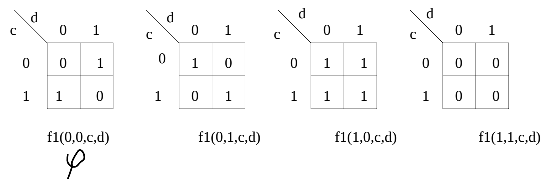

Figure 4:

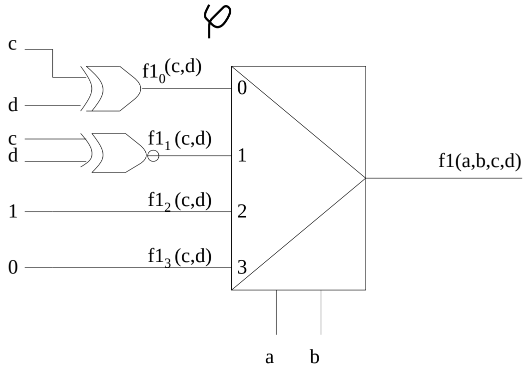

Figure 5:

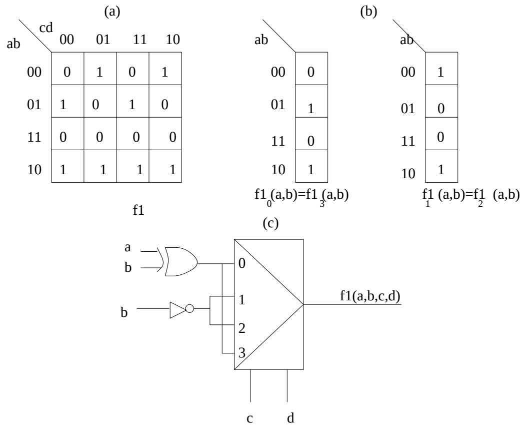

Figure 6:

tion methods. The corresponding circuit realization is shown in Fig. 5.

Note that contrary to well-known approaches we do not need to use Marquand or decomposition charts to find multiplexer realizations for other sets of address variables. The same K-map will suffice. The process of finding the circuit when variables c and d are assumed as the address variables for example is shown in Fig. 6.
This realization costs less than the previous one since an inverter is used instead of an equivalence gate (exnor). Similarly, assuming address variables a and c (Fig. 7a), the functions $f1_0(b,\ d)$, $f1_1(b,\ d)$, $f1_2(b,\ d)$, $f1_3(b,\ d)$ will be as shown in Fig. 7b and the realization as shown in Fig. 7c. Note the inverse order of variables in the headings of some of maps from Fig. 7b. If the designer has difficulties in using the "inverse" K-maps, he can rewrite some of them as shown in Fig. 7d. After solving few examples, however, the student becomes proficient enough to omit this step as well and operate directly from the "inverse submaps" of the K-map. The realization shown in Fig. 7c has the same cost as one from Fig. 5.

We are now able to formulate the design method for designs implemented with multiplexers. Corresponding loops in the K-map must be drawn for each possible subset of variables that are used as the address variables.

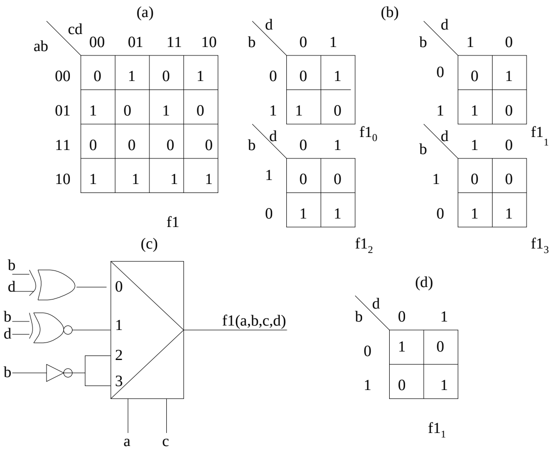

Figure 7:

Fig. 8 shows loops for all of the possible sets of address variables for 4-variable functions. It is easy to start with the loop corresponding to the product made up of noninverted address variables (such products are shown in bold - for instance group 3 for ab in Fig. 8a). The remaining loops for these variables can quickly be drawn by taking the mirror image of the initial loop with respect to all the possible combinations of values of the corresponding address variables. Group 1 in Fig. 8a is, for example, a mirror image with respect to variable a, the group 2 with respect to variable b, and group 0 with respect to both variable a and variable b.

We denote the set of address variables by B. Later this set will also be referred to as a set of *free variables* (or *free set*) to be consistent with the notation used for decomposition. The set of all the remaining input variables will be denoted by A. This is a set of *bounded variables* or a *bounded set.* The *set of all input variables* is denoted by I. Therefore, I = A ∪ B, and , of course, A ∩ B = ∅.

The design method for designs implemented with multiplexers is then as follows:

1. Draw the K-map for the function f(I).
2. Find all the possible subsets B of input variables to be used as the address variables.
3. Draw the initial loop and then all the remaining loops for each set B of address variables, as shown in Fig. 8.
4. For each subset B of address variables:
   - find loop functions $f_i(A)$ for all the loops of this free set.
   - realize loop functions $f_i(A)$ using any Boolan minimization method but use the same function for multiple multiplexer inputs as often as possible.

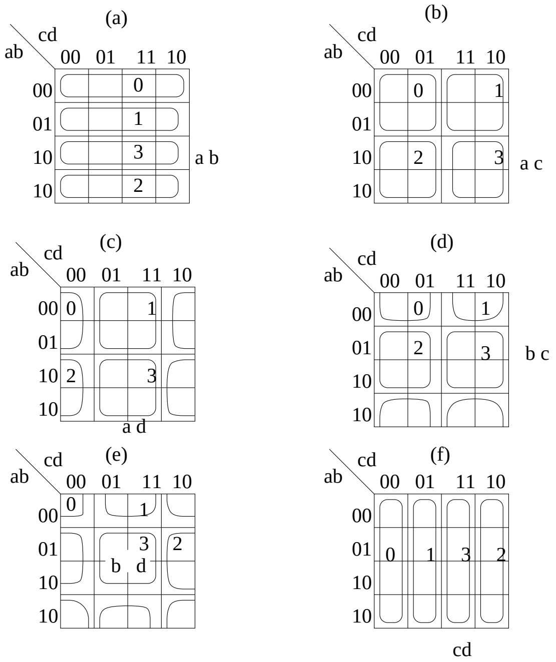

Figure 8:

   - evaluate the cost of the solution by adding the realization costs of various loop functions.
5. Select the best solution for all the free sets out of those found in step 4.

Experienced designers can practically apply this method in K-maps using not more than ten variables. Many design short-cuts are possible to find good solutions without investigating all free sets. Its application is now mainly didactical, since students can easily solve functions of this size on a program written for the personal computer.

B. DECOMPOSITION

Let us first observe that function $f1_0(c,\ d)$ in Fig. 5 is the negation of function $f1_1(c,\ d)$. Therefore, the realization from Fig. 5 can be re-drawn as shown in Fig. 9.

Next, the function from Fig. 9 can be represented in block diagram form as in Fig. 10 (and Fig. 1). This figure represents a general scheme of the classical *(simple) disjunctive decomposition* of a single output Boolean function with the *set of bound variables* {c, d} and *the set of free variables* {a, b}. Such decompositions are discussed in [1] and [2].

At this point the question arises as to how to find whether this function is decomposable for the set of address variables selected by us from the function's K-map. It is obvious from Fig. 9 and Fig. 10 that if we want to have a single connection from subfunction 2 to subfunction 1 then all of the data inputs to the multiplexer have

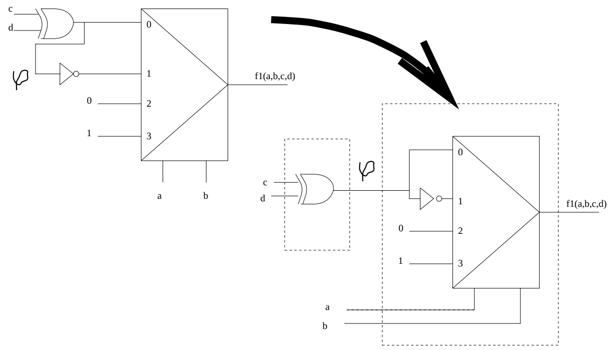

Figure 9:

Figure 10:

to be a constant 0, a constant 1, an output of subfunction 2 or a negation of subfunction 2 (the negation is created inside of subfunction 1). Patterns 0,1,φ, and $\overline{\varphi}$ in A are equivalent to two patterns in B. This is basically a proof of the Ashenhurst theorem.

The well-known Ashenhurst theorem can now be expressed as follows.

The function f(I) is decomposable with free set B and bounded set A if each of the subfunctions $f_i(A)$ is at most one of the following types:

- constant 0,
- constant 1,
- some function, call it $\varphi(B)$,
- negation $\overline{\varphi(B)}$ of this said function $\varphi(B)$.

Let us now observe that the decomposition process can be performed graphically even faster than the process of design with multiplexers using K-maps. First, all the loops are drawn for the selected free variables as previously. Then, we discard all the loops consisting of only zeros or only ones. Finally, we assume that one of the remaining (if any) loop functions is φ. When all of the remaining loops describe functions that are either φ or $\overline{\varphi}$ we have found a decomposition. If we find at least one function that is neither a φ and nor a $\overline{\varphi}$, the decomposition **does not** exist for the selected free variables. We can often quickly establish that the decomposition for a given set of free variables does not exist without checking too many loops for this set. Therefore with some experience the designer can quickly scan the loops to find nonexistence of a decomposition. He does this subsequently for all other sets of free variables. There is no need to redraw the sub-K-maps. Finding the existence of the decomposition can be more time consuming and it may require the re-drawing of some submaps for K-maps with many variables.

Example 2.

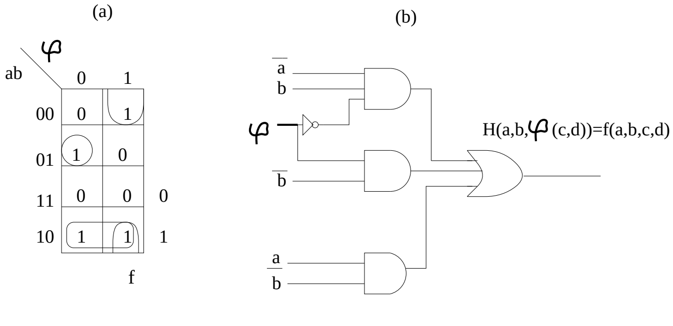

Figure 11:

Let us go back to function $f1(a,b,c,d)$ from Example 1. We assume address variables a, b. Loop $a\overline{b}$ of ones and loop $a\ b$ of zeros are discarded so only loops $\overline{a}\ \overline{b}$ and $\overline{a}b$ remain. Let us assume that $\varphi = f1_0$. Now it is easy to check that $f1_1 = \bar{\varphi}$ and the theorem is, therefore, satisfied. The decomposed realization of this function is shown in Fig. 11. The subfunction 1 K-map is created from the K-map shown in Fig. 2, φ, and $\bar{\varphi}$, as in Fig. 11a. This function is minimized to the circuit shown in Fig. 11b using two-level Boolean minimization

Example 3.

For the same function let us now select variables c and d as address variables (Fig. 6a). No loops can be discarded. Assuming $\varphi = f_0(a,\ b)$ in column 0 (loop $\overline{c}\ \overline{d}$) we immediately see that function from loop $\overline{c}d$ is neither φ nor $\bar{\varphi}$, so no decomposition exists with c and d as free variables.

Let us observe that if the decomposition exists the designer has a choice to implement each function either with a multiplexer for subfunction 1, efficiently realizing the data input functions, or subfunction 1 could also be realized using any other Boolean minimization method, for instance as a PLA, minimized using a two-level Boolean minimization program (as in Example 2). If the decomposition does not exist, the results of the above loop functions comparison are still useful for the efficient design for the implementation using the multiplexer in which simple loop functions are realized and the smallest possible number of them exist.

C. DECOMPOSITION OF INCOMPLETELY SPECIFIED FUNCTIONS

The graphical method can also be applied to incompletely specified functions with very little modification. First, we remove loops that can be completed to all zeros or that can be completed to all ones. Next we check whether the remaining loops can be *completed* to functions φ and $\bar{\varphi}$.

Example 4.

Given is function $f2(a,b,c,d,e,f)$ from Fig. 12.

Assuming address variables a, b the loops are as shown in Fig. 12. We can immediately discard loop $\overline{a}b$ that can be completed to all zeros, as well as the loop $a\ b$ that can

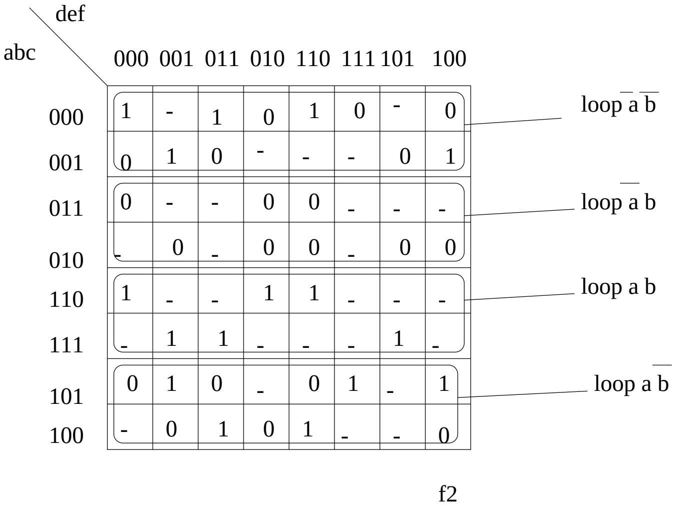

Figure 12:

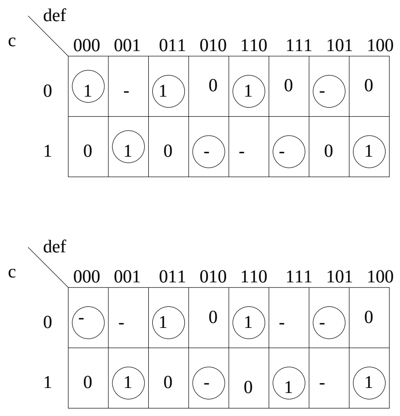

Figure 13:

be completed to all ones. To help the reader the loops $\bar a\ \bar b$ and $a\ \bar b$ are rewritten to those shown in Fig. 13. It is clear that both of them can be completed to the same function

$\phi = \bar c\ \bar d\ \bar e\ \bar f + c\ \bar d\ \bar e f + \bar c\ \bar d e f + c\ \bar d e\ \bar f + \bar c d e\ \bar f + cdef$ $+\ \bar c d\ \bar e f + c d \bar e\ \bar f = c \oplus d \oplus e \oplus \bar f$

It is important to note that with some experience one does not have to draw all the loops in the K-map, but, remembering their shapes, the existence of functions φ and $\bar\varphi$ can be verified directly.

When checking the decomposition possibility for free sets, it is also often not neccessary to generate loops systematically for all possible free sets. Since we know that the existance of loops of zeros and loops of ones for some free set increases the possibility of a decomposition, we will first find a large loop of zeros or a large loop of ones, or a loop that can be completed to a loop of such a type. Next, we will test the decomposition possibility for other loops created for the free set of the variables from this loop. If no decomposition exists for this free set, the decomposition for the next large loop of the above types is

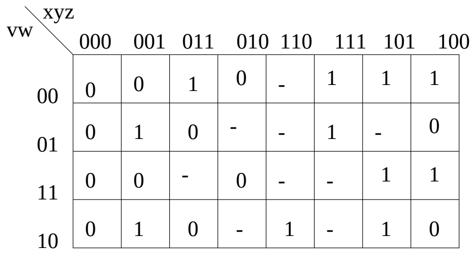

Figure 14:

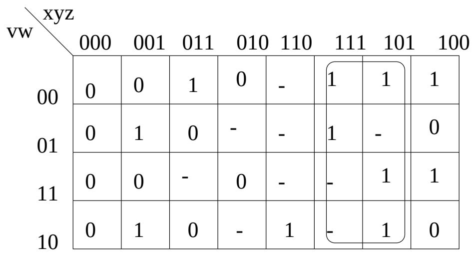

Figure 15:

investigated.

Example 5.

We will solve the example from p. 171 in [1].

$f(v, w, x, y, z) = \sum$ (3,4,5,7,9,15,17,21,22,28,29) + $\sum_{\emptyset}$ (10,13,14,18,23,27,30,31).

The K-map for this function is shown in Fig. 14.

First, we try to find large loops that can be completed to loops of all zeros or to loops of all ones. A loop xz is found as a loop that can be completed to ones - see Fig. 15. This suggests selecting the set {x,z} as a free set. By examining the other loops for this set we find that loop $\bar x\ \bar z$ can be completed to all zeros. Now the remaining loops are found and compared (Fig. 16). As we see, assuming that loop $x\ \bar z$ corresponds to φ, we notice that the loop $\bar x z$ will correspond to $\bar\varphi$. The completion is shown in Fig. 17a.

Therefore the decomposition for free set {x, z} is found, as shown in Fig. 17. Fig. 17b, c presents two methods for realization of function $f(x, z, \varphi)$. The final circuit is presented in Fig. 17e.

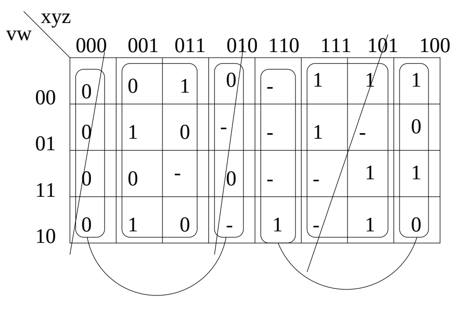

Figure 16:

3. ANALYTICAL DECOMPOSITION OF INCOMPLETELY SPECIFIED SINGLE-OUTPUT BOOLEAN FUNCTIONS

The analytical method presented below serves to explain the computer algorithm that follows in section 4, but it is not necessary for the teacher to explain it in the class.

Our goal is to find an analysis method to check if the completion to φ and $\bar{\varphi}$ exists for a given free set.

Definition 1.
By a *loop pair* we will understand a pair:
[loop, respective loop function] = [loop, f(loop)].

Definition 2.
Two functions are called the *compatible functions* when they can be completed to the same function.

Definition 3.
Two loop pairs $p_1 = [l_1, \mathrm{f}(l_1)]$ and $p_2 = [l_2, \mathrm{f}(l_2)]$ are called *compatible loop pairs* when f(l_1) and f(l_2) are compatible functions.

Definition 4.
The operation of merging, with two functions f_1 and f_2 as arguments, produces the least defined function f = $f_1 \odot f_2$ that completes both of them by replacing dont-cares with zeros or ones. This is done on a cell-by-cell basis. If a cell has the same value in both functions, this value is retained. Don't care and cell will yield the value of this cell:
$1 \odot 1 = 1$, $0 \odot 0 = 0$, $1 \odot - = 1$, $0 \odot - = 0$. If the respective cells have values 1 and 0, the merging cannot be done (we denote this by $1 \odot 0 = \emptyset$) and we say that functions f1 and f2 are *not compatible.*

Definition 5.
Two loop pairs $p_1 = [l_1,\ \mathrm{f}(l_1)]$ and $p_2 = [l_2,\ \mathrm{f}(l_2)]$ are *merged* to a loop pair
$[\{l_1,\ l_2\},\ \mathrm{f}(l_1) \odot \mathrm{f}(l_2)]$ if f(l_1) and f(l_2) are compatible functions.

Loop pairs are introduced to make it possible to perform operations on the corresponding loop functions while keeping track of the loops from which the merged functions are derived.

First note that the *loop pair compatibility* relation is not an equivalence relation.

Example 6.
Let us consider a function given in a Karnaugh map of Fig. 18, assuming B = {c, d, e}.
The free set B = {c,d,e}, as can be seen in the loop functions that correspond to the columns of the map. After removing the loop $\overline{c}\ \overline{d}\ \overline{e}$ of 1's and the loop $cd\overline{e}$ of 0's, the *loop pair compatibility graph* as shown in Fig. 19 is created with the remaining loop pairs as nodes.

The compatibility of two loop pairs in the graph is denoted by an edge between the corresponding nodes. Let us, for instance, compare loop functions for columns 011 and 010 (loops $\overline{c}de$ and $\overline{c}\ d\ \overline{e}$). The loop function for a column will be denoted in the current section, for the sake of explanation, as the string of symbols "0", "1", and "-"

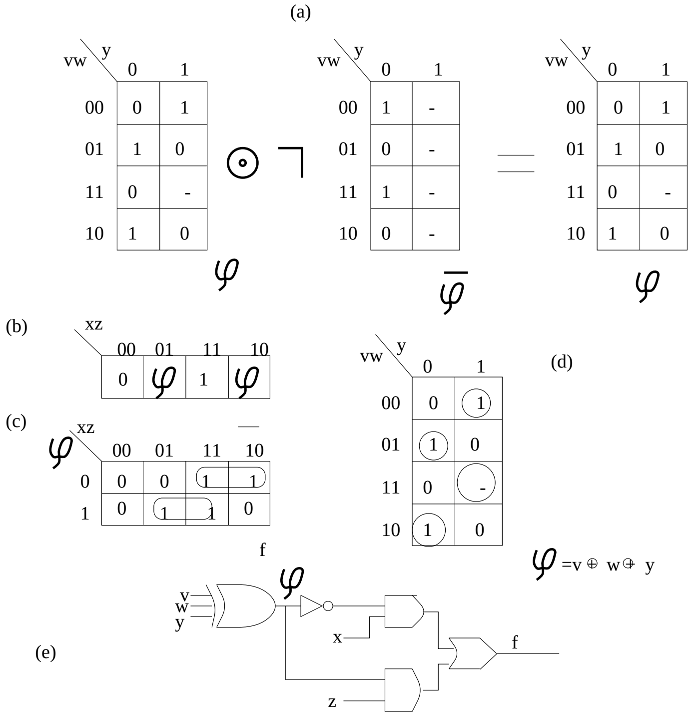

Figure 17:

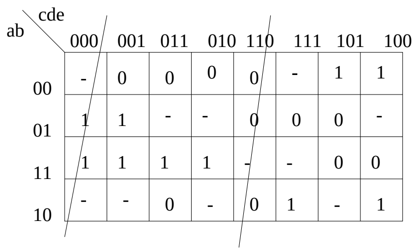

Figure 18:

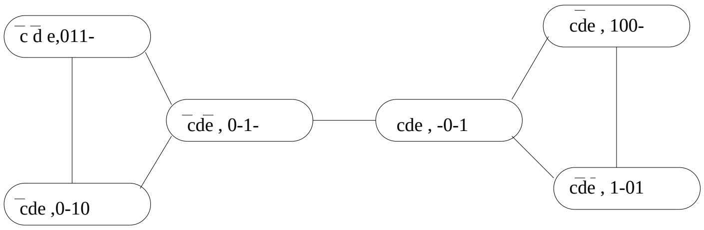

Figure 19:

, and in the same order as they are in the column from the top to the bottom. The computer representation of loop functions will be discussed in the next section when the computer program will be presented. The loop function for column 011 (loop $\bar{c}de$) is 0-10, and the loop function for column 010 is 0-1-. These loop functions are compatible. This is denoted in cube calculus as:
0-10 $\odot$ 0-1- = 0-10. We denote the merging (common minimal completion) operation of functions represented by $\odot$. The corresponding loop pairs are then also compatible:
[$\bar{c}de$, 0-10] $\odot$ [$\bar{c}\ d\bar{e}$, 0-1-] = [{$\bar{c}de$, $\bar{c}d\bar{e}$}, 0-10]. Compatibility of loop pairs is denoted as an edge between nodes [$\bar{c}de$, $f(\bar{c}de)$] and [$\bar{c}d\bar{e}$, $f(\bar{c}d\bar{e})$]. In our notation the nodes: [$\bar{c}de$, 0-10] and [$\bar{c}d\bar{e}$, 0-1-] (see Fig. 19). Similarly, loop pairs [$\bar{c}d\bar{e}$, 0-1-] and [cde, -0-1] are compatible, since 0-1- $\odot$ -0-1 = 0011. The loop pairs [$\bar{c}de$, 0-10] and [cde, -0-1], however, are not compatible since 0-10 $\odot$ -0-1 = 001$\emptyset$ (we denote the incompatible 0 and 1 in the lowest row of the K-map by $\emptyset$). **The relation of compatibility is, therefore, not transitive and hence it is not an equivalence relation**.

We create the entire loop pair compatibility graph from Fig. 19 using the above method. The next stage is to find **a partition of this graph to two maximum cliques.** A *clique* is a subgraph with any two nodes linked by an edge. *The maximum clique* is a clique with maximum number of nodes. If we created function φ after merging the loop functions from the first clique, the merging from the other clique would be $\bar{\varphi}$, otherwise there is no decomposition. The classical decomposition of the function does not exist in one of two cases:

1) there is more than two cliques in the minimal partition.

2) $= \bigodot_{c_i \in C_1} c_i \neq \overline{\odot_{c_j \in C_2} c_j}$
i.e. the two mergings from the cliques C_1 and C_2 can not be completed to functions φ and $\bar{\varphi}$ of each other.

The fact that the relation of compatibility is not an equivalence relation makes the search for maximum cliques more difficult.

We can create a simple sequential clique-growing algorithm in which the two cliques are extended step-by-step, starting from any loop pair. By first selecting the loop pair [$\bar{c}\ \bar{d}e$, 011-] and then the pair [$\bar{c}de$, 0-10], for example, we create a merged loop pair [{$\bar{c}\ \bar{d}e$, $\bar{c}de$}, 0110]. Now [$\bar{c}d\bar{e}$, 0-1-] is selected and we join it to this loop pair: [{$\bar{c}\ \bar{d}e$, $\bar{c}de$, $\bar{c}\ \bar{d}e$}, 0110]. Then we select [cde, -0-1]. This cannot be merged with the previous merged loop pair, so is becomes a beginning of the new merged loop pair. Next we select [$c\bar{d}\ \bar{e}$, 1-01], creating the merged loop pair [{cde, $c\ \bar{d}\ \bar{e}$}, 1001]. When next the pair [$c\bar{d}e$, 100-] is selected it is joined to the last merged loop pair, creating a pair: [{cde, $c\bar{d}\ \bar{e}$, $c\bar{d}e$}, 1001]. After scanning all loop pairs two merged loop pairs: [{$\bar{c}\ \bar{d}e$, $\bar{c}de$, $\bar{c}\ \bar{d}e$}, 0110] and [{cde, $c\bar{d}\ \bar{e}$, $c\ \bar{d}e$}, 1001], have then been created. They are mutual negations, so the respective decomposition exists. Let us however observe that this method would not work for the other order of selection. If we first selected the loop pair [$\bar{c}d\bar{e}$, 0-1-] and then the pair [cde, -0-1], we would create the pair [{$\bar{c}d\bar{e}$, cde}, 0011]. Then, we select loop pair [$\bar{c}\ \bar{d}e$, 011-] and we see that the loop functions, 011-, from this pair, and 0011 from the pair [{$\bar{c}d\bar{e}$, cde}, 0011] cannot be negations of each other and also cannot be merged. The decomposition was, therefore, not found for this order of loop pair selection even if it existed, as was the case for another order.

The problem of finding maximum clique partitioning is in general an NP-hard one. We will, therefore, propose a simple and fast algorithm below that yields good results for most practical problems. It refers to an optimal NP-hard clique partitioning algorithm, only when necessary.

Algorithm 1.

1. UNSPECIFIED := $\emptyset$.

   Select any loop function. Call it φ.

   Make a pair $p_i = [\emptyset, \varphi]$.

2. **For each** loop pair $p_i = [l_i, c_i] \in$ LOOP_PAIRS **do**

   if loop c_i has no bits in the same positions as φ or $\bar{\varphi}$

   then UNSPECIFIED := UNSPECIFIED $\cup$ $\{p_i\}$

   else

   if $c_i \odot \varphi$ and $\bar{c_i} \odot \bar{\varphi}$

   then

   $[S', \varphi] := [l_i, c_i] \odot [S, \varphi]$; $[S', \bar{\varphi}] := [l_i, \bar{c_i}] \odot [S, \bar{\varphi}]$

   if

   $c_i \odot \bar{\varphi}$ and $\bar{c_i} \odot \varphi$

   then

   $[S', \varphi] := [\bar{l_i}, \bar{c_i}] \odot [S, \varphi]$; $[S', \bar{\varphi}] := [l_i, c_i] \odot [S, \bar{\varphi}]$

   else

   return "no decomposition"

3. **If** UNSPECIFIED = φ

   then

   return

4. **For each** loop pair $p_i \in$ UNSPECIFIED **do if** column c_i has no bits in the same positions as φ or $\bar{\varphi}$

   then

   UNSPECIFIED1 := UNSPECIFIED1 $\cup$ p_i

   else

   if $c_i \odot \varphi$ and $\bar{c_i} \odot \bar{\varphi}$

   then

   $[S', \varphi] := [l_i, c_i] \odot [S, \varphi]$; $[S', \bar{\varphi}] := [l_i, \bar{c_i}] \odot [S, \bar{\varphi}]$

   if $c_i \odot \bar{\varphi}$ and $\bar{c_i} \odot \varphi$

   then

   $[S', \varphi] := [l_i, \overline{c_i}] \odot [S, \varphi]$; $[S', \bar{\varphi}] := [l_i, c_i] \odot [S, \bar{\varphi}]$

   else

   return "no decomposition"

5. **If**

   CARD(UNSPECIFIED) = CARD(UNSPECIFIED1)

   then apply optimal clique partitioning algorithm of loop pair compatibility graph

   else

   UNSPECIFIED := UNSPECIFIED1;

   UNSPECIFIED1 := $\emptyset$

   Go to 3.

   End

Example 7.

We continue the previous example and solve it using Algorithm 1. Let us assume, that first loop pair $[\bar{c}d\bar{e}$, 0-1-] is selected. It is marked as φ. Let us now assume that as a second loop pair, we select the pair $[cde$, -0-1]. The function from this pair has no common specified bits common to function φ. The pair is, therefore, added to the set UNSPECIFIED. The pair selected next is $[c\overline{d}\,\overline{e}$, 1-01]. The negation of this pair's function has a common complement to the function φ. Therefore, $\bar{\varphi}$:= 1-01. The pair $[c\overline{d}e$, 100-] selected as the next one is compatible with $\bar{\varphi}$. Therefore, $\bar{\varphi}$ = 1-01 $\odot$ 100- = 1001, so φ = 0110. Then, pair $[\overline{c}\,\overline{d}e$, 011-] is selected and merged with φ. Next $[\overline{c}de$, 0-10] is merged with ϕ. Now, the UNSPECIFIED set's functions are taken into consideration again. The pair $[cde$, -0-1] can only be joined with $\bar{\varphi}$. Decomposition exists with φ = 0110 and $\bar{\varphi}$ = 1001.

4. COMPUTER IMPLEMENTATION OF ANALYTICAL DECOMPOSITION METHOD

A Boolean function of n *input variables,* $X_1, X_2, ..., X_n$ will be given in a form of *arrays of cubes.* Each *cube* will have n positions. Each position of a cube can be a value of the respective variable: 0 - in negative form, 1 - in positive form, X - don't care form. The function will be specified as *ON-array* and *OFF-array.* ON-array is an array of *ON-cubes.* OFF-array is an array of *OFF-cubes.* ON-cube is a cube with value 1. OFF-cube is a cube with value 0. By ON(00XXX) we will denote the set of ON-cubes of $f(0, 0, c, d, e)$. Similarly by OFF(XX010) we will denote the set of OFF-cubes of $f(a, b, 0, 1, 0)$.

Theorem 1.

Two functions f_1 and f_2 are *compatible* if and only if

ON(f_1) $\sqcap$ OFF(f_2) = $\emptyset$

and

ON(f_2) $\sqcap$ OFF(f_1) = $\emptyset$

Proof is obvious from Definition 2.

Definition 6.

The *loop pair incompatibility graph* is an undirected graph

G = (N, RS) defined as follows:

N - is the set of loops,

a = (c_1, c_2) $\in$ RS $\Leftrightarrow$ loop functions f(c_1) and f(c_2) are **not compatible** (such functions are called *incompatible*).

The nodes of the graph are loops. The edge a is created in the incompatibility graph between two nodes when the corresponding loop functions of these loops are not compatible.

Let us observe that the loop pair incompatibility graph is a complement of the loop pair compatibility graph, i.e. has an edge betwen two nodes if there is no edge in the other graph, and vice versa.

The optimal algorithm for loop pair compatibility graph clique partitioning is based on minimal proper coloring of the loop pair incompatibility graph. The proper coloring is an assignment of colors to graph nodes in which any nodes linked by an edge receive different colors. Minimal coloring is a proper coloring with the minimum number of colors. If the number of colors is two then the classical decomposition is possible: Function $F1$ being the merging of loop functions from all nodes of color 1 is found. Next function $F2$ being the merging of loop functions from nodes of color 2. If $F1$ and $\overline{F2}$ are compatible then decomposition exists. Otherwise, even when the number of colors exceeds two - the minimal proper coloring result can still be used for the efficient realization with multiplexers.

Example 8.

Let us assume a function $f(a, b, c, d, e)$ specified as follows: ON(f) = {10000, 110X1, 00X11, 01101, 10110, 11100}, OFF(f) = {110X0, X1010, 0XX10, 101X1, 1010X, X0101, 00000, 01001}. The function is presented in a Karnaugh map from Fig. 20.

We illustrate application of the optimal graph-coloring algorithm for B = {c, d, e}.

1. Find loop functions in the form of arrays of cubes.

ON(XX000) = {10XXX},

OFF(XX000) = {00XXX, 11XXX},

ON(XX001) = {11XXX},

OFF(XX000) = {01XXX},

ON(XX010) = {},

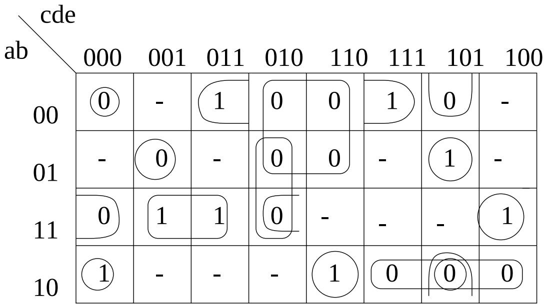

Figure 20:

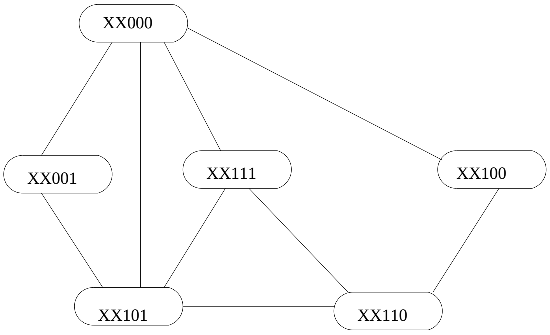

Figure 21:

OFF(XX010) = {0XXXX, X1XXX, 11XXX},
ON(XX011) = {00XXX, 11XXX},
OFF(XX010) = {},
ON(XX100) = {11XXX},
OFF(XX100) = {10XXX}
ON(XX101) = {01XXX},
OFF(XX101) = {X0XXX},
ON(XX110) = {10XXX},
OFF(XX110) = {0XXXX},
ON(XX111) = {00XXX},
OFF(XX111) = {10XXX}.

2. Discard loops of zeros and loops of ones.
After removing the loop XX011 of ones and the loop XX010 of zeros (such loops can be easily found, since they have empty sets OFF(XX011) and ON(XX010), respectively) we obtain the following set of loops

{XX000, XX001, XX100, XX101, XX110, XX111}.

These loops become the incompatibility graph nodes.

3. Create the loop pair incompatibility graph.
For the pair of loops XX000, XX001 we have
ON(XX000) $\sqcap$ OFF(XX001) = {10XXX} $\sqcap$ {01XXX} = $\emptyset$,
OFF(XX000) $\sqcap$ ON(XX001) = {00XXX, 11XXX} $\sqcap$ {11XXX} = {11XXX} $\neq \emptyset$
then these loops are incompatible and an edge is added to the graph between nodes XX000 and XX001.

Similarly, using this method the entire loop pair incompatibility graph from Fig. 21 is created.

4. Find minimal proper coloring of the loop pair incompatibility graph.

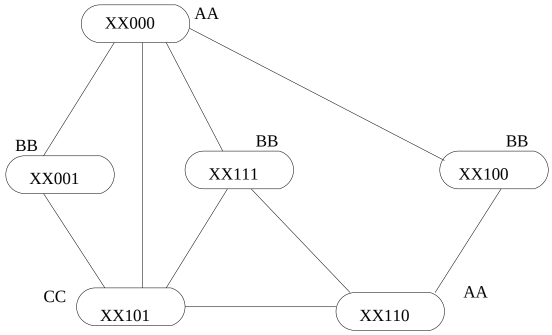

Figure 22:

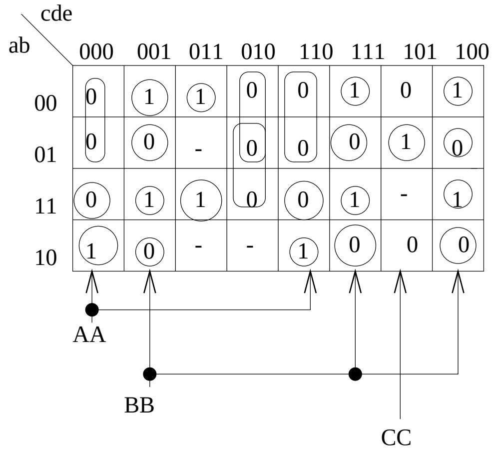

Figure 23:

After coloring with a minimum number of colors the graph is as in Fig. 22. The minimal coloring has then 3 colors: AA, BB, and CC. The decomposition thus does not exist but this result is useful to minimize the design using the multiplexer. We have then to realize only three different functions: AA, BB, and CC to feed the output subfunction 1 (multiplexer) for B = {c, d, e}. The set of columns colored with color AA is = {XX000, XX110}. The set of columns with color BB is {XX001, XX111, XX100} and the set for color CC is {XX101}.

The next stage is to merge the compatible loop pairs.

For color AA:
ON(XX000) $\sqcup$ ON(XX110) =
{10XXX} $\sqcup$ {10XXX} = {10XXX}
OFF(XX000) $\sqcup$ OFF(XX110) =
{00XXX, 11XXX} $\sqcup$ {0XXXX} = {0XXXX, 11XXX}
(since 00XXX $\sqsubseteq$ 0XXXX)
Let us observe that this is not a set of prime implicants for this column.

For color BB:
ON(XX001) $\sqcup$ ON(XX111) $\sqcup$ ON(XX100) =
{11XXX} $\sqcup$ {00XXX} $\sqcup$ {11XXX} = {11XXX, 00XXX};
OFF(XX001) $\sqcup$ OFF(XX111) $\sqcup$ OFF(XX100) =
{01XXX} $\sqcup$ {10XXX} $\sqcup$ {10XXX} = {01XXX, 10XXX}.

For color CC:

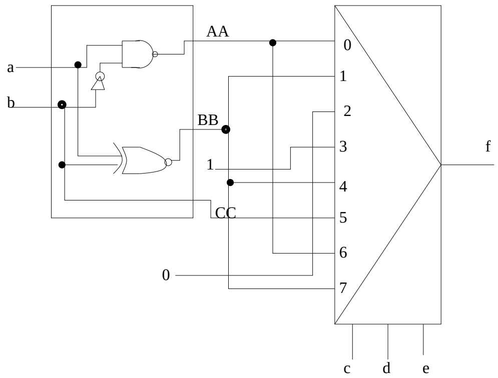

Figure 24:

ON(XX101) = {01XXX}.
OFF(XX101) = {X0XXX}.
5. Complete the function f according to the optimal coloring found in the previous step: functions being mergings of all loop functions of the same color replace the original loop functions.
Now our K-map looks as in Fig. 23.
6. Depending on the number of colors and compatibility of $F1$ and $\overline{F2}$: either find classical decomposition or find a realization using a multiplexer that minimizes the number of various functions given on this multiplexer's data inputs. The respective realization with multiplexers is shown in Fig. 24. The names of the wires: AA, BB, and CC correspond to the colors of the combined functions. Let us observe that the don't cares have been utilized to simplify function CC.

The above method based on minimal proper graph coloring of the loop pair incompatibility graph is a starting point to generalized decompositions introduced in [6].

5. CONCLUSION

We have presented a new approach to design with multiplexers and decomposition of Boolean functions. Both a quasi-optimal graphical method and an optimal analytical method as well as a respective computer program have been presented.

Before we began to teach the material as in this paper, in many undergraduate classes, the students complained that decomposition and design with multiplexers, especially classical approach to decomposition (based on decomposition charts [1]) is too difficult, error-prone, and time consuming.

The graphical method was class-tested in four universities and two countries in classes ranging from sophomores to Ph.D. graduates and from short review courses to advanced logic design courses for practicing VLSI digital design engineers and professional CAD tools developers. The student responses on questionairies indicated that they felt that our new methods were not any more difficult than the rest of the material presented in these classes. Exam results were quite satisfactory and showed that this material was well retained. The students also liked the methods used very much.

Several expansions of the analytical method has been designed in [6] and are now being programmed.

- For multioutput functions.
- For multioutput functions with multiple-valued inputs (application in design of PLA with pair decoders [4]).
- For generalized decomposition, as discussed in [3], but for the incompletely specified functions.

6. LITERATURE.

[1] Kohavi, Z.: Switching and Finite Automata Theory, McGraw-Hill Book Company, 1978.

[2] Ashenhurst, R.L.: The Decomposition of Switching Functions, Proceedings of an International Symposium on the Theory of Switching, April 2-5, 1957, Ann. Computation Lab., Harvard Univ., Vol. 29, pp. 74-116, 1959.

[3] Sasao, T.: Functional Decomposition of PLA's, Proceeedings of the International Workshop on Logic Synthesis, Research Triangle Park, North Carolina, May 12-15, 1987, Vol. I.

[4] Sasao, T.: Input Variable Assignment and Output Phase Optimization of PLA's, IEEE Trans. on Comp., October 1984, Vol. C-33, No. 10, pp. 879-894.

[5] Rudell, R., and Sangiovanni-Vincentelli, A.: Multiple-Valued Minimization for PLA Optimization, Proceedings of 17th Intern. Symposium on Multiple-Valued Logic, Boston MA, May 26-28, 1987, pp. 198-208.

[6] Perkowski, M., Uong, H.: Generalized Decomposition of Incompletely Specified Multioutput, Multivalued Boolean Functions, submitted.

[7] Brayton, R.K., Mc Mullen, C.T.: The Decomposition and Factorization of Boolean Expressions, International Symposium on Circuits and Systems, Rome, Italy, 1982, pp. 49 - 54.