Technology mapping from boolean expressions to standard cells

tu/e
Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

Technology Mapping from Boolean Expressions to Standard Cells

by
M.R.C.M. Berkelaar
and
J.F.M. Theeuwen

EUT Report 87-E-174
ISBN 90-6144-174-9
June 1987

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven The Netherlands

ISSN 0167- 9708 Coden: TEUEDE

TECHNOLOGY MAPPING
FROM BOOLEAN EXPRESSIONS TO STANDARD CELLS

by

M.R.C.M. Berkelaar

and

J.F.M. Theeuwen

EUT Report 87-E-174

ISBN 90-6144-174-9

Eindhoven

June 1987

COOPERATIVE DEVELOPMENT OF AN INTEGRATED, HIERARCHICAL AND MULTIVIEW VLSI-DESIGN SYSTEM WITH DISTRIBUTED MANAGEMENT ON WORKSTATIONS.

(Multiview VLSI-design System ICD)

code: 991

DELIVERABLE

Report on activity 5.2.D: Attempt to build optimisation routines into the logic editor such that the editor generates logic structures geared to the CMOS families found in activity 5.2.C.

Abstract:
In this report an approach is presented towards the problem of making a standard cells IC implementation from a previously optimised and decomposed set of boolean functions. The algorithms were implemented in Commonlisp.

The library of standard cells should contain *and-or-invert*-gates up to a certain maximum size. Libraries of standard cells for NMOS or CMOS technologies usually do. This maximum size is a parameter to the algorithms, so the program can be used for different libraries of standard cells.

The total problem is split up into two subproblems which are treated separately. First, functions which do not fit into standard gates directly, are split up in pieces small enough to fit. Second, the inverters, which are necessary between the gates in a technology with *and-or-invert*-gates, are removed if possible. Special attention is paid to the inverters on the critical path to speed up the circuit.

Both problems are attacked in a partly heuristical and partly analytical way. The heuristical part limits the problem to a size small enough for an analytical approach, which then solves the limited problem completely optimal.

Trial runs with international benchmark examples have shown that the approach shows good results

deliverable code: WP 5, *task:* 5.2, *activity:* 5.2.D.

date: 01 - 06 - 1987

partner: Eindhoven University of Technology

author: M.R.C.M. Berkelaar, J.F.M. Theeuwen.

This report was accepted as a M.Sc. Thesis of M.R.C.M. Berkelaar by Prof.Dr.-Ing. J.A.G. Jess, Automatic System Design Group, Faculty of Electrical Engineering, Eindhoven University of Technology. The work was performed in the period from July 1987 to May 1987 and was supervised by Dr.ir. J.F.M. Theeuwen.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Berkelaar, M.R.C.M.

Technology mapping from Boolean expressions to standard cells / by M.R.C.M. Berkelaar and J.F.M. Theeuwen. - Eindhoven: University of Technology, Faculty of Electrical Engineering. - Fig. - (EUT report, ISSN 0167-9708; 87-E-174)
Met lit. opg., reg.
ISBN 90-6144-174-9
SISO 663.42 UDC 621.382:681.3.06 NUGI 832
Trefw.: elektronische schakelingen; computer aided design.

CONTENTS

LIST OF FIGURES

1. ABSTRACT

In this report an approach is presented towards the problem of making a standard cells IC implementation from a previously optimised and decomposed set of boolean functions. The algorithms were implemented in Commonlisp.

The library of standard cells should contain *and-or-invert*-gates up to a certain maximum size. Libraries of standard cells for NMOS or CMOS technologies usually do. This maximum size is a parameter to the algorithms, so the program can be used for different libraries of standard cells.

The total problem is split up into two subproblems which are treated separately. First, functions which do not fit into standard gates directly, are split up in pieces small enough to fit. Second, the inverters, which are necessary between the gates in a technology with *and-or-invert*-gates, are removed if possible. Special attention is paid to the inverters on the critical path to speed up the circuit.

Both problems are attacked in a partly heuristical and partly analytical way. The heuristical part limits the problem to a size small enough for an analytical approach, which then solves the limited problem completely optimal.

Trial runs with international benchmark examples have shown that the approach shows good results.

2. INTRODUCTION

2.1 A Short Overview

Nowadays big efforts are made to generate the layout of (VLSI) integrated circuits automatically, or at least with a lot of computer support. In an ideal situation, the designer would feed some kind of functional description of the desired integrated circuit into the computer, push a button, and after a certain time the complete layout description, ready to go to the mask fabrication, would be produced. Currently, only some stages of the whole process are completely automatic. The designer still has to make a lot of choices at various points of this process.

This report treats some of the problems which arise in the this field of automatic generation of layout for integrated circuits. Specifically, we will describe some steps in the process from algebraic description of a piece of combinational logic to a so-called *standard cell implementation*. This process consists roughly of two steps, the first being *logical optimisation*, and the second *technology mapping*. This last step is the subject of this study.

2.2 Some Definitions

Throughout this report a number of terms will be used with a specific meaning in this context. The logical inversion of an expression will be denoted with a bar over the expression (i.e. $\overline{\text{expression}}$). The meaning of the terms "literal", "operator", "expression" and "function" can be derived from the following description:

```
<letter>      ::= "a" | "b" | ... "z" | "A" | "B" | ... "Z"
<digit>       ::= "0" | "1" | ... "9"
<subscript>   ::= {<digit>}+
<special>     ::= "-"
<character>   ::= <letter> | <digit> | <special>
<variable>    ::= <letter> {<character>}* [<subscript>]
<literal>     ::= <variable> | <variable>
<sum>         ::= <literal> | {<product> "+"}+ <product>
<product>     ::= <literal> | {"(" <sum> ")" "."}+ "(" <sum> ")"
<expression>  ::= <product> | <sum>
<function>    ::= <literal> "=" <expression>
```

Furthermore the terms "cube" and "sum of cubes" are important and can

be defined like this:

<cube> ::= <literal> | <literal> "." <cube>

<sum of cubes> ::= <cube> | <cube> "+" <sum of cubes>

The last notion we have to define here is "subexpression". Only if an expression is not a single literal, it has so-called subexpressions. There are two possibilities:

1. The expression is a sum: the products which build the sum are the subexpressions.

2. The expression is a product: the sums which build the product are the subexpressions.

Boolean functions will be denoted according to the above syntax in the report, except in the sections called "The Lisp Implementation", which will describe the Lisp implementation of a specific part of the program. In those sections the representation will be as described in chapter 3, a prefix representation. This prefix representation is the exact image of the way the boolean functions are stored in the data-structure used by the program. The reader who is interested only in the ideas and algorithms and not in the Lisp implementation can skip these sections without missing essential information, and is not required to get acquainted with the prefix notation.

3. THE IMPLEMENTATION IN LISP

3.1 Introduction

First it has to be noticed that whenever we use the word Lisp in this report, we actually mean the Lisp dialect Commonlisp. For a complete description of this dialect lit. [4] can be referenced.

There are several reasons Lisp has been chosen to implement the algorithms for the technology mapping operation. The most important one is the possibility to make use of the Lisp standard data structure, the list. As will be shown in the next section, it is very easy to store boolean expressions, functions, and whole sets of functions in such a list. And, what is almost equally important, the implementation of operations on the boolean functions can be written as Lisp operations on lists in a very simple and clear-to-everybody way. No complicated types and structures have to be declared, no obscure pointer statements written, and no memory management done by the program; as would have been necessary when using pascal or C. All of this is hidden on the Lisp program level. And last but not least there is the beauty and simplicity of the basic Lisp structure, which makes it possible to write the Lisp functions in a very simple and beautiful way.

3.2 The Representation Of Expressions And Functions

As "data-structure" we use the Lisp list structure. In the following table the representation of the structures we need is defined:

boolean structure:		Lisp representation:
<variable>	::=	Lisp symbol, symbol-name starting with letter
<~variable>	::=	Lisp symbol, symbol-name starting with "~"
<literal>	::=	<variable> | <~variable>
<sum>	::=	<literal>
		| "(" "+" <product> {<product>}+ ")"
<product>	::=	<literal>
		| "(" "*" <sum> {<sum>}+ ")"
<expression>	::=	<sum> | <product>
<function>	::=	"(" "=" <literal> <expression> ")"
<set-of-functions>	::=	"(" {<function>}+ ")"

The term "subexpression" is defined for the prefix notation in exactly the same way as for infix. If an expression is not just a single

literal, it is built by an operator and a list of subexpressions. Again, there are two possibilities:

1. The expression is a sum: the products which build the sum are the subexpressions.

2. The expression is a product: the sums which build the product are the subexpressions.

An example of a correct set of functions in our Lisp notation:

((= *F1* (+ (* *a* ~*b*) (* *c d*) *e*)) (= *F2* (* *d e f* (+ ~*a* (* ~*e* ~*f*)))))

In infix notation this set would be denoted as:

$F_1 = a.\bar{b} + c.d + e$

$F_2 = d.e.f.(\bar{a} + \bar{e}.\bar{f})$

As you can see we now denote boolean expressions and functions in a prefix notation. This is standard practice in Lisp expressions, and will prove very useful when we start writing Lisp functions which operate on boolean functions and expressions. It makes it for example especially easy to evaluate the boolean functions in the Lisp environment, although this is something we do not need during the technology mapping. It will however be useful in future extensions. Some practical examples can be found in the next section.

3.3 Some Basic Boolean Function Manipulation In Lisp

3.3.1 *Logical inversion of a literal*:

This is very simple: if the symbol-name of a literal starts with a "~", it has to be removed, otherwise a "~" has to be added to the front of the symbol-name.

In Lisp this may look like this:

```
(defun invert-literal (literal)
  (if (eq (subseq (symbol-name literal) 0 1) "~")
      (intern (subseq (symbol-name literal) 1))
    ;; else
    (intern (concatenate 'string "~" (symbol-name literal)))
  )
)
```

3.3.2 *Logical inversion of an expression*:

Expressions can be inverted in a very simple way using the Morgan's rules, which (in a prefix notation) state that:

1. $\overline{(+\ a_1\ a_2\ \ldots\ a_n)} = (*\ \bar{a}_1\ \bar{a}_2\ \ldots\ \bar{a}_n)$, and:
2. $\overline{(*\ a_1\ a_2\ \ldots\ a_n)} = (+\ \bar{a}_1\ \bar{a}_2\ \ldots\ \bar{a}_n)$

So all we have to do is change the operator from * to + or vise versa, and invert each of the subexpressions a_i. All we have to watch out for is if the expression we get is a simple literal, in which case our function *invert-literal* will do the job.

Implemented in Lisp this might look like this:

```
(defun invert-expr (expr)
  (let (result)
    (cond

     ;; is expr perhaps a single literal?

     ((symbolp expr)
      (invert-literal expr))

     ;; expr is a list, a complex expression

     (t
      (setq result (if (eq '* (car expr))
                       '(+)
                     '(*)))
      (dolist (sub-expr (cdr expr) (reverse result))
        (setq result (cons (invert-expr sub-expr) result))
      )
     )
    )
)
```

As you can see the implementation is very short and very clear. The above two functions only serve as an example to show the possibilities we get after choosing Lisp and the prefix notation for the implementation of the algorithms.

3.4 The Data-structure

3.4.1 *The set of functions*

In all of our Lisp functions we will assume that the set of functions is stored in the global variable *FUNCTION-LIST*. This is done according to the format defined in section 3.2.

Also available is the global variable *FUN-NAME-LIST*, which is a list with the names of the functions stored in *FUNCTION-LIST*, in exactly the right order. This list is used to facilitate referencing functions in *FUNCTION-LIST*.

To make changes to *FUNCTION-LIST* the following functions are defined:

- (add-function-to-function-list <function>)
  Adds function <function> to the global function-list *FUNCTION-LIST*. It is added in front, and global variable *FUN-NAME-LIST* is updated.

- (replace-function-in-function-list <function> <literal>)
  If a function with name <literal> is found in *FUNCTION-LIST*, it is replaced by <function> and t is returned. *FUN-NAME-LIST* is updated if necessary. In all other cases nil is returned.

- (remove-function-from-function-list <literal>)
  Removes the function with name <literal> from the global function-list *FUNCTION-LIST*. Returns t if successful, nil otherwise. Global variable *FUN-NAME-LIST* is updated.

Each of these Lisp functions have the obvious effect of making the desired changes to *FUNCTION-LIST* destructively. To safeguard the contents of *FUNCTION-LIST*, these are the only functions allowed to change it.

3.5 The Representation Of The Technology

The technology is represented by a single integer called *gate-size*, which represents the maximum size of a standard cell in the library. This parameter is an argument of the Lisp functions which need it. See also section 6.2.

4. THE STEPS BEFORE AND AFTER THE TECHNOLOGY MAPPING

4.1 Introduction

The whole process from the designer-specified description of a piece of combinational logic to the actual geometrical layout description of this piece of IC area is performed in a series of steps. We can describe them as follows:

1. The designer specifies the functional description as a set of logical (= boolean) functions.

2. These functions are simplified as far as possible, which means that redundancies are removed.

3. The now nonredundant set of functions is decomposed to a certain degree, which means that subexpressions which occur more often than a users-specified minimum are replaced by a new variable, and the subexpression is added as a new function to the set of functions.

4. Now the result of the previous step is mapped onto a certain technology (NMOS, CMOS), using some kind of structure (gate array, pla, standard cells, gate matrix...).

5. In case of gate arrays and standard cells placement and routing have to be performed as a last step.

Steps 1, 2, 3 and 5 will be looked at in this chapter. The mapping onto standard cells is described in the following chapters.

4.2 The Functional Description

The designer will have to specify a piece of combinational logic as a set of logical (boolean) equations.

With i inputs, j users-specified intermediate functions, and k outputs we get:

$$F_1 = f(inp_1 \ldots inp_i, \overline{inp}_1 \ldots \overline{inp}_i, int_1 \ldots int_j, \overline{int}_1 \ldots \overline{int}_j)$$

....

$$F_k = f(inp_1 \ldots inp_i, \overline{inp}_1 \ldots \overline{inp}_i, int_1 \ldots int_j, \overline{int}_1 \ldots \overline{int}_j)$$

$$int_1 = f(inp_1 \ldots inp_i, \overline{inp}_1 \ldots \overline{inp}_i, int_2 \ldots int_j, \overline{int}_2 \ldots \overline{int}_j)$$

....

$$int_j = f(inp_1 \ldots inp_i, \overline{inp}_1 \ldots \overline{inp}_i, int_1 \ldots int_{j-1}, \overline{int}_1 \ldots \overline{int}_{j-1})$$

These functions have to be specified as sums of products.

The following small set of functions is provided as a realistic example. It is part of the benchmark set which is used for testing purposes all over the world, and known under the name of "alul". It does not contain intermediate functions.

Example:

$F_1 = \overline{a} + e.\overline{l} + \overline{e}.\overline{k}$
$F_2 = \overline{b} + \overline{l}.f + \overline{k}.\overline{f}$
$F_3 = \overline{c} + \overline{l}.g + \overline{k}.\overline{g}$
$F_4 = \overline{d} + \overline{l}.h + \overline{k}.\overline{h}$
$F_5 = e.\overline{a}.\overline{i} + \overline{e}.\overline{a}.\overline{j}$
$F_6 = f.\overline{b}.\overline{i} + \overline{f}.\overline{b}.\overline{j}$
$F_7 = g.\overline{c}.\overline{i} + \overline{g}.\overline{c}.\overline{j}$
$F_8 = h.\overline{d}.\overline{i}$

4.3 Logical Simplification

The logical simplification at this moment is performed by the program *log_sim*, which removes only certain types of redundancy. Removed is cube containment, resulting in a *minimum term prime implicant cover* of all the functions.

To give an example of this simplification:

$F = a.b.c + a.b + c.d + c.\overline{d}$

will be reduced to:

$F = a.b + c$

because cube $a.b.c$ is contained in cube $a.b$, which means that $a.b \Rightarrow a.b + a.b.c$, and because $c.d + c.\overline{d}$ can be written as $c.(d + \overline{d})$, which is clearly logically equivalent to c.

For more information on this subject see lit. [1].

4.4 Decomposition

The decomposition process performed on the set of functions can be guided by the designer with a number of parameters. He can choose a minimum size for subexpressions (measured in literals) to be substituted, and a minimum number of occurrences for them as well, all independently. Also a limit can be put to the logical depth of the resulting set of functions, measured in gate-delays. In this way the size and speed of the resulting circuit can be influenced. For an example of this influence the benchmark results in appendix A. can be referenced.

To give an example of this process, assuming no restrictions on the decomposition, we take the following set of functions:

$F_1 = a.b + c.d + e.f$

$F_2 = a.b + c.d + g.h$

The common term $a.b + c.d$ will be found and substituted, after which the set of functions will become:

$F_1 = subst + e.f$

$F_2 = subst + g.h$

$subst = a.b + c.d$

It should be noted that this step introduces an extra level of logic, and therefore extra delay in the resulting circuit. Inputs a, b, c and d will have to propagate through two gates now before they reach an output. This means that decomposition will make the resulting circuit smaller, but it also makes it slower.

For more information on this subject see lit. [2] and [3].

4.5 Placement And Routing

When the cells necessary to implement the combinational logic are found, they will have to be interconnected. This is called routing. In order to be able to do this with as little wiring as possible, the cells will have to be placed smartly, so the routing is preceded by a placement procedure. Only now the piece of combinational logic is implemented completely.

5. WEAK DIVISION

5.1 Introduction

As we have seen in chapter 3, the starting point of the technology mapping is a set of boolean functions in a sum of cubes format. This sum of cubes format is not very useful for us, because it often contains the same literal more than once. Because each literal which occurs in more than one cube will also mean that number of transistors in a standard cell in NMOS technology extra, or even twice that number of transistors in CMOS, reduction of these multiple occurrences will in the end reduce the necessary number of transistors, and therefore the occupied area on the chip. We will try to get rid of these multiple occurrences by using weak division.

Weak division will split up a function:

$$F = f(a_1 \ldots a_n, \bar{a}_1 \ldots \bar{a}_n)$$

in which a_i occurs more than once, into:

$$F = a_i.f(a_1 \ldots a_{i-1}, a_{i+1} \ldots a_n, \bar{a}_1 \ldots \bar{a}_n) + f(a_1 \ldots a_{i-1}, a_{i+1} \ldots a_n, \bar{a}_1 \ldots \bar{a}_n)$$

which can also be written like:

$$F = term.cofactor + remainder$$

To give a simple example, weak division will change the sum of cubes format:

$$F = a.b + a.c$$

into a complicated expression format:

$$F = a.(b + c)$$

5.2 The Approach

All functions from our initial set of functions are treated one by one. As they are expressed as sums of cubes, with each cube containing only literals, weak division upon them is relatively easy. The strategy chosen takes the literal occurring in the biggest number of cubes, and divides that one out. If there are more cubes with the same number of occurrences, the first one found is used. This process is then called recursively on the cofactor and the remainder, and is continued until all multiple occurrences are divided out.

Example:

$F = a.b.c.d + a.b.\bar{e}.f + a.g.h + c.i.\bar{j} + i.\bar{j}.k.l$

will be divided in the following steps:

1. $F = a.(b.c.d + b.\bar{e}.f + g.h) + c.i.\bar{j} + i.\bar{j}.k.l$
2. $F = a.(b.(c.d + \bar{e}.f) + g.h) + i.(c.\bar{j} + \bar{j}.k.l)$
3. $F = a.(b.(c.d + \bar{e}.f) + g.h) + i.\bar{j}.(c + k.l)$

which can not be reduced further.

It is clear that after this step the functions are no longer sums of cubes, but complicated expressions with an undefined level of bracket nesting.

It is also clear that the strategy of starting with the most often occurring literal does not necessarily give the optimal result, the minimum number of literals. If we take the function:

$F = a.b + a.c + a.d.e.f.g.h + d.e.f.g.h.i$, containing 16 literals

then the result of our strategy will be:

$F = a.(b + c + d.e.f.g.h) + d.e.f.g.h.i$, containing 14 literals

whereas the best solution would be:

$F = a.(b + c) + d.e.f.g.h.(a + i)$, containing 10 literals

It would therefore be recommendable to improve the strategy in the respect of choosing the right literal(s) to divide by. At the moment however it is not clear how to do this economically.

5.3 The Lisp Implementation

If we assume the following Lisp functions to be defined:

- (get-most-freq-term <expression>)
  Returned is a list with the most frequent element in <expression> followed by the number of occurrences. For comparison #'equal is used. <expression> is scanned from left to right, so the leftmost element with the highest number of occurrences is found.

- (remove-1-elt <item> <sequence>)
  Returned is <sequence> with the first occurrence of <item> removed.

then we can write down the weak division algorithm in Lisp like this:

```
(defun dirty-simpl-by-div (expr)
  (let ((divterm (get-most-freq-term expr))
        remainder
        cube
        cofactor)

     (cond
      ((= 1 (cadr divterm)); no term occurs more than once
       expr)

      (t ; there is something to divide
       (setq divterm (car divterm))
       (dolist (cube expr)
         (cond
          ((operatorp cube)
           nil)

          ;; is cube perhaps a single literal?

          ((symbolp cube)
           (if (eq divterm cube)
               (setq cofactor (cons '1 cofactor))
             ;; else
             (setq remainder (cons cube remainder))))

          ;; is divterm in this cube?

          ((member divterm cube :test #'equal); yes, it is!
           (if (> (length cube) 3); removing divterm will not
                                  ; change cube to single literal
               (setq cofactor (cons (remove-1-elt divterm cube)
                                    cofactor))
             ;; else, cube becomes single literal
             (setq cofactor (cons (cadr (remove-1-elt divterm cube))
                                  cofactor))))

          ;; divterm is not in this cube

          (t
           (setq remainder (cons cube remainder)))
         )
       )
```

```
        ;; format output

        (if (null remainder)
            (dirty-simpl-by-div (list '*
                                      divterm
                                      (dirty-simpl-by-div
                                       (cons '+ cofactor))))
          ;; else
          (dirty-simpl-by-div
           (cons '+
                 (append remainder
                         (list
                          (list '*
                                divterm
                                (dirty-simpl-by-div
                                 (cons '+ cofactor)))))))))
    )
  )
)
```

This Lisp function is called dirty-simpl-by-div because it sometimes produces "dirty" output like (* *a* (* *b* (+ ...))) instead of the correct notation (* *a b* (+ ...)). This direct nesting of products is not allowed by the definition in section 3.2., and therefore has to be removed by a filtering function. This of course is the case in the total program.

6. THE ACTUAL MAPPING ONTO STANDARD CELLS

6.1 Introduction

At this stage of the process, we are left with a set of boolean functions of unknown structure and size. Many of these functions will be too complex to fit directly on one of the standard cells available in the current technology. Therefore a mapping operation will have to take place, which will split up the big functions into a set of smaller ones, each of which should have a direct counterpart in the set of standard cells. This mapping operation should be as optimal as possible in two respects: big functions should be split up in as few standard cells as possible, and the resulting multiple-level structure of these standard cells should be as fast as possible, i.e. the logical depth should be limited as far as possible.

6.2 The Library Of Standard Cells

For this program the library of standard cells should contain all *and-or-invert* gates up to a certain number of inputs for the *and* and the *or*. This maximum number of inputs must be equal for the *and* and the *or* gate, to make sure that for any gate of the library the dual gate does also exist. By dual gate we mean the gate with *and* changed to *or* and vice versa. The necessity of this requirement will become clear in the next chapter, when we will remove inverters by inverting functions. This maximum number of inputs is a parameter to the program and will henceforth be called *gate-size*.

To give a formal definition of which gates are in the library for a given *gate-size*, we can denote an expression as a directed graph, with every literal forming a node and an *or* relation giving rise to two parallel edges, opposed to the and relation, which connects two nodes with an edge in series. This seems to be a bit complicated, but a few examples will soon clarify the situation. The expression $a + b + c + d$ has a graph representation as follows:

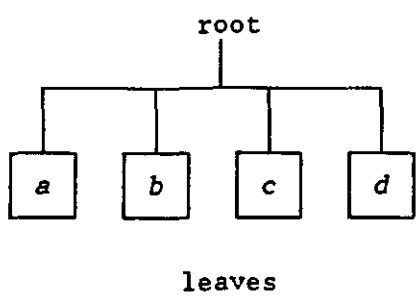

Figure 1. Graph of expression $a + b + c + d$

The expression *a.b.c.d* has a graph representation like:

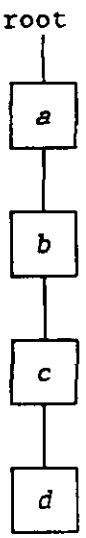

Figure 2. Graph of expression *a.b.c.d*

And the more complicated expression *a.b.c + d.e.(f + g).(h + i) + j.k* has a graph representation:

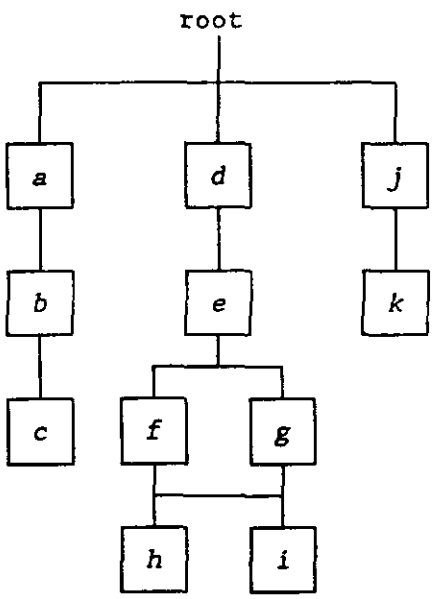

Figure 3. Graph of expression *a.b.c + d.e.(f + g).(h + i) + j.k*

In this last case the leaves are formed by the nodes *c*, *h*, *i* and *k*. The graph is directed in the sense that all edges connect downward or horizontally, but never upward. A horizontal part of an edge is therefore bidirectional.

Now the formal definition: A gate belongs to the library with parameter *gate-size* if and only if:

1. There is no path from root to any of the leaves which passes by more than *gate-size* nodes, and:

2. There is no horizontal cut in the graph which cuts more than *gate-size* edges.

Clearly all 3 above examples are part of all libraries with *gate-size* ≥ 4.

So, if *gate-size* equals 3 (as it does in the technology currently used on the TUE), the following combined gates for example are in the library:

- $F = \overline{(a_1 + a_2 + a_3).(b_1 + b_2 + b_3).(c_1 + c_2 + c_3)}$
  And of course all gates with the same form but missing one or more of the inputs.

- $F = \overline{a_1.a_2.a_3 + b_1.b_2.b_3 + c_1.c_2.c_3}$
  And of course all gates with the same form but missing one or more of the inputs.

- $F = \overline{a.(b_1 + b_2).(c_1 + c_2) + d_1.d_2.d_3}$

And the following gates for example are not included in the library:

- $F = \overline{a_1 + a_2 + a_3 + a_4}$
- $F = \overline{a.b.(c_1 + c_2) + (d_1 + d_2).e.f}$

Note that the fact that all standard cells have an inverted output will not bother us at this stage. We simply pretend they have straight outputs and add an extra inverter to each gate output later. In chapter 7 an approach to remove as many as possible of these inverters will be presented.

6.3 The Approach

6.3.1 *Representation of a complicated function*

The approach we use to split up the big functions is partly heuristical and partly analytical. The heuristical part consists of the

following two limitations:

1. We use a so-called *top-down* approach, i.e. the function is attacked at its highest level, the output level. In this way we hope to get a situation where we map onto big, complicated standard cells near the output of the function, and get possibly smaller and simpler ones near the inputs. If we take for example the function:

   $F = a.b.c.d + e.f.g.h + i.j.k.l + m.n.o.p$

   and the library contains standard cells with a maximum *gate-size* of 3, then the *top-down* approach breaks down F into the set of functions:

   $F = a.b.subst_1 + e.f.subst_2 + subst_3$

   $subst_1 = c.d$

   $subst_2 = g.h$

   $subst_3 = i.j.subst_4 + m.n.subst_5$

   $subst_4 = k.l$

   $subst_5 = o.p$

   The logical depth (= the biggest number of gates through which an input signal must propagate to reach an output) of the circuit represented by this set of functions is 3. The input signals which have a path of this length, are k, l, o and p.

   The *bottom-up* approach however would result in the following set of functions:

   $F = a.subst_1 + subst_5$

   $subst_1 = b.c.d$

   $subst_2 = f.g.h$

   $subst_3 = j.k.l$

   $subst_4 = n.o.p$

   $subst_5 = e.subst_2 + i.subst_3 + m.subst_4$

   The logical depth of this circuit is also 3, but now input signals f, g, h, j, k, l, n, o and p have a propagation path of this length.

   The reason for this difference in number of input signals with the longest path is that the *top-down* approach results in bigger gates near the output, and smaller gates nearer to the inputs of the circuit. The *bottom-up* approach however tries to fit big gates near to the inputs, and near to the output only a small

gate may be left. This certainly increases the number of inputs with maximum path length, and possibly in some situations even the logical depth. So we prefer the *top-down* approach here.

2. At each moment we limit our view of a certain expression to 1 level, which means we represent it like:

   $expr = subexpr_1 + subexpr_2 + \ldots subexpr_n$, or:

   $expr = subexpr_1 . subexpr_2 . \ldots subexpr_n$

   whichever is appropriate, and solve the problem of mapping this expression on a tree of standard gates completely optimal, in an analytic way. If any of $subexpr_i$ is a complex expression itself, and not just a single literal, this process is continued recursively with these subexpressions until literals are found.

6.3.2 *Size restriction*

To control this recursive process, the size an expression is allowed to have is represented by a pair of numbers (a, b), with a indicating the number of subexpressions the expression is allowed to have, and b how many sub-subexpressions each of the subexpressions will be allowed to have. Size restriction parameter a is always important at the current level, and b at the next level. In fact, at each level the second size restriction parameter is never used or changed, but only passed on as the first size restriction parameter for the next (lower) level. So, if *gate-size* equals 3, then at top level, when we enter a function, the size restriction will be (3, 3). Each of the subexpressions of the function will get a size restriction of (3, ...). The value of the second size restriction parameter for the subexpressions depends on what has happened with the expression on the current level, and will be important again for the possible sub-subexpressions, and so forth for every second level. In 6.4.2. this recursive process will be explained in more detail.

A more formal definition of size restriction can be formulated if we picture the expression as a graph, as defined in section 6.2. We must now distinguish two possibilities:

1. Expression is a sum. The first size restriction parameter a now is equivalent to the maximum number of edges any horizontal cut in the graph may cut, and the second size restriction parameter b is equivalent to the maximum path length in the graph which is allowed.

2. Expression is a product. Now a limits the longest path and b the horizontal cut.

6.4 The Algorithm

6.4.1 *The solution of the one level problem*

If we take the problem of mapping the expression:

$expr = subexpr_1\ op\ subexpr_2\ op\ \ldots\ subexpr_n$;
op being either . or +

and the size restriction (*a*, *b*),

then we can calculate the exact structure of the tree of standard-gates required to implement this expression. This solution of the one level problem will be optimal in two senses: we will use the minimal amount of standard cells to do it, and the logical depth of the solution will also be minimal. For this we only need the size restriction parameter *a*, telling us how many subexpressions we can leave in the expression, and the technology parameter *gate-size*, informing us on the maximum size of the potential substitution gates.

6.4.1.1 *How does substitution on one level work?*

If we see that an expression has more subexpressions than the first size restriction parameter *a* allows, then we have to limit this number of subexpressions somehow. What we do is to take away a number of subexpressions from the expression and substitute one literal for them instead. This literal has to be unique in the set of functions. To keep the set of functions equivalent to the originally user specified set, we have to add a new function to it, which defines the value of the newly chosen literal. So, if we substitute *i* subexpressions in an expression with originally *n* subexpressions, the process is as follows:

$expr = subexpr_1\ op\ subexpr_2\ op\ \ldots\ subexpr_n$

is transformed to the logically equivalent set of functions:

$expr = subexpr_1\ op\ \ldots\ subexpr_{n-i}\ op\ subst$
$subst = subexpr_{n-i+1}\ op\ \ldots\ subexprn$

Now if $n-i$ is still bigger than *a*, more of these substitutions will be necessary.

It is wise to take away exactly *gate-size* subexpressions per substitution, because then the newly formed functions will automatically be mapped to a standard cell which is as big as possible at least in one "dimension". All other possibilities have disadvantages:

- If we would take away less than *gate-size* subexpressions at a time, more substitutions would be necessary for every big expression, resulting in too many new functions and therefore in too

many standard cells in the resulting circuit. This is a waist of area.

- If we take away more than *gate-size* subexpressions per substitution, but not a multiple of *gate-size*, then the resulting substitution functions will have to be split up in their turn, before they fit in a standard cell. But this will then result in at least one function with less than *gate-size* subexpressions, again not optimal.

- Taking away a multiple of *gate-size* would be acceptable, but then these substitution functions would themselves have to be split into pieces with exactly *gate-size* subexpressions, so we might as well do this right away.

6.4.1.2 *The number of necessary substitutions*

It is now easy to verify that the number of necessary substitutions for an expression with n subexpressions and a current size restriction (a, b) is:

$$nr\text{-}of\text{-}subst = ceiling((n - a)/(gate\text{-}size - 1))$$

Verification: we start with n subexpressions, of which a may be left in the end. So we have to get rid of $n - a$ subexpressions. Each substitution takes away *gate-size* subexpressions, but adds the substitution variable as an extra subexpression to the expression. So the actual number of subexpressions removed per substitution equals *gate-size* - 1. We have to take the ceiling of the division, because even if one of the substitutions does not have to take away *gate-size* - 1 subexpressions completely, there still has to be a substitution for this smaller but excessive number of subexpressions.

6.4.1.3 *The one level substitution algorithm*

Once we have found the number of substitutions, we have to start thinking about how these substitutions should be performed. To give an example of different possibilities for these substitutions we will consider the function:

$$F = a.b.c.d.e.f.g.h$$

We will take *gate-size* to be 3, and accordingly the size-restriction at top level will be (3, 3). This gives as the necessary number of substitutions *ceiling*((8-3)/(3-1)), which equals 3. Now there are a lot of possibilities to split up F using 3 substitutions, most of which do not have the minimal logical depth. To give a few examples:

$$subst_1 = f.g.h$$

$$subst_2 = c.d.e$$

$$subst_3 = subst_1.subst_2$$

$F = a.b.subst_3$

This solution gives us a logical depth in gates of 3. Let's look at another solution:

$subst_1 = g.h$

$subst_2 = e.f.subst_1$

$subst_3 = c.d.subst_2$

$F = a.b.subst_3$

This solution gives us a logical depth of 4! Clearly the previous solution was better. But let's look at a third solution:

$subst_1 = g.h$

$subst_2 = d.e.f$

$subst_3 = a.b.c$

$F = subst_1.subst_2.subst_3$

This one gives us a logical depth of only two, and seems to be the best solution possible.

6.4.1.4 *The minimal logical depth*

It is possible to calculate the minimum logical depth needed to map a function with n subexpressions, and given *gate-size* and size-restriction. If the first size-restriction parameter is a, then:

$$min\text{-}depth = ceiling(^{gate\text{-}size}\log\ n/a) + 1$$

This formula can also be verified quickly. With a depth of 1 we can handle a maximum of a subexpressions, with a depth of 2, $a * gate\text{-}size$ subexpressions, and with a depth of k, $a * gate\text{-}size^{k-1}$ subexpressions. Clearly, the inversion of this formula results in the expression for the minimal depth.

This minimal depth can only be reached if we choose the right substitutions. The algorithm presented here will always find the optimal solution, without actually calculating it. To understand it, we have to switch to the prefix notation now. In this prefix notation, an expression can be represented by:

$expr : (operator\ subexpr_1 \ldots subexpr_n);\ n > gate\text{-}size$

Because the operator is of no importance for the substitution operation, we will take only the list with the subexpressions $(subexpr_1 \ldots subexpr_n)$, and formulate the substitution operation on that list.

6.4.1.5 *The small gate*

The first thing we have to worry about is that there is possibly one gate which does need less than *gate-size* inputs. This gate will always be one of the substitutes, because we will want the small gate to be one of the leaves of the tree of gates after the substitutions have taken place. The expression which tells us how many inputs this smaller gate will have is:

size-of-small-gate = *remainder*((*n* + *nr-of-subst* - *a*)/*gate-size*)

Verification: A total of *n* + *nr-of-subst* - *a* subexpressions have to be substituted into gates of size *gate-size* or smaller. Clearly we want as many substitutions as possible to have a size of *gate-size*. Then the one gate which can be smaller is of the above size. (If *n* + *nr-of-subst* - *a* is a multiple of *gate-size*, no small gate will be necessary).

6.4.1.6 *The actual substitution*

Now the substitutions are performed as follows:

1. If *size-of-small-gate* > 0 substitute the small gate. Take away *size-of-small-gate* subexpressions from the end of the list, and add the substitution variable to the front of the list.

2. Now substitute all the other gates. Every substitution takes away the last *gate-size* subexpressions from the end of the list, and adds its substitution variable to the front of the list. This is continued until the list contains exactly *a* subexpressions, the number allowed by the size restriction.

The result of this algorithm is an optimal tree of gates. This can be proven as follows:

We start with a logical depth of 1 in the expression:

(*operator* $subexpr_1$ ... $subexpr_n$)

Now, because we always take away subexpressions from the end of the list for each substitution, and add the substitution variables at the front, the logical depth connected to each subexpression of the list will always be decreasing in the direction of the list. This guarantees that we take away the subexpressions with the lowest possible logical depth. In this way the logical depth will be kept minimal.

6.4.2 *The complete mapping algorithm*

We now know how to perform mapping on a one level expression. But of course most expressions are much more complicated. We will now discuss the problem of mapping such a complicated expression onto standard cells.

We take the complicated expression

$subexpr_1$ *op* $subexpr_2$ ... *op* $subexpr_n$

and a size restriction (a, b). Of course initially both a and b will be equal to *gate-size*. Each $subexpr_i$ is itself an expression of unknown size and structure. Now we distinguish two possibilities:

1. $n > a$: There has to be substitution at this level. Solve the one level problem and then map each of the a subexpressions left in the expression with size restriction (b, 1). Because all substitution functions are added to the set of functions, they will be mapped later as an independent function.

2. $n \leq a$: No substitutions on this level. Map all n subexpressions subsequently now, the first subexpression with a size restriction (b, a-n+1). Of each subexpression after it has been mapped the size it actually needs is checked. The size an expression needs is also expressed as a pair of numbers, like the size restriction. This pair (x, y) has $x \leq b$ and $y \leq a$-n+1 of course, otherwise substitutions would have been done. Now y is important for us. The second subexpression gets size restriction (b, a-n+1-y+1), and the i-th subexpression, after the other subexpressions have returned an actual size of (x_1, y_1) ... (x_{i-1}, y_{i-1}), a size restriction of:

$$(b,\ a\text{-}n\text{+}1 - \sum_{j=1}^{i-1} y_j + i\text{-}1).$$

So each subexpression gets a chance to expand on the "second level" as far as the situation up to now allows. If this space is not or not fully used up, the next subexpression gets the space which is left over.

The above algorithm requires something we have not yet discussed, the actual size an expression occupies. Its definition is simple: it is equal to the smallest size restriction necessary to get no substitutions with this expression. Therefore, if we picture expression as a graph, it is equal to:

- If expression is a sum, the pair (longest horizontal cut, longest path) of that graph, and

- If expression is a product, to the pair (longest path, longest horizontal cut).

To calculate it we use an equivalent recursive definition:

- An expression with only n literals occupies a space of (n, 1). So a single literal occupies a space of (1, 1).

- An expression with n subexpressions, each of which occupies a space of (x_i, y_i), occupies a total space of

$$\left(\sum_{i=1}^{n} y_i, \max(x_1 \ldots x_n) \right)$$

This recursive definition also has its counterpart in graph theory:

- If we connect subgraphs in parallel, which is equivalent to making a sum out of (product-)subexpressions, then the longest horizontal cut of the total graph will be the sum of the longest cuts of the subgraphs, and the longest path in the graph will be equal to the maximum of the longests paths of the subgraphs.

- If expression is a product, then we connect the subgraphs in series. Now the longest horizontal cut of the resulting graph will be equal to the maximum of the horizontal cuts of the subgraphs, and the longest paths of the graph is equal to the sum of the longest paths of the subgraphs.

6.4.3 *A complete example*

To illustrate the mapping algorithm, we will perform the mapping of a function step by step. To make the process reasonably interesting, we will assume a *gate-size* of 4. The function to be mapped is:

$$F = a.(b + c).(d + e.(g + h + i)).(j + k.(l + m)) + n.o.p.(q + r + s.(t + u))$$

It is clear that F is a sum, and consists of 2 subexpressions. To write down the steps the algorithm performs, we will make use of some formatting definitions. Each time we enter the mapping algorithm, we will write a line like "->entering: expression, size-restriction". The return value of the algorithm will be stated with "->returning: expression, actual-size". The recursive nesting will be shown by indenting the lines when we move down a level.

So, we enter the algorithm with the complete expression and a size restriction of (4, 4).

```
→entering:  a.(b + c).(d + e.(g + h + i)).(j + k.(l + m))
          + n.o.p.(q + r + s.(t + u)), (4, 4).
        expression is a sum with 2 subexpressions.

    →entering: a.(b + c).(d + e.(g + h + i)).(j + k.(l + m)), (4, 3).
            expression is a product with 4 subexpressions.

        →entering:  a, (3, 1).
        →returning: a, (1, 1).

        →entering: b + c, (3, 1).
                expression is a sum with 2 subexpressions

            →entering:  b, (1, 2).
            →returning: b, (1, 1).

            →entering:  c, (1, 2).
            →returning: c, (1, 1).

        →returning: b + c, (2, 1).

        →entering: d + e.(g + h + i), (3, 1).
                expression is a sum with 2 subexpressions

            →entering:  d, (1, 2).
            →returning: d, (1, 1).

            →entering: e.(g + h + i), (1, 2).
                    expression is a product with 2 subexpressions
                    n > a, so substitution has to be performed
                    the new function subst_1 = g + h + i is
                    formed
            →returning: subst_1, (1, 1).

        →returning: d + subst_1, (2, 1).
```

→entering: $j + k.(l + m)$, (3, 1).
expression is a sum with 2 subexpressions

→entering: j, (1, 2).
→returning: j, (1, 1).

→entering: $k.(l + m)$, (1, 2).
expression is a product with 2 subexpressions
$n > a$, so substitution has to be performed
the new function $subst_2 = k.(l + m)$ is formed
→returning: $subst_2$, (1, 1).

→returning: $j + subst_2$, (2, 1).

→returning: $a.(b + c).(d + subst_1).(j + subst_2)$, (4, 2).

→entering: $n.o.p.(q + r + s.(t + u))$, (4, 2).
expression is a product with 4 subexpressions

→entering: n, (2, 1).
→returning: n, (1, 1).

→entering: o, (2, 1).
→returning: o, (1, 1).

→entering: p, (2, 1).
→returning: p, (1, 1).

→entering: $q + r + s.(t + u)$, (2, 1).
expression is a sum with 3 subexpressions
$n > a$, so substitution has to be performed
the new function $subst_3 = r + s.(t + u)$ is formed
→returning: $subst_3 + q$, (2, 1).

→returning: $n.o.p.(q + subst_3)$, (4, 2).

→returning: $a.(b + c).(d + subst_1).(j + subst_2) + n.o.p.(q + subst_3)$, (4, 4).

The result of this mapping operation is a set of 4 functions:

$F = a.(b + c).(d + subst_1).(j + subst_2) + n.o.p.(q + subst_3)$

$subst_1 = g + h + i$

$subst_2 = k.(l + m)$

$subst_3 = r + s.(t + u)$

6.4.4 *Possible future improvements to the algorithm*

The main disadvantage of this algorithm is that the subexpressions of an expression are not investigated at all before determining how to substitute. It might well be useful to change this in future. Some kind of sorting operation on the subexpressions based upon their size could make the algorithm better.

6.5 Optional Removal Of Equivalent Functions

6.5.1 *Introduction*

If the decomposition previous to the technology mapping was not exhaustive, meaning that it was limited to subexpressions with a minimum size greater than 2, or a minimum amount greater than 2, then it is possible that some of the remaining equal subexpressions are found during our mapping operation. This will mean that there will be 2 or more functions in the set which are exactly the same. This also implies having two different gates on the chip doing exactly the same job. This is a waist of area, and therefore we will try to remove this redundancy.

Because this test is only necessary if the decomposition was limited, it is optional and can be switched on and off by the designer.

6.5.2 *The algorithm*

Because testing for logical equivalence is a very time intensive operation, and because we know that the set of functions is freed as far as possible of redundancies, we will use a simpler algorithm instead. If we compare two functions, we will sort them uniquely, and then see if the results are subexpression by subexpression exactly equal. Only in that case we will declare them "equivalent" and remove one of them from the set of functions. Starting with leaf functions and working towards the outputs we test every function of the set (and its inverse) with every other function in this way, and remove the "equivalent" ones immediately. Of course all references to the removed functions are changed too. In this way we also find equivalences which occur only after another equivalence was removed.

6.5.2.1 *The sorting of expressions*

Expressions are sorted with the relation *expr<*. (*expr<* $expr_1$ $expr_2$) returns *t* when:

1. In case both $expr_1$ and $expr_2$ are literals: if (*string<* $expr_1$ $expr_2$) returns *t* (so they are put in alphabetical order).

2. If one is a literal and the other is a complicated expression: the literal is considered *expr<* than the complicated expression. The complicated expression is sorted recursively.

3. When both are complicated expressions: The first subexpressions of the sorted (!) expressions are recursively compared with *expr<*.

Thus recursively sorted, all functions will have a unique representation.

6.6 The Lisp Implementation

6.6.1 *The mapping algorithm*

The algorithm described in 6.4.2. is implemented in the Lisp function (implement-expr <gate-size> <expression> <size-restr1> <size-restr2>). The Lisp code follows the algorithm step by step and should not be difficult to understand.

```
(defun implement-expr (gate-size expr max-nr-of-subexprs
                                      max-subexpr-length)

  ;; We need quite a lot of local variables, here they come:

  (let* ((expr-length (1- (length expr)))
         (current-operator (car expr))
         (result-expr (list current-operator))
         result-of-sub-impl
         (max-incr-subexpr-length-used 1)
         (total-incr-subexpr-width-used 0)
         incr-subexpr-width-used
         incr-subexpr-length-used
         new-function-def
         new-name
         new-expr
         subexprs-left
         first-strip
         nr-of-subst)
```

```
(cond

 ;; are there too many subexpressions?

 ((> expr-length max-nr-of-subexprs)

  ;;first we calculate how many substitutions will have
  ;;to be done, and how many subexpressions should be taken
  ;;away the first time ('first-strip')

  (setq nr-of-subst
        (ceiling (/ (- expr-length max-nr-of-subexprs)
                    (float (1- gate-size)))))
  (setq first-strip
        (rem (- (+ expr-length nr-of-subst)
                max-nr-of-subexprs)
             gate-size))

  ;; is there a gate with less than gate-size subexpressions?

  (unless (zerop first-strip)
    ;; make new function out of last first-strip subexprs
    (setq new-function-def
          (cons current-operator
                (nthcdr (1+ (- expr-length first-strip)) expr)))
    ;; update expr
    (setq expr
          (subseq expr 0 (1+ (- expr-length first-strip))))
    ;; get new name for substitution function
    (setq new-name (intern (symbol-name (gensym))))
    ;; and add it to the front of expr
    (setq expr (cons current-operator (cons new-name (cdr expr))))
    (setq expr-length (1- (length expr)))
    ;; add new function to FUNCTION-LIST
    (add-function-to-function-list (list equivalence
                                         new-name
                                         new-function-def))
  )
```

```
  ;; now we start taking away 'gate-size' subexpressions at a
  ;; time until the expression is no longer too long. Substitution
  ;; is exactly the same as above

  (while (> expr-length max-nr-of-subexprs)
    (setq new-function-def
          (cons current-operator
                (nthcdr (1+ (- expr-length gate-size)) expr)))
    (setq expr (subseq expr 0 (1+ (- expr-length gate-size))))
    (setq new-name (intern (symbol-name (gensym))))
    (setq expr (cons current-operator
                     (cons new-name (cdr expr))))
    (setq expr-length (1+ (- expr-length gate-size)))
    (add-function-to-function-list (list equivalence
                                         new-name
                                         new-function-def))
  ))

;; all substitutions ready, at most max-nr-of-subexprs now

(setq subexprs-left (length (cdr expr)))
(dolist (subexpr (cdr expr))
  (setq subexprs-left (1- subexprs-left))
  (cond
   ((symbolp subexpr); subexpr is a literal
    (setq result-expr (cons subexpr result-expr))
    (setq max-nr-of-subexprs (1- max-nr-of-subexprs))
    (setq total-incr-subexpr-width-used
          (1+ total-incr-subexpr-width-used)))

   (t ; subexpr is a list, a complicated expression
    (setq result-of-sub-impl (implement-expr subexpr
                                             max-subexpr-length
                                             (- max-nr-of-subexprs
                                                subexprs-left)))
    (setq new-expr (car result-of-sub-impl))
    (setq incr-subexpr-length-used (cadr result-of-sub-impl))
    (setq incr-subexpr-width-used (caddr result-of-sub-impl))
    ;; update Σ y_i
    (setq total-incr-subexpr-width-used
          (+ incr-subexpr-width-used
             total-incr-subexpr-width-used))
```

```
        ;; update max(x_i)
        (if (> incr-subexpr-length-used max-incr-subexpr-length-used)
            (setq max-incr-subexpr-length-used
                  incr-subexpr-length-used)
        )

        (setq result-expr (cons new-expr result-expr))
        (setq max-nr-of-subexprs
              (- max-nr-of-subexprs incr-subexpr-width-used)))
      )
    )

    ;; Make return-value
    (if (= 2 (length result-expr)); result-expr of form (sgnlx +)
        (list (car result-expr)
              total-incr-subexpr-width-used ;Σ y_i
              max-incr-subexpr-length-used) ;max(x_i)
      (list (nreverse result-expr)
            total-incr-subexpr-width-used ;Σ y_i
            max-incr-subexpr-length-used) ;max(x_i)
    )
  )
)
```

6.6.2 *The optional removal of equivalent functions*

If we compare two functions, we sort them using the Lisp function (sort <sequence> <test>). After this they are compared with the Lisp function (equal <item1> <item2>). As test for (sort ...) we define the following function:

```
(defun expr< (expr1 expr2)
  (cond
   ((and (symbolp expr1) (symbolp expr2))
    (string< (symbol-name expr1) (symbol-name expr2)))

   ((symbolp expr1) ; But not expr2!
    (sort (cdr expr2) #'expr<)
    t)

   ((symbolp expr2) ; But not expr1!
    (sort (cdr expr1) #'expr<)
    nil)

   ;; Now both expr1 and expr2 are complicated expressions

   (t
    (expr< (car (sort (cdr expr1) #'expr<))
           (car (sort (cdr expr2) #'expr<))))
  )
)
```

To understand this function fully, we must bear in mind that the car of an expression is always an operator, and that (sort ...) sorts destructively. So we always sort the cdr of an expression only.

Two complicated expressions are sorted comparing only their first subexpressions. This is of course not unique for all expressions, because subexpressions like (+ *a b c*) and (+ *a d e*) could not be distinguished. But in our case, where redundancies are removed and literals which occur more than once are divided out, this situation never occurs.

7. GETTING RID OF THE INVERTERS AND CRITICAL PATH OPTIMISING

7.1 Introduction

The situation now is that we have a set of logical functions, all broken down to a size small enough to have a standard cell in the library which can implement them. All of the functions represent *and-or* or *or-and* combined gates, or simple *and* or *or* gates. The problem is however, that the libraries of standard gates contain only *and-invert*, *or-invert*, *and-or-invert* and *or-and-invert* gates, because these are the most natural ones in the technology which is being used. This is the case for NMOS and CMOS for example.

This implies that every standard cell we use will have to be followed by an inverter to restore the function we actually need. This has several great disadvantages. First, the extra inverters enlarge the necessary area on the chip, something which is undesirable. Secondly, the whole circuit is slowed down considerably, if we take a unit gate delay, even to half the speed that we expected when we were only looking at the set of functions. So it is clear that we will have to try to improve this situation.

7.2 The Set Of Functions Represented As A Graph

As we have seen in the previous chapter, a system of connected gates can be represented by a graph. Each gate becomes a node, and the relation "being an input of" creates an edge of the graph. If we introduce an extra node, which we call *root*, and declare all outputs of the circuit to be connected to *root*, then we have a directed a-cyclic graph (DAG) in which we can easily find these outputs. Each edge is directed in the sense that it points towards the outputs, and therefore follows the flow of the signals in the circuit. For reasons of simplicity we do not represent inverters as nodes in the DAG, but instead introduce two types of edges, one meaning "straight connection", and the other "connection through inverter". Associated with these two types of edges are numbers which indicate the delay in that connection, being 1 and 2 respectively.

To give a small example we consider the following set of functions:

$F_1 = a.b + int_1 + c.int_2$
$F_2 = \bar{a}.\bar{b} + d.int_1 + int_3$
$F_3 = \bar{d} + int_2$
$int_1 = e.f + \bar{e}.\bar{f}$
$int_2 = g.\bar{h} + int_2$
$int_3 = i.j + \bar{k}$

If we make a DAG out of this set directly we get the following result:

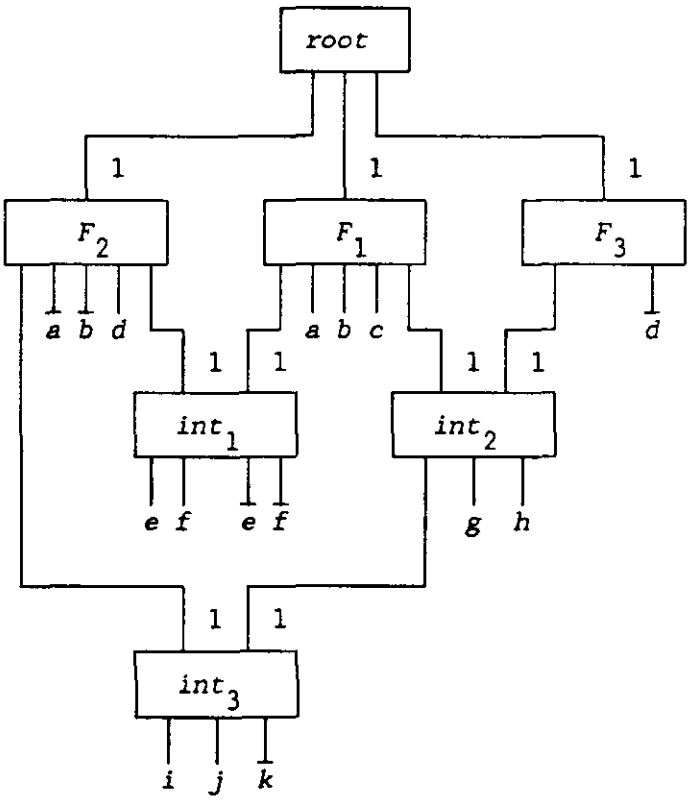

Figure 4. DAG of gates.

If we have a technology which requires extra inverters after each gate, we get exactly the same DAG, with the exception that the number next to each edge must be changed to "2".

7.3 The Delay In A Circuit

If we take a look at Fig. 4., it is clear that signals i, j and $\bar{k}$ have the longest way to travel before they reach an output. If we assume the normal situation with inverters following every output, and therefore all edges get a delay of 2, then, if we add up all delays on the path of i,j and k, we get 6. In fact, these signals determine the speed of the whole circuit. All other signals have a shorter way to the output. So it is clear that the speed of the whole circuit is determined by the longest path in the DAG, if we define path length as the sum of the delays as described by the type of edge. This longest path is therefore named the critical path. It is clear that if we want to influence the speed of a circuit we will have to do something about the delay on the critical path.

7.4 Inverting A Function In A DAG

If we consider the operation of inverting a function in the set of functions without changing the logical behaviour of the whole set, we can consider the Morgan's laws again. For reference, they were written down in section 3.3.2. With these we can see that a function is logically equivalent to its inversion with inverters added to both the output and all of the inputs.

Now if we note also that two inverters in series add up to no inverter at all, we have found a way to manipulate inverters, without changing the logical behaviour of a circuit. If we face the situation, as we often do, that all functions are connected through inverters, then we can try to get rid of as many of them as possible by selectively inverting some functions in the set. In the example of section 7.2. it would be possible to get rid of most of them by inverting F_1, F_2, F_3 and int_3. The resulting set of functions then becomes:

$$\bar{F}_1 = (\bar{a} + \bar{b}).\overline{int}_1.(\bar{c} + \overline{int}_2)$$
$$\bar{F}_2 = (a + b).(d + \overline{int}_1).\overline{int}_3$$
$$\bar{F}_3 = d.\overline{int}_2$$
$$int_1 = e.f + \bar{e}.\bar{f}$$
$$int_2 = g.\bar{h}$$
$$\overline{int}_3 = (\bar{i} + \bar{j}).k$$

And the resulting DAG:

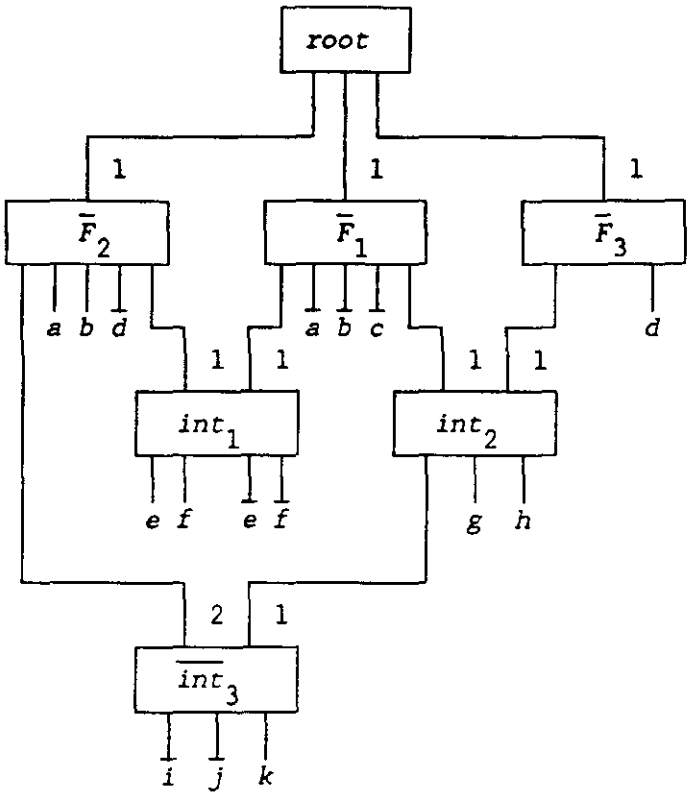

Figure 5. Optimised DAG.

The length of the longest path is now reduced to 3. In fact there are three paths with this length, $\overline{int_3} \rightarrow \bar{F}_2 \rightarrow root$, $\overline{int_3} \rightarrow int_2 \rightarrow \bar{F}_3 \rightarrow root$ and $\overline{int_3} \rightarrow int_2 \rightarrow \bar{F}_1 \rightarrow root$. As you can see there is still an inverter left between $\overline{int_3}$ and $\bar{F}_2$. Trying to remove it would however lengthen the critical path. It could be done by inverting either $\overline{int_3}$ or $\bar{F}_2$, but in both cases other inverters are added to the circuit which only make the situation worse.

7.5 Which Functions Must Be Inverted?

The answer to this question is by no means simple. It has been proven that to find the optimal solution, with the minimal number of inverters, is a N.P. complete problem. So finding this solution for big examples is probably not feasible. But then we must realise what goals we are really after. Mainly, we are interested in the speed of the circuit. But in section 7.3. we have seen that this speed is only determined by the critical path, and therefore by a few inverters only. On the other hand we still regard area as being important too. So the inverters in those parts of the circuit which are not on the

critical path bother us as well.

To solve this situation the following strategy has been chosen:

1. At first we make a global run through all the functions in the set, and determine with the use of some simple heuristics whether or not to invert this function. At this stage we limit our view to the output and the inputs of the function and base our decision upon that view only. In this way we hope to get rid of most of the inverters, but we can not claim an optimal solution.

2. Secondly, we find out what the critical path is (or: paths are), and shorten this path (these paths) rigorously. In the simple case of straight paths this means that at most one inverter is left in each of them. This process is then repeated for the new critical path(s), until no progress is possible any more.

This approach will yield a result with few inverters in the whole circuit, if we choose the right heuristics for step 1. The speed of the whole circuit, determined by the delay on the critical path, should be (near) optimal. Yet the implementation in Lisp can be very fast.

7.5.1 *The global optimising phase*

All functions are considered for inversion only once, in an order which takes functions near the root first, and leaf functions last. They are inverted if:

- In case of leaf functions:
  - if all input variables are inverted, *OR*
  - if they do not have straight (= connected without inverter) parents *AND* more inverted variables as inputs than straight variables.
- In case of an intermediate or output function:
  - if they have more children through inverters than straight children *AND* no straight parents.

The above strategy is simple and fast, and mainly based on common sense. Very probably it could still be improved, not enough experimenting has been done with it. It has however shown to give good results in many trial examples, very often even a result with the optimal critical path length. However, our main goal was minimising delay, and this problem will be attacked much more thoroughly in the next section.

7.5.2 *Critical path optimising*

The algorithm is as follows:

1. First, we have to determine the critical path(s). This is done with a standard longest path algorithm from lit. [5]. The path-length of *root* is equal to 0, and all other nodes get a path-length entry equal to the maximum of the path-length of its parents plus the delay determined by the type of edge that connects this parent. The leaf nodes with the longest path-length then determine the critical paths. The paths themselves are found by tracing backward from those leaf nodes. From the set of parents of a node on a critical path only those parents are on the path which have a path-length exactly equal to the local node's path-length minus the delay on the connecting edge.

2. After these paths have been found, they are shortened one by one. Starting at *root* all inverters are wiped off the path by making the necessary inversions of functions. Finding an inverter means invert the next function on the path. If the path forks, each branch is treated in this way. If there is reconvergent fanout on the path, which means that after forking several branches come together again, the rest of the path gets wiped more than once. When a leaf function is reached, the inverter in its output is removed unless the function has straight variables as inputs only.

3. Last, all path lengths are updated to the new situation. The new critical paths are determined. If these new critical paths are not shorter than the old ones were, stop. Otherwise go to step 2 and shorten the new critical paths.

7.6 Optional Resubstitution

7.6.1 *Introduction*

In some cases where speed is very important, it can be useful to try to turn back some of the kernel or cube substitutions done during the decomposition program. The implementation will have a few extra transistors, and therefore a larger area, but the trade-off between speed and area may be worth wile. This resubstitution can however only be possible if the minimum kernel-size or the minimal cube-size specified for the decomposition was smaller than or equal to *gate-size*, and even then it is not guaranteed to succeed. Because of this fact resubstitution is an option for the designer, who can determine beforehand whether it could be successful or not.

7.6.2 *The strategy*

The resubstitution is currently limited to the leaf functions of the critical paths only. Perhaps in future it could be extended to functions in other places of the critical path(s), but that will complicate matters considerably.

After the critical path optimisation stops, it is determined if it would be possible to resubstitute all of those leaf functions of the critical paths into their parents on the critical paths. Only if all of those resubstitutions are allowed by the *gate-size* restriction, the circuit can be speeded up. Even if only one should not be allowed, then there would still be a critical path left which keeps the circuit at the old speed. So all are resubstituted, or none.

After a successful resubstitution, the resulting new critical paths are determined, and critical path optimisation is started again. After this resubstitution is tried again, and the program keeps on resubstituting and optimising until the resubstitution fails. In appendix A. the number of successful resubstitution and optimising runs for the benchmark tests are recorded under "resubstitution level". Several times 3 consecutive runs were possible, speeding up the circuit considerably.

7.7 The Lisp Implementation

7.7.1 *The representation of the DAG*

To create the DAG of functions the following data-structure is defined. Each function-name is bound to a vector of length 11, and the definition of the contents of this vector are on position:

0: <expression> denoting the function definition.

1: a list of children which are connected straight.

2: a list of children which are connected through an inverter.

3: a list of parents which are connected straight.

4: a list of parents which are connected through an inverter.

5: an integer denoting the maximum path length from this function to *root*.

6: a list with all inputs of this function.

7: if this function is on (one of the) critical path(s), then a list of children which are also on the path and connected straight.

8: if this function is on (one of the) critical path(s), then a list of children which are also on the path and connected through an inverter.

9: if this function is on (one of the) critical path(s), then a list of parents which are also on the path and connected straight.

10: if this function is on (one of the) critical path(s), then a list of parents which are also on the path and connected through an inverter.

Of course at certain stages of the program not all of these items are specified. There is however a minimum set of specified items, set to the right value by the function (initialise-function-DAG). This functions sets items 0, 1, 2, 3, 4 and 6. The information for this is taken from global variable *FUNCTION-LIST*.

To handle the information stored in the vector a set of macros is defined. In this way the data-structure is hidden from the other functions which use the information, and could be changed without changing any of the other functions. Also the algorithms are easier to read with the macros. Mapping the entries in the vector the following macros have been defined:

0: (get-function-def <literal>) and
(set-function-def <literal> <expression>)

1: (get-children-straight <literal>) and
(set-children-straight <literal> <list>)

2: (get-children-through-inverter <literal>) and
(set-children-through-inverter <literal> <list>)

3: (get-parents-straight <literal>) and
(set-parents-straight <literal> <list>)

4: (get-parents-through-inverter <literal>) and
(set-parents-through-inverter <literal> <list>)

5: (get-max-path-length <literal>) and
(set-max-path-length <literal> <integer>)

6: (get-signal-sort <literal>) and
(set-signal-sort <literal> <list>)

7: (get-path-children-straight <literal>) and
(set-path-children-straight <literal> <list>)

8: (get-path-children-through-inverter <literal>) and
(set-path-children-through-inverter <literal> <list>)

9: (get-path-parents-straight <literal>) and
(set-path-parents-straight <literal> <list>)

10: (get-path-parents-through-inverter <literal>) and
(set-path-parents-through-inverter <literal> <list>)

7.7.2 *Calculation of the critical paths*

For any given node in the DAG we calculate the maximum of the max-path-lengths of its parents, to which the delay on the edge is added. This gives the correct entry for the node's max-path-length. Because the structure of the DAG is not known, it might happen that a parent's max-path-length is not yet defined. In that case it is calculated first. This is a more simple procedure than determining an order in which to calculate the max-path-length in the DAG first. We simple backtrack if necessary instead. When the current node's max-path-length is determined, the function calls itself recursively for the children of the node. Leaf functions do not have children, so it stops automatically.

For practical reasons the function (set-path-length ...) is able to handle a whole list of nodes to calculate the max-path-length for. To calculate all path-lengths in a whole DAG it is enough to call it on *root*.

```
(defun set-path-length (function-name-list)
  (let ((max-length-parents-straight -1)
        (max-length-parents-through-inverter -2)
        ;; they are set to -1 and -2 to give correct results for root
        ;; who does not have parents.
        length-parent)

    (dolist (fun function-name-list)
      (dolist (parent (get-parents-straight fun))
        ;; is max-path-length of parent defined yet?
        (if (setq length-parent (get-max-path-length parent))
            ;; then see if it has the longest path-length so far:
            (if (< max-length-parents-straight length-parent)
                (setq max-length-parents-straight length-parent)
            )
          ;; else, calculate it:
          (set-path-length (list parent))
        )
      )
      ;; add edge delay to the longest path length found:
      (setq max-length-parents-straight
            (1+ max-length-parents-straight))
```

```
      ;; and now the same for parents-through-inverter:
      (dolist (parent (get-parents-through-inverter fun))
        (if (setq length-parent (get-max-path-length parent))
            (if (< max-length-parents-through-inverter length-parent)
                (setq max-length-parents-through-inverter
                      length-parent)
            )
          (set-path-length (list parent))
        )
      )
      (setq max-length-parents-through-inverter
            (+ max-length-parents-through-inverter 2))

      (set-max-path-length fun
                           (max max-length-parents-straight
                                max-length-parents-through-inverter))

      ;; reset variables for next fun in dolist:

      (setq max-length-parents-straight -1)
      (setq max-length-parents-through-inverter -2)
    )

    ;; and now calculate the path-lengths of all children:

    (dolist (fun function-name-list)
      (set-path-length (get-normal-children fun))
      (set-path-length (get-inverted-children fun))
    )
  )
)
```

7.7.3 *Global optimising*

All functions in *FUNCTION-LIST* are picked up one by one, in the order they are stored there. Because substitutes are always added to the front of *FUNCTION-LIST*, this more or less assures that we start with leaf functions. The test as described in section 7.5.1. is performed, and the function is inverted if necessary. This inversion of a function in the DAG is performed by the function (invert-function-in-DAG <literal>), which does all the necessary bookkeeping to maintain the correctness of the parents- and children- entries which bind the DAG of functions together. We will not provide the Lisp functions here, because they are algorithmically simple and not interesting.

7.7.4 *Critical path optimising*

The Lisp implementation of the critical path shortening as described in section 7.5.2. is straightforward. It is assumed that the critical paths are all made traceable with the right entries for the path-parents and the path-children. In *root* all starting nodes of these

paths can then be found. The function (shorten-path <node>) inverts all path-children-through-inverter of <node>, unless <node> is a leaf, in which case <node> is inverted if and only if its input variables are all inverted. As a last step (shorten-path <node>) calls itself recursively on all path-children of <node>.

```
(defun shorten-path (node)
  (let ((path-children-inv (get-path-children-through-inverter node))
        (path-children-str (get-path-children-straight node)))

    (cond
     ((and (null path-children-inv)
           (null path-children-str)) ; leaf-function reached
      (let ((invert! t))
        (dolist (signal (get-signal-sort node))
          (setq invert! (and invert! (inverted-signalp signal)))
        )
        (if invert!
            (invert-function-in-DAG node)
        )
      ))

     (t ; current node not a leaf
      (dolist (child path-children-inv)
        (invert-function-in-DAG child)
      )

      ;; beware! children could have been changed
      ;; by invert-function-in-DAG

      (dolist (child (get-path-children-through-inverter node))
        (shorten-path child)
      )
      (dolist (child (get-path-children-straight node))
        (shorten-path child)
      ))
    )
  )
)
```

7.7.5 *Resubstitution*

Like global optimising, resubstitution is mainly a question of book-keeping. If a certain resubstitution is allowed is checked with the recursive size determining algorithm of section 6.4.2. Care is taken that functions which loose all parents in the resubstitution process are removed from *FUNCTION-LIST*.

8. CONCLUSIONS

In appendix A. the results of a benchmark test are presented. They show that the whole program works well. The number of inverters between gates is in most cases about 20 - 30% of the total number of gates. Perhaps this number could still be reduced by working on the heuristics for the global optimising phase (see section 7.5.1).

The framework in which the whole technology mapping problem is split up in subproblems, each of which can be solved in a simple way, and implemented in a relatively simple Lisp function, is probably the biggest achievement of this research. The heuristics are not extensively tested yet, and probably the overall performance could still be enhanced by working on them. But new ideas can very easily be inserted into the whole program because of its high degree of modularity.

Therefore the conclusion I would like to draw here is: The program as developed here performs well. Future enhancements can be implemented easily because of the high degree of modularity.

9. APPENDIX

9.1 Appendix A: Some Benchmark Results

In this appendix the results are listed of running the benchmark set through the simplification, the decomposition and the technology mapping phase. The technology is NMOS with a *gate-size* of 3.

The decomposition restriction as can be found in the table is of the form is either "no decomp", which indicates that the decomposition step has been skipped completely, or $digit_1 digit_2 digit_3 digit_4$, which means:

$digit_1$: minimum kernel size before substitution takes place.

$digit_2$: minimum kernel amount before substitution takes place.

$digit_3$: minimum cube size before substitution takes place.

$digit_4$: minimum cube amount before substitution takes place.

The other columns in the table have the following meaning:

#gates: This is the number of standard gates used by this implementation. The inverters are not included in this number!

#inverters: This is the number of inverters needed. The first number indicates the number of inputs which must be inverted, and the second number is the number of inverters between gates.

#trans.: This is the number of transistors used by the implementation. Load transistors are not included however. Of course the total number of load transistors is equal to the sum of #gates and #inverters.

delay: This is the length of the critical path(s), expressed in unit gate delays. if there is a "+ 1" in this column, it means that one or more of the inputs which feed the critical path(s) have to be inverted first.

resubst. levels: This number indicates how many consecutive resubstitution runs were successful.

example	decomp. restr.	#gates	#inverters	#trans.	delay	resubst. levels
alu2	2222	56	10 + 18	201	6	1
alu2	2323	54	10 + 25	214	6	0
alu2	2424	51	9 + 20	201	6	0
alu2	3232	98	10 + 38	362	8	0
alu2	3333	99	10 + 38	376	10	0
alu2	4242	106	10 + 27	376	8	0
alu2	no decomp	60	10 + 16	293	6	0
alu3	2222	42	10 + 11	149	5 + 1	2
alu3	2323	38	10 + 17	155	5 + 1	1
alu3	2424	37	10 + 10	148	4 + 1	0
alu3	3232	59	10 + 19	210	7	0
alu3	3333	50	10 + 14	202	6 + 1	1
alu3	4242	56	9 + 15	209	7	0
alu3	no decomp	39	10 + 8	180	5 + 1	0
apla	2222	80	9 + 25	273	7 + 1	1
apla	2323	82	9 + 29	300	7 + 1	2
apla	2424	66	10 + 25	260	9	0
apla	3232	101	10 + 40	343	9	0
apla	3333	85	10 + 35	317	9	0
apla	4242	101	10 + 40	338	9	0
apla	no decomp	122	10 + 14	481	5 + 1	0
in6	2222	118	23 + 46	424	8 + 1	0
in6	2323	117	24 + 42	442	7 + 1	1
in6	2424	117	27 + 49	463	8	0
in6	3232	132	25 + 34	456	8	0
in6	3333	121	28 + 37	469	8 + 1	0
in6	4242	134	25 + 36	472	8	0
in6	no decomp	148	28 + 31	587	7	0
in7	2222	59	17 + 18	202	7	0
in7	2323	64	19 + 24	230	7 + 1	2
in7	2424	75	21 + 22	267	8 + 1	0
in7	3232	69	21 + 19	242	7	0
in7	3333	61	17 + 20	234	8	0
in7	4242	72	20 + 19	255	9	0
in7	no decomp	64	21 + 15	286	6	0
col4	2222	31	14 + 11	130	7	0
col4	2323	24	10 + 5	113	7 + 1	0
col4	2424	24	10 + 5	113	7 + 1	0
col4	3232	31	14 + 11	130	7	0
col4	3333	24	10 + 5	113	7 + 1	0
col4	4242	31	14 + 11	130	7	0
col4	no decomp	24	10 + 5	113	7 + 1	0

example	decomp. restr.	#gates	#inverters	#trans.	delay	resubst. levels
dc1	2222	19	4 + 6	67	3 + 1	2
dc1	2323	21	4 + 5	67	3 + 1	3
dc1	2424	16	4 + 4	64	3	0
dc1	3232	18	3 + 3	64	3 + 1	0
dc1	3333	17	4 + 3	62	3 + 1	0
dc1	4242	12	4 + 1	61	3	0
dc1	no decomp	12	4 + 1	61	3	0
dc2	2222	48	7 + 18	170	5 + 1	3
dc2	2323	50	7 + 15	185	5	2
dc2	2424	55	7 + 21	210	7	0
dc2	3232	59	7 + 13	197	7	1
dc2	3333	57	7 + 18	210	7	0
dc2	4242	62	7 + 19	226	7 + 1	0
dc2	no decomp	40	7 + 9	194	4	0
dk17	2222	51	8 + 10	163	7	2
dk17	2323	44	10 + 11	159	5 + 1	0
dk17	2424	46	9 + 17	170	7	0
dk17	3232	48	9 + 16	163	7	0
dk17	3333	48	8 + 18	173	9	0
dk17	4242	48	9 + 16	163	7	0
dk17	no decomp	85	10 + 12	304	5 + 1	0
dk27	2222	33	9 + 7	110	5 + 1	1
dk27	2323	33	9 + 10	124	5 + 1	0
dk27	2424	36	8 + 11	133	6 + 1	0
dk27	3232	34	9 + 6	121	5	0
dk27	3333	34	9 + 13	139	6 + 1	0
dk27	4242	36	8 + 6	126	5	0
dk27	no decomp	37	9 + 4	141	4 + 1	0
misg	2222	47	15 + 10	143	5	0
misg	2323	40	18 + 15	144	5	0
misg	2424	48	21 + 14	173	5	1
misg	3232	56	15 + 19	188	7	0
misg	3333	50	20 + 12	180	5	1
misg	4242	45	22 + 13	171	5	0
misg	no decomp	41	19 + 15	163	5	0
mish	2222	35	10 + 7	105	3 + 1	0
mish	2323	32	15 + 8	118	3	0
mish	2424	32	15 + 8	118	3	0
mish	3232	34	12 + 7	116	3	0
mish	3333	31	16 + 8	117	3	0
mish	4242	31	16 + 8	117	3	0
mish	no decomp	31	16 + 8	117	3	0

example	decomp. restr.	#gates	#inverters	#trans.	delay	resubst. levels
rd53	2222	18	5 + 7	75	4 + 1	1
rd53	2323	22	5 + 3	89	3 + 1	3
rd53	2424	25	5 + 2	93	4	2
rd53	3232	43	5 + 9	135	6 + 1	0
rd53	3333	32	5 + 9	132	6 + 1	0
rd53	4242	33	5 + 13	128	7 + 1	0
rd53	no decomp	17	5 + 6	93	5	0
risc	2222	56	8 + 17	146	5 + 1	0
risc	2323	50	8 + 13	141	5 + 1	0
risc	2424	46	8 + 7	139	3 + 1	1
risc	3232	56	8 + 17	154	5 + 1	0
risc	3333	47	8 + 21	152	5 + 1	0
risc	4242	54	8 + 13	161	5 + 1	0
risc	no decomp	54	7 + 9	188	3 + 1	0
sqn	2222	51	7 + 11	182	6 + 1	1
sqn	2323	55	7 + 19	195	6 + 1	1
sqn	2424	54	7 + 25	207	7 + 1	1
sqn	3232	56	7 + 16	211	6 + 1	0
sqn	3333	49	7 + 20	205	6 + 1	0
sqn	4242	51	7 + 15	204	7	0
sqn	no decomp	33	7 + 1	169	4 + 1	0
wim	2222	19	4 + 5	63	3 + 1	3
wim	2323	17	4 + 8	63	3 + 1	2
wim	2424	16	4 + 7	60	3 + 1	1
wim	3232	15	4 + 4	62	3 + 1	0
wim	3333	13	3 + 3	57	3	0
wim	4242	11	4 + 4	59	3 + 1	0
wim	no decomp	12	4 + 3	60	3	0

10. REFERENCES

[1] Smits, A.J.
MANIPULATION OF BOOLEAN FUNCTIONS.
M.Sc. Thesis. Laboratory of Automatic System Design,
Department of Electrical Engineering, Eindhoven
University of Technology, 1984.

[2] Paassen, P.T.H.M. van
OPTIMISATION OF COMBINATIONAL LOGIC.
M.Sc. Thesis. Laboratory of Automatic System Design,
Department of Electrical Engineering, Eindhoven
University of Technology, 1985

[3] Brayton, R.K. and C. McMullen
THE DECOMPOSITION AND FACTORIZATION OF BOOLEAN EXPRESSIONS.
In: Proc. 15th Int. Symp. on Circuits and Systems, Rome,
10-12 May 1982.
New York: IEEE, 1982. P. 49-54.

[4] Steele, Jr., G.L. et al.
COMMON LISP: The language.
Burlington, Mass.: Digital Press, 1984.

[5] Christofides, N.
GRAPH THEORY: An algorithmic approach.
London: Academic Press, 1975.
Computer science and applied mathematics

Eindhoven University of Technology Research Reports
Department of Electrical Engineering

ISSN 0167-9708
Coden: TEUEDE

(157) Lodder, A. and M.T. van Stiphout, J.T.J. van Eijndhoven
ESCHER: Eindhoven SCHematic EditoR reference manual.
EUT Report 86-E-157. 1986. ISBN 90-6144-157-9

(158) Arnbak, J.C.
DEVELOPMENT OF TRANSMISSION FACILITIES FOR ELECTRONIC MEDIA IN THE NETHERLANDS.
EUT Report 86-E-158. 1986. ISBN 90-6144-158-7

(159) Wang Jingshan
HARMONIC AND RECTANGULAR PULSE REPRODUCTION THROUGH CURRENT TRANSFORMERS.
EUT Report 86-E-159. 1986. ISBN 90-6144-159-5

(160) Wolzak, G.G. and A.M.F.J. van de Laar, E.F. Steennis
PARTIAL DISCHARGES AND THE ELECTRICAL AGING OF XLPE CABLE INSULATION.
EUT Report 86-E-160. 1986. ISBN 90-6144-160-9

(161) Veenstra, P.K.
RANDOM ACCESS MEMORY TESTING: Theory and practice. The gains of fault modelling.
EUT Report 86-E-161. 1986. ISBN 90-6144-161-7

(162) Meer, A.C.P. van
TMS3001C EVALUATION MODULE CONTROLLER.
EUT Report 86-E-162. 1986. ISBN 90-6144-162-5

(163) Stok, L. and R. van den Born, G.L.J.M. Janssen
HIGHER LEVELS OF A SILICON COMPILER.
EUT Report 86-E-163. 1986. ISBN 90-6144-163-3

(164) Engelshoven, R.J. van and J.F.M. Theeuwen
GENERATING LAYOUTS FOR RANDOM LOGIC: Cell generation schemes.
EUT Report 86-E-164. 1986. ISBN 90-6144-164-1

(165) Lippens, P.E.R. and A.G.J. Slenter
GALI: A Gate Array Description Language.
EUT Report 87-E-165. 1987. ISBN 90-6144-165-X

(166) Dielen, M. and J.F.M. Theeuwen
AN OPTIMAL CMOS STRUCTURE FOR THE DESIGN OF A CELL LIBRARY.
EUT Report 87-E-166. 1987. ISBN 90-6144-166-8

(167) Oerlemans, C.A.M. and J.F.M. Theeuwen
ESKISS: A program for optimal state assignment.
EUT Report 87-E-167. 1987. ISBN 90-6144-167-6

(168) Linnartz, J.P.M.G.
SPATIAL DISTRIBUTION OF TRAFFIC IN A CELLULAR MOBILE DATA NETWORK.
EUT Report 87-E-168. 1987. ISBN 90-6144-168-4

(169) Vinck, A.J. and Pineda de Gyvez, K.A. Post
IMPLEMENTATION AND EVALUATION OF A COMBINED TEST-ERROR CORRECTION PROCEDURE FOR MEMORIES WITH DEFECTS.
EUT Report 87-E-169. 1987. ISBN 90-6144-169-2

(170) Hou Yibin
DASM: A tool for decompesition and analysis of sequential machines.
EUT Report 87-E-170. 1987. ISBN 90-6144-170-6

(171) Monnee, P. and M.H.A.J. Herben
MULTIPLE-BEAM GROUNDSTATION REFLECTOR ANTENNA SYSTEM: A preliminary study.
EUT Report 87-E-171. 1987. ISBN 90-6144-171-4

(172) Bastiaans, M.J. and A.H.M. Akkermans
ERROR REDUCTION IN TWO-DIMENSIONAL PULSE-AREA MODULATION, WITH APPLICATION TO COMPUTER-GENERATED TRANSPARENCIES.
EUT Report 87-E-172. 1987. ISBN 90-6144-172-2

(173) Zhu Yu-Cai
ON A BOUND OF THE MODELLING ERRORS OF BLACK-BOX TRANSFER FUNCTION ESTIMATES.
EUT Report 87-E-173. 1987. ISBN 90-6144-173-0

(174) Berkelaar, M.R.C.M. and J.F.M. Theeuwen
TECHNOLOGY MAPPING FROM BOOLEAN EXPRESSIONS TO STANDARD CELLS.
EUT Report 87-E-174. 1987. ISBN 90-6144-174-9

(175) Janssen, P.H.M.
FURTHER RESULTS ON THE McMILLAN DEGREE AND THE KRONECKER INDICES OF ARMA MODELS.
EUT Report 87-E-175. 1987. ISBN 90-6144-175-7

(176) Janssen, P.H.M. and P. Stoica, T. Söderström, P. Eykhoff
MODEL STRUCTURE SELECTION FOR MULTIVARIABLE SYSTEMS BY CROSS-VALIDATION METHODS.
EUT Report 87-E-176. 1987. ISBN 90-6144-176-5