MASTER

Searching for complex functions in boolean circuit descriptions using kernel matching

Wijdeven, J.H.P.

Award date:
1993

Eindhoven University of Technology
Department of Electrical Engineering
Design Automation Section (ES)

Searching for complex functions in boolean circuit descriptions using kernel matching

J.H.P. Wijdeven

Master Thesis

performed: September 1992 - August 1993
by order of Prof. Dr. Ing. J.A.G. Jess
supervised by Dr. Ir. J.F.M. Theeuwen

Abstract

In order to decrease the size of integrated circuits, all kinds of optimisation techniques have been developed to accomplish this. Also in the Design Automation Section of the Department of Electrical Engineering, at the Eindhoven University of Technology, people are working on logic optimisation. One part of logic synthesis where optimisation can be applied is technology mapping. Technology mapping is the mapping of a circuit description onto a cell library. That cell library usually exists of simple standard cells. But in some cases circuits contain complex structures that can be mapped more economical onto a more complex cell. The complex cell uses less space than an implementation with standard cells and in most cases the complex cell is faster.

Basically three methods could be used for finding those complex structures. They are boolean matching, graph covering and kernel matching. When searching for multi output structures this is very difficult to do with boolean matching while graph covering would be very suitable for it. But the disadvantage of graph covering is that the given circuit description has to be converted to a representation with only, for example, two input nands and inverters. This conversion process takes a lot of CPU time and after that the structures have still to be found.

A whole new approach is the use of kernels to find the complex (multi output) structures. Whit this method the functions of the structure to be found are identified by their kernels of level 0. These kernels are then matched against the kernels of the given circuit. The advantage of kernel matching is that the circuit description doesn't have to be preprocessed, only the kernels have to be computed.

The algorithm for kernel matching is integrated in the program log_decom. Log_decom is a logic optimisation program and is developed in the Design Automation Section of the Department of Electrical Engineering at the Eindhoven University of Technology. The advantage of integrating the

algorithm in log_decom is that log_decom already contains a lot of tools for handling kernels.

Although not all the results of the algorithm are mapped onto a cell library, to check the gain in size of the circuit layout, for some circuits there is already a slight gain in the amount of transistors. Especially for the results of the search for exclusive or's and exclusive nor's. Mapping some results on a library shows that in most cases there can be a gain in delay. In cases were the increase of transistors is small compared to the amount of found patterns, it is possible that there still can be a gain in area.

A disadvantage of the algorithm is that it becomes very slow for large circuits due to the large amount of kernels.

Table of Contents

Chapter 1

Introduction

The Design Automation Section of the Department of Electrical Engineering at the Eindhoven University of Technology is doing research on logic synthesis. One step in logic synthesis is called technology mapping. In this step a circuit description will be mapped onto a cell library. Until some years ago this had to be done manually, with the assistance of computers. Because circuits were increasing in both size and complexity, methods were designed so that this step can be done automatically.

The cell library usually exist of only standard cells like for example AND, OR and inverter cells. In some cases circuits contain complex structures that can be mapped on those standard cells, but could be mapped more economical onto a more complex cell. The advantage of such a complex cell is that it uses less space than an implementation with standard cells. Another advantage is that in most cases the complex cell is faster. Examples of these complex cells are (half/full) adders, exclusive or's and multiplexers.

This report describes how these complex structures can be found in boolean circuit descriptions with the use of kernels.

Chapter 2

Comparing different searching methods

This chapter shortly describes the comparison of three methods that can be used for searching for complex functions. These three methods are boolean matching, graph covering and kernel matching.

2.1 Boolean matching

Using boolean or functional matching, a library gate is a candidate for implementing a pattern of logic in the target network if both of them compute the same logic function ([1]).

In [2] a system is presented that uses boolean matching for technology mapping. The logic circuit to be mapped is partioned into subject graphs $\{F_1,\ldots,F_k\}$, that are decomposed into an interconnection of two-input gates.

Define a library L which contains only the complex function searched for. So for a complex function with m ($m \geq 1$) outputs L is defined as $\{L_1,\ldots,L_m\}$ where L_j is the function belonging to output j of the complex function.

A boolean match between two boolean functions F and L can be determined with recursive Shannon decomposition. These two functions are recursively cofactored generating two decomposition trees. F and L match if they have the same logic value for all the leaves of the recursion.

For a function with n input variables, in the worst case, all permutations ($n!$) and phase assignments (2^n) of the input variables are considered. Therefore, up to $(n! \cdot 2^n)^m$ different Shannon decompositions have to be considered when that function has m outputs. In case of a full adder, which has 3 input variables and 2 outputs, this comes to 2304 different possibilities.

2.2 Graph covering

Graph covering is a much used method for technology mapping. It can be divided into tree matching ([3], [4], [5]) and graph matching ([3]).

The basic step of graph covering is that a circuit description is converted into a graph. Each node in that graph then represents a standard cell (like AND or NOR). A common approach is the Nand2/Inverter representation.

In case of tree matching the next step is to partition the graph into a forest of trees. The library elements will be represented as trees also. Then the trees from the circuit are covered with trees from the library.

The main disadvantage of tree matching is that, for example, an XOR or a multiplexer can't be represented by a tree (see figure 1), but only by a graph. This also applies to most multiple output functions.

With graph matching the network graph is not partitioned. In this way multiple output functions can be treated as one searchpattern, instead of several searchpatterns for every output. Also the XOR and multiplexer can be handled easily, and this would make graph matching very interesting.

One disadvantage of graph matching is the number of different Nand2/inverter representations of large functions.

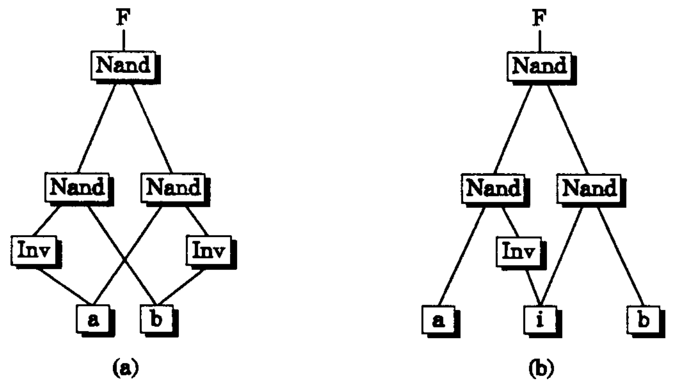

***Figure 1** : Nand2 / Inverter representation of (a) exclusive or and (b) multiplexer*

2.3 Kernel matching

Another approach to the problem is kernel matching (for the definition of a kernel see paragraph 3.1), where each function (L_i) to be found is represented by its kernels of level 0.

Using kernel matching, all the kernels of the boolean circuit description (F) are determined. Also the kernels of L are determined. The next step is to find for every kernel of L an equivalent kernel from F. Then one of the kernels of L is substituted and the equivalent kernel is substituted in F. Again the kernels of L and F are determined, equivalents are searched and one kernel is substituted. This will be repeated until L has no more kernels. This method is illustrated in example 2.1, in which is searched for a full adder.

Example 2.1

The boolean description of a full adder is as follows:

co : x1.x2 + x1.x3 + x2.x3;
so : x1.x2.x3 + x1.x2'.x3' + x1'.x2.x3' + x1'.x2'.x3;

Where *co* is the carry output and *so* is the sum output of the adder.

Now suppose that there is a simple boolean circuit description F as follows:

$f1$: $a.b + a.c + b.c;$
$f2$: $a.b.c + a.b'.c' + a'.b.c' + a'.b'.c;$

The kernels of co are:

$x2 + x3$	(divide by $x1$)
$x1 + x3$	(divide by $x2$)
$x1 + x2$	(divide by $x3$)

The kernels of $f1$ are:

$b + c$	(divide by a)
$a + c$	(divide by b)
$a + b$	(divide by c)

By matching b to $x2$, c to $x3$ and a to $x1$ it is easy to see that for every kernel of co there is an equivalent kernel in $f1$. Now substitute $x2+x3$ by k in co and $b+c$ by m in $f1$. The results are:

co : $x1.k + x2.x3;$
k : $x2 + x3;$

$f1$: $a.m + b.c;$
m : $b + c;$

The kernels of co and $f1$ are co and $f1$ themselves. Substituting these kernels by l and n results in:

co : $l;$
l : $x1.k + x2.x3;$
k : $x2 + x3;$

$f1$: $n;$
n : $a.m + b.c;$
m : $b + c;$

Finally k and m can be substituted back.

co : $l;$
l : $x1.x2 + x1.x3 + x2.x3;$

$f1$: n;
n : $a.b + a.c + b.c$;

The same is done with *so* and *f2*. In this way the whole full adder is extracted from the circuit. □

The approach in example 2.1 may look a bit superfluous, because in just a glance can be seen that *f1* contains something that matches with *co*, and which could be extracted in one step. But what if *f1* would look like this:

$f1$: $d.e.a.b + a.e.c.d + e.f.g + e.b.c.d$;

Now it is a bit more difficult to see that *f1* 'contains' *co*, but the level 0 kernels of *f1* are still the same as before.

The advantage of kernel matching is that the boolean description doesn't have to be partioned first and/or converted into a two input gate representation. Only the kernels have to be computed. Using the kernels of a function divides the problem into several smaller subproblems, which can be handled easier.

That a set of kernels can belong to different functions is of small concern. This problem is eliminated by the substitution of the kernels (see example 2.2).

Example 2.2

Given the searchpattern *p*:

p : $a.c + b.c + b.d$;

with the kernels:

$a + b$ (divide by c)
$c + d$ (divide by b)

And given a function *f1* with the same kernels as *p*:

$f1$: $a.e + b.e + c.f + d.f$;

Then substitute the kernel $a+b$ by k in both p and $f1$, which results in the following:

$p \quad : k.c + b.d;$
$f1 \quad : k.e + c.f + d.f;$
$k \quad : a + b;$

The only kernel of p doesn't have an equivalent kernel in $f1$ so the process can be stopped here and $f1$ is not the function that was searched for. □

Another problem of kernel matching is that functions that don't have kernels can't be handled. For example the half adder where the carry output is described as $a.b$ where a and b are the inputs. How this could be solved is described in chapter 6.

Chapter 3

The matching problem

This chapter gives a solution to the problem of searching for functions with the use of kernels.

3.1 Basic definitions

The basic definitions as they are presented in [6].

- A variable is a symbol representing a coordinate in the boolean space. A symbol is a string of characters not starting with a digit, containing no special characters like "," "." ";" and the like.
- A variable can have two values: "1" or "0". The complement of a variable is denoted by *<variable>'*. For instance the complement of the variable a is a'.
- Variables and their complements are called literals.
- A cube is the product of a set of literals, such that it contains either a variable or it's complement. For instance "*a.b.c*" is a cube. The "." is denoted by "and" being equivalent to the boolean "and" operator.
- A boolean expression can be represented by a sum of cubes.
  For instance *f1 : a.b.c+d.e.f;*

Where "+" is denoted by "or" being equivalent to the boolean "or" operator. A ":", denoted as "is defined as", indicates the start of a boolean expression. A ";" indicates the end of a boolean expression.

- Weak division of a boolean expression f by a boolean expression g is defined as the largest set of cubes common to the result of dividing the numerator f by each cube of the denominator g.
  For instance:
  $(a.b+a.c+c.d)\,/\,a = b+c$
  $(a.b+a.c+c.d)\,/\,(b+c) = a$

- If $(f/g)\cdot g=f$ holds, g divided f evenly. For instance: a divides $a.b+a.c$ evenly.

- An expression f is cube free if and only if 1 divides f evenly. For instance $a.b+c.d$ is cube free but $a.b+a.c$ is not cube free.

- A primary divisor of a boolean expression f is defined as:

  $\{\ f/c \mid c = cube\ \}$

  In words: a primary divisor is the result of the weak division of f by a cube.

- A kernel is a cube free primary divisor.

- A kernel of level zero is a kernel which contains no other kernel.
  For instance: $f : a.b+a.c+b.c;$
  The kernels of level 0 of f are: $b+c$ (divide by a)
  $a+b$ (divide by c)
  $a+c$ (divide by b)
  The expression itself $a.b+a.c+b.c$ is also a kernel but not of level 0.

- A kernel of level n is a kernel which contains at least one level n-1 kernel, but no kernels of level n (other than itself) or greater.

3.2 Finding equivalent kernels

A given function g might occur in a function f, if for every kernel of g there is an equivalent kernel in f. An equivalent kernel is a kernel that matches the given kernel. To find a kernel equivalent to a kernel k, first a lookalike (see definition 3.1) of k is searched for.

Definition 3.1 : a lookalike of a kernel k is a kernel that has the same amount of cubes as k has. For each cube in k there exists a cube in the lookalike with the same amount of literals and the same amount of negated literals. □

Once this lookalike is found and the amount of literals is the same as the amount of literals in k then each literal of k will be bound to a literal of the lookalike. If a literal of k will be bound to two different literals, then the lookalike is not equivalent to k (see example 3.1). This binding is done per cube.

Example 3.1

Given two kernels *x1'.x2'+x1.x2* and *x1+x3*. A lookalike of the first kernel could be *a'.b'+a.b*.

First cube of kernel:
x1 is bound to *a*
x2 is bound to *b*

Second cube of kernel:
x1 is bound to *a* and that matches the previous binding of *x1*
x2 is bound to *b* and that matches the previous binding of *x2*

A lookalike of *x1+x3* could be *b+c*. Now *x1* must be bound to *b* or *c* but neither of these possibilities matches with the previous binding of *x1* so *b+c* is a lookalike but not an equivalent kernel. □

By binding the first literal encountered in given kernel to the first free literal in the lookalike, all permutations of that kernel have to be checked to find all possible matches. If for example the equivalent kernel to *x1+x2+x3.x4* is wanted, then there has to be searched with *x1+x2+x3.x4*, *x1+x2+x4.x3*, *x2+x1+x3.x4* and *x2+x1+x4.x3*. These permutations all have to be computed, and when a function has more kernels, then there has to be kept track of which permutation of each kernel is already used. A much simpler way is to apply a fixed binding to a group of selected literals from the searchpattern before the search is started. To each of these literals a literal from the given network is bound (fixed binding). Then the search is repeated with all possible combinations of binding these literals to those of the network. How the literals are selected is defined in definition 3.2.

Definition 3.2 : the criterions for selecting literals for fixed binding.

1) If a kernel has n cubes that contain only one literal, then mark n-1 of these literals as *fixed*. The remaining literal will be bound automatically during the searching procedure and is marked as such.
2) If a kernel has a cube that contains m literals, then mark m-1 of these literals as *fixed*. The remaining literal will be bound automatically during the searching procedure and is marked as such.
3) The literals for fixed binding are selected in such way that the one remaining literal is not already marked as *fixed* or *automatic*. And if a kernel does contain marked literals then less than n-1 or m-1 are marked as *fixed*. □

Fixed binding could also be applied on every literal of the pattern, but in this way less literals are needed (thus saving time) to find the same matches.

The selection process will be illustrated in example 3.2.

Example 3.2

Given two kernels (in order of searching):

$a + b + c.d.e$
$e + f + c.g$

First kernel : two cubes of one literal, mark a as *fixed*, mark b as *automatic*.
one cube of three literals, mark c and d as *fixed*, mark e as *automatic*.
Second kernel : two cubes of one literal, e is already marked so mark f as *automatic*.
one cube of two literals, c is already marked so mark g as *automatic*.

So a, c and d are used for fixed binding. □

Applying the rules of definition 3.2 to a full adder, multiplexer or exclusive or results in only one literal for fixed binding.

3.3 Extending the searching algorithm

Sofar only in *simple* examples of networks the kernels of the searchpattern can be found. With *simple* is meant something like the following:

searchpattern	:	*co : a.b + a.c + b.c;*
function of network	:	*f1 : d.a.b + d.a.c + d.b.c + d.e;*

Each cube of the pattern to be found can be multiplied by a cube that is the **same** for every cube of the pattern, but that is essential because only then the pattern is present and can be extracted. The extra cubes (besides the ones of the pattern, like *d.e*) do not contain any literals of the pattern. When they do then the kernels of the pattern and the kernels of the function do not match (see example 3.3).

Example 3.3

searchpattern	:	*co : a.b + a.c + b.c;*	
kernels	:	$b + c$	(divide by a)
		$a + c$	(divide by b)
		$a + b$	(divide by c)
function of network	:	*f1 : a.b + a.c + b.c + a.d;*	
kernels	:	$b + c + d$	(divide by a)
		$a + c$	(divide by b)
		$a + b$	(divide by c)

Kernel b+c+d doesn't match with kernel b+c but now it contains b+c. □

To solve the problem described in example 3.3 the kernel b+c+d can be split in kernels that look like the wanted kernel b+c. This would result in the kernels b+c, b+d and c+d. To recognize this situation, definition 3.1 has to be adapted slightly.

Definition 3.3 : a lookalike of a kernel k is a kernel that has an amount of cubes that is equal to or greater than the amount of cubes of k. For each cube in k there exists a cube in the lookalike with the same amount of literals and the same amount of negated literals. □

With that new definition, $b+c+d$ is a lookalike of $b+c$ and can be split into smaller kernels which from now on will be referred to as *splits*. Then one of the splits is chosen according to the fixed or previous binding of the literals. The remaining splits may not be used as lookalikes for other kernels. The original kernel (before splitting) can be used for splitting again. But only under the condition that the new chosen split doesn't have any cubes in common with earlier used splits of that kernel. This means that after a kernel is split it can't be used in one piece as a lookalike but only in parts (see also example 3.4).

Example 3.4

Given three kernels to find:

$a + b$
$d'.e + d.e'$
$d'.e + c$

A lookalike of these three kernels is:

$a + b + c + d'.e + d.e'$

From the lookalike both $a+b$ and $d'.e+d.e'$ can be used. But after one of these two is chosen, $d'.e+c$ can't be used. □

Another problem is the following:

The searchpattern L contains an exclusive or which has one kernel of level 0 namely $a.b'+a'.b$. Given a function from a circuit which contains an exclusive or:

$f1 : a.b' + a'.b + a.c;$

Now $f1$ also has one kernel of level 0 but that kernel is $b'+c$. This means that the Xor wouldn't be found. This is easily solved by computing not only the kernels of level 0 of the boolean network description, but also the kernels of higher levels. In this way *f1* is a kernel of level 1 and can be split.

In practice the highest kernel level is about 6 or 7.

To speed up the search for a function that has more than one kernel of level 0, the divisor(s) of the kernel can be used. How these divisors are used is given in given in definition 3.4.

<u>*Definition 3.4*</u> : using divisors.

1) If a kernel has no divisors then it doesn't matter what divisors the lookalike has.
2) If a kernel does have divisors then the lookalike must have at least the same amount of divisors.
3) *c1* is a cube of literals that are divisors of the current kernel of the function searched for but not of the other, already found, kernels of that function.

   c2 is a cube of literals that are divisors of the lookalike but not of the other, already found, equivalent kernels.

   If the amount of literals in *c1* is the same as the amount of literals in *c2*, then the lookalike is a candidate for an equivalent kernel. □

Using the divisors in this way results in a simple checking algorithm without the need for binding the divisors. Because when a lookalike has more divisors than the given kernel then these divisors could be bound in different ways.

In example 3.5 the use of divisors is illustrated.

<u>Example 3.5</u>

Given the carry output of a full adder as the function to search for :

co : x1.x2 + x1.x3 + x2.x3;

Kernels :	*x2 + x3*	(divide by *x1*)
	x1 + x3	(divide by *x2*)
	x1 + x2	(divide by *x3*)

Given a function *f1* :

f2 : $a.b.d + a.c.d + b.c.d;$

Kernels :	$b + c$	(divide by $a.d$)
	$a + c$	(divide by $b.d$)
	$a + b$	(divide by $c.d$)

Suppose b+c is already found as an equivalent kernel of $x2$+$x3$ and a+c is considered as a lookalike of $x1$+$x3$. Then $c1$ (as mentioned in definition 3.4) contains $x2$ and $c2$ contains b (the only literal that is in $b.d$ but not in $a.d$). The amount of literals in $c1$ is equal to the amount of literals in $c2$ so a+c is a lookalike. □

Although this way of using the divisors does not guarantee that the right kernels will be found, the increase of speed can be enormous. For example for the benchmark Rd53 (see chapter 5) which contains one full adder. Without the use of divisors it took 7.5 CPU minutes to find it. With the divisors it took only 11 CPU seconds.

Chapter 4

Integrating the searching algorithm in log_decom

This chapter describes the program log_decom and the changes that are made to it.

4.1 What is log_decom

Log_decom is a program that can optimise a set of logic expressions. This optimisation will lead to a so called multi level implementation of the specified combinational logic.

The basic operation performed by log_decom is searching for common subexpressions in the set of expressions. If a certain subexpression appears a number of times then it could be beneficial to realise that subexpression only once, and to use the result on the different places in the expressions. This will lead to three main effects:

1. An extra logic level will be added to the circuit.

2. The number of transistors in the final circuit will be smaller, which will result in a smaller used area of the final layout.

3. The expressions will become less complex. This will increase the probability that the expressions can be mapped straight onto library cells.

The four basic operations log_decom uses for the optimisation process are:

simplification : this is the first step in the process and must be executed before any other operation is executed. Simplification is applied on every expression separately and writes every expression as a minimal sum of primary cubes. Minimal means here that it's impossible to leave out a cube. Generally the number of literals in the set of expressions will decrease. But in some cases the number of literals can even increase.

Distillation : During the distillation process there is searched for equal kernels. This process consist of three steps:

1. Of every expression all kernels of level 0 are computed.
2. All kernels are compared with each other and a list of equal kernels is made.
3. From this list of equal kernels a number of times the most favourable kernel is determined and is realised as a separate expression. Everywhere this kernel appears the new created (internal) variable is substituted. The number of times this process is repeated depends on the user who can adjust this. The next consideration is important here:

   By substituting a kernel it can happen that a number of other kernels will cease to exist. Therefor the list of (equal) kernels won't be 100% correct any more and a next equal kernel could be a wrong one. Because computing all kernels of an expression costs a lot of CPU time it isn't feasible to determine all kernels again after a substitution.

Condensation : The condensation is similar to the distillation process. However now is not searched for equal kernels but for equal cubes. Like the distillation process it consist of a number of steps:

1. All cubes of all expressions will be compared to each other and a list of equal cubes is made.

2. From this list of equal cubes a number of times the most favourable cube is determined and is realised as a separate expression. Everywhere this cube appears the new created (internal) variable is substituted. The number of times this process is repeated depends on the user who can adjust this. Therefor the same consideration as with distillation applies for condensation.

Collapsing : It can occur that during the distillation and condensation process substitutions have found place that aren't very meaningful later. Log_decom can collapse in many different ways certain variables.

The advantage of integrating the searching algorithm in log_decom is that log_decom already contains a lot of tools for handling literals, cubes and kernels.

4.2 Adapting log_decom

(The original data structure of log_decom is completely left intact to ensure compatibility with newer versions.)

Log_decom is now able to work with two files simultaneously. This was done by doubling the global variables of log_decom. With the command *'swap'* the values of these variables are exchanged which results in swapping between the two files. The first file is loaded as before, namely on the prompt. The second file, or searchpattern, is read from within log_decom with the command *'read* <file>'. On this file all operations of log_decom can be applied and therefore it can be used for other purposes also. With the command *'fp'* (find pattern) the search for the pattern is started.

4.2.1 Data structures

The data structures added to log_decom are described below.

```
struct _bind_record      *bind_ptr;
struct _bind_record      {
        literal    real_literal;   /* literal in searchpattern */
        literal    bound_literal;  /* literal in network file */
        bool       bound;
        bool       temp;
        int        bind_stage;
        int        fixed;
        bool       automatic;
        bind_ptr   next;
                         } bind_record;
```

This structure is used to keep track of each literal in the searchpattern. The record corresponds with one literal. The variable *temp* is set to TRUE if the corresponding literal of a kernel from the pattern is bound successfully. If all literals of that kernel are bound with success then *bound* is set to TRUE for those literals. If not all literals can be bound successfully (e.g. wrong lookalike kernel) then the literals with the *temp* variable set are cleared. *Bind_stage* is a number that corresponds with the number of the kernel (in searching order) in which the literal was bound for the first time. The variables *fixed* and *automatic* have the same function as in definition 3.2.

For every kernel of the searchpattern exists a record which is defined below.

```
struct _kern_list_record *kl_ptr;
struct _kern_list_record {
        kernel_ptr kernel;
        int        kernel_nr;
        int        expr_nr;
        kernel_ptr lookalike;
        bool       split;
        kernel_ptr split_kernel;
        int        nr_of_splits;
        expr_ptr   lookalike_owner;
        kl_ptr     next;
        kl_ptr     prev;
                       } kern_list_record;
```

The variable *kernel* corresponds with a kernel from the searchpattern. The equivalent kernel in the network is pointed to by *lookalike*. If a kernel is split then *split_kernel* points to that kernel.

Chapter 5

Results

The algorithm is tested on some benchmark circuits which are given in table 1. After each circuit the amount of transistors is given according to log_decom. These amounts are determined before any optimisation is done.

Table 1 : Original sizes of benchmarks given in amount of transistors

Name	# trans.	Name	# trans.	Name	# trans.
5xp1	163	F2	32	Radd	113
9sym	272	Misex1	88	Rd53	74
Alu4	1881	Misex3x	908	Rd73	229
Bw	292	Primes8	505	Rd84	419
Clip	279	Primes9	983	Sao2	198
Duke2	801	Primes10	2100	Xor5	28

Only three patterns were used for testing, namely the exclusive or $(a.b'+a'.b)$, the exclusive nor $(a.b+a'.b')$ and the multiplexer $(a.b+a'.c)$. This was done because these structures appeared the most in the given circuits. The results of the xor and xnor are shown in table 3 and the results of the multiplexer are given in table 2. In table 2 only the examples containing multiplexers are given. The whole program runs on a HP 9000/S735, a 108 MIPS machine.

The number of, for example, xor's represents the amount of different xor's. Each of these xor's can occur one or more times in the circuit.

(1) and (2) have the following meaning (in tables 1 to 4):

(1) After simplification, distillation, condensation and collapsing common kernels and cubes that occur only once (applied on original circuit).

(2) After simplification, distillation and condensation (applied on circuit after the search).

Table 2 : Results of searching for Multiplexers

Name	# (1) trans.	Multiplexer # (2) trans.	CPU (min)	# Mux
Alu4	1580	1708	4:17	8
Bw	218	229	0:00	4
Duke2	483	**471**	0:02	1
Misex1	72	79	0:00	1
Misex3c	729	784	0:15	14
Primes8	380	449	0:01	14
Primes9	888	924	0:26	22
Primes10	1882	2097	2:46	42

In the table 4 the results of successive searches for the exclusive or and the exclusive nor are given. First a search for one of these is done and on the resulting file a search for the other one is performed. After that the file is optimised using simplification, distillation and condensation. The CPU time needed is in the worst case the sum of the CPU times given in table 3.

Only one benchmark circuit contained a full adder. This was Rd53 and the CPU time needed to find it was 11 seconds. The amount of transistors after the search and optimisation was 76.

Table 3 : Results of searching for Xor's and Xnor's

		Exclusive Or			Exclusive Nor		
Name	# (1) trans.	# (2) trans.	CPU (min)	# Xor	# (2) trans.	CPU (min)	# Xnor
5xp1	114	155	0:00	5	138	0:00	4
9sym	198	301	0:38	25	-	-	-
Alu4	1580	1584	5:17	3	**1541**	5:27	4
Bw	218	221	0:00	8	229	0:00	4
Clip	169	179	0:03	3	205	0:03	5
Duke2	483	484	0:05	4	**448**	0:04	3
F2	24	28	0:00	2	-	-	-
Misex1	72	84	0:00	2	75	0:00	1
Misex3c	729	754	0:17	17	**708**	0:18	6
Primes8	380	423	0:09	12	427	0:06	12
Primes9	888	**844**	0:57	19	**870**	0:38	17
Primes10	1882	1933	5:26	29	2001	6:29	33
Radd	64	66	0:01	4	71	0:01	3
Rd53	61	75	0:00	5	**57**	0:00	2
Rd73	181	**176**	0:14	10	**164**	0:12	3
Rd84	274	**184**	1:26	5	**242**	1:22	4
Sao2	161	215	0:04	10	182	0:02	6
Xor5	28	29	0:00	2	29	0:00	2

Table 4 : Successive search for Xor and Xnor

		Xor/Xnor		Xnor/Xor	
Name	# (1) trans.	# (2) trans.	# xor/xnor	# (2) trans.	# xnor/xor
5xp1	114	174	5/4	166	4/4
Alu4	1580	**1506**	3/3	**1525**	4/4
Bw	218	218	8/1	**217**	4/5
Clip	169	191	3/5	206	5/3
Duke2	483	**467**	4/2	496	3/4
Misex1	72	83	2/1	83	2/1
Misex3c	729	772	17/5	743	6/16
Primes8	380	434	12/9	439	12/11
Primes9	888	899	19/17	**860**	17/14
Radd	64	71	4/3	74	3/4
Rd53	61	72	5/2	72	2/5
Rd73	181	**177**	10/3	**173**	3/10
Rd84	274	**198**	5/4	**208**	4/7
Sao2	161	217	10/2	206	6/4
Xor5	28	**26**	2/2	**26**	2/2

In the previous tables only the gain in transistors is considered. But as can be seen only in a few cases the results are slightly better. In some cases the results also show that the algorithm is very slow. This because of the fact that after a successful substitution all of the kernels of the function, in which the substitution found place, are computed again and of the way the literals are used for fixed binding. More literals in a circuit means more computing time.

In table 5 for a some examples the circuit is mapped on a library (before and after the search for exclusive or's).

Table 5 : Results after technology mapping

	without searching			after searching for xor's			
Name	# trans.	delay (ns)	width (λ)	# trans.	# Xor	delay (ns)	width (λ)
5xp1	114	13.69	2032.8	155	5	13.44	2675.4
9sym	198	20.36	3775.8	301	25	14.44	4431.0
Alu4	1580	40.80	33747.0	1584	3	35.95	34276.2
Bw	218	15.43	3074.4	221	8	12.13	3061.8
Clip	169	13.26	2835.0	179	3	12.30	3330.6
Duke2	483	17.23	6879.6	484	4	20.31	7224.0
F2	24	5.52	554.4	28	2	4.08	512.4
Misex1	72	7.17	886.2	84	2	8.04	1507.8
Misex3c	729	23.85	12364.8	754	17	21.04	11659.2
Primes8	380	19.84	6442.8	423	12	18.13	7639.8
Primes9	888	32.17	17623.2	844	19	26.37	14422.8
Radd	64	12.07	1323.0	66	4	11.27	1453.2
Rd53	61	11.39	1764.0	75	5	7.76	1226.4
Rd73	181	22.97	5476.8	176	10	15.24	3591.0
Rd84	274	26.25	10332.0	184	5	18.79	5300.4
Sao2	161	20.80	3305.4	215	10	17.88	3809.4
Xor5	28	11.66	1024.8	29	2	8.57	655.2

In the used library all the cells have the same height, so the area is measured by the width of the all the cells when they would be ordered after each other in one line ($\lambda = 0.6\ \mu m$).

When looking at table 5 it shows that only in two cases (*Duke2* and *Misex1*) there is no gain in delay. And in five examples (*Bw*, *F2*, *Misex3c*, *Rd53* and *Xor5*) there is even a gain in area although the amount of transistors is increased after the search. But those differences in transistors were small with regard to the amount of found xor's. In cases where the difference in transistors is large and the amount of found patterns is small, it shows that this difference can't be made up with a gain in area. But when there is already a gain in transistors then there is also a gain in area.

Especially for *Rd53*, *Rd73* and *Rd84* the overall results are impressive with 30 to 49 percent gain in area and 28 to 34 percent gain in delay. This can be explained by the large number of times the different xor's appear in these circuits, compared to the amount of transistors. In *Rd53* they appear 11 times, in *Rd73* 45 times and in *Rd84* even 125 times.

Looking at the overall results, searching for complex functions does show some perspectives to continue with it. Most importantly when it concerns the delay and in some cases the area of the circuit.

Chapter 6

Future work

As mentioned before in paragraph 2.3, functions that don't have kernels are a problem. This problem can be solved because of the flexible data structure of log_decom, the program in which the whole algorithm is integrated (see chapter 4). If a function has no kernels then the whole function is defined as a kernel and log_decom will treat it as a kernel. One disadvantage is that not only the splitting of kernels has to be considered but also the splitting of cubes. This would slow down the algorithm even more.

In paragraph 3.2 is described how and why literals for fixed binding are chosen. When the circuit in which to find the pattern contains a lot of different literals, this process of fixed binding can slow down the algorithm enormously. In that case it would be better to use its more complex counterpart, namely searching with every permutation of a kernel.

Chapter 7

Conclusions

Although not all the results produced were mapped onto a cell library, to check the size of the layout, for some circuits there is already a slight gain in the amount of transistors. Especially for the results of the search for exclusive or's and exclusive nor's. And with the fact that an exclusive or is smaller than when this function is made with standard gates (one or, two and's and two inverters) it may be concluded that a gain in area is achieved. Mapping some results onto a cell library shows that in almost all cases there is a gain in delay even when the amount of transistors and area increases. This because the delay of an exclusive or is smaller than the representation with standard cells. But this gain can only be achieved if such an exclusive or occurs on the longest path of the circuit.

The algorithm becomes very slow on large circuits because they have a large amount of kernels and every kernel has to be checked. Also the algorithm will become considerable slower when the amount of literals in the circuit increases. This because of the way how these literals are used to find a kernel.

References

[1] Kung D.S. and R.F. Damiano, T.A. Nix, D.J. Geiger
BDDMAP: A TECHNOLOGY MAPPER BASED ON A NEW COVERING ALGORITHM.
Proceedings of the 29th ACM/IEEE Design Automation Conference,
Anaheim CA, 8-12 june 1992.
Los Alamitos CA: IEEE, 1992.
P. 484-487.

[2] Mailhot F. and G. De Micheli
TECHNOLOGY MAPPING USING BOOLEAN MATCHING AND DON'T CARE SETS.
Proceedings of the European Design Automation Conference,
Hamburg, 7-10 september 1990.
Brussels: IEEE, 1990.
P. 212-216.

[3] Detjens E. and G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, A.Wang
TECHNOLOGY MAPPING IN MIS.
Proceedings of the ICCAD,
Santa Clara, 9-12 november 1987.
New York: IEEE, 1987.
P.116-119.

[4] Keutzer K.
DAGON: TECHNOLOGY BINDING AND LOCAL OPTIMIZATION BY DAG MATCHING.
Proceedings of the 24th ACM/IEEE Design Automation Conference,
Miami Beach, 28 june - 1 july 1987.
New York: IEEE, 1987.
P. 341-347.

[5] Crastes M. and K. Sakouti, G. Saucier
A TECHNOLOGY MAPPING METHOD BASED ON PERFECT AND SEMI-PERFECT MATCHINGS.
Proceedings of the 28th Design Automation Conference,
San Francisco, 17-21 june 1991.
New York: IEEE, 1991.
P.93-98.

[6] Janssen, G.L.J.M. and L. Stok, G.G. de Jong, M.R.C.M. Berkelaar, J.T.J. van Eijndhoven, J.F.M. Theeuwen
C.A.D.-Systemen.
1st edition. Eindhoven: Eindhoven University of Technology.
nr. 5699.

Appendix

This section is a short manual on how to use log_decom when searching for patterns.

First there have to be two files, *file.log* (in which is searched) and *pattern.log* (containing the searchpattern). Then log_decom is started as follows :

decom <file>

Next from within log_decom the searchpattern is read with the command :

load <pattern>

With the command *'swap'* can be switched between the file and the pattern. The search for the pattern is started with *'fp'* (find pattern). Finally there is a command named *'pk'* to print the kernels of all functions of the current file or pattern.

The first output of log_decom is the number of times a copy of the pattern is found. The second output is the amount of different copies of the pattern (one copy of a pattern can occur one or more times).

On both the file and the pattern all the operations of log_decom can be applied.