Optimisation of combinatorial logic

van Paassen, P.T.H.M.

Award date:
1985

Eindhoven University of Technology
Department of Electronic Engineering
Laboratory Automatic System Design

OPTIMISATION OF COMBINATIONAL LOGIC

P.T.H.M. van Paassen

Thesis performed during 1.2.1984 - 21.2.1985
by order of prof. dr. -ing. J.A.G. Jess
and supervised by ir. J.F.M. Theeuwen.

ABSTRACT.

An optimisation of a set Boolean expressions, describing a combinational network, is implemented in PASCAL. An algebraic technique is used to decompose the expressions. After identification of common subexpressions, these are substituted by new variables. Owing to this a set of less complex expressions is obtained, together with a set of common subexpressions, represented by new variables.
A heuristic is presented for a realisation in NMOS random logic. The realisation consists of a NAND/NOR gate configurations and the (sub)expressions, obtained from the optimisation, are decomposed and translated into a description of such gates.

CONTENTS

1. INTRODUCTION.

Design of digital circuits can be divided into three parts: High-level synthesis, Logic synthesis and Physical synthesis. The topic of this report should be placed in the second part.
Logic circuitry can be categorised in terms of combinational/sequential, static/dynamic and the way of arrangement. The arrangement of logic circuitry varies from completely random via structured random (gate structures) to highly structured, for instance a Programmable Logic Array (PLA).
This report considers the design of combinational logic of more or less random nature, which can be part of a larger, different category. It handles an automatic optimisation of a combinational network, given in a two-level-logic (Boolean) description. The outputs of the network are given as a Boolean function of the inputs and in a realisation, many intermediate results, which can be common to several outputs, are computed. This commonality implies a minimisation of the network description and to some extend of the circuit.
Since this synthesis process is automatic and does not require designers intervention, it can be used as a component of a larger system. However, the automatic character holds a structured optimisation, which minimises design time and with that the design costs at the expense of physical performance. So it should be used only when achievement of functionality is not too closely tied to achievement of operational performance.
The second part of this report considers a network description. A description in static logic, using distributed input gate structures in NMOS technology. The decomposition needed to obtain this description is based on the same technique as used for the optimisation.

2. OPTIMISATION OF BOOLEAN EXPRESSIONS.

To obtain a minimal realisation of a set of Boolean expressions, we have to look for common subexpressions. Decomposing the expressions will give us the subexpressions and by comparing them we obtain the common ones.
A common subexpression need to be realised only once and therefore can reduce area, the number of components and power consumption. The problem is to define a subexpression which is easy to compute and by which all subexpressions can be acquired.
Brayton and McMullen present in [1] a technique for the decomposition of Boolean expressions. By giving some basic definitions we will derive a suitable definition for a subexpression.

2.1 Definitions.

A variable is a symbol (usually a character) representing a coordinate of the Boolean space. Variables and their negations are called literals.
A function can be given as a sum of products. A product is a set of literals such that it contains either a variable or its complement and will be referred to as a cube.
A Boolean expression is a set (read sum) of cubes, which as opposed to a function has to be nonredundant.
The support of an expression f (sup(f)) is the set of variables which are appearing straight or as a negation in that expression.

E.g. let

$$f = A + B'C,$$

then

$$sup(f) = \{A,B,C\}.$$

Using the support of an expression we define that an expression f is orthogonal to g if their supports do not hold common variables. This definition (orthogonality) makes it possible to define the algebraic operation product and its inverse operation, division.

The product of two expressions f and g, containing the cubes f1,f2,...fn and g1,g2,...fm respectively, consists of all cross-terms fi*gj.

E.g.

$$f = A + B$$

and

$$g = C + D,$$

then

$$fg = AC + BC + AD + BD.$$

A zero product, a variable multiplied with its complement, can never occur, because the product of two expressions is only defined if they have disjoint supports and therefore this product can be called algebraic.
The inverse operation, division of an expression (f) by another one (g), is defined as the largest set of cubes common to the results of dividing the 'numerator' (f) by each cube of the 'denominator' (g), presuming that sup(f/g) is not empty.

Let

$$f = AB + AC + AD + BC + BD,$$

then

$$f/A = B + C + D$$

and

$$f/B = A + C + D.$$

When

$$g = A + B,$$

then

$$f/g = C + D$$

and

$$f = (A + B)(C + D) + AB.$$

This division is called weak division as opposed to strong division, which is used in [3] for simplification. The result of strong division would have been:

$$f/A = B + C + D + BC + BD.$$

If (f/g)g = f holds, we say g divides f evenly. If only 1 divides f evenly then f will be called cube free (no cube can be factored out, comparable with a prime number).
The set of cubes and subexpressions that divide an expression will be called the set of primary divisors. From this set a subset is taken, that satisfies the properties that its elements are easy to compute and that the set is large enough to obtain all subexpressions. This subset consists of the cube free primary divisors and such a divisor will be referred to as a kernel.

Although the subset satisfies the two properties, still the kernels can be difficult to manage.

E.g. k = A(B(C + D) + E) + F

Suppose k is a kernel, occurring in some expression. Considering the comparison for acquisition of subexpressions common to two or more expressions, k is still a complex expression which is not easy to identify.
Further, k contains the kernel B(C + D) + E, which in its turn contains the kernel C + D. By assigning a level to these kernels, according their relation, the following table is obtained.

A(B(C + D) + E) + F	level 2
B(C + D) + E	level 1
C + D	level 0

Table: level assignment.

The motivation to choose the set of kernels of level zero is twofold. In the first place, they have the property that they are subexpressions containing at least two cubes which do not have literals in common. Consequently, they are easy to identify and therefore the change of finding a larger number of equal kernels is bigger. secondly, it is easy to retain (by collapsing) the kernels of a higher level, so there is no loss of effectiveness.

2.2 **Algorithms.**

The minimisation of a set of Boolean expressions consists of three stages. The first stage is finding common subexpressions (kernels) and substituting them by new variables. The next stage, called condensation, is extracting common cubes, which will be substituted too. Collapsing, the final stage, is a correction or finding a certain balance between the first two.

Distillation.

Distillation, the first stage, consists of computing the set of kernels of each expression. This set contains all quotients f/c such that c is a cube, f/c is cube-free and no literal in f/c appears twice. Now, having a set of kernels of each expression, the comparison of these kernels can start in order to acquire the common ones.
The comparison of two kernels consists of taking the intersection of their set of cubes. If this intersection contains at least two cubes, a common subexpression is found. This subexpression will be substituted by a new variable and after all possible substitutions we obtain a set of expressions with reduced complexity and a set of new variables representing the substituted kernels.
Also complements of kernels are substituted. From each kernel its complement is taken and all expressions are divided by that complement. If an expressions contains the complement, the negation of the new variable, representing that kernel, is inserted.
This procedure is repeated until no common kernels can be found. At this stage the set of expressions is called relatively kernel free, which means that only cubes are their common divisors.

Condensation.

The relatively kernel free expressions have left only single cubes as their common divisors. Since the representation of the expressions as sum of cubes, it is straightforward that these are much easier to identify than the kernels.
In this stage all cubes are compared and they will be pulled out when their intersections contain more than one literal,
After condensation only single literals are left as common divisors.

Collapsing.

Collapsing is the reverse action of extraction. It is possible, as shown in the example, that a new variable, representing a subexpression, appears only once in another subexpression and it is obvious that this does not optimise that subexpression. Therefore the variable can be pushed back into that expression. However, when such a subexpression describes, in a certain technology, a cell or gate, performing this back-substitution can be unnecessary (chapter 4).
Pushing back a new variable reduces the number of delay stages. Every new variable is an extra stage (an

intermediate signal) in the circuit, which implies an extra delay. So, collapsing can also be used to control the number of delay stages or to obtain a smaller set of somewhat larger subexpressions. By the latter, kernels of a higher level can be obtained.

Example.

Consider the expressions:

$$f1 = AB(C(D + E) + F + G) + H$$
$$f2 = AI(C(D + E) + F + J) + K$$

For surveyability they are given in a factorised representation.

First the kernels of level zero for both expressions are computed:

$$f1 : (ABC) \quad D + E$$
$$f2 : (AIC) \quad D + E$$

They are equal and the substitution of L = D + E will result in:

$$f1 = AB(CL + F + G) + H$$
$$f2 = AI(CL + F + J) + K$$

The kernels obtained by the next pass of distillation are:

$$f1 : (AB) \quad CL + F + G$$
$$f2 : (AI) \quad CL + F + J$$

They are not equal, but they have two cubes in common and consequently the substitution M = CL + F is performed:

$$f1 = AB(M + G) + H$$
$$f2 = AI(M + J) + K$$

Another pass gives:

f1 : (AB) M + G
f2 : (AI) M + J

They are not equal either, but now they do not have (at least) two cubes in common and consequently the expressions are relatively kernel free and condensation can start.
Condensation consists of comparing the cubes of the expressions and substituting the common cubes, which contain at last two literals.
Only the cubes:

f1 : ABM
f2 : AIM

are having such a cube in common. That cube is AM and by substituting N = AM we obtain the expressions:

f1 = B(N + AG) + H
f2 = I(N + AJ) + K

and the new variables:

L = D + E
M = CL + F
N = AM

The expressions are now relatively cube free, only single literals are their common divisors.
Now the final stage, collapsing, can be started. Looking at the new variables, we see that L and M appear only once. This does not optimise the subexpressions and therefore they can be collapsed (pushed back).

Collapsing will give the final result:

$$f1 = B(N + AG) + H$$
$$f2 = I(N + AJ) + K$$
$$N = A(C(D + E) + F)$$

where N contains a kernel of level one.

An example with a small set of expressions is chosen for a clear explanation of the algorithm at the expense of showing a large gain. Furthermore, the gain depends on the way of realisation.

3. IMPLEMENTATION.

Before describing the implementation of the three stages of optimisation, the internal representation of the expressions and the necessary data structures are considered.

3.1 Data structure.

An expression is defined as a sum of products (cubes) and as such its internal representation is chosen.
A literal is represented by an integer value obtained from the Pascal ORD-operation. A cube, which is a product-term, is a Pascal set of these integers.
A variable is usually represented by a character and the ORD-operation gives the ASCII-code for that character.
E.g. ORD(A) = 65. By subtracting 64 we obtain 1 as a value for the first character (A has the lowest ASCII-code). The set of variables which can be used is:

```
 A(1)   - Z(26) , a(33)  - z(58),
#A(59)  - #Z(84) , #a(91)  - #z(116),
@A(117) - @Z(142), @a(149) - @z(174).
```

The numbers denote the value by which the variables are represented. The highest value is 174 and will be used to form the negations.
A negation is denoted by ' and the value for internal representation will be the one of the variable + 174, e.g.: a' is represented by 33 + 174 = 207.

E.g. let

f = AB + C'D

then the internal representation is:

f = [1,2] + [4,175]

A cube is stored in a Pascal pointer record and an expression is formed by a linked list of suchs records.
An expression consist of two kinds of pointer records. First a pointer record (register), which contains information about that expression and second a pointer record (link), which contains a cube of that expression.

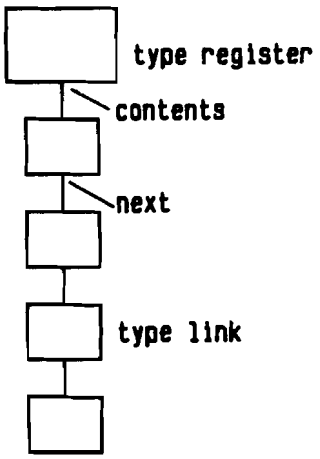

Figure 1. Internal representation of an expression, a record of type register and a linked list of link records.

pointer record fields of type register:

union	:	set of all literals of the expression,
contents	:	pointer to linked list of link records,
nextexpr	:	pointer to register record of next expression in expression list,
nextkernel	:	pointer to register record of next kernel in kernel list,
owner	:	index of original expression,
numberofcubes	:	number of cubes the expression consists of,
divisors	:	divisors by which the expression is obtained,
newvariable	:	variable representing the expression.

Note: the use of the fields of register record depends on being part of one of the expressions or of one of their subexpressions (kernels).

pointer record fields of type link:

info	: set of literals the cube consists of,
next	: pointer to next link record.

The 'head' record (type register) of an expression contains the record field contents, which is a pointer of type link, pointing to the first cube of that expression (fig. 1).
The new variables, representing the substituted subexpressions, are stored in these records too. The set of expressions is contained in a list. This list is formed using the record field nextexpr (fig. 2).
A kernel is stored in a similar way as an expression. The record field owner registers to which expression the kernel belongs. The record field divisors is a cube containing the divisors by whose division the kernel is obtained.

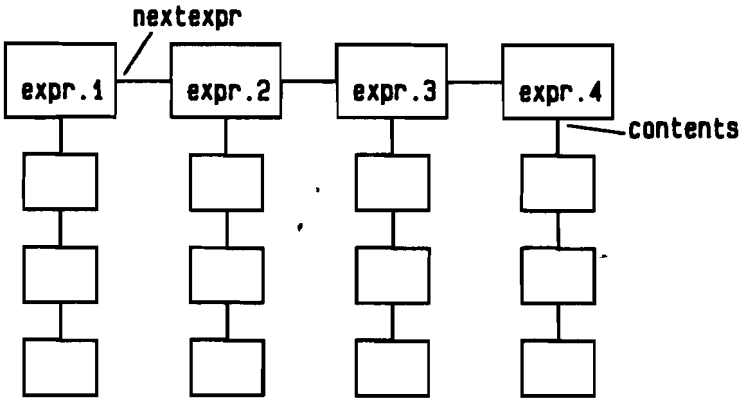

Figure 2. List of expressions.

All the computed kernels of an expression are also stored in a list. The record field nextkernel in the register pointer record of an original expression points to the first kernel and in every kernel 'head' it points to the next kernel in the kernel list. If a kernel appears more than once in an expression, the record field nextexpr in the 'head' of that kernel will point to the 'head' of the equal one.

When the kernels of all expression are computed, we obtain the structure given in fig. 3.

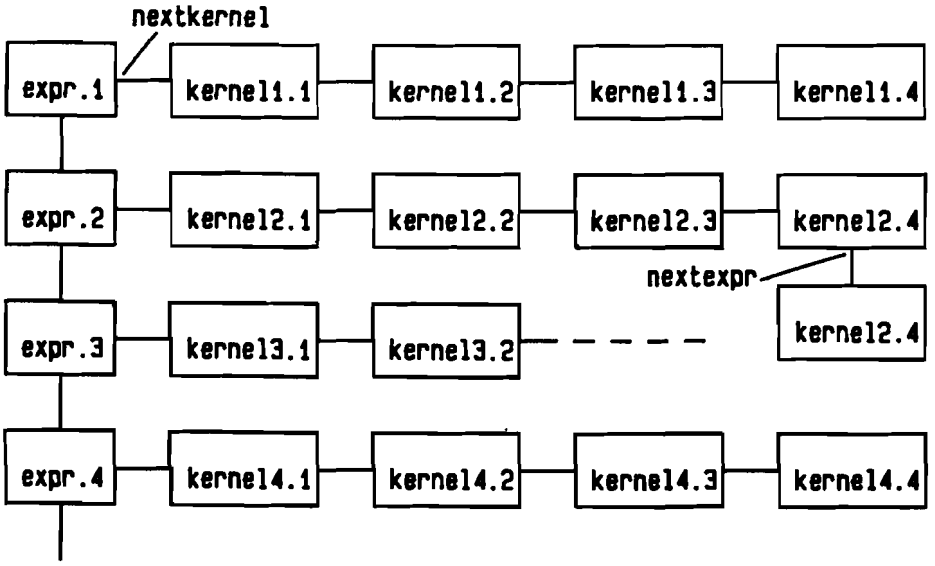

Figure 3. List of expressions with kernels.

Comparison of these kernels gives strings of equal kernels. Here the word string instead of list is used, to indicate the difference with the list in which these strings are stored. These strings are stored in a list of a third kind of pointer record.

pointer record fields of type point:

equalkernel	: pointer to string of equal kernels,
next	: pointer to next point pointer record,
previous	: pointer to previous point pointer record,
setofexpr	: a set of expressions containing the kernel,
numofkernel	: number of equal kernels in string.

A somewhat similar pointer record is used for condensation.

pointer record fields of type drip:

drop	: cube containing literals common to two or more cubes,
setofexpr	: a set of expressions containing the drop,
amount	: number of appearances of drop,
count	: number of literals in drop,
next	: pointer to next drip pointer record.

The lists are ordered according an object function. For the result of distillation the object function is the number of literals, which the kernel contains, multiplied by the number of appearances of that kernel.

The results of condensation are sorted according the number of literals in the cube and when there are cubes with that number of literals, the number of appearances of that cube is considered. The small difference is caused by the fact that the list with equal kernel strings is always in the right order. A new string is placed according the object function. The list with the results of condensation is sorted afterwards and its records are not provided with a record field previous.

3.2 Distillation.

The structure of the program is shown in procedure 1:

```
procedure decomposition;

  "read boolean expressions";

  while "new equal kernels"
  do
    distillation;
    kernelnegation;
  od;

  condensation;
  collapsing;

corp;
```

Procedure 1.

In this and the following sections will be dealt with distillation, kernelnegation, condensation and collapsing.

The first step in the optimisation is distillation, this is the extraction of subexpressions (kernels of level zero) from each expression in the set of Boolean expressions and substituting the common subexpressions by new variables.
This is described in the procedure distillation (proc. 2).

This procedure is repeated until no more common kernels can be found.
As discussed before, the computed kernels are listed behind their expression.
The kernels are computed recursively by the procedure kernel (proc. 3).

The procedure divide executes the weak division and the procedure leveltest tests if the subexpression is a kernel of level zero.
The result (quotient) of the division consists of those cubes of the expression containing the divisor, with the divisor left out. At the same time the number of cubes (numcube) of the result is registered.
The procedure leveltest tests if the subexpression satisfies

```
procedure distillation;

  for "all expressions"
  do
    "compute kernels";
  od;

  "compare kernels";
  "replace equal kernels";
  "substitution of equal kernels";

corp;
```

Procedure 2.

```
procedure kernel(divisor,expression);

  while divisor <= 'highest' literal
  do
    if divisor in expression
    then
        divide(divisor,expression,quotient,numcube);
        if numcube > 1
        then
            if divisor <> 'highest' literal
            then
                kernel(divisor + 1,quotient);
            else
                leveltest(quotient,divisor);
            fi;
        else
            leveltest(expression,divisor);
        fi;
    else
        if divisor = 'highest' literal
        then
            leveltest(expression,divisor);
        fi;
    fi;
    divisor := divisor + 1;
  od;

corp;
```

Procedure 3.

the property of a kernel of level zero: a literal appears only once. If it is a kernel it will be placed at the end of the list of kernels except when it appeared before, then it is placed 'under' that equal kernel (record field nextexpr, fig. 3).
At the first call of the procedure kernel, it starts with dividing an expression by its 'lowest' literal. Then a check on the result (quotient) is performed. If the result contains more than 1 cube, the procedure kernel will be called recursively with the result of the previous division. So the result will be divided by the next literal and checked again. This will be repeated until the check (number of cubes > 1) is negative or the divisor was the 'highest' literal (appearing in the expression). In the latter case the procedure leveltest will check if the result is a kernel of level zero.
However, if the result does not contain more than 1 cube (a kernel of level zero has at least 2 cubes), the expression on which the division was performed has to be checked by procedure leveltest.
The procedure kernel will be left when the divisor reaches its 'highest' value or a kernel is found.
In case all checks are positive the next call of the procedure kernel will be with the result of the division of the expression, whose previous quotient was involved in the previous call of the procedure kernel.
Let an expression f contain amongst others the literals X and Y (Y is 'higher' than X). When that expression f is divided by X, the procedure kernel will be called with f/X and divisor Y, assuming that f/X at least two cubes. When leaving this call, the one which was entered with f/X and Y, f will be divided by the next literal Y and the procedure kernel will be called with f/Y.
Following this procedure we achieve that an expression will be divided by each possible combination of literals, which could lead to a kernel of level zero.

E.g. let f = AB(G + H) + AC(I + J)

with the kernels of level zero:

f/AB = G + H and
f/AC = I + J.

The procedure kernel will be called with f and divisor A, then recursively with f/A and divisor B. This leads to f/AB = G + H. f/AB contains more than 1 cube and B is not the 'highest' literal. Consequently, the procedure kernel is entered with f/AB and divisor C. C is not contained in

f/AB, so the divisor is increased till it reaches the first ('higher') literal in f/AB which is G.
f/ABG = 1, which has only one cube. Now, leveltest is called with f/AB. This is a kernel and is placed in the kernel list. Leveltest raises the divisor to the 'highest' literal, so the procedure kernel, entered with f/A and divisor B will be left, because a kernel is found. Next the divisor, B, is 'increased' and procedure kernel is entered with f/A and divisor C, which, eventually, leads to kernel I + J.

After computation of all kernels of each expression we start to compare them in order to obtain the common ones. The procedure comparekernel is performing this comparison (proc. 4).

It uses the boolean function equalcubes and the procedures save, interkernel and place.
The function becomes true when the two kernels have two or more cubes in common. If the kernels are equal, same_expression will stay true. When the kernels have at least two but not all cubes equal, same_expression becomes false.
The procedure interkernel is called when same_expression is false. This means that not all kernels in the string are the same and interkernel will take the intersection of the sets of cubes of the kernels in the string.

E.g. If the string contains:

A + B + C
A + C + D
A + B'+ C

then the result will be a string
with 3 equal cubes A + C.

The procedure savekernels puts the basic and all its equals according equalcubes in string. The procedure place places that string of equal kernels in the equal kernel list according the object function.

The procedure comparekernel starts to compare the first (basic) kernel of the first (basic) expression with the first (current) kernel of the second (current) expression. It will run down the string of the current expression and then the next expression of the current one will become the next current expression. This continues until the last expression has been the current one.
So the basic kernel has been compared with all kernels of

```
procedure comparekernel;

  "first basic expression";

  while "basic expression"
  do
    "first basic kernel";

    while "basic kernel"
    do
      "first current expression";
      same_expression : = true;
      equal := false;

      while "current expression"
      do
        "first current kernel";
        while "current kernel"
        do
          if equalcubes(basic kernel, current kernel,
                                              same_expression)
          then
              savekernels(basic kernel, current kernel);
              equal := true;
          else
              "next current kernel";
          fi;
        od;

        "next current expression";
      od;

      if not same_expression
      then
          interkernel;
      fi;

      if equal
      then
          place;
      fi;

      "next basic kernel";
    od;

    "next basic expression";
  od;

corp;
```

Procedure 4.

the expressions following the basic expression in the expression list.
Every time that equalcubes becomes true, the current kernel or both the basic and the current one are put in the string. After, if necessary, calling the procedure interkernel, place will put the sting in the equal kernel list.
Further, the next kernel in the basic expression will become the basic one and will be compared with all others kernels that were not equal to previous basic kernels. When leaving the procedure comparekernel a list of strings of equal kernels as described in [2.1] has been formed.

After comparison a test has to be performed in order to find out which kernels can be substituted together. It is possible that two kernels of one expression appear in more than one expression and both containing a part of the same cube of that expression. If this is the case, only one of the two kernels can be substituted.
Suppose an expression contains the kernels E + F and C + D, which are obtained by dividing the expression by ABC and ABE respectively. It is clear that both kernel share a part of the cube ABCE of that expression. Therefore the kernels are replaced behind their expression. Before replacement of the next string of equal kernels, the test is performed with each kernel that already has been replaced. They are replaced before substitution to be able to test the less frequently appearing kernels.
The procedure replace (proc. 5) is performing this by using the procedures kerneltest and execute.

The most frequently appearing kernels, which are stored in the first string of the equal kernel list, can be replaced without the test. The procedure execute places the kernels behind their expression and assigns a new variable to them.
The basic string is the string whose kernels have to be replaced and the basic set is containing the indices of the expressions in which that kernel is appearing. The current string is the one whose kernels have already been replaced.
The Boolean variable empty indicates if the contents of the intersections of the expression sets of the basic string with all the current strings, contains any expression number.
If indices of expressions are appearing in both the basic and the current set, the kernels of each of those expressions in the basic string have to be tested with the already replaced kernels.
The procedure kerneltest is performing this test and disposes the kernels in the basic string which cannot be substituted together with the already replaced kernels. Further, the expression concerned will be removed from the basic string (later this string will become a current one).

```
procedure replace;

  "first basic string";
  execute(basic string);

  while "basic string"
  do
    empty := false;
    "first current string";
    while basic set * current set = [] and not empty
    do
      "next current string";
      if current string = basic string
      then
          empty := true;
      fi;
    od;

    if not empty
    then
        while current string <> basic string
        do
          if basic set * current set <> []
          then
              kerneltest;
          fi;
          "next current string";
        od;
    fi;

    if "more than 1 kernel left"
    then
        execute(basic string);
    fi;
    "next basic string";
  od;

corp;
```

Procedure 5.

If only one kernel survives the test, it will not be replaced.
After replacing all the kernels the substitution can start. The replaced kernels and their divisors are needed for removal of the cubes concerned. Their divisors are inserted

together with the new variable, which represents the kernel. At the same time a list with the new variables (in record field newvariable) and what they represent, is formed.

3.3 Kernel negation.

After each pass of distillation the complement of the new substituted kernels will be taken, to check if they appear in the set of expressions or in the already substituted subexpressions.
The complements are taken, using the fast recursive Boolean function manipulation [4]. These manipulations have been implemented by Ad Smits [3].

The complementation is based on the Shannon expansion:

$$f = xf\hat{}x + x'f\hat{}x',$$

where f^x and f^x' are the cofactors obtained by dividing f

Complementation and the strong division commute, therefore the expansion can be written as:

$$f' = xf'\hat{}x + x'f'\hat{}x'$$

and using that property:

$$f' = x(f\hat{}x)' + x'(f\hat{}x')'$$

By this expansion a function is broken down to unate functions and because of that property only the unate functions have to be complemented.
Using this tool the complement of the kernels are obtained. The expressions will be divided by the complements and if a complement is contained in an expression, the complement of the variable representing that kernel will be substituted.
The procedure kernelnegation describes this (proc. 6).

Each kernel from the last pass of the distillation is copied, by the procedure copy, to the data structure used for the complementation [3]. The Shannon expansion and the complementation are performed on this data structure. The

```
procedure kernelnegation;

  for "each new kernel"
  do
    copy(kernel);
    kernelcomplementation(kernel,complement);
    copyback(complement);

    for "each expression"
    do
      divisionofexpression(expression,complement,result);

      if "result exists"
      then
          "substitute negation";
      fi;
    od;

    if "complement contains 1 cube"
    then
        for "each kernel in list"
        do
          if "cube in kernel"
          then
              "substitute negation";
          fi;
        od;
    fi;
  od;

corp;
```

Procedure 6.

result will be copied back to the original data structure. Each expression is divided by the complement of the kernel. This is performed by the procedure divisionofexpression. This procedure divides the expression by the first and second cube of the complement. If they contain common cubes, then the expression will be divided by the next cube of the complement, assuming there is one. The result of the division by the next cube will be compared with the result of the previous divisions. As long as these result have cubes in common, the complement can be contained in the expression. The final result will be the set of cubes common to the results of the separate divisions.

If there is a result, the complement is substituted in the expression concerned. When the result contains only one cube, e.g. the complement of A + B is A'B', then it is possible that an already substituted kernel contains this cube and it will be substituted too.

3.4 Condensation.

This stage is entered with a set of Boolean expressions which are relatively kernel free, only cubes are their common divisors. All cubes have to be compared (taking the intersection) to obtain these common cubes. The cubes which appear in more than one cube have to be substituted by a new variable, which will be placed in the list of new variables. This is described in the procedure condensation (proc. 7).

The structure of the comparison of the cubes is similar to the comparison of kernels in procedure comparekernel.
A cube (basic cube) is compared with all other cubes (current cube). The first basic cube is the first cube of the first (basic) expression and the first current expression is the basic one too, because the first current cube is the cube next to the basic cube.

The procedure comparecube compares the basic and the current cube. It takes the intersection of both cubes and this intersection will be stored in the condensation list if it contains more than one literal. However, if a basic cube has equal intersections with two other cubes and when the first of these two cubes becomes the basic one, then this basic cube will have an equal intersection with the third cube. This intersection has been registered before and therefore it should not be registered again.
Suppose the first basic cube has been compared with all other cubes and some common intersections have been found and are registered in the condensation list. During changing of basic cube the end of the condensation list is stored and the comparison of the new basic cube can start. Every time a new common intersections is found, we have to compare it with the previous found ones. Starting with the first in the condensation list, we run down that list till an equal one is found or the end of the list is reached. When the end has been reached, the intersection is stored in a newly linked record. When, however, an equal one is found, depending on having passed the previous stored end of the condensation list, we can tell whether or not the new intersection has been registered before. If the stored end is passed the intersection has not been registered before.

```
procedure condensation;

  "first basic expression";

  while "basic expression"
  do
    "first basic cube";

    while "basic cube"
    do
      "first current expression";

      while "current expression"
      do
        "first current cube";

        while "current cube"
        do
          comparecube(basic cube,current cube);
          "next current cube";
        od;

        "next current expression";
      od;

      "next basic cube";
      "store end of condensation list";
    od;

    "next basic expression";
  od;

  sortlist(condensation list);
  cubesubstitution;

corp;
```

Procedure 7.

Thus, by using a Boolean variable, which tells if the stored end has been passed, and storing the end of the condensation list, when changing the basic cube, it is possible to avoid this double registration.

The condensation list is sorted by the procedure sortlist. The records are sorted according the number of literals in

the intersection and second according the number of appearances. The largest cubes will be substituted first and this substitution is performed by the procedure cubesubstitution. Intersections with less literals can be part of bigger ones and therefore they are only substituted where they are not part of these bigger ones.

One pass of condensation does not guarantee that all common cubes with at least two literals are substituted. E.g. let the condensation list contain the following cubes:

```
ADEF    in expressions  [1,3]
DHIJ    in expressions  [2,5]
BCD     in expressions  [1,2]
```

which are contained in the cubes ABCDEF and BCDHIJ of the expressions 1 and 2 respectively. Further, the condensation list does not contain the cube BC. So, if the first two cubes are substituted,then the cube BCD cannot be found any more and the common cube BC will be left unsubstituted. Therefore the list with the new variables is put at the end of the expression list and the common subexpressions will be numbered, like the original set of expressions. Then the condensation is performed on this extended set of expressions and the cubes which were not found in the first pass will be found now. Also the cubes common to the cubes which were found in the first pass of condensation will be found.

3.5 Collapsing.

The collapsing starts with the determination of the variables which appear only once. This is done by taking the intersection of every combination of two sets of literals (record field union) of expressions and subexpressions. The union of these intersections contains the literals appearing in more than one (sub)expression. So all variables in the list of new variables, representing subexpression, which do not appear in that union, appear only once and therefore can be pushed back (collapsed).
However, if that variable represents a subexpression which is easy to realise in a chosen implementation structure, it is clear that it should not be pushed back. And if only the negation of that variable is used, it should not either.

4. REALISATION OF A BOOLEAN FUNCTION IN NMOS RANDOM LOGIC.

The reason for choosing NMOS technology is that this process is running in the EFFIC, the IC-fabrication laboratory of EINDHOVEN UNIVERSITY OF TECHNOLOGY. In NMOS technology, circuits can be constructed with NAND, NOR and inverter configurations. The distributed input gate structure is chosen to be able to use an automatic layout generations followed by placement and routing. The constraints are a maximum number of 3 drivers in series or parallel. A function is represented as a sum-of-products and for realisation in such gates, it has to be decomposed. We try to do this using similar techniques as used before, namely division. This can be either "weak" or "strong" division. "Weak" division is a better choice, because realisation of circuits using "strong" division leads to realisations with redundancy. A network description for a realisation in the gate structure is obtained and written down in a file called gate file.

4.1 Decomposition

Decomposing a function f can be done in the following way:

$$f = x\, f\hat{}x + r. \qquad (1)$$

The function is represented as a product of a literal x and its cofactor plus a remaining part r. Here, the cofactor f^x is the result of dividing f "weakly" by x.
For a realisation in NMOS technology a function has to be realised in universal logic gates (ulg.) like:

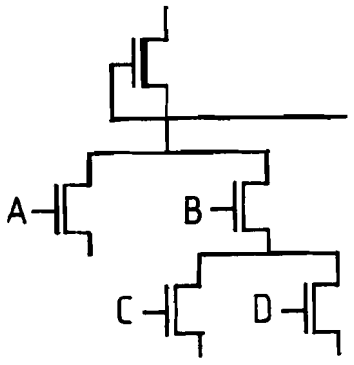

Figure 4. An example of a universal logic gate.

Actually, the drivers are realising the complement of the function. Therefore we can write (1) as:

$$f = (x\ f\hat{}x + r)''. \qquad (2)$$

Every gate contains a complementation and the drivers are realising the function:

$$f' = (x\ f\hat{}x + r)'. \qquad (3)$$

This can be written as:

$$f' = (x\ f\hat{}x)'\ r' \qquad (4)$$

or

$$f' = (x' + f\hat{}x')\ r'. \qquad (5)$$

Which of these possible notations will be chosen depends on realisation of f^x and r. These subexpressions can be handled in a similar way as f. If for instance the literal x is a complement of a variable, not yet realised, (5) could be a right choice, because you do not need to invert the variable.
On the other hand, when f^x is easier to realise than f^x', (3) or (4) would be a better choice. So, depending on x and f^x and in a same way on r a choice has to be made.
However, our starting-point is the output of the optimalisation program and therefore only r has to be handled in the same way. The cofactor cannot contain another literal twice, because this would indicate another condensation result.
Another point is the choice of the literal x. It is not difficult to see that to obtain a minimal realisation the most frequently appearing literal should be chosen.

When to take a decision and what are the constraints?
Two constraints are:

- a maximum of 3 drivers in series,
- a maximum of 3 drivers in parallel.

Till a maximum of 3 drivers in series, it is possible to use a standard load transistor for automatic layout generation. The number of parallel drivers is limited to prevent a too large delay. By dividing the function and the subexpressions until no literal appears more than once we obtain kernels of level zero and single cubes. When those kernels are obtained decisions can be made on how to realise

them and doing so we are also taking decisions on a higher level (stage).

First we look for one of the most frequently appearing literals and divide the function by it. The cofactor will be a kernel and for the remainder this process will be continued until the remainder is a kernel of level zero itself or a single cube. Depending on the obtained kernel we have to determine if the kernel has to be realised on his own or together with a driver of a former divisor in series. The latter is the case if the kernel contains 3 cubes or less each with no more than 2 literals. When a gate has been formed, it can be considered as a single variable or a negation.

4.2 Kernel realisation.

Now the problem of realising a kernel of level zero. A kernel with less than 4 cubes and with cubes containing 3 literals or less can be realised in one ulg., which, depending on the number of complemented variables, has to be invertered (compare fig. 5 and 6).

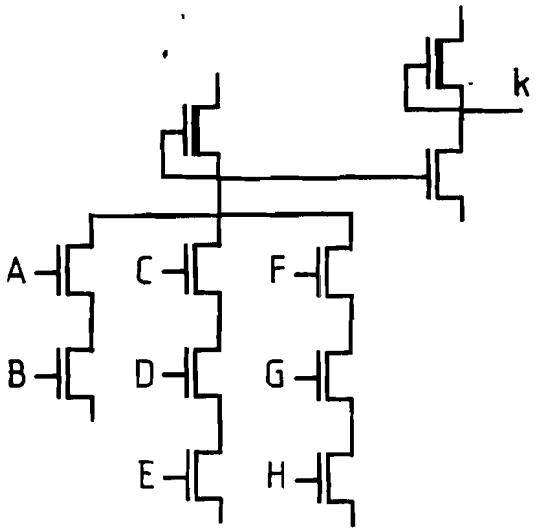

Figure 5. Realisation of
k = AB+CDE+FGH.

A kernel formed by 4 cubes or more has to be divided in groups of preferable 3 cubes. Each group has to be handled as a kernel of 3 cubes or less and on a higher level (stage) it can be considered as a variable or a complemented one.

The realisation of a kernel depends on the number of negations in that kernel. If a kernel does not contain any negations, it is profitable to realise it with an inverter. This can be seen in figure 5. The drivers realise the kernel. However, if a kernel contains only negations, it is realised as shown in figure 6. Now, the drivers are realising the complement of the kernel.

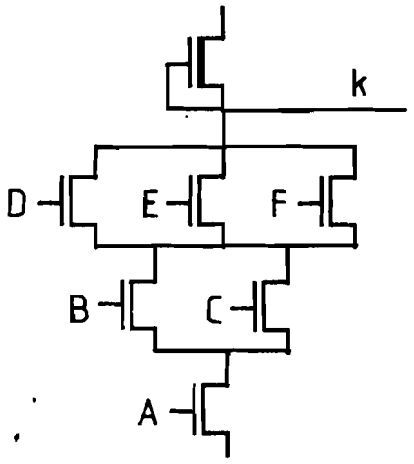

Figure 6. Realisation of
k = A'+B'C'+D'E'F'.

These are the two extreme situations, if not all literals are only variables or only negations, an intermediate solution has to be formed (fig. 7). Due to the constraints, the 'big' cubes, containing more than 3 literals, have to be decomposed and can be considered as a cube with 1 to 3 literals (possibly output of gates) as shown in the next section. A kernel containing 'big' cubes can only be realised after the decomposition of these 'big' cubes.

A close examination of the figures 5, 6 and 7 leads to the following heuristics. A cube with only variables has to be realised as a NAND whose output has to be complemented in

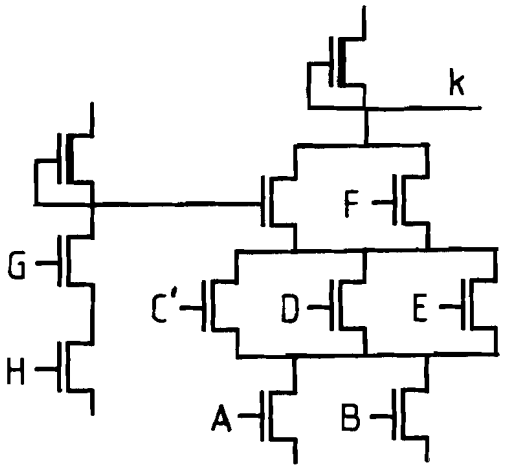

Figure 7. Realisation of
k = A'B'+CD'E'+F'GH.

the realisation of the kernel, containing that cube. A cube with only negations has to be realised in a NOR configuration. The cubes containing both variables and negations can be realised in either a NAND or a NOR configuration.

The following table contains some possible configurations.

cube	of kernel	inverted NAND		NOR
1	ABC	(ABC)''		
2	ABC'	((A+B+C')''	or	((AB)'+C)'
3	AB'C'	(A(B+C)')''	or	((A'+B+C)'
4	A'B'C			(A+B+C)'
5	AB'	(AB')''	or	(A'+B)'

Table. Realisation of single cubes as part of a kernel.

The cubes in a kernel, which can be formed with an inverted NAND, hold a realisation in 2 logic stage, indicated by a

double '. The NOR configuration holds a single stage.
When a kernel is realised, it is clear that this should be done with as few stages as possible. Thus, whenever a NOR configuration can be used, it should be. However, one of the constraints is the maximum (depth) number of drivers in series is 3. A NAND configuration with the necessary inverter costs 1 driver in depth and one NOR configuration too. So, in the cases 2, 3 and 5 of the table, the NOR configuration is preferred, but if a depth of 3 is exceeded, they possibly should be included in a NAND configuration (see example). A NAND can contain a maximum of three cubes (fig. 5).

This leads to the following algorithm. It starts with sorting the kernels according the 3 groups. The order is cubes without negations, cubes with both negations and variables and at the end the cubes containing only negations. The kernels are formed by filling up the gates till maximum width (drivers parallel) or maximum depth has been reached.
Every time a depth or width of three is reached, a new gate is formed and these new gates have to be combined to realise the kernel.

E.g. let k = BCD+EFG+KLM+NOP'+QRS+T'U'V'+HI'J'.

This kernel contains the following cubes in the chosen order:

BCD
EFG
KLM
QRS
HI'J'
NOP'
T'U'V'

Grouping them gives:

(BCD+EFG+KLM)'
(QRS+H(I+J)'+NOP')'
(T'U'V')'

The kernel can be realised as a NAND of these 3 groups, which leads to the gates:

```
1    (BCD+EFG+KLM)'
2    (I+J)'
3    (QRS+H(2)+NOP')'
4    ((1)(3)(T+U+V))'
```

The kernel is realised by gate 4.

For realising this kernel a depth a depth of 3 has been reached. In case k would have contained the cube A'W' too, a depth of 4 would have been reached. This would lead to an extra gate for realising A'W':

```
5    (A+W)'
```

And the kernel would be realised as:

```
6    (4+5)''
```

The kernel realised by the complemented output of gate 6, can be combined with the divisor by whose division this kernel was obtained. Let Z be that divisor, a possible form for gate 6 will be:

```
6    (Z'+4+5)''
```

4.3 Realisation of a 'big' cube.

Cubes with more than 3 literals have to be decomposed into a complemented sum of complemented product terms, preferably containing 3 literals. Complemented variables should be grouped together (fig. 8). When there are more than 3 complemented product terms, they in turn have to be grouped too. When a cube has been grouped in such a complemented sum, then such a sum can be considered as a complemented variable on a higher level (stage).

The algorithm used for decomposing the 'big' cubes is quite simple, because the complement of a cube is a kernel. So, a simple form of the algorithm for the kernel realisation can

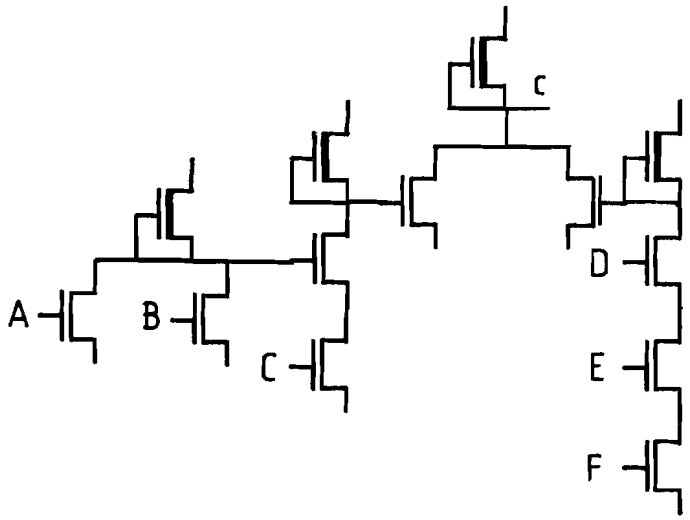

Figure 8. Realisation of 'big' cube
c = A'B'CDEF.

be used. Simple, because a cube of such a kernel contains only one variable or one negation.

E.g. let c = ABC'D'E'FGH'I'J'

and its complement is:

c' = A'+B'+C+D+E+F'+G'+H+I+J

This leads to:

1	(C+D+E)'
2	(H+I+J)'
3	((1)(2)A)'
4	(BFG)'
5	(3+4)'

Gate 5 is the realisation of the 'big' cube and can be considered as a variable in the kernel in which this cube appeared.
Literals are taken out of the 'big' cube one by one. The first literal will be the one with the 'highest' internal representation value. Groups of 3 literals are formed. Such a group will be written to the gate file and depending on the contents of that group, the output of the gate is

considered as a variable or a complemented one.

5. IMPLEMENTATION.

5.1 Data structure.

The expansion:

$$f = x\, f^{x} + r,$$

where r has to be expand further, lends itself to a recursive procedure. Before starting to describe that procedure, the data structure for decomposition is given. For the realisation, the following type of pointer record is used.

pointer record fields of type net:

info	: contains a cube,
plus	: pointer to other record,
product	: pointer to other record,
ands	: number of literals in cube,
ors	: number of cubes in subexpression,
negs	: number of negations in cube,
compl	: complement of gate,
gatenr	: number of gate described in gate file,
divisor	: divisor of expression.

Note that the storage, given in figure 9, would occur only when an expression is converted into this data structure and is not followed by the network description generation. In reality the conversion of a kernel or a cube is immediately followed by the translation into the network description.

An expression, internally represented as in the previous chapters, is decomposed in order to obtain subexpressions, which are easier to realise in universal logic gates. A more clear use of the record fields is given during the explanation of the recursive procedure network, which describes the decomposition and network description in universal logic gates. An expression is decomposed and converted into these records as shown in figure 9. By this conversion special data about is obtained, which is necessary for the translation to the network description. This data is stored in the records of type net.

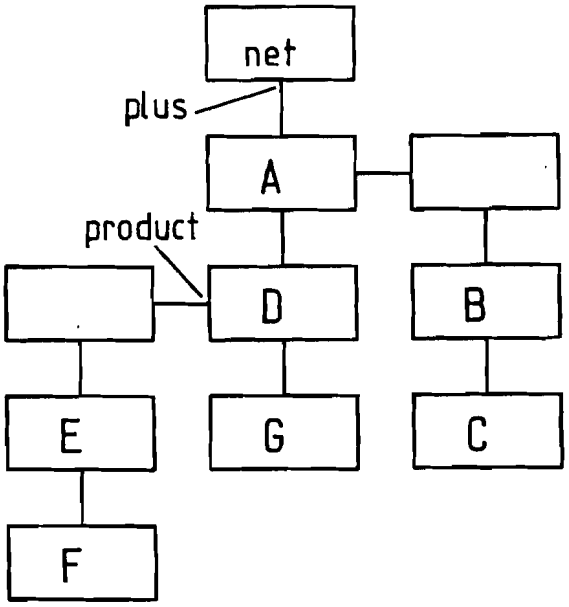

Figure 9. Storage of the expression:
A(B + C) + D(E + F) + G.

5.2 Decomposition.

The procedure network (proc. 8) is using the Boolean function literalfordivision and the procedures division, convert, writekernel and writenetwork.
The function literalfordivision is a boolean function and becomes true when a literal appears more than once. If the function is true, divisor will be the literal which appears most frequently. If there is such a divisor, the procedure division divides the expression by that literal and the result will be a quotient and remainder. The quotient is a cofactor of the expression and is a kernel of level zero.
During the division of the expression a part of the new data structure is build up. The 'head' of this linked data structure is net (see heading of procedure network). When the expression is divided, which means that there was a literal appearing more than once, a new record is created and contains this divisor. It is contained in the record field with the same name.
The record field plus of the 'head' record (net) of the list is pointing to this new record (fig. 9).

The procedure convert converts cubes of kernels to the new data structure. The record field info contains the cubes and in the fields ands and negs the number of literals and negations are registered.
The record field product of the record, containing the divisor, points to a record which is the 'head' of a kernel

```
procedure network(expression,net);

  if literalfordivision(expression,divisor)
  then
      division(expression,divisor,
                          quotient,remainder);
      convert(quotient,p_net);
      bigcubes(p_net);
      writekernel(p_net);

      if "remainder"
      then
          network(remainder,p_net);
      fi;

  else
      convert(expression,p_net);
      bigcubes(p_net);
      writenetwork(net);
  fi;

corp;
```

Procedure 8.

(p_net). The record field plus of this 'head' points to the linked list of records, containing the converted kernel (fig. 7) and the record field ors contains the number of cubes of that kernel.
The procedure bigcubes checks if the cubes of the subexpression contain more than 3 literals. Such cubes have to be decomposed. The procedure decomcube, called by bigcubes, performs this decomposition.
The procedure writekernel writes a network description of the kernel concerned to the gate file, which consists of lines with the number of a gate and the NAND/NOR configuration it represents.

E.g. let k = ABC + DE

then writekernel will write in the gate file:

1 (ABC+DE)'

Further, record field gatenr of the 'head' of the kernel will contain number 1 and the field compl will become true.

This means that the divisor concerned ought to be multiplied by the complement of the output of gate 1. The kernel can be considered as a complemented variable.
For every call of network, a new record, containing the divisor, is linked in the list of 'net' formed by the plus record fields. The procedure writenetwork can consider this string as a kernel, because every record is representing a cube formed by divisors, output of gates and original cubes.

5.3 Kernels.

The procedure writekernel writes the gates, which are realising a kernel, to the gate file.
The procedure writekernel proceeds from cubes containing less than 4 literals. These literals can be output of gates, realising a 'big' cube, or original literals. If such a gate number, an output of a gate, is stored in a record of which the field compl is true, it is considered as a complemented variable. The same applies to the procedure writenetwork, every cube consists of literals represented by a divisor together with a gate number, realising a kernel or a single gate output or an original cube, whether or not decomposed and consequently containing gate numbers.
First, the cubes are sorted in the right order and their numbers are counted. Then the cubes without negations are grouped together to form NAND configurations. Every NAND configuration contributes to the depth of the gate , which combines the inverted NAND and the NOR configurations. If necessary, a NAND, containing 1 or 2 cubes without negations, can filled up with cubes, containing both literals and negations, till a width of 3.
When all cubes without negations are implemented in gates, the total depth (number of NANDs and NORs left over) is counted and till maximum depth of 9, the output of the NANDs and the NOR configurations are divided in groups, containing a maximum depth of 3 and each group forms a NAND configuration, which are combined in 1 NOR, by whose complemented output the kernel is realised. The procedure writekernel, depending on the depth and whether or not the divisor is a complemented variable, can included the divisor in the realisation of the kernel.
The procedure writenetwork performs the same operations and considers this inclusion as a single literal.

5.4 'Big' cubes.

The 'big' cubes have to be complemented and the cubes of that complemented cube have to be ordered, first the ones with a variable and next the ones with the negation. The actual complementation is not performed, because the sorting of these cubes comes to the same thing as taking out the negations first and forming a NOR configuration of preferably 3 drivers parallel. Each output of such a NOR configuration is considered as a single variable. At the moment that all negations are taken out and are realised in gates, written down in the gate file, a cube with only variables is left over. This cube is realised by grouping preferably 3 variables together in a NAND configuration, whose outputs are combined in a NOR configuration. However, if more than 3 NAND configurations are created a larger number of NOR configurations has to be formed. Their outputs are forming a product, unless there are more than 3 NORs, they are combined in a NAND whose complemented output is realising the 'big' cube.
This decomposition is performed by the already mentioned procedure decomcube, which is called by the procedure bigcubes.

6. CONCLUSIONS.

By running some examples, it can be stated that the optimisation of a set of Boolean expressions, performed by the three stages distillation, condensation and collapsing leads, in a relative short time, to set of less complex expressions and some subexpressions.
By the third stage of the optimisation, collapsing, it is possible to steer the optimisation to make the final result more suitable for a particular circuit realisation, to control the number of delay stages or to obtain a starting-point for further optimisation, using Boolean instead of algebraic substitution. This Boolean substitution, which makes use of substituting subexpressions, containing redundant product terms, can lead to a better optimisation at the expense of computation time [2].
The obtained network description can be used for automatic generation of the layout of the gates and can be transformed into a net list for their placement.

7. REFERENCES.

[1] R.K. Brayton, C.T. McMullen,
"The Decomposition and Factorization of Boolean Expressions",
Proceeding of the International Symposium on Circuits and Systems, Rome, 1982, pp. 49-54.

[2] R.K. Brayton, C.T. McMullen,
"Synthesis and Optimization of Multistage Logic",
Proceedings of IEEE International Conference on Computer Design: VLSI in Computers, New York, 1984, pp. 23-28.

[3] A. Smits,
"Manipulation on Boolean Functions",
Thesis Report, Eindhoven University of Technology, Eindhoven, October 1984.

[4] R.K. Brayton, J.D. Cohen, G.D. Hachtel, B.M. Trager, D.Y.Y. Yun,
"Fast Recursive Boolean Manipulation",
Proceeding of the International Symposium on Circuits and Systems, Rome, 1982, pp. 58-62.

[5] R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli,
"Logic Minimization Algorithms for VLSI Synthesis",
Kluwer Academic Publishers, 1984.

[6] J. Mavor, M.A. Jack, P.B. Denyer,
"Introduction to MOS LSI Design",
Addison-Wesley Publishing Company, 1982.

[7] J. Welsh, J. Elder,
"Introduction to Pascal",
Prentice-Hall International, 1982.

[8] F.J. Hill, G.R. Peterson,
"Introduction to Switching Theory & Logical Design",
John Wiley and Sons, Second Edition, 1974.

8. APPENDIX.

An example of the optimisation.

The input is a set of 17 functions.

```
F 1 = BCDEHI'JKL'+BCDFHI'JKL'+DEHIJKL'O+DFHIJKL'O+
      D'EHIJKL'MN+D'FHIJKL'MN+ABCDE+ABCDF+AD'G+AE'F'G+
      D'GI'JKL'+E'F'GI'JKL'+GH'IKL'+GH'JKL'+GH'I'J'K'+
      H'I'JKLP'+DE'F'HIJKL'M
F 2 = D'EH'I'JKLMNP'+D'FH'I'JKLMNP'+DEH'I'JKLOP'+
      DFH'I'JKLOP'+DEHI'JKL'O+DFHI'JKL'O+AD'EMN+AD'FMN+
      ADEO+ADFO+D'I'JKL'Q+E'F'I'JKL'Q+DE'F'H'I'JKLMP'+
      H'I'JKQ+H'IKL'Q+ADE'F'M
F 3 = AB'P'R+AC'P'R+B'HJKL'P'R+C'HJKL'P'R+AD'R+AE'F'R+
      D'HJKL'R+E'F'HJKL'R+H'I'J'K'R+H'I'JKR+H'IKL'R+APS+
      HJKL'PS
F 4 = AB'P'T+AC'P'T+B'HJKL'P'T+C'HJKL'P'T+AD'T+AE'F'T+
      D'HJKL'T+E'F'HJKL'T+H'I'J'K'T+H'I'JKT+H'IKL'T+APU+
      HJKL'PU
F 5 = AB'P'V+AC'P'V+B'HJKL'P'V+C'HJKL'P'V+AD'V+AE'F'V+
      D'HJKL'V+E'F'HJKL'V+H'I'J'K'V+H'I'JKV+H'IKL'V+APW+
      HJKL'PW
F 6 = AB'P'X+AC'P'X+B'HJKL'P'X+C'HJKL'P'X+AD'X+AE'F'X+
      D'HJKL'X+E'F'HJKL'X+H'I'J'K'X+H'I'JKX+H'IKL'X+APY+
      HJKL'PY
F 7 = AB'P'Z+AC'P'Z+B'HJKL'P'Z+C'HJKL'P'Z+AD'Z+AE'F'Z+
      D'HJKL'Z+E'F'HJKL'Z+H'I'J'K'Z+H'I'JKZ+H'IKL'Z+APa+
      HJKL'Pa
F 8 = AB'P'b+AC'P'b+B'HJKL'P'b+C'HJKL'P'b+AD'b+AE'F'b+
      D'HJKL'b+E'F'HJKL'b+H'I'J'K'b+H'I'JKb+H'IKL'b+APc+
      HJKL'Pc
F 9 = AP'x+HJKL'P'x+APd+HJKL'Pd+H'I'J'K'x+H'I'JKx+H'IKL'x
F10 = AP'e+HJKL'P'e+APy+HJKL'Py+H'I'J'K'e+H'I'JKe+H'IKL'e
F11 = AP'g+HJKL'P'g+APf+HJKL'Pf+H'I'J'K'g+H'I'JKg+H'IKL'g
F12 = AP'i+HJKL'P'i+APh+HJKL'Ph+H'I'J'K'i+H'I'JKi+H'IKL'i
F13 = AP'k+HJKL'P'k+APj+HJKL'Pj+H'I'J'K'k+H'I'JKk+H'IKL'k
F14 = AP'm+HJKL'P'm+APl+HJKL'Pl+H'I'J'K'm+H'I'JKm+H'IKL'm
F15 = AP'o+HJKL'P'o+APn+HJKL'Pn+H'I'J'K'o+H'I'JKo+H'IKL'o
F16 = AP'q+HJKL'P'q+APp+HJKL'Pp+H'I'J'K'q+H'I'JKq+H'IKL'q
F17 = AP'w+HJKL'P'w+APv+HJKL'Pv+H'I'J'K'w+H'I'JKw+H'IKL'w
```

Note: the variables are represented by characters of the alphabet. A '#' in front of a character denotes another variable.

The optimisation results in a set of less complex expressions:

```
F 1 = I#J#Q+A#D'#F'+#F'#M+AG#D+G#N+I#O+J#O+GH'#S+#L
F 2 = A#J+#J#L+O#M+Q#B#T+Q#N
F 3 = R#K+S#P
F 4 = T#K+U#P
F 5 = V#K+W#P
F 6 = X#K+Y#P
F 7 = Z#K+a#P
F 8 = b#K+c#P
F 9 = x#H+d#P
F10 = e#H+y#P
F11 = g#H+f#P
F12 = i#H+h#P
F13 = k#H+j#P
F14 = m#H+l#P
F15 = o#H+n#P
F16 = q#H+p#P
F17 = w#H+v#P
```

and a set of subexpressions represented by the new variables:

```
#A = A+#Q
#B = I'J+IL'
#C = E+F
#D = D'+#C'
#E = K#B+#S
#F = B'+C'
#H = #U+P'#A
#J = DM#C'+D'MN#C+O#D'
#K = P'#A#F+#A#C'+D'#A+#U
#L = H'LP'#R
#M = I'#D'#Q
#N = L'#D#R
#O = GL'#T
#P = P#A
#Q = HJKL'
#R = I'JK
#S = I'J'K'
#T = H'K
#U = H'#E
```

The optimisation of this set took approximately a cpu time of 100 seconds.