EINDHOVEN UNIVERSITY OF TECHNOLOGY

Generating layouts for random logic : cell generation schemes

Citation for published version (APA):
Engelshoven, van, R. J., & Theeuwen, J. F. M. (1986). *Generating layouts for random logic : cell generation schemes*. (EUT report. E, Fac. of Electrical Engineering; Vol. 86-E-164). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1986

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 22. May. 2024

Research Report

Eindhoven
University of Technology
Netherlands

Department of Electrical Engineering

Generating Layouts for Random Logic: Cell Generation Schemes

by
R.J. van Engelshoven
and
J.F.M. Theeuwen

EUT Report 86-E-164
ISBN 90-6144-164-1
ISSN 0167-9708

November 1986

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

Eindhoven The Netherlands

GENERATING LAYOUTS FOR RANDOM LOGIC:
Cell generation schemes

by

R.J. van Engelshoven

and

J.F.M. Theeuwen

EUT Report 86-E-164

ISBN 90-6144-164-1

ISSN 0167-9708

Coden: TEUEDE

Eindhoven

November 1986

COOPERATIVE DEVELOPMENT OF AN INTEGRATED, HIERARCHICAL AND MULTIVIEW VLSI-DESIGN SYSTEM WITH DISTRIBUTED MANAGEMENT ON WORKSTATIONS.
(Multiview VLSI-Design System ICD). Code: 991.
Report on activity 5.3.A: Generating layouts for random logic: Cell generation schemes.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Engelshoven, R.J. van

Generating layouts for random logic: cell generation schemes /
by R.J. van Engelshoven and J.F.M. Theeuwen. - Eindhoven:
University of Technology. - Fig.,tab. - (Eindhoven University of
Technology research reports / Department of Electrical Engineering,
ISSN 0167-9708; 86-E-164)
Met lit. opg., reg.
ISBN 90-6144-164-1
SISO 664.3 UDC 621.382:681.3.06 NUGI 832
Trefw.: elektronische schakelingen; computer aided design.

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 5.3.A: Generating layouts for random logic: Cell generation schemes.

Abstract:
Starting from a boolean expression the process of generating a linear transistor array, also called a Cell, is described for NMOS. This result is used to obtain a practical method for generating CMOS cells.

The adopted Depth First Search provides a clearly structured basic algorithm consisting of modules that can easily be adjusted or extended. In this way the cell generator may be tuned to calculate solutions in minimum cpu-time or solutions that need minimum ,area or even have fitting dimensions regarding their surrounding cells.

finally a data structure is presented that makes it possible to modify the structure of the network without changing the implemented logic function. This approach is then applied to the basic algorithm.

deliverable code: WP 5, task: 5.3, activity: 5.3.A.

date: 01 - 11 - 1986

partner: Eindhoven University of Technology

author: R.J. van Engelshoven, J.F.M. Theeuwen.

Abstract

Starting from a boolean expression the process of generating a linear transistor array, also called a cell, is described for NMOS. This result is used to obtain a practical method for generating CMOS cells. The adopted Depth First Search provides a clearly structured basic algorithm consisting of modules that can easily be adjusted or extended. In this way the cell generator may be tuned to calculate solutions in minimum cpu-time or solutions that need minimum area or even have fitting dimensions regarding their surrounding cells. Finally a data structure is presented that makes it possible to modify the structure of the network without changing the implemented logic function. This approach is then applied to the basic algorithm.

Engelshoven, R.J. van and J.F.M. Theeuwen
GENERATING LAYOUTS FOR RANDOM LOGIC: Cell generation schemes.
Department of Electrical Engineering, Eindhoven University of Technology, 1986.
EUT Report 86-E-164

Address of the authors:

Automatic System Design Group,
Department of Electrical Engineering,
Eindhoven University Of Technology,
P.O. Box 513,
5600 MB EINDHOVEN,
The Netherlands

CONTENTS

1. INTRODUCTION.. 1

2. LINEAR ARRAYS FOR NMOS AND CMOS..................... 2

3. SOME USEFUL DEFINITIONS AND THEOREMS CONCERNING GRAPHS... 6

4. REFLECTIONS OVER A STRATEGY......................... 8

5. BASIC ALGORITHM..................................... 10

6. THE INTERCHANGE FACILITY............................ 13
   6.1 A NEW DATA STRUCTURE............................ 13
   6.2 THE INTERCHANGE TOOL USED IN THE BASIC ALGORITHM... 19

7. SUPPORTING PROCEDURES............................... 21

8. SUGGESTIONS... 23

9. REFERENCES.. 24

APPENDIX A : EXAMPLE OF NETDECOMPOSITION OUTPUT..... 25

APPENDIX B : TYPICAL OUTPUT OF THE CELL GENERATOR.. 26

APPENDIX C : STORAGE AND USE OF THE PROGRAM......... 34

1. INTRODUCTION

At the laboratory ES (Automatic System Design) of the Department of Electrical Engineering of the Eindhoven University of Technology effort is made on the construction of silicon compilers. That is the design of systems that automate the design of integrated circuits. One project is concerned with the construction of a macro cell generator for combinatorial logic. This combinatorial logic can be used in larger systems to control the dataflow in the datapath. A control function of a finite state machine can be described by a function of boolean variables representing inputs and internal states of that machine.

The macro cell generator consists of a logic editor, a netdecomposition unit, a cell generator, a placement unit and a router. The logic editor first simplifies the functions by removing redundancy. Then it decomposes the functions, that is tracking them for common parts for which a new variable is introduced. To make those functions suitable for a certain technology (Nmos or Cmos) a netdecomposition is done. The next step is the implementation of these separate functions into small layout islands, called cells , consisting of nands, nors, inverters and combinations of them. The two next steps, placement and routing, speak for themselves.

Up till now only a cell generator for Nmos was available. But now that the introduction of Cmos technology at the EFFIC, the IC-fabrication laboratory at the Eindhoven University of Technology, is at hand there is the necessity to adjust the macro cell generator for this technology. There is a growing desire to come to a flexible system suited for a wide range of technologies. It was our job to write a flexible cell generator for Cmos that generates the cell layout for a given boolean function.

2. LINEAR ARRAYS FOR NMOS AND CMOS

An Nmos gate exists of a load (depletion) transistor and depending on the function to be realized some input or driver transistors. Both transistors are of the n-channel type (p-substrate) and can be constructed in the same type of diffusion. Figure 2.1. shows the circuit of a 3-input Nand gate (a), and the realization of a more complex non-optimized function f: ~(a.(b.c+d+c.(e.f+b.d))); (b), where "~" means the negation.

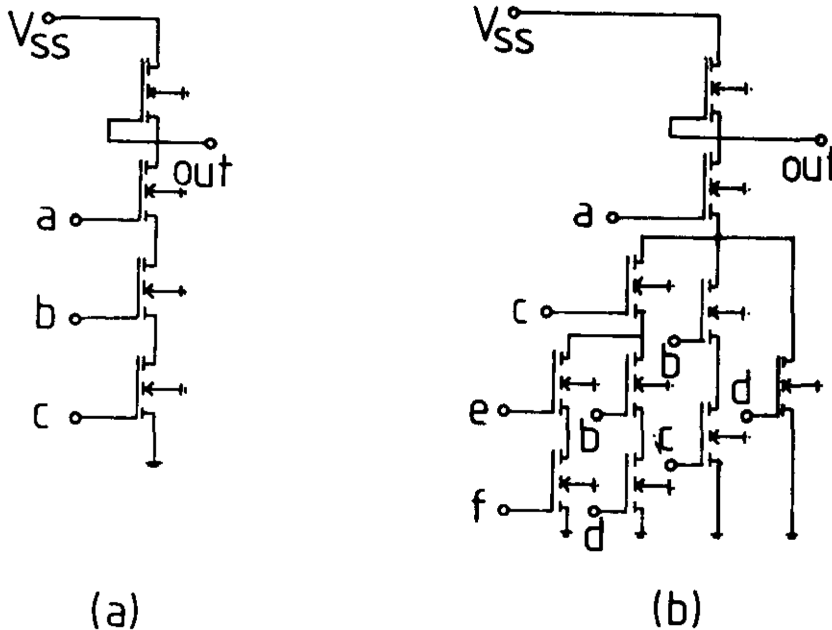

figure 2.1

The number of series and parallel connections is restricted by the used technology. The Nmos process currently running at the Effic limits the number of parallel and series connections to a maximum of three. As already mentioned in the introduction a boolean control function is transformed by the logic editor and the netdecomposition unit into a set of equations that are suited for implementation in a certain target technology. Together they realize the original boolean function. The function "f" will yield the following set of implementable equations for the current EFFIC Nmos process (figure 2.2). Appendix A gives the same results for a more complex function.

```
gat008  : (b +f e );
gat009  : (a d );
f       : (gat009 (int007' +gat008' ));
gat011  : (a c );
int007  : (gat011 );
gat008' : (gat008 );
int007' : (int007 );
```

figure 2.2

This set of equations has to be realized in the hardware of the target technology. Each function is realized in a configuration called a *cell*. Experience has thought that a *linear transistor array* is the best suited form to implement those functions. A *linear array* consists of a diffusion area over which polysilicon tracks are laid and thus forming the transistors of the circuit. Over this array aluminium interconnections are made to accomplish the network connections. To make things clear figure 2.3 shows the layout of the cell representing the function "f" of figure 2.1.b .

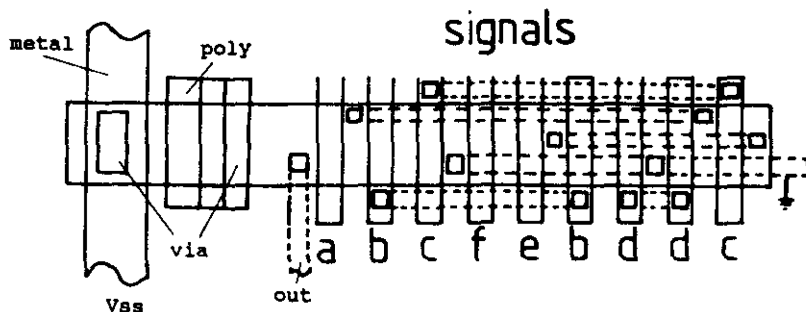

figure 2.3

The layout of such an array has to be optimized in length and width. This realization is not always possible without modifying (not changing) the network. Modification of a network means, changing the sequences in series connections, duplicating transistors, introducing transistors with grounded gates (breaks), etc.. The logic function of the

network must not be changed.

Turning to static Cmos one may notice that the most striking difference between Nmos and Cmos when observing the circuits is the number of transistors needed for both technologies (see fig. 2.4). Nmos only needs one time the number of inputs plus one (load transistor) as opposed to Cmos that needs exactly twice the number of inputs. In Cmos the gate function and its complement are realized in resp. the N-field and the P-field. Both functions are realized in two separate arrays, each of which is projected on one field. Because both functions need the same inputs a significant gain in space and complexity can be made by placing the arrays side by side and arranging the transistors so that no additional routing in the space between both arrays is needed. Figure 2.4.a shows the optimized circuit and a non-optimized cell layout (b) of the function "f" mentioned above.

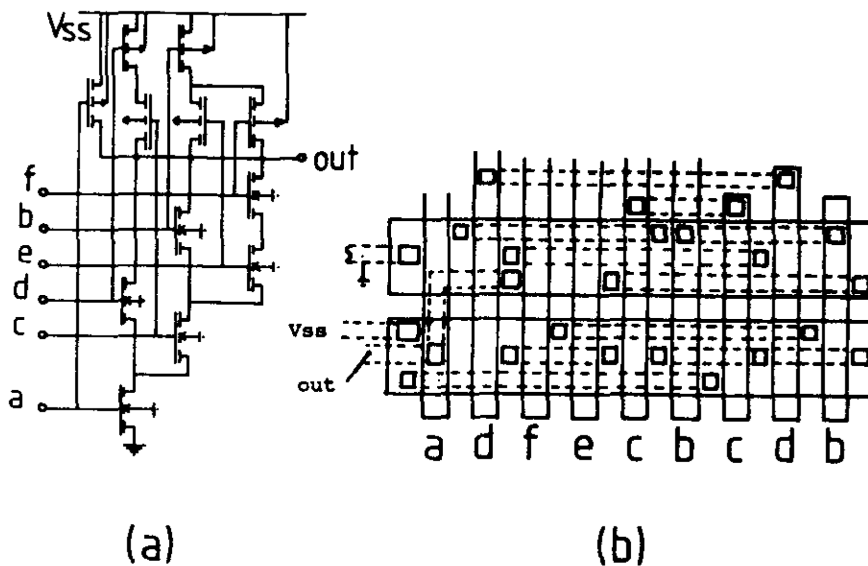

figure 2.4

A network of transistors (mathematically a graph with labeled edges) will be realized into a diffusion strip, i.e. a linear list of transistors (a linear array), in which all nodes that were connected in the original network (graph) have to be connected again by strips of metal. In future we will talk, in mathematical terms, about *intervals* instead of metal strips because they are due to the nodes in the graph. One or more metal strips can be placed behind each other in

one layer, called a *track*. If two metal strips (intervals) should have an overlap they can not be placed in the same track. They must be placed in separate tracks.

3. SOME USEFUL DEFINITIONS AND THEOREMS CONCERNING GRAPHS

As mentioned in the previous chapter a transistor network can be represented by a graph with labeled edges. Figure 3.1 gives the graph representation of the network that belongs to the function "f" that was presented earlier.

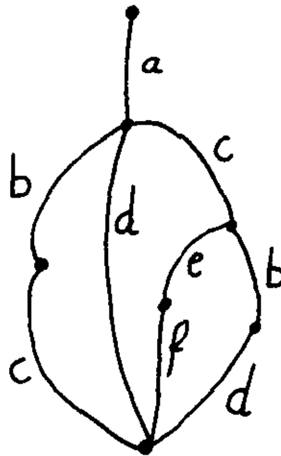

figure 3.1

In the next we will talk about graphs and edges instead of circuits and transistors. Before proceeding with the algorithms, a few items related to graph theory are defined.

Definition 1.

A path in a graph G is any sequence of edges where the final vertex of one is the initial vertex of the next one except perhaps the first and last vertex when those two are not the same.

Definition 2.

The degree of a node v in graph G, denoted degree(v), is the number of edges incident with v.

Since every edge is incident with two nodes, it contributes 2 to the sum of the degrees of the nodes. Thus we find the following result.

Theorem 1.

The sum of the degrees of the nodes of a graph G is twice the number of edges:

SUM(i){degree(v(i))} = 2 * t

Theorem 2.

In any graph the number of nodes of odd degree is even.

Definition 3.

An Eulerpath is a path which contains each edge exactly once.

If an Eulerpath exists, it means that the graph can be drawn on paper by following this path and without lifting the pen from the paper.

Definition 4.

An Eulergraph is a graph in which an Eulerpath exists.

The basic theorem on the existence of an Eulerpath is the next:

Theorem 3.

A connected, undirected graph G contains an Eulerpath if and only if the number of nodes of odd degree is 0 or 2.

4. REFLECTIONS OVER A STRATEGY

The main concern while constructing a cell is that its area is minimized. So both length and width have to be optimized. To minimize the length the following points are of importance:

- use each edge only once
- use a minimum number of breaks

For a minimum width the next rules can be mentioned:

- minimize the number of tracks
- minimize the number of intervals
- make the intervals short

Before proceeding with these observations, first a short explanation about the phenomenon called *break*. A break is realized with a transistor that is permanently blocked. The diffusion area is divided in two separate parts, where the diffusion area at the left-hand side and the right-hand side of the break-transistor each represent a different node in the graph which are not connected. The break-transistor contributes to the length of the array and in cases where a point of even degree is involved in a break it contributes to the number of intervals which may eventually increase the number of tracks.

Looking at the conditions for which the array length is minimized, we may notice a strong resemblance with the properties of an Eulerpath. The minimum number of breaks that have to be added is halve the number of points with odd degree minus one, as a path can be made in a graph with two odd points.

In fact it is easy to see that an eulerpath will yield minimum length of the transistor array. An eulerpath will however not always give minimum width. Often a large number of eulerpaths can be constructed in a graph each having different interconnections that will fit in a specific number of tracks.

During my practical assignment I have implemented the method of *local cycles* described by [Talsma]. It is a method to reduce the length of the intervals which is useful as this increases the packing of the tracks which reduces the number of tracks. I did not proceed with this idea because the algorithm took more than 2 times the amount of cpu-time than a Depth First Search (DFS) oriented algorithm. Another argument for rejecting this approach is the complexity of

the method certainly when using it in Cmos cell generation. The method of *local cycles* may be reconsidered when networks of over 20 transistors have to be handled because the time used by DFS increases more than linear with the number of edges where local cycles is linear. For these amount of transistors DFS needs additional techniques (chapter 6) which considerably increase the processing time.

With the DFS strategy I have adopted a large number of possible eulerpaths are found. Out of these solutions the path that gives minimum width or that has the shortest sequence of edges can be selected. The selection can also be made on the kind of sequence. A suitable sequence of edges may for instance reduce the interconnection length between different cells.

5. BASIC ALGORITHM

We assume the data of the 2 boolean functions (f and f')to be stored in two incidence matrices called matrix1 and matrix2. The rows represent the nodes and the columns represent the edges in the graph representation of such a function. If edge i is connected to node k and node l, element (i,k) and (i,l) are one while the rest of that column is zero. Further we choose two startpoints, both matching a node in one of the graphs. The job is now to find two identical **eulerpaths**, one in each graph.

In the proceeding descriptions we will deal with two graphs representing both boolean functions. A great deal of the variables apply to both graphs and have a postscript (1 or 2) to indicate the referenced graph. Where variables are used without postscript, both variables are intended. Here in short some variables of interest for a good comprehension of the text.

- start : initial startpoints
- matrix : the incidence matrices
- edgepath : gives the sequence of edges the path exists of
- nodepath : gives the sequence of nodes the path exists of
- point : last point that was included in nodepath
- rdegree : array that keeps the restdegree of all nodes in the graph i.e. the initial degree minus the number of edges adjacent to that node that are part of the path.

For a flowchart of the basic algorithm, refer to fig. 5.1.

Out of the startpoints or out of the points that were reached earlier a common edge is sought for. When this edge is found, it is appended to both paths stored in *edgepath*. The new nodes are appended to *nodepath*. These nodes are assigned to point1 and point2. The appropriate elements in *matrix* are reset and the restdegrees are updated. With the newly found points a new *trace operation* can be started.

When no common edge can be found, a *break* might be a solution. A break is only considered if from the newly found nodes by procedure BREAK a successful *trace operation* is possible. Breaks are only made from one odd point to another. This to limit the additional length of the path.

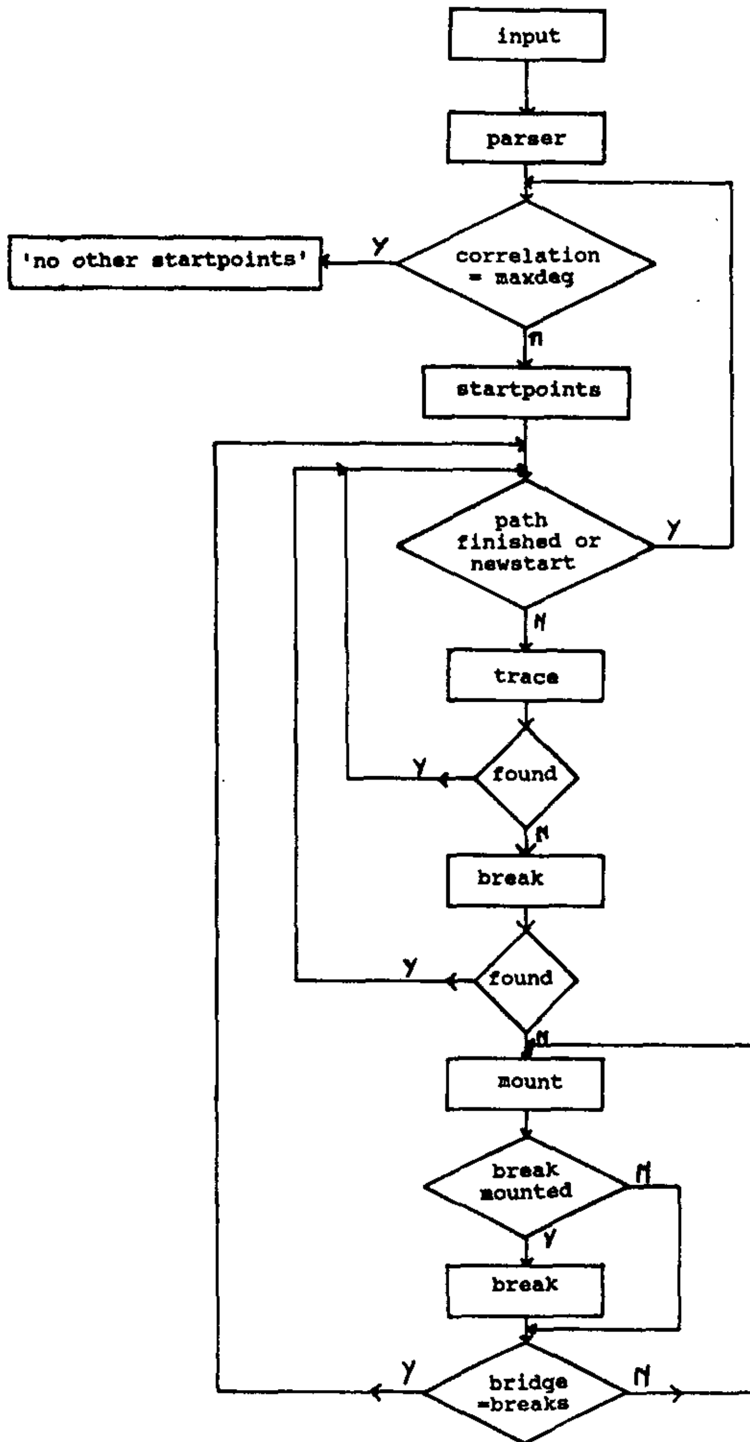

figure 5.1

Three different situations can be distinguished when making a break:

- only point1 is odd : This may result in a *break* in edgepath1 and a *space* (no transistor) in edgepath2.

- only point2 is odd : This may result in a *break* in edgepath2 and a *space* (no transistor) in edgepath1.

- both points are odd : A break may be inserted in both paths.

When neither a *TRACE* nor a *BREAK* was possible one step back is done in both paths, that is the last edges and points are deleted from edgepath and nodepath. This operation is done by procedure *MOUNT*. If a break was *mounted*, which means that point1 and/or point2 is of odd degree, a new break will be tried to make, for a *trace operation* has proved unsuccessful here. If the *break operation* is not successful the sequence is mounted one more step, otherwise a new *trace operation* can be done. And of course, when no break was mounted a new *trace operation* is done.

6. THE INTERCHANGE FACILITY

6.1 A NEW DATA STRUCTURE

With the help of breaks between odd points it is always possible to find an Eulerpath in a graph. To find however identical sequences in different graphs, though closely related, has proved difficult and in some cases impossible with the basic concept previously described. There are several techniques to enlarge the probability for a solution for this problem. One could think of doubling edges, inserting breaks between even points etc. They all have one major disadvantage, that is the additional length of the final array. *Changing the sequence in series connections* does not have this disadvantage. In a logic function the order of appearance of the factors in a product can be changed without penalty except for the loss of some time efficiency. The logic function of the network is not changed but only the appearance is different. The left and right part of the next equation realize the same function.

a.b.(c+d)+e.f.(g+h.(i+j).k) = (c+d).a.b+e.(g+(i+j).k.h).f

A main condition to implement this technique is a good knowledge of the structure of the graph. Changing sequences in series connections implies that the separate terms and factors can be identified. Information about the hierarchy of these factors and terms is needed as factors from a low level can not be interchanged with factors of a higher level. To explain what is meant with *level* : The level of a variable is defined as the number of pairs of brackets between which the variable is mentioned. So is the level of the variable "*c*" in the preceding expression higher than the level of factor "*a*".

All this information can be stored in an incidence matrix if only the two end nodes can be distinguished. The end nodes are the points in the electrical network where this is connected to the rest of the circuit. Access to the information stored in the incidence matrix implies a lot of search work in order to identify a specific factor or term. Manipulation of the stored data is even worse. It is clear that for a smooth operation on the data a better accessible data structure is required.

Therefore we observe a boolean function. Generally it consists of a sum of sub-expressions called terms. These terms may consist of other terms or a product of factors. A factor may consist of an other sub-expression or out of a signal. Note that functions of just one product are not excluded

from this observation. On the basis of this observation a new data structure or term oriented structure is introduced. A universal syntax of a boolean expression is given below :

boolean expression ::= {<term>} "+" {<boolean expression>} |
{<term>} ;

term ::= <term><opt-factor-separator><term> | <factor> ;

factor ::= {<boolean expression>} | {<signal-name>} ;

A signal-name is a legal name of a boolean variable.

The proposed data structure gives optimal access to the separate parts of the graph i.e. terms and factors. Changing the sequence of factors in a product simply means moving an element in a linked list. The data structure exists of two different elements. First there are the *index records* that point to a separate term or factor. Secondly there are the *element records* in which the elements are stored or which carry information about the hierarchy and the kind of sequence (sum or product) that is pointed at. Fig. 6.1. illustrates the two types used in the new data structure.

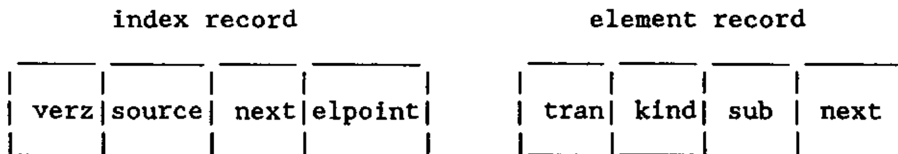

figure 6.1

The record variables of index are :

- verz : Used to determine the hierarchy with respect to another row. (see procedure findroot)

- source : An index pointer to a higher row where the reference is made to this row.

- next : An index pointer to the next row.

- elpoint : This is an element pointer that points at the first element of a row.

The record variables of element are :

- tran : If an element record contains data of an edge (transistor), this variable gives the name of the according transistor.

- kind : If the element contains data of an edge, this variable is equal to that edge number. If this element represents a compound factor it has value (-1). When this element represents a term "kind" will be zero.

- sub : This index pointer points at the row (index record) where the description of the factor or term (compound) is continued. When no compound is referenced, but just an edge, this pointer is nil.

- next : points at the next element record of the row.

To give an idea of how a boolean function looks like when it is fed into this new data format, an example (see figure 6.2) is presented for the function

$$f = a.b.(c+d.(e.f+g)+h)+i.j$$

The advantage of this new data structure is the clear hierarchical build-up so that the different levels can be easily recognized. All the levels can be easyly accessed as only one level is stored in a row. The structure can be walked through in both directions, top-down and bottom-up, because the references in both directions are unique. We always know through which lower level we reached the actual level, but also where the reference was made in this higher row.

During the description of the data structure I will say that an element has the value "x" if the *kind* part of that record is assigned "x". In this structure two kinds of rows can be distinguished. The rows in which all the elements are zero will be called the *sum-rows* as they represent an or-function. The *sub* parts of these elements point at a compound which is a term of the or-function. All the other rows are called *product-rows* as they represent an and-function. An element with value (-1) is a factor representing a compound. The *sub* part of this special factor element points at

function_ptr

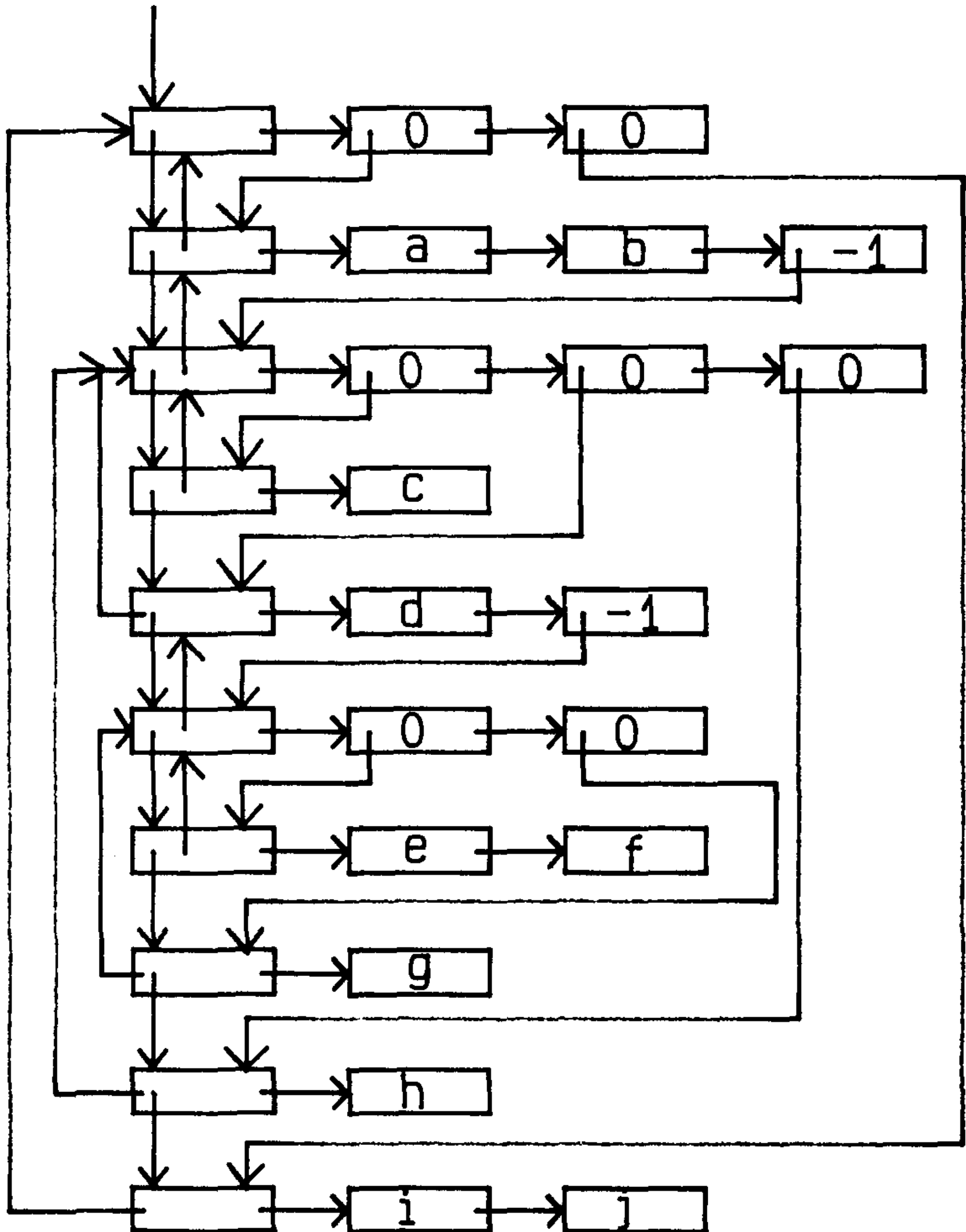

figure 6.2

this compound.

There are two elements involved with the presented interchange operation. The first is the last edge of the uncompleted path. It is called "used". The second is the element, called "object", which is tried to be moved. An interchange run is successful if at the end "*used*" and "*object*" are connected to each other.

The elements "*used*" and "*object*" are both members of a branch in the data structure, not necessarily distinct branches. A branch is usually part of a tree in which a root can be distinguished. The former tree may be part of another tree which has another root. In this way the final root of the data structure is reached. So when mounting the trees out of the elements, a number of roots is traversed. But somewhere there exists a tree for "*used*" and one for "*object*" which will have an equal root. The first root that is encountered in this way is now called "*root*".

To make things even more clear let "g" in figure 6.2 be the value of "used" and "h" the value of "object". The third row is now called the root-row of "g" and "h". The reference to "g" and "h" can be tracked back by bottom-up search. The element "h" is part of row nine, and this row is referenced from row three. The element "g" is part of row eight. This row is referenced from the second element of row six. Row six is referenced from the second element of row five. Row five in turn is referenced from the second element of row three. The second element of row three is the root of the tree that contains "g". The third element of row three is the root of the tree that contains "h".

Due to the hierarchy there always exists a row from which both "*used*" and "*object*" have a different reference. This row is called the "root". In this root-row at least two elements are subroots. One of the used-tree, the other of the object-tree. If the element is an edge, the reference is the element itself. If the root-row is a product row it is important to know whether the reference to "*used*" is situated before or after the reference to "*object*" when reading the rows from left to right. The boolean *first* is true if "*used*" was first referenced, otherwise false. The variable *position* gets a coded value for the position of "*used*" in its subtree:

```
position =  1 : used at top of the tree
            2 : used is the only element in its tree
            0 : used somewhere in between top and bottom
           -1 : used at the bottom of the tree
```

If the root-row is a sum-row the boolean *first* is of no importance. There are a few values of the pair (first,position) for which no shifting of "*object*" is useful. If "*root*" is a product-row they are:

```
first = true          position = 1 or 0
        false                   -1 or 0
```

If "*root*" is a sum-row *position = 0* will yield no useful shift. If after this selection "*object*" is still in for a switch operation, all the elements from behind *used-tree* , i.e. the tree containing "*used*" , up to and including the *object-tree* are screened for availability. If there are elements found that are already part of the path a shift operation may be impossible because the path and the data structure could become inconsistent with each other. This screening is done by procedure "*still_free*". I will illustrate all this with an example.

Let 7 - i - 1 - a - 3 be a sequence of an uncompleted path in the graph shown in figure 6.3 . The numbers in the sequence correspond with the node numbers in this graph. If we want to shift edge "f" next to edge "a", i.e. used = "a" and object = "f" the following steps are made. The root is row 2 and we see that "a" is stored in row 2 and "f" in row 7. The tree containing "f" has its top behind "a". So *first* becomes true. As "a" is the only member in its tree, its position is assigned the number 2 (connection to top and bottom). These two values mean that "f" is still in for switching. Now all elements between "a" and the "f-tree" are scanned by procedure *still_free*.

The element behind the one containing "b" is a factor pointer (kind = -1). It points at row 3. Row three is a sum-row. All the terms of this row are consecutively scanned. Successively the following edges are screened : c - d - e - f - g - h . In our example all these edges are still free for use and thus "*object*" can be switched. If "*root*" is a product-row than procedure "*switch*" evaluates the boolean *first* to determine the shift direction. If *first=true* than "*object*" must be moved up in its tree, otherwise it must be moved down. If however "*root*" is a sum-row, the value of *position* determines the shift direction. A position value of

1 means a shift up in the tree, a value of 2 a shift down. For position=2 both directions are possible, but I choose to move *"object"* up in that case. In our example *"root"* is a product-row and first is true, so "f" will be moved up in the tree.

The shift operation starts in the row containing *"object"*. Here "f" is shifted to the first place in row 7. Row 7 is referenced out of row 6 which is a sum-row. A sum-row is of no importance for the tree position of *"object"*. Row 6 is referenced out of row 5 which is a product-row. The pointer to the previous compound is moved to the first position of row 5. Row 5 is referenced from row 3 which is again a sum-row and this has no implications for the position of *"object"*. Row 3 is referenced from *"root"*. In *"root"* we notice that "b" lies next to the pointer to the previous compound and so no shift needs to be done. "f" will now be connected to "a" in the transformed graph (see figure 6.3).

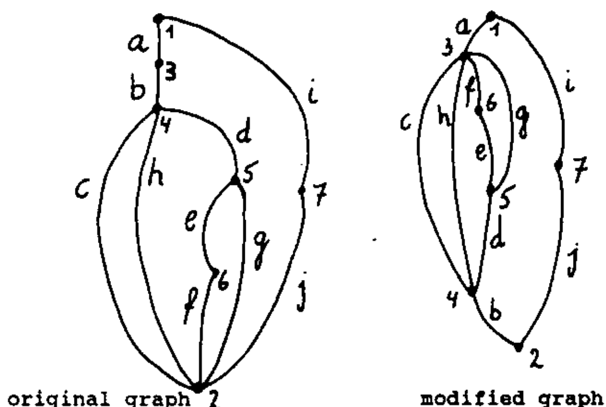

figure 6.3

6.2 THE INTERCHANGE TOOL USED IN THE BASIC ALGORITHM

The procedure *INTERCHANGE* is called from the procedure *TRACE*. Because this utility may take a considerable extra amount of time it will only be run in case no solution was found for a certain pair of startpoints applying the basic algorithm. The boolean *menu2* marks whether a *normal trace* or an *extended trace* operation incorporating the interchange

mechanism is executed.

As this program job was already taking too long I have implemented the procedure *INTERCHANGE* in a simple version. The presented implementation of "interchange" only covers a small percentage of all possible cases in which it can be applied. A more elaborate use of this mechanism will yield increased speed and performance.

Only minor changes are made in procedure *trace* to fit in the interchange mechanism. The actions performed by *trace* can be described as follows. The procedure first searches in graph1 for an adjacent edge (=object) with the last edge in the uncompleted path (=used). If no according edge is found in graph2 the procedure *interchange* is executed. This procedure searches in graph2 for an edge equal to the one found in graph1. If one is found, an attempt is made to switch the sequences in graph2 in a way that this edge becomes adjacent to "*used*". If the interchange operation was successful a new version of matrix2 is constructed for the changed graph. Because there already exists an uncompleted path the according elements in matrix2 have to be deleted. As procedure *buildmatrix* will generally change some nodenumbers, *nodetree* must be adjusted too. This adjustments on those arrays are done by procedure *adjustarray*. After this intervention of *interchange* a trace operation can be performed which will append one more edge to the uncompleted path.

7. SUPPORTING PROCEDURES

Apart from the procedures discussed above some additional procedures are applied to secure smooth operation. The input for the cell generator consists of two boolean functions, the second being the inverse of the first. A possible input could like :

```
f1 : ~(a+b.(d+e.(f+g.k.l.(h+i))));
f2 : ~(a.(b+d.(e+f.(g+k+l+h.i))));
```

These two functions are parsed by the procedure *col_parser*. The output is stored according the *term structure* discussed under chapter 6. This program was available from the group. Some additional code had to be inserted to adjust the program for its task. The input for this procedure is a file with two boolean functions, like the set described above.

The procedure *buildmatrix* is called from *col_parser* as well as from *trace*. It establishes the incidence matrix out of the term datastructure. Node number 1 is assigned to the first node in the first row of this structure and node number 2 to the last node of this row. The other nodes are assigned in order of appearance (see also example).

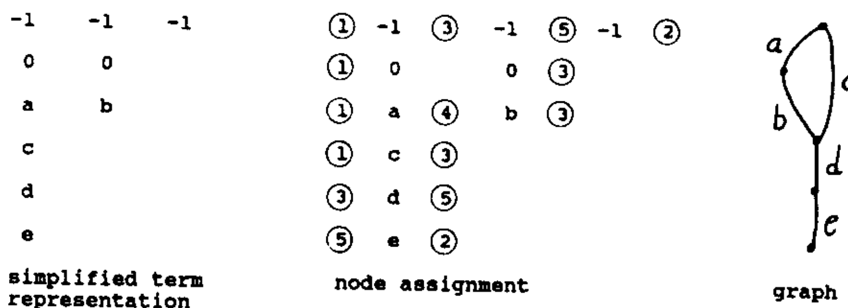

figure 7.1

A sum-row only needs a top and a bottom node, for all terms are connected between those nodes. Top and bottom node are resp. the first and the last position of the row. If the sum-row is not the first row in the term representation

structure these numbers are inherited from the source row.

In a product-row a new node number is introduced for every connection of factors within that row except for the first and the last node which are inherited from the source row or in case the product-row is the first row in the structure, they are automaticly assigned the number 1 and 2.

8. SUGGESTIONS

There are still a number of things to be done. The program is slow for large boolean expressions and has difficulty in solving them.

In general there is no need to process arrays for all the start-couples currently generated by procedure *startpoints*. As the number of tracks normally differs no more than 1 or 2 it could be sufficient to process only a certain number of solutions and choosing the best one out of them.

The current version of procedure *trace* is a first start to a powerful routine capable of tracing paths in large graphs. Two menus are used, one without the interchange utility, the other with the simpliest form of this tool. The latter only has the possibility of searching for an object in one tree and in case of success modifying that part of the tree. To enjoy the full power of the interchange technique a search and modification operation in both trees must be possible.

Because using the interchange mechanism may take a considerable amount of time when large boolean expressions are evaluated it may be interesting to process a less optimal solution but in a shorter time. The exchange between accuracy and cpu-time can be achieved by introducing more breaks than strictly necessary. One could think of first trying breaks between an odd-even pair of points and where needed inserting breaks between two even points. Notice that the length will always increase but that also the width of the cell may increase by doing so.

9. REFERENCES

[1] Harary, F.
GRAPH THEORY.
Reading, Mass.: Addison-Wesley, 1969.
Addison-Wesley series in mathematics

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

[3] Talsma, J.E.
OPTIMIZATION OF LINEAR TRANSISTOR ARRAYS. 2 parts.
M.Sc. Thesis. Department of Mathematics and Informatics,
Delft University of Technology, 1985.

APPENDIX A : EXAMPLE OF NETDECOMPOSITION OUTPUT

The function that is to be split up into implementable gates is :

```
f1: a.b.g+ c.d.f+ b.g.l+  c.d.e.i+  j.k.l+  a.c.h.k.l+  b.e+
    d.l+ e.f.g+ b.c.d+ f.i.l+ a.c.j;
```

The output of the net-decomposition unit is :

```
gat017 :~(int013 +g e +l i );
gat018 :~(int016 +int014 );
gat019 :~(int015 +d );
gat020 :~(e +int013 );
gat021 :~(int015 a +i int013 e +int014 h int016 );
gat022 :~(gat021 (f' +gat017' )(j' +gat018' ));
gat023 :~((l' +gat019' )(b' +gat020' ));
gat024 :~(gat022 +gat023 );
f1     :~(gat024);
gat026 :~(d c );
int013 :~(gat026 );
gat028 :~(a c );
int014 :~(gat028 );
gat030 :~(b g );
int015 :~(gat030 );
gat032 :~(l k );
int016 :~(gat032 );
gat020':~(gat020 );
b'     :~(b );
gat019':~(gat019 );
l'     :~(l );
gat018':~(gat018 );
j'     :~(j );
gat017':~(gat017 );
f'     :~(f );
```

APPENDIX B : TYPICAL OUTPUT OF THE CELL GENERATOR

The input for the Cell generator looks like :

```
f1 : ~(a+c.d.(e+g.h.i.j.(k.l+m)));
f2 : ~(a.(c+d+e.(g+h+i+j+m.(k+l))));
```

(f2 is the complement of f1)

```
start1=  8start2=  5
start1=  1start2=  1
start1=  1start2=  2
start1=  2start2=  1
```

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	2	1	a	a
2	1	3	c	c
3	3	2	d	d
4	4	3	e	e
5	2	4	br	sp
6	4	4	g	g
7	5	2	h	h
8	6	4	i	i
9	7	2	j	j
10	8	4	m	m
11	2	5	l	l
12	9	2	k	k
13	8	5		

number of tracks for this array= 3

Appendix B : Typical output of the Cell Generator

start1= 2start2= 2

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	2	2	l	l
2	9	5	k	k
3	8	2	j	j
4	7	4	i	i
5	6	2	h	h
6	5	4	g	g
7	4	2	d	d
8	3	3	c	c
9	1	2	sp	br
10	1	1	a	a
11	2	3	e	e
12	4	4	br	sp
13	2	4	m	m
14	8	5		

number of tracks for this array= 4

start1= 1start2= 3
start1= 2start2= 3

start1= 2start2= 4

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	2	4	e	e
2	4	3	d	d
3	3	2	c	c
4	1	3	a	a
5	2	1	br	br
6	4	4	g	g
7	5	2	h	h
8	6	4	i	i
9	7	2	j	j
10	8	4	m	m
11	2	5	l	l
12	9	2	k	k
13	8	5		

number of tracks for this array= 4

start1= 2start2= 5

Appendix B : Typical output of the Cell Generator

start1= 3start2= 2

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	3	2	c	c
2	1	3	a	a
3	2	1	sp	br
4	2	2	l	l
5	9	5	k	k
6	8	2	j	j
7	7	4	i	i
8	6	2	h	h
9	5	4	g	g
10	4	2	d	d
11	3	3	br	sp
12	4	3	e	e
13	2	4	m	m
14	8	5		

number of tracks for this array= 5

start1= 3start2= 3

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	3	3	d	d
2	4	2	g	g
3	5	4	h	h
4	6	3	i	i
5	7	4	j	j
6	8	3	m	m
7	2	5	sp	br
8	2	1	a	a
9	1	3	c	c
10	3	2	br	sp
11	4	2	e	e
12	2	4	l	l
13	9	5	k	k
14	8	4		

number of tracks for this array= 5

Appendix B : Typical output of the Cell Generator

start1= 4start2= 2

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	4	2	d	d
2	3	3	c	c
3	1	2	sp	br
4	1	1	a	a
5	2	3	e	e
6	4	4	g	g
7	5	2	h	h
8	6	4	i	i
9	7	2	j	j
10	8	4	m	m
11	2	5	l	l
12	9	2	k	k
13	8	5		

number of tracks for this array= 4

start1= 4start2= 3

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	4	3	d	d
2	3	2	c	c
3	1	3	a	a
4	2	1	sp	br
5	2	3	e	e
6	4	4	g	g
7	5	2	h	h
8	6	4	i	i
9	7	2	j	j
10	8	4	m	m
11	2	5	l	l
12	9	2	k	k
13	8	5		

number of tracks for this array= 5

start1= 4start2= 4

Appendix B : Typical output of the Cell Generator

start1= 5start2= 2

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	5	2	g	g
2	4	4	e	e
3	2	3	a	a
4	1	1	sp	br
5	1	2	c	c
6	3	3	d	d
7	4	2	br	sp
8	5	2	h	h
9	6	4	i	i
10	7	2	j	j
11	8	4	m	m
12	2	5	l	l
13	9	2	k	k
14	8	5		

number of tracks for this array= 5

start1= 5start2= 4

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	5	4	g	g
2	4	3	e	e
3	2	4	m	m
4	8	5	k	k
5	9	3	l	l
6	2	5	sp	br
7	2	1	a	a
8	1	3	c	c
9	3	2	d	d
10	4	3	br	sp
11	5	3	h	h
12	6	4	i	i
13	7	3	j	j
14	8	4		

number of tracks for this array= 4

Appendix B : Typical output of the Cell Generator

start1= 6start2= 2

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	6	2	h	h
2	5	4	g	g
3	4	2	d	d
4	3	3	c	c
5	1	2	sp	br
6	1	1	a	a
7	2	3	e	e
8	4	4	br	sp
9	6	4	i	i
10	7	2	j	j
11	8	4	m	m
12	2	5	l	l
13	9	2	k	k
14	8	5		

number of tracks for this array= 5

start1= 6start2= 4

start1= 7start2= 2

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	7	2	i	i
2	6	4	h	h
3	5	2	g	g
4	4	4	e	e
5	2	3	a	a
6	1	1	sp	br
7	1	2	c	c
8	3	3	d	d
9	4	2	br	sp
10	7	2	j	j
11	8	4	m	m
12	2	5	l	l
13	9	2	k	k
14	8	5		

number of tracks for this array= 5

Appendix B : Typical output of the Cell Generator

start1= 7start2= 4

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	7	4	i	i
2	6	3	h	h
3	5	4	g	g
4	4	3	e	e
5	2	4	m	m
6	8	5	k	k
7	9	3	l	l
8	2	5	sp	br
9	2	1	a	a
10	1	3	c	c
11	3	2	d	d
12	4	3	br	sp
13	7	3	j	j
14	8	4		

number of tracks for this array= 4

start1= 8start2= 2

length of cel	node in graph1	node in graph2	transistor in array1	transistor in array2
1	8	2	j	j
2	7	4	i	i
3	6	3	h	h
4	5	4	g	g
5	4	3	e	e
6	2	4	m	m
7	8	5	k	k
8	9	3	l	l
9	2	5	sp	br
10	2	1	a	a
11	1	3	c	c
12	3	2	d	d
13	4	3		

number of tracks for this array= 5

Appendix B : Typical output of the Cell Generator

startl= 8start2= 4

length of cel	node in graphl	node in graph2	transistor in arrayl	transistor in array2
1	8	4	br	sp
2	4	4	g	g
3	5	2	h	h
4	6	4	i	i
5	7	2	j	j
6	8	4	m	m
7	2	5	sp	br
8	2	1	a	a
9	1	3	c	c
10	3	2	d	d
11	4	3	e	e
12	2	4	l	l
13	9	5	k	k
14	8	4		

number of tracks for this array= 5

startl= 8start2= 5

startl= 9start2= 2

length of cel	node in graphl	node in graph2	transistor in arrayl	transistor in array2
1	9	2	l	l
2	2	5	m	m
3	8	4	j	j
4	7	2	i	i
5	6	4	h	h
6	5	2	g	g
7	4	4	e	e
8	2	3	a	a
9	1	1	sp	br
10	1	2	c	c
11	3	3	d	d
12	4	2	br	sp
13	8	2	k	k
14	9	5		

number of tracks for this array= 4

startl= 9start2= 5
no other startpoints

APPENDIX C : STORAGE AND USE OF THE PROGRAM

The directory of the Cmos-Cell-Generator is called Cmos_cell. It is located under /users/rob_e/stage . The files *cel.p* and *main.p* are similar and contain the pascal source. The files *main* and *cel* are the respective executable versions. They need a file like *inputexample* as their input. The output is directed to *cel.out*. The complete directory structure is given below.

```
Cmos_cell:

a.out           cel             cel.out         cel.p
change          chtest.con      chtest.typ      chtest.var
colparse        inputexample    main            main.p
parser          spath

Cmos_cell/change:

findroot        interch.var     interchange.h   still_free
switch          switchable      verzameling

Cmos_cell/colparse:

C_commhent.h    addtokwtab.h    col_parser.h    comment.h
error.h         finalscan.h     getch.h         handle_esc.h
identifier.h    initscan.h      kwtab.h         lex.h
list.h          nested_comm.h   number.h        options.h
peeknextch.h    scanner.h       stringconst.h   symbuf.h
trans.h         trans_esc.h

Cmos_cell/parser:

col_parser.con  col_parser.typ  col_parser.var  col_struc.con
col_struc.typ   col_struc.var   cor.con         cor.typ
cor.var         error.con       error.var       general.typ
kwsyms.con      kwtab.con       kwtab.var       lex.con
lex.ext         lex.typ         lex.var         list.con
list.var        scanner.con     scanner.var

Cmos_cell/spath:

adjustarray.h   break           buildmatrix.h   copyarray2.h
exist           mount           readtree        resetarray2.h
startpoints     trace           tracks          writematrix.h
```

ISSN 0167-9708
Coden: TEUEDE

(144) Dijk, J. and A.P. Verlijsdonk, J.C. Arnbak
DIGITAL TRANSMISSION EXPERIMENTS WITH THE ORBITAL TEST SATELLITE.
EUT Report 84-E-144. 1984. ISBN 90-6144-144-7

(145) Weert, M.J.M. van
MINIMALISATIE VAN PROGRAMMABLE LOGIC ARRAYS.
EUT Report 84-E-145. 1984. ISBN 90-6144-145-5

(146) Jochems, J.C. en P.M.C.M. van den Eijnden
TOESTAND-TOEWIJZING IN SEQUENTIËLE CIRCUITS.
EUT Report 85-E-146. 1985. ISBN 90-6144-146-3

(147) Rozendaal, L.T. en M.P.J. Stevens, P.M.C.M. van den Eijnden
DE REALISATIE VAN EEN MULTIFUNCTIONELE I/O-CONTROLLER MET BEHULP VAN EEN GATE-ARRAY.
EUT Report 85-E-147. 1985. ISBN 90-6144-147-1

(148) Eijnden, P.M.C.M. van den
A COURSE ON FIELD PROGRAMMABLE LOGIC.
EUT Report 85-E-148. 1985. ISBN 90-6144-148-X

(149) Beeckman, P.A.
MILLIMETER-WAVE ANTENNA MEASUREMENTS WITH THE HP8510 NETWORK ANALYZER.
EUT Report 85-E-149. 1985. ISBN 90-6144-149-8

(150) Meer, A.C.P. van
EXAMENRESULTATEN IN CONTEXT MBA.
EUT Report 85-E-150. 1985. ISBN 90-6144-150-1

(151) Ramakrishnan, S. and W.M.C. van den Heuvel
SHORT-CIRCUIT CURRENT INTERRUPTION IN A LOW-VOLTAGE FUSE WITH ABLATING WALLS.
EUT Report 85-E-151. 1985. ISBN 90-6144-151-X

(152) Stefanov, B. and L. Zarkova, A. Veefkind
DEVIATION FROM LOCAL THERMODYNAMIC EQUILIBRIUM IN A CESIUM-SEEDED ARGON PLASMA.
EUT Report 85-E-152. 1985. ISBN 90-6144-152-8

(153) Hof, P.M.J. Van den and P.H.M. Janssen
SOME ASYMPTOTIC PROPERTIES OF MULTIVARIABLE MODELS IDENTIFIED BY EQUATION ERROR TECHNIQUES.
EUT Report 85-E-153. 1985. ISBN 90-6144-153-6

(154) Geerlings, J.H.T.
LIMIT CYCLES IN DIGITAL FILTERS: A bibliography 1975-1984.
EUT Report 85-E-154. 1985. ISBN 90-6144-154-4

(155) Groot, J.F.G. de
THE INFLUENCE OF A HIGH-INDEX MICRO-LENS IN A LASER-TAPER COUPLING.
EUT Report 85-E-155. 1985. ISBN 90-6144-155-2

(156) Amelsfort, A.M.J. van and Th. Scharten
A THEORETICAL STUDY OF THE ELECTROMAGNETIC FIELD IN A LIMB, EXCITED BY ARTIFICIAL SOURCES.
EUT Report 86-E-156. 1986. ISBN 90-6144-156-0

(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
TMS32010 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