GM : a gate matrix layout generator

tu
Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

GM:
A Gate Matrix Layout Generator

by
G.J.P. van Lieshout
and
L.P.P.P. van Ginneken

EUT Report 87-E-179
ISBN 90-6144-179-X
September 1987

Eindhoven University of Technology Research Reports
EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven The Netherlands

ISSN 0167- 9708 Coden: TEUEDE

GM:
A gate matrix layout generator

by

G.J.P. van Lieshout

and

L.P.P.P. van Ginneken

EUT Report 87-E-179
ISBN 90-6144-179-X

Eindhoven
September 1987

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

(Multiview VLSI-design System ICD)

code: 991

DELIVERABLE

Report on activity: 5.3.D: Implement totally integrated cell generator and place- and route scheme.

title: GM: a gate matrix layout generator

Abstract: We present a gate matrix cell generator which can layout any nMOS circuit. The circuit is specified as a netlist of parametrizable transistors. The size of the transistors can be freely specified thereby giving the opportunity of optimizing parameters as power, speed and fanout. Also shape, pinout and design rules can be parametrized. The pinout can be determined while building the gate matrix structure, by simply extending certain gates or nets to the correct side. The design rule parameters can be satisfied by a simple two dimensional grid-based compaction algorithm. The horizontal connections are realized in the metal layer and are called the nets. The vertical connections are realized in poly silicon and are called the gates. We used a new two dimensional folding algorithm, to improve the layout density and to manipulate the shape of the cell. Two dimensional folding allows nets to be assigned to the same row, aswell as gates assigned to the same column. The area used for small logic examples is smaller on average (upto 40% smaller) than the area used by a pluri cell style random logic generator.

deliverable code:	WP 5, *task:* 5.3, *activity* D.
date:	21-08-1987
partner:	Eindhoven University of Technology
authors:	G.J.P. van Lieshout and L.P.P.P. van Ginneken

This report was accepted as a M.Sc. Thesis of G.J.P. van Lieshout 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 1 January 1987 to 27 August 1987 and was supervised by ir. L.P.P.P. van Ginneken.

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Lieshout, G.J.P. van

GM: a gate matrix layout generator / by G.J.P. van Lieshout and L.P.P.P. van Ginneken. - Eindhoven: University of Technology, Faculty of Electrical Engineering. - Fig., tab. - (EUT report, ISSN 0167-9708; 87-E-179)
Met lit. opg., reg.
ISBN 90-6144-179-X
SISO 663.42 UDC 621.382:681.3.06 NUGI 832
Trefw.: elektronische schakelingen; computer aided design.

CONTENTS

LIST OF FIGURES

Acknowledgements :

Before I start, I want to thank Lukas van Ginneken and Jos van Eijndhoven. Without their support and suggestions, I would never have come so far. I also want to thank Reinier van den Born, who made the laser jet layout pictures shown in the appendices.
There is one other group of people I should not forget. I want to thank all other students of the Automatic System Design group for never missing a single chance to delay my work.

Abstract

Lopez and Law introduced a new layout style in 1980: the gate matrix layout style. This matrix is made of intersecting rows and columns. Often, the columns are implemented in the polysilicon layer and the rows in both the metal and the diffusion layer; the transistors are situated on the intersections of rows and columns.

In this report, a gate matrix layout generator, GM, will be discussed. GM can generate layouts with any size of transistors; GM will always come up with a feasible result and GM allows the user to manipulate the shape of the final layout.

We will present the algorithms used in GM for signal ordering and folding and discuss the heuristics used for net placement. We will describe the addition of the power lines to the matrix and we will describe the compaction of the layout.

At the end, we will show some promising results generated by GM and compare them with layouts obtained from other layout generators.

1. General Introduction

One of the current projects of the Automatic System Design group at the Eindhoven University deals with the application of the stepwise refinement technique in the layout design part of a silicon compiler.
The stepwise refinement technique was first described by Wirth [WIRT71] for programming purposes. Starting with a clear problem statement the problem is progressively redefined. Each decision should leave enough freedom to following stages to satisfy the constraints it created and at the same time rearrange the available data such that further meaningful decisions can be made in the next step.

The principles of stepwise refinement obviously apply to any complex design task based on a top-down strategy. It can also be used for layout generation *[GINN84]*.
Say we want to generate a layout, consisting of several *modules*, defined as functional layout parts with a flexible shape. The size and shape of a module are constrained by the amount and type of circuitry that have to be accommodated into the module. These limitations can be found in the *shape constraints*.
Using the stepwise layout refinement technique, we start with the generation of a *floorplan*. First the modules are appointed to one out of two cells. Cells are collections of modules. They have a relative position; cell *a* can be situated to the right of cell *b*. The decision of which cell is suited for a module is primarily based on the resulting wiring length between the modules. Next, the modules within one cell are divided again, into smaller cells. This process continues until every cell contains only one module.
When the floorplan is complete, we know the relative position of every module; we know in what part of the layout it is going to be situated; we know the neighbouring modules; we know the *topology* of the layout. The exact shape of the layout is still unknown at this stage since we only have the shape constraints to work with. The geometrical details of the floorplan can now be determined.
We compute the shape constraints of all cells by adding the shape constraints of the different cells and modules it contains. This way, we will get the shape constraints for the total layout and are able to choose a certain shape for it. Then we work our way down again, specifying a shape for each cell and module.
The stepwise layout refinement technique is quite the contrary of other placement techniques. These work with totally specified modules, difficult to handle because of their inflexibility. In most cases, there is no possibility for adapting the modules to their environment. Using the stepwise layout refinement technique, the exact geometries of the modules are determined only after the generation of the floorplan.
If we want to exploit the benefits of this layout generation technique completely, we should be able to generate modules with the property of having very flexible shape constraints. In that case, the shapes of different modules can be well adapted to each other and cells containing several modules, can be made efficiently.

In recent years, a new layout style was introduced: the gate matrix layout style. Apart from being efficient, this style can be made to have the right shape flexibility.

To exploit the gate matrix layout style, a program, called GM, to generate gate matrix layouts has been developed. GM should generate the modules needed by the stepwise layout refinement technique. There were two demands set for GM:

1)The generated layout would have to be reasonably efficient.

2)The shape of the layout should be flexible.

Chapter 2 discusses the gate matrix layout style. The chapters 3 to 11 describe the different techniques used in GM to generate a gate matrix layout. Results obtained with GM are shown in chapter 12, followed by some suggestions for continuation in chapter 13. Finally, some concluding remarks are made in chapter 14.

2. The gate matrix layout style

The gate matrix layout style was introduced by Lopez and Law [LOLA80] in 1980. It can be regarded as a generalization of the earlier developed Weinberger layout style [WEIN67].
The gate matrix layout problem has received some attention in the past few years and various algorithms ([WING82], [WING83], [JTLI83], [WIHU85], [DELA87]...) have been designed to automate the layout procedure. In this chapter I will introduce the gate matrix layout style and give a definition of the gate matrix problem (compare [WIHU85]).

2.1 Introduction

In the vertical direction of the gate matrix, the *gates* are situated in the different columns. If we only allow one gate in a single *column*, we get something like figure 1.

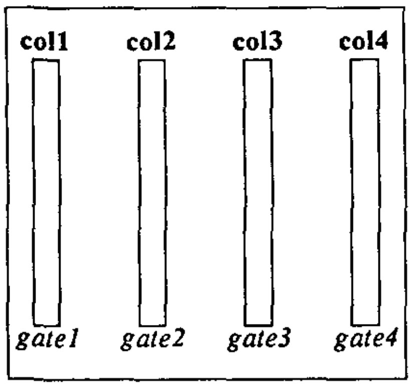


Figure 1. Unfolded gates

The gates will be implemented in the polysilicon layer. They will serve the dual role of transistor gates and interconnection.
There may be a gate for every *signal* present in the circuit or just for a distinct subset of all the signals. This depends on how we represent the circuit with the different gate matrix elements.
If we allow more than one gate into a single column, we could get the gate ordering shown in figure 2. Placing more than one gate into a single column is called *folding* (compare PLA folding).

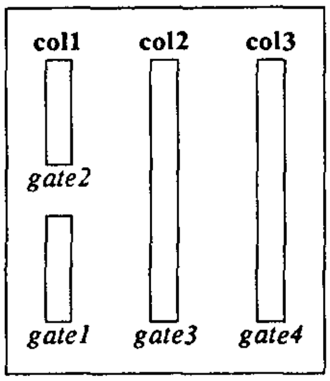


Figure 2. Folded gates

What we want to realize next, is a transistor. Figure 3 shows a possible gate matrix layout for a transistor. The gate of the transistor is situated in column 2. The drain and the source of the transistors are connected with gates 1 and 3 by two *nets*. Nets are usely implemented in the metal layer. What we call drain or source is unimportant for the whole transistor is symmetric.

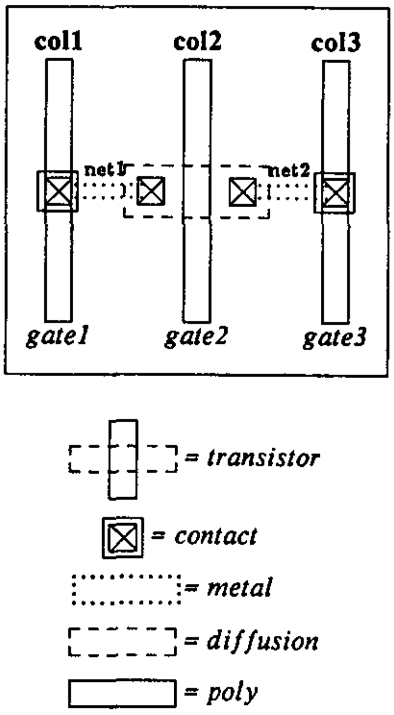
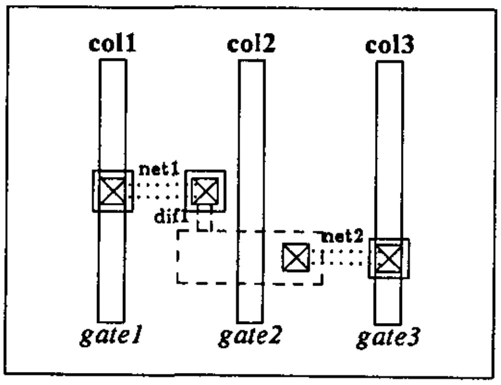


Figure 3. Possible realization for a transistor

In the last figure, the nets were placed into the same *row* of the gate matrix. In that case, the transistor will always be placed in the same row also. We have to choose a position for the transistor if the two nets are situated in different rows. Look at figure 4; the resulting circuit is the same for all four. Only the position of the gate matrix elements near column 2 varies.



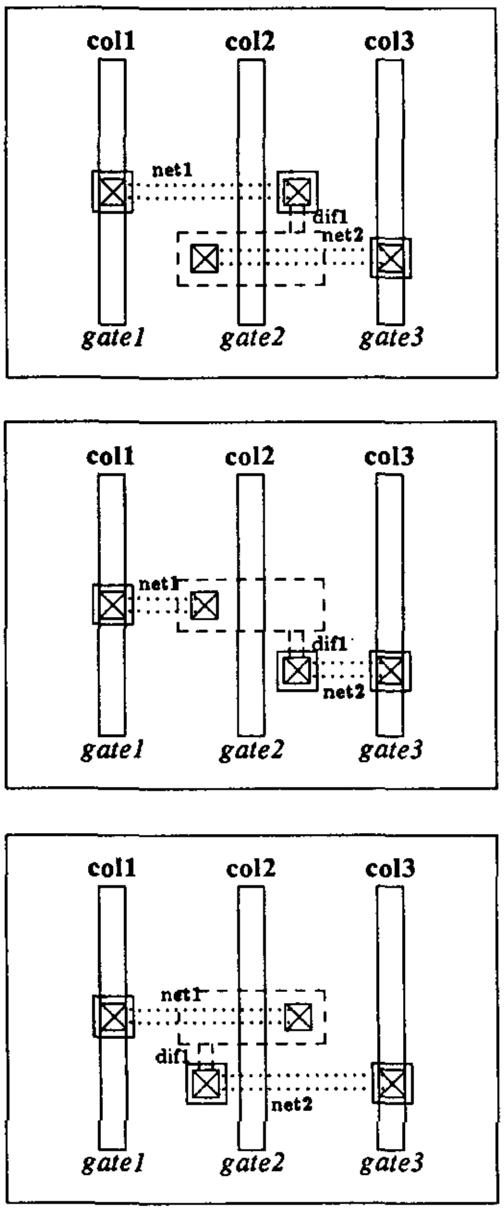


Figure 4. Four different implementations of a transistor

We use a diffusion run to get from one row to the other. This diffusion can be placed on either side of the transistor gate. If we allow no more than one diffusion run for every transistor, the transistor will have to be placed in one of the two rows containing a net.
Contrary to the situation of figure 3, in this case the layout is not completely specified after the net placement. The layout generator will have to decide upon where the diffusion runs and the transistors are going to be placed.

In the figures 3 and 4, the gate signal of the transistor was always situated in the middle column. An example of a gate signal in an outer column is shown in figure 5.

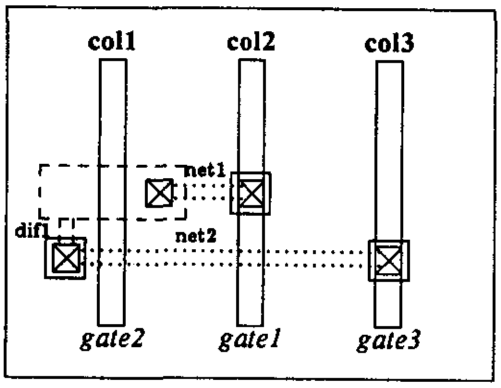


Figure 5. Example of a gate situated in an outer column

Not all nets are implemented in the metal layer. Figure 6 shows an example of a net (net 2) implemented in the diffusion layer.

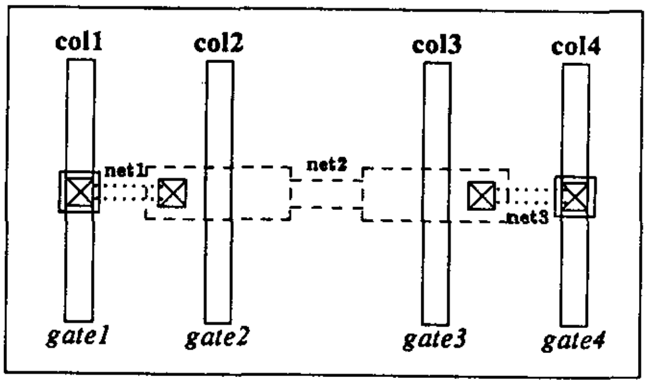


Figure 6. Example of a net implemented in the diffusion layer

We will gain some improvement of performance by implementing net 2 in the diffusion layer because of loosing two metal/diffusion contacts we would have had otherwise.

Having read this paragraph, the reader should have gained enough insight into the gate matrix layout style to understand the definition of the optimalization problem given in the next paragraph.

2.2 Definition of the problem

One of the inputs for GM is a description of the circuit to be generated. The circuit is described on the transistor level. The transformation of this description into a layout can be divided into two separate problems:

1) First, we have to obtain a circuit description using gate matrix elements only.

2) Next, these elements have to be ordered efficiently.

Out of the first step, we get a description of the circuit using elements of the following sets:

$T = \{ t_i \mid 1 \leq i \leq \textit{number of transistors} \}$: set of transistors

$G = \{ g_i \mid 1 \leq i \leq \textit{number of gates} \}$: set of gates

$N = \{ n_i \mid 1 \leq i \leq \textit{number of nets} \}$: set of nets

We also get information about the relations between these elements. This information can be found in the next Incidence matrix:

Define: Incidence matrix I[number of gates][number of nets]:

I[i][j] =

-1 : there is a connection between gate i and net j; no transistor is present.

0 : there is no connection between gate i and net j.

1..(number of transistors) : on the intersection of gate i and net j a transistor is present with transistor number I[i][j].

Apart from this circuit description, there are two other inputs to the placement problem:

1) The structure of a gate matrix.

2) Details about the technology in which the layout is going to be generated.

ad 1)
The program has to know certain details about the structure of a gate matrix. In this description the following sets are defined:

$C = \{ c_i \mid 1 \leq i \leq \textit{number of columns} \}$: set of columns of the gate matrix

$C' = \{ c'_i \mid 1 \leq i \leq \textit{number of C columns} + 1 \}$[1] : set of columns, situated in between the polysilicon columns of C

$R = \{ r_i \mid 1 \leq i \leq \textit{number of rows} \}$: set of rows of the gate matrix

ad 2)
While generating the gate matrix, we have to know details about the technology in which the gate matrix is going to be realized. It will specify facts like which layers are allowed to overlap or what the distance between two metal wirings should be to avoid short circuiting... This technology description is the third input.

One last gate matrix element has to be mentioned: the diffusion runs.

$D = \{ d_i \mid 1 \leq i \leq \textit{number of transistors} \}$: set of diffusion runs in the gate matrix

Which transistors will cause a diffusion run is determined during the net placement.

1. The first column of C' is situated at the left of the first column of C; the last column of C' is situated at the right of C thus C' has one element more than C.

Knowing the three inputs for the placement problem, we are now able to define the placement problem. The gate matrix layout is completely specified if we have determined the next four functions:

1) The gate assignment function *f* assigns the different signals of the set G, to the columns:

$f: G \rightarrow C$

2) The net assignment function *h* assigns nets to the rows of the gate matrix :

$h: N \rightarrow R$

In the previous paragraph we discussed the problem of the diffusions. Consider a transistor whose drain and source are connected to nets m and n respectively. If *h(n) <> h(m)*, there will be a vertical diffusion run between *h(n)* and *h(m)*. For these diffusions, we define a third function:

3) The diffusion assignment function *k* assigns diffusion runs to the intercolumn area of the gate matrix :

$k: D \rightarrow C'$

One last function is needed to complete the description. We have to choose the positions of the different transistors.

4) The transistor assignment function *l* assigns transistors to the rows of the gate matrix.

$l: T \rightarrow R$

A generated layout is said to be realizable if all the diffusion runs defined by *k* are realizable without *collisions*. Say we have two diffusion runs: diffusion 1 from net *dif1_begin* to net *dif1_end*, and diffusion 2 from net *dif2_begin* to *dif2_end* (*h(difx_begin) < h(difx_end)*). A collision occurs if:

1) *k(dif1) = k(dif2)*

2) *h(dif1_begin) >= h(dif2_end)* and *h(dif1_end) <= h(dif2_begin)*
or
h(dif2_begin) >= h(dif1_end) and *h(dif2_end) <= h(dif1_begin)*

If we take the "unfolded" approach, the optimal gate matrix layout problem can be stated as follows:

> **Given a set of transistors T together with the set of distinct gates G and the set of nets N, find functions f,h,k and l such that in the layout :**
>
> **1) the number of rows is minimum.**
> **2) the layout is realizable (no diffusion collisions).**

The layout generator we developed is able to place more than one signal into a column. If we allow folding, the problem should be stated in a more general way:

> **Given a set of transistors T together with the set of distinct gates G and the set of nets N, find functions f,h,k and l such that in the layout :**
>
> **1) the area used is minimum.**
> **2) the layout is realizable (no diffusion collisions).**

Most layout generators use a two-stage approach to the problem. First the function *f* is optimized and afterwards *h* is optimized. Some try to optimize the two functions in one stage [DEVA86]. I chose for the two-stage approach because it is generally not as time consuming as the second approach and because more literature is available about this approach.
However, the probability of choosing a function *f'* with a bad column order for the resulting *h* is probably smaller in the one-stage approach because of the closer relation between *f* and *h*.

If *f* and *h* are determined, it is quite easy to determine satisfying functions *k* and *l*, as long as they still exist. The functions *f* and *h* may disable the possibility of creating functions *k* and *l* causing no diffusion collisions. While determinating *f* and *h*, we will have to keep the realization of *k* and *l* in mind.

The purely on efficiency based problem descriptions given above, can be extended with several extra constraints e.g. a maximum width or height or some kind of aspect ratio. In the algorithms we will discuss, the user is able to manipulate the shape of the gate matrix during the determination of *f*.

3. Structure of a Gate Matrix Layout Generator

This chapter discusses the structure of our gate matrix layout generator GM. The layout generator is subdivided into several functional blocks. Figure 7 shows a diagram of these blocks .

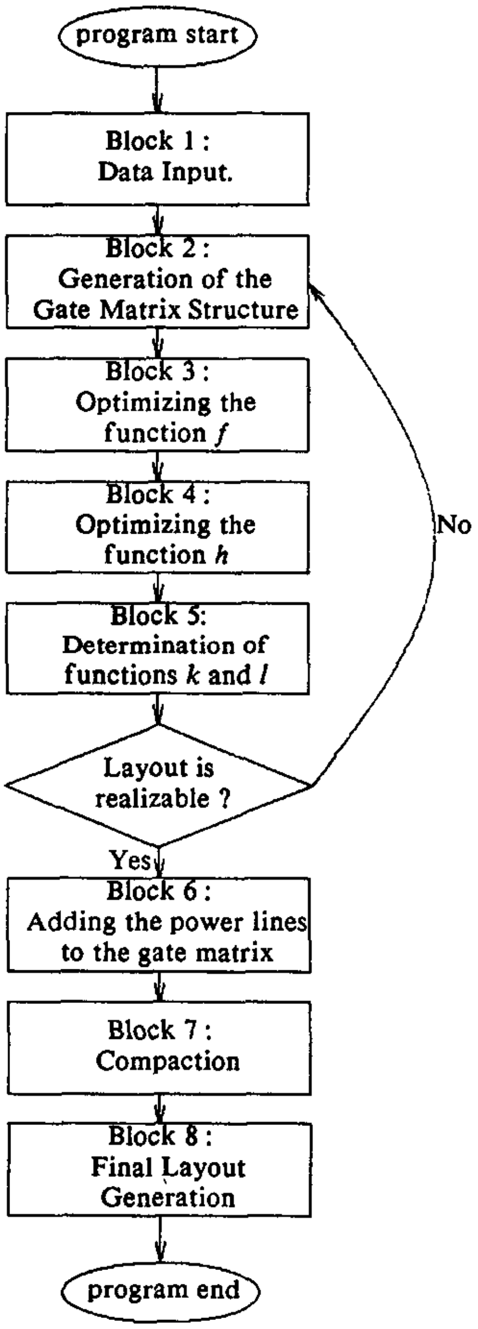

program start
Block 1 : Data Input.
Block 2 : Generation of the Gate Matrix Structure
Block 3 : Optimizing the function *f*
Block 4 : Optimizing the function *h*
Block 5: Determination of functions *k* and *l*
Layout is realizable ?
No
Yes
Block 6 : Adding the power lines to the gate matrix
Block 7 : Compaction
Block 8 : Final Layout Generation
program end

Figure 7. Main Program Structure

1) We start with loading in the necessary data. The layout generator checks the data format of the input.

2) Block 2 generates a gate matrix structure for the circuit. It determines what signals are going to be represented by nets, what signals are going to be represented by gates, what extra nets have to be added for output signals etc. . Chapter 4 discusses this block in more detail.

3) If the two-stage approach for the optimization of the gate matrix is taken, the next step is the optimization of the function *f*. This is a very important part of the layout generator. A bad gate order can enlarge the layout area with a factor 2 or even more. Chapter 5 discusses different approaches to this problem .
If we allow more than a single gate into one column, *f* becomes a bit more complicated. An extension of *f* for this purpose, is discussed in Chapter 6 on column folding.

4) The second stage of the two-stage approach is the optimization of the function *h*. We place the different nets into the rows of the gate matrix.
Chapter 7 discusses the approach for net placement in an unfolded gate matrix and in chapter 8 this approach is expanded into one suitable for a gate matrix with folded gates.

5) Chapter 9 discusses the determination of *k* and *l*. If GM generated an unrealizable layout (a layout with diffusion collisions), it will have to start anew at block 2. This time, we would like to have an increased probability of generating a realizable layout. Chapter 9 also deals with these questions concerning the realizability of the layout.

6) For reasons to be explained in chapter 4, many gate matrix layout generators place the power lines after the complete placement of the signal containing part of the gate matrix. The "normal" gate matrix nets do not supply the power to the circuit. Chapter 10 shows how the positions of the power lines can be determined.

7) Moving towards the end, the final coordinates of every element of the gate matrix are determined. This " compaction step " is discussed in chapter 11.

8) The final step is the actual generation of the layout. This is more a matter of solid bookkeeping than of great algorithmic expertise. If all details about the gate matrix are exactly computed in the previous part, the gate matrix is uniquely described.
In our implementation of GM, only the center coordinates of every gate matrix element arrive at block 8. The coordinates of the different element parts still have to be computed. Although this block is not really all that simple, it is not interesting enough for a detailed discussion.

4. Generating the gate matrix structure

Before any optimization methods can be used, the circuit has to be represented by gate matrix elements. This gate matrix structure is generated in three stages:

The first stage deals with all the transistors. Every transistor has three connections with the outer world. We generate a single gate for every single signal.
One transistor will generate (at most) two nets, each net containing one half of the transistor. Thus, one transistor will result in the gate matrix elements given below:

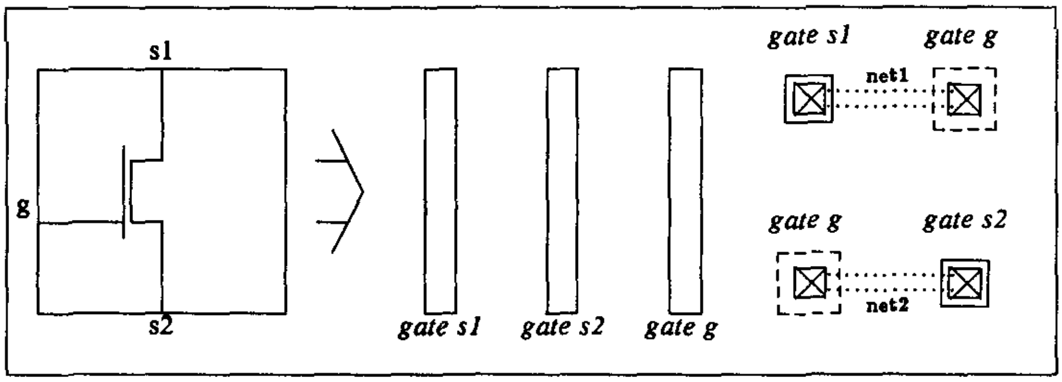


Figure 8. Transformation of a transistor into a layout structure

In the final layout, the different components are able to realize a transistor as shown in chapter 2.

The second step has to remove some overhead produced in the first step. We can delete a gate out of a column if:

-The gate is not used as a transistor gate.

-The gate does not have to be connected to "the outside world".

-The gate is connected with two nets.

The deletion is shown in figure 9.

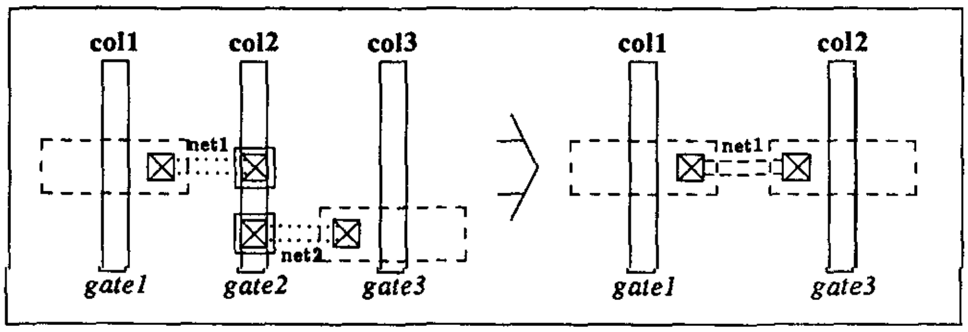


Figure 9. Deleting a superfluous gate

As a result of this step, we now have nets connecting two transistors. No such nets were generated by the first step.

The third step generates extra nets for output signals. If the user wants to have an output to the right or the left side of the gate matrix, an extra net is generated, connecting that specific gate and the right or left border of the gate matrix[2].

Note that all the generated nets have exactly two terminals. They can be divided in 3 categories:

0T: nets with on one side a transistor piece and at the other side a poly/metal contact. These nets are generated in step 1.
TT: nets with a transistor piece on both sides. These nets are generated in the second step.
00: nets with on one side a poly/metal contact, and at the other side a terminal. These nets are generated in the third step.

The reader might wonder if there are not any "but's" for the gate matrix structure just generated. And indeed, there is one. Look at the following example:

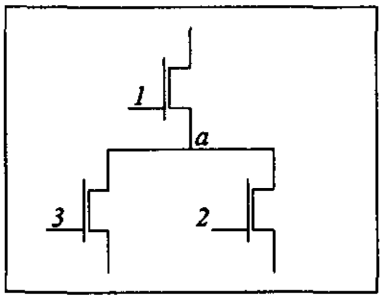


Figure 10. Circuit example

Signal *a* will result in a gate placed in a column and three nets will leave from this column. Say, the gate representing signal *a* is placed into column 2, and the gates representing the transistor gates are placed into columns 1,5 and 6. The three nets leaving from column 2 are connected with columns 1,5 and 6 respectively. After the placement of the first two nets, signal *a* is already present as far as column 5. Generating the third net, this information is not used: not a net from 5 to 6 is generated but a net from 2 to 6. This disadvantage becomes particularly clear if we have signal connected to many other signals.

The caused disadvantage may seem bigger then it really is:

1) We can reduce the disadvantage by taking a different approach to some nets. Chapter 10 discusses the special approach we used for the power supply (a heavily connected "signal").

Note : Because of this special approach to the power supply, no nets resulting from connecting a transistor to the power supply will be found in the gate matrix structure. These connections are added to the

2. If the user wants to have a certain signal as an output to the bottom or top side of the gate matrix, we simply extend the gate to that specific border of the gate matrix.

gate matrix afterwards (see chapter 10). A consequence of this approach is the fact that there is only one net present in the gate matrix structure, containing a transistor piece for a transistor connected to the power supply.

2) In a technology with only one metal layer, both sides of most transistors will have to be reached by nets placed in this metal layer. It will not be possible to continue a net, coming from one direction, into the other direction because a net is already present at the other side of the transistor.

Apart from the nets connected to the power supply, there is one other type of nets that will not be found in the gate matrix structure. In this case it concerns nets connecting a transistor gate with the source or drain of the same transistor. We remove these nets from the gate matrix structure and mark the transistor. In the layout, we will place a diffusion/poly contact to realize the connection.

At this stage, we have determined the gate matrix structure. Now we know the different elements of the gate matrix, which order can be optimized in the next parts of the layout generator.

5. The gate order

The first optimization problem is to determine the gate order in the columns. The order generated in the previous chapter (gate 1 into column 1, gate 2 into column 2 ...), does not have to be efficient at all. At first, only one gate is placed into a column. Placing more than one gate in a column is seen as a separate optimization problem which will be discussed in the next chapter.
We want to minimize the number of rows of the gate matrix, needed by the nets. Although not equal, this problem is highly correlated with the problem of finding a gate order resulting in a minimal total net length.

Probably because the importance of this problem, most of the literature on gate matrix layouts deals with this optimization problem. After a brief description of several possible solution methods, I will discuss the algorithms used in GM.

5.1 Some ordering methods

Probably the most productive author on gate matrix layout generators is Omar Wing. Since the early eighties, he writes about this subject. He started with an approach to this linear gate array problem, similar to the one dimensional logic gate assignment introduced by [OHMO79]. First he generates a matrix containing all connections of nets and gates. Look at the next matrix[3]:

$$\begin{array}{lcccc} & \text{gate1} & \text{gate2} & \text{gate3} & \text{gate4} \\ \text{net1} & 0 & 1 & 0 & 1 \\ \text{net2} & 1 & 0 & 0 & 1 \\ \text{net3} & 1 & 0 & 1 & 1 \\ \text{net4} & 0 & 1 & 1 & 0 \end{array}$$

In this matrix, we can not see the net intervals clearly. The matrix does not show that net 1 is also positioned at col3. We can change this by filling up the matrix : substitute on every row, the 0's situated between 1's by 1's. Now the matrix becomes:

$$\begin{array}{lcccc} & \text{gate1} & \text{gate2} & \text{gate3} & \text{gate4} \\ \text{net1} & 0 & 1 & 1 & 1 \\ \text{net2} & 1 & 1 & 1 & 1 \\ \text{net3} & 1 & 1 & 1 & 1 \\ \text{net4} & 0 & 1 & 1 & 0 \end{array}$$

This last matrix has the "consecutive 1's property": in every row the ones are grouped together.
If in one column of the matrix, two nets have 1, this means that if we would take the gate order of the columns in the matrix, those two nets would overlap at that particular column. The matrix is a representation for an interval graph [WIHU85]: the columns of the matrix are the vertices and there is an edge from vertex *a* to *b* if they have a 1 situated in the same column.
Nets which have a 1 positioned at the same column, form a clique in the interval

3. This matrix resembles the Incidence matrix of chapter 2. The matrix shown above can be obtained out of the Incidence matrix by substituting a one for every non zero element in I.

graph. The number of tracks needed in the final layout is equal to the largest clique number thus equal to the largest number of 1's positioned in a column. Now we can state the optimization problem:

> Find a permutation of the columns such that if each row without the consecutive ones property is filled with ones, the largest number of ones positioned in any column is minimized.

Unfortunately, it is shown in [KAFU79] that this problem is NP-hard.
Omar Wing used several heuristics described in [WING82], [WING83] and [WIHU85] to find a reasonable result. Columns are placed one by one, at each stage trying to keep the number of necessary 0 to 1 substitutions as small as possible.

A some what different approach is used by [JTLI83]. He uses a different matrix to describe the problem but he also has to solve the problem of filling a matrix in order to get the consecutive ones property.
In [DEKR87], it is shown that there is a family of problems for which the ratio of the number of tracks, resulting from the two optimization methods described above, and the minimum number of tracks is unbounded.

Another approach is described by Leong [LEON86]. He used an algorithm based on simulated annealing. A temperature schedule, a cost function and several types of moves are described.

As mentioned in chapter 3, some optimization methods use a two dimensional placement to obtain the gate order and the gate folding in one step. In GENIE [DEVA86], Devadas and Newton use an algorithm again based on simulated annealing, to obtain a 2 dimensional order. Although the results shown look very promising, time complexity will become a problem for larger layouts. It would be nice to implement a two dimensional approach into GM in the near future and compare the results with the results obtained by the algorithm of paragraph 5.2.

The first algorithm I implemented, was the one described by Omar Wing in one of his more recent publications [WIHU86]. The gate order was based on the approximation algorithm described by Asano [ASAN82]. Some changes were made to this basic algorithm, for example Wing's algorithm uses the information about the size of the different components.
If we compare the results obtained by this algorithm, with the results from the algorithm we will discuss in the next paragraph, we have to decide in favor of the second algorithm. Wing only looks at the number of new nets he is introducing while placing a new gate. He does not take into account the number of finishing nets. We would need more time to determine the exact cause of the difference in performance between the two algorithms but at a first glance, this looks the main reason.

5.2 A Kang based algorithm for gate ordering

The algorithm finally implemented in GM is based on an algorithm described by Kang [KANG83]. This linear ordering algorithm was already used for standard cell and gate array layout. The algorithm is given below:

1) Read in the information about the circuit and put all the gates into OUT.

2) Select the most lightly connected gate from OUT. (This first column is called the seed of the placement).

3) Move the selected gate from OUT or ACTIVE to IN, and all gates connected to it from OUT to ACTIVE.

4) If ACTIVE is empty, go to 2. Otherwise select a gate from ACTIVE and go to 3.

5) Repeat 3-4 until OUT is empty.

Figure 11 should illustrate the different groups. The gates in IN are already placed. The ones in ACTIVE are possible candidates for placement and the rest is situated in OUT.

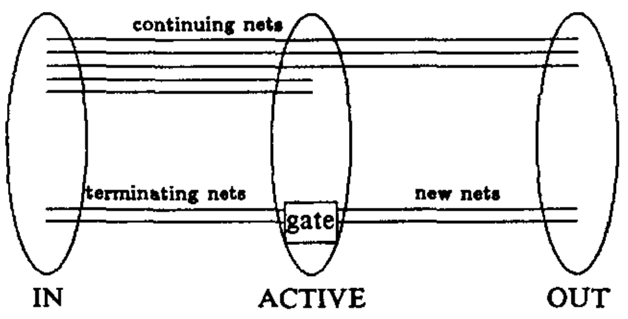


Figure 11. Illustration of the different gate groups

The algorithm works very fast; O(col*col). This is partly due to the fact that only gates situated in ACTIVE are candidates for placement, not all the unplaced gates. We could take the groups ACTIVE and OUT together; the algorithm becomes more simple and a better performance will be the result (we remove an extra constraint). It will increase the time usage of the algorithm.

In step 4 of the algorithm, a gate is chosen. Kang defines the *net gain* : the *net gain* is the number of new nets minus the number of terminating nets if a certain gate would be placed next.
The selection rules described by Kang are:

1) First, select a gate with minimum *net gain.*

2) For a tie, select one with larger number of terminating nets.

3) For a tie, select one with larger number of continuing nets[4].

4) If tie again, select lighter one (least number of connected nets).

In figure 12 , two realizations of a transistor are shown. It may be clear that we prefer the left situation. The right situation uses more diffusion space. Can we incorporate this fact into the selection rules?
The answer is Yes (of course): we do not want to place a gate functioning as a transistor gate, if both source and drain are not placed yet. A function, say *two_track()*, should compute the number of these drain source conflicts for a certain gate, would this gate be placed next. If *two_track(gate1) > two_track(gate2)* , we prefer *gate2* to be placed instead of *gate1*.
The remaining difficulty is where this selection rule should be fitted into the old

4. I did not implement this selection rule because if two gates introduce the same number of terminating nets, they will always have the same number of continuing nets.

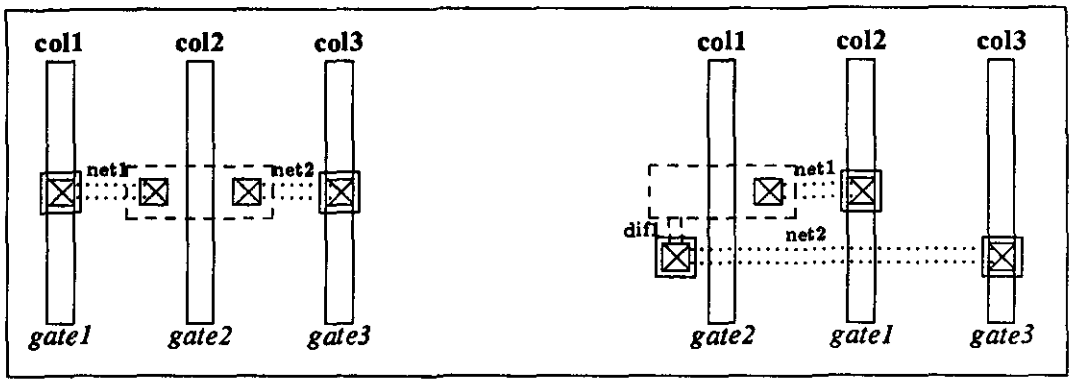


Figure 12. *Two realizations of a transistor*

selection rules. In GM, we choose to integrate it into the first selection rule, which becomes:

1) Select gate with Minimum *net gain + two_track(gate).*

I would probably be wise if the weight of *two_track* in this selection rule could be varied; one could think of an input variable specifying the weight of a gate source conflict in terms of an extra *net gain. Two_track()* will add the input variable to his total for every gate source conflict a column placement would introduce.

5.2.1 Seed selection

Step 2 of the Kang algorithm chooses the most lightly connected gate from OUT to start with. It may be clear that we do not want to start with a heavily connected gate, but does a lightly connected gate always result in an efficient placement?
I tested the placement algorithm using different seeds and found that the placement depends very strongly on the seed. In general, it is true that the best results are obtained starting with a relatively lightly connected seed. But, it is also true that the difference in total net length between two placements both starting with a 'most lightly connected seed', can be very large. In some examples, differences in total net length up to 30% were reached.
For this reason, we placed an extra option into GM. GM will then try out all most lightly connected seeds and continue with the one resulting in the smallest total net length.

Note: If the user has specified terminals at the left side of the gate matrix, we do not have the problem of choosing a seed. In that case the "column", connected with all nets realizing an output at the left side, will be the seed.

5.2.2 Improving the result

We were still not satisfied with the result obtained. There should be some extra improvement possible. This time, the time complexity was of no importance; these extra steps should be optional. After some experimental work, the user could use these options to get a final result. Two strategies are implemented:

1) The first thing we did is, if the algorithm has decided what gate should be placed, an extra step decides whether this gate should be placed at the back of all the gates placed until now or in front of these gates. This decision is based on total net length.

2) We also thought of improving the result by performing swap operations; we started with trying to swap every gate with one of the adjacent columns and a considerable improvement of the total net length was obtained. Continuing this idea, we came up with the next swapping routine:

```
BEGIN

 WHILE ( new_net length < old_net length )
 {
    old_net length = new_net length;
   FOR (dist = 0 ; dist <= MAX_SWAP_DIST ; dist++)
        WHILE (new_netl < old_netl)
            FOR (c = 1 + dist ; c < lastcol ; c++)
               {
                     old_netl = new_netl;
                     swap (col[c-dist],col[c]);
                     temp_netl =the net length now obtained;
                         IF (old_netl < temp_netl)
                               swap (col[c-dist],col[c]);
                         ELSE
                                    new_netl = temp_netl;
               }
    new_net length = new_netl;
 }

END
```

lastcol = number of columns.

Notice that the algorithm is purely based on net length and does not consider the diffusion size of the different transistors or the number of tracks the gate matrix is going to use. So in a way, we spoil the former result.

The swapping routine does not always result in a decreased number of tracks but, certainly for larger layout examples, the improvement was drastic. For a large unfolded gate matrix example, the number of tracks was reduced from 39 to 29.

We also tried to achieve improvement by using mirroring routines. We mirrored a certain number of adjacent columns. They proved very useful working at the traveling salesman problem. Nevertheless, probably due to the different connectivity structure of the problem[5], the total net length resulting from these routines had a worse average and a larger spread than the results from the swapping routines.

5. In the traveling salesman problem every element is only connected to two other elements. If we mirror a certain group of adjacent elements, only two connections between this group and all other elements change. Mirroring a group of gates in the gate matrix will generally affect more than just two nets. It will also affect nets starting somewhere in the middle of the group and connected to elements outside the group.

6. Column folding

What we have up till now, is a linear ordering of gates in the columns, every gate using a single column. If we want to fold different gates in one column, we will have to establish what columns are allowed to be folded[6].
There are a few reasons why two gates can not be folded into one column:

- If two gates are in- or output gates at the same side (bottom or top) of the gate matrix, the gates can not be placed into one column.
- If two gates are interconnected by a net, the two gates can not be placed into the same column (the net would have to be folded).
- For reasons to be explained in the chapter on net placement with column folding, we do not want to fold gates connected to the same *net path*. A *net path* is a sequence of nets interconnected by diffusion runs.

We can generate a matrix FOLD using the facts stated above:

FOLD[gate1][gate2] = TRUE : if only one pair of gates of the gate matrix is going to be folded, it is allowed to fold the gates gate1 and gate2.

FOLD[gate1][gate2] = FALSE : otherwise.

Taking care of the matrix FOLD, does not guarantee that we will be able to place all nets into the gate matrix. This is due to the following fact:

Say gate a and b are placed in the same column and gate a is placed above gate b. All nets connected to gate a will have to be placed before any net connected to gate b can be placed. Only then we know the row in which gate b can start.

The following example might put things into a clearer perspective:
We have 4 columns (a,b,c,d) and 3 nets (1,2,3). Their relations are given in the table below:

	a	b	c	d
1	-	-		
2		-	-	
3	-	-	-	-

Explanation of the figure: e.g. net 3 is connected to the gate a and the gate d. Recall that all the nets are two terminal nets.

Knowing the connections, we can generate the matrix FOLD:

6. The reader may wonder why the gate ordering and the gate folding are separated into two different parts of the layout generator. It is true indeed, that the resulting ordering would probably be better if the intergrated gate placement was used but then again this could be much more time consuming.

FOLD				
	a	b	c	d
a	0	0	1	0
b	0	0	0	1
c	1	0	0	0
d	0	1	0	0

Everything looks fine if we fold gates a and c into column 1, and gate b and d into column 2. But what happens if we start to place the nets:
First, we place net 1. So gate a has started in column 1 and gate b has started in column 2. Next, we want to place net 2. We can make a connection to gate a, but not to gate d because gate b is not placed completely yet. So we skip net 2 and try to place net 3. Again no luck; gate c can not be used because of the incomplete gate a. So there is no placeable net and the net placement will fail.

We can display the represent the situation by a graph G_fold. G_fold is made out of:

nodes: one for every gate.

labeled directed edges: a directed edge from node a to node b means that placement of nets connected to gate b can only occur after complete placement of all the nets connected to node a. The edge is labeled with the number of the column that caused it.

labeled undirected edges: an undirected edge from node a to node b indicates that the gates represented by the nodes are connected to the same net path. The placement of a net represented by an edge between node a and b depends on both nodes. The edge is labeled with the net number.

In figure 13, the G_fold graph for the folding chosen above is shown.

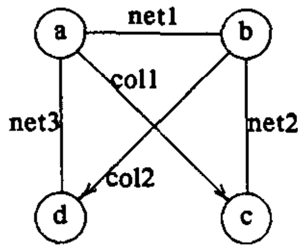


Figure 13. The G_fold graph for the example

The matrix FOLD excludes several gates from folding but does not guarantee successful placement if we fold gates which are allowed to be folded by FOLD. In figure 14, possible foldings are represented by dashed directed edges.

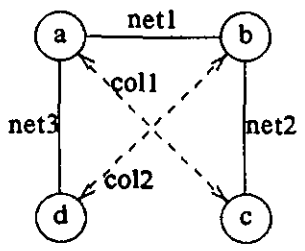


Figure 14. Possible foldings for the example

If we want to perform as many foldings as possible, we have to find a graph G_fold with as many directed edges as possible and still describing a placement, guaranteed to succeed.
In the next two paragraphs, two possible approaches to this problem will be discussed. The approach discussed in 6.1 will be able to find the optimal G_fold but is also very time consuming and for reasons to be explained in paragraph 6.1, the generated layout may still be very unefficient. Paragraph 6.2 describes a more simple approach, not as time consuming but then again also not as smart as the first approach.

6.1 The folding problem 1

Take another look at figure 13. The reader may wonder how G_fold displays a failing placement. It can be found using the next algorithm:

1) Label all the nodes with no incoming directed edges; these gates can be placed right away.
2) Label all nets connected to two labeled nodes.
3) Try to label the unlabeled nodes. A node may be labeled if all the connected undirected edges are labeled and if all the directed edges pointing at the node come from a labeled node.
4) If there are no edges or nodes labeled in the last two steps and not all nodes and edges are labeled yet, the folding indicated by the graph will result in a failing net placement.
If all edges and nodes are labeled, the folding indicated by the graph will result in a successful net placement. If not all edges and nodes are placed yet, goto step 2.

If we apply the algorithm to the example given in figure 13, we get:

1) label node a and b
2) label net 1
3) no new nodes labeled
4) goto 2
2) no new nets labeled
3) no new nodes labeled
4) --> failing net placement

If we apply the algorithm to the example given in figure 15, we get:

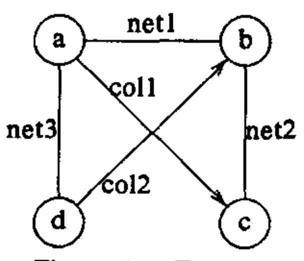


Figure 15. Example of a graph indicating a successful net placement

1) label a and d

2) label net 3

3) label node b

4) goto 2

2) label net 1

3) label node c

4) goto 2

2) label net 2

3) no new nodes labeled.

4) --> successful net placement.

What is the complexity of the search. Say the number of nodes is *n* and the number of edges is *e*. In the worst case, every time we apply step 2 and 3 only one additional node is labeled. This way, we cycle O(*n*) times through the loop. Rule 3 is the most time consuming step. Every time all unlabeled nodes have to be examined for possible labeling. A node can be connected to *e* edges at most so we will have to perform *e* comparing operations at most. Applying step 3 will thus cos t O($n * e$). This results in a total complexity of O($n * n * e$).

Having determined a search algorithm, we could try out all possible combinations of gate foldings and choose the one resulting in most foldings, still representing a successful net placement. This way, we will find the gate order with most foldings for sure. Apart from being very time consuming, there is another disadvantage to this method. Look at the configuration in the next figure:

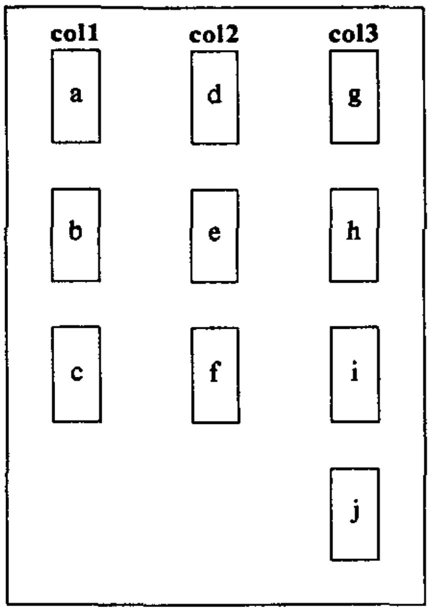


Figure 16. Example of unefficient folding

If gates a and d only have connections with one gate from col 3, say gate j, and a and d are placed at the top of col 3, we would like gate j to be placed at the top of column 3 also. If gate j, like in the figure, is placed at the bottom of col3, gates a and d will become much longer than necessary.

We could avoid this situation by choosing the gate order in a column during the net placement. If we have placed gates a and d, nets from these gates will need gate j, and gate j will be placed at the top of col 3. The exact mechanism is described in chapter 8. As a consequence of this approach we can only determine what gate is going to be placed in what column during the folding. The order within a column is not determined yet. This approach is described in the next paragraph.

6.2 The folding problem 2

If we do not know the ordering of the gates in a column, the directed edges in G_fold become undirected edges. Look at fig 17.

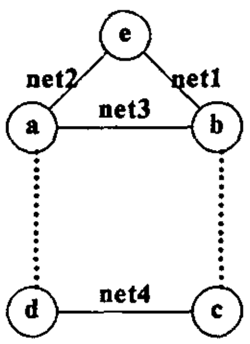


Figure 17. G_fold with undirected edges

This figure corresponds to the four possible orderings shown in figure 18:

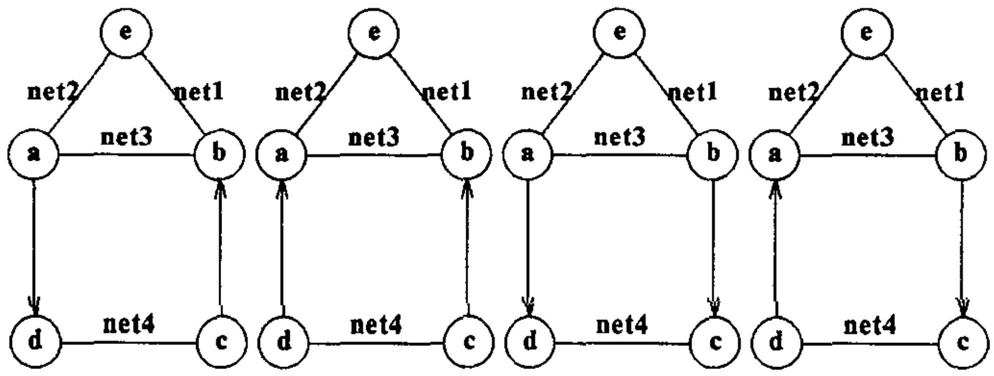


Figure 18. Possible realizations of fig.17

If we want to be sure of a succeeding net placement, no possible gate ordering should result in a failing net placement. Out of the four possible realizations of fig 18, two result in a failing net placement, so the situation of figure 17 will have to be forbidden[7].

How can we find out if a folding is legal ?
We will change the fold graph a bit. If gates are folded into the same column, we will represent those gates by one node. We also substitute edges connected to the same two

7. Note that if we forbid this situation, we also throw away two legal foldings.

nodes by one edge, at each node labeled with the number of gates the nets, the edge represents, are connected to. E.g. a new edge representing one old edge will be labeled with 1-1. An edge representing two nets, connected to different gates out of the same column, will be labeled with 2-2....
The graph of figure 17 now changes into the graph shown in figure 19.

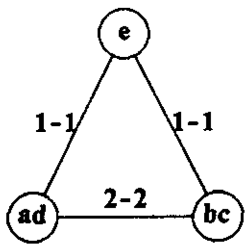


Figure 19. New G_fold

What kind of structures are forbidden in this new G_fold graph. Having read the beginning of this chapter, it may be clear that we do not allow an edge with label 2-2 or higher between two nodes.
Another situation is shown in figure 20.

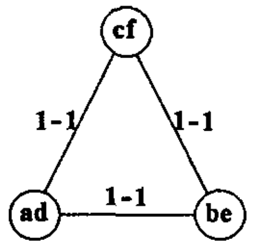


Figure 20. Example of a new G_fold graph

The graph in figure 20 represents 8 possible foldings. One of them is shown in figure 21.

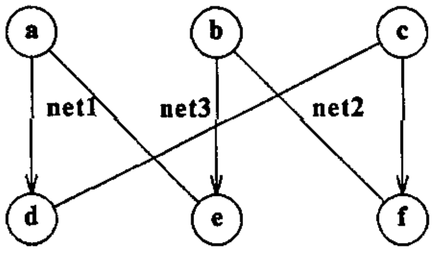


Figure 21. Possible realizations of fig.19

This folding will result in a failing net placement so we have to forbid the situation of figure 20. We could change it into figure 22.
In the new G_fold graph, we have to check for loops between nodes containing more than one gate. It is easy to check for this property: for every subgroup of interconnected nodes containing more than one gate, the number of edges has to be one less than the number of nodes[8].

8. Due to a lack of time, this 'cycle check' is not implemented in GM yet.

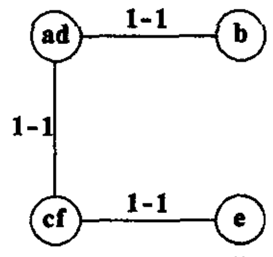


Figure 22. Acyclic new G_fold graph

Resuming; the second approach is less time consuming but will forbid a number of foldings unnecessarily. It does have the advantage of giving the net placement the freedom of choosing the gate ordering. The second approach is implemented in GM.

6.3 The folding algorithm

Before we give the final folding algorithm, a few problems will have to be discussed.

Should we try to fold a certain gate with every other gate or just with a certain subgroup of the other gates ?
Say we have a linear ordering of 20 gates coming from the gate optimization. The ordering is primarily based on the number of connections between the different gates. If we would fold gate 18 with gate 3 into column 3, the result would probably be poor. Signal 18 will be highly connected with gates near 18 (16,17,19,20). If we fold 18 and 3, many nets will have to go from the left side of the gate matrix (column 3), to the right side of the gate matrix (16,17,19,20). This would cost a lot of rows.
So we only want to try to fold gates with their "surrounding" gates. A solution may be to let the user specify this surrounding by an input variable BACK_COL. BACK_COL is the number of columns we go backwards during the folding, to see if folding is allowed.

Now another problem becomes evident. Let the gates 1,2 and 3 be placed into columns 1,2 and 3. If the user has specified that we are allowed to go 3 columns backwards for a folding attempt, gate 4 could be folded into columns 1,2 or 3. *If all the foldings are legal, which one of them do we prefer ?*
One might think column 3 because if gate 5 is placed in column 4, and gate 4 and 5 are highly interconnected, the nets will stay short. If we use this "thought", we should work our way, checking for possible folding, form column 3 to column 2 to column 1.
In GM, we choose for the reverse order (1,2,3), because of the following consideration: Say, we fold gate 4 into column 3. Again I want to point at the fact that the linear ordering is primarily based on the interconnecting nets. So in the described situation, it is very likely for gate 5 to have an interconnection with gate 4 or 3. This will disable the folding of 5 into column 3. Maybe 5 can be folded into column 2. Then we are reversing the column order, which will cause difficulties at the left end.
The idea is shown in figure 23. Distance A, generated by checking for possible folding from the left to right to the left, is smaller than distance B, generated by checking for possible folding from the right to the left.

```
1   2   3             1   2   3
<-- A -->             6   5   4
4   5   6   7         <--- B --->
                                7
```

Figure 23. Idea behind checking order for column folding

I admit that the given evidence is not completely satisfying and further research could result in better folding. It might even be true that it is best to change the direction of the search depending on how much folding we want to have in our gate matrix (columns <--> rows).

One last consideration has to mentioned. If we, having taken all the facts mentioned above into account, are allowed to fold two gates, will we always want to fold them ?
If we have a layout which is unfolded realisable using 8 rows, and the folding of the gates a and b would result in using 15 rows just for these two gates, we probably do not want to execute the folding. We have to arm ourselves against such a mistake:

First we make an estimation of the number of rows we want the gate matrix to use. The user may specify ESTIMATION_CO and the number of rows wanted, ESTimation_NUMBer_TRAcks, is determined by dividing the number of transistors in the gate matrix with the ESTIMATION_CO. An estimation for the number of rows needed for a certain column, after folding another gate into that column, can be divided into 3 groups:

T1 : Number of nets connected to gates in the column before the latest folding.

T2 : Number of nets that cross the column.

T3 : Number of nets connected to the gate, candidate for folding.

Now, if (T1 + T2 + T3) <= ESTimation_NUMBer_TRAcks, we decide to fold the gate into the column.

The total folding algorithm is given below:

```
BEGIN

  place gate 1 into column 1;
  PLA_COUNT = 1;
  FOR ( all gates )
       FOR ( COL = PLA_COUNT - BACK_COL ; COL <= PLA_COUNT ; COL ++)
           IF (the matrix FOLD allows the folding  &&
              T1 + T2 + T3 <= EST_NUMB_TRA        &&
              folding is allowed by new G_fold graph )
                fold gate into column COL;
            ELSE
           {
                PLA_COUNT = PLA_COUNT + 1;
                place gate into column PLA_COUNT;
           }

 END
```

PLA_COUNT = number of columns (out of C), containing gates.

7. The Net placement

In step four of the total gate matrix realization the nets have to be placed in the rows of the gate matrix. I have chosen for a greedy routine, based on the left edge algorithm. If unconstrained left edge would be used, the realizability of the diffusions between the nets could not be guaranteed; diffusion collisions (contacts between diffusions of different transistors) might occur.
So alterations had to be made in the original left edge algorithm. In this chapter, several heuristics will be discussed. Heuristics, used during the placement in order to increase the probability that the generated layout is realizable.

We recall the division of the nets. In chapter 3, we defined three categories:

00) nets with a poly/metal contact at one side and a terminal at the other side (a net for an output signal).

0T) nets with a transistor piece at one end, and a poly/metal contact at the other end.

TT) nets with one transistor piece at each side.

In Chapter 4, it was explained that if we had two nets for the representation of every transistor, nets of the type TT will not exist. For a start we will only look at nets of the first three types.

Heuristic 1.

If a net of group 0T is placed, always search for the other transistor part and place the two nets while checking for diffusion collisions.

If all nets of the gate matrix are in the first three groups, Heuristic 1 guarantees that the generated layout is realizable because during the placement of every single diffusion we check for a collision and all nets of one net path are placed at the same time. Problems arise if nets of category TT are present. In that case we may have more than one diffusion in a single net path. The placement of two nets with a realizable diffusion in between at the start of a net path, may disable a diffusion between t wo nets in another part of the net path. To avoid this situation, the following Heuristic should help:

Heuristic 2:

If a net of type TT is placed, and both connected transistors have a second half at another net, the net is placed in combination with one of the two nets containing the other transistor pieces. The third net is placed on a stack, and will be placed next.

Note that in this way, it can still not be guaranteed that the diffusion between the net of the type TT and the net on the stack is realizable. In order to lower the chance of a collision, another heuristic is used. Look at the next example:

example 1 net	col1	col2	col3	col4
a)	-	-	-	T0
b)	T1:	-	-	T2
c)	T1:	-	-	T3
d)	.	-	-	T2

Syntax :

Tx *->* *a transistor*
: *->* *a diffusion part*
- *->* *a net part*
. *->* *empty*

Note : transistor T3 is a transistor connected to a power line thus only one transistor piece will be present among the nets.
Net a) was already placed and net b) has to be placed next. Net c) is placed at the same time and the diffusion of T1 will be realizable. Net d) is put on the stack and will be placed next. The diffusion of T2 cannot be placed because the transistors T0 and T3 block the way in both directions. A solution is brought by the next heuristic:

Heuristic 3:

Never place a net of type TT without having placed one of the other nets containing a transistor half first.[9]

Using this Heuristic, the placement of example 1 becomes:

example 2 net	col1	col2	col3	col4
a)	-	-	-	T0
c)	T1:	-	-	T3
b)	T1:	-	-	T2:
d)	.	-	-	T2:

Now no collision occurs for T1 or T2; both diffusions can be placed.

The heuristics explained above were implemented into GM. Very seldom a collision occurred and within 3 attempts all the layouts could be realized.

9. **Note: this Heuristic does not apply to nets of type TT) with one of the transistors having only one transistor piece.**

8. Net placement with column folding

The placement described so far, does not take any notice of column folding. How can we adapt the net placement to the column folding? From chapter 6, we know what gates have to be assigned to what columns, but we do not know the order of the gates within a certain column; we do not know which rows are going to be crossed by what gates.

In the horizontal direction we used a left edge algorithm. Why not use the same idea in the vertical direction as well. Start a gate in a columns and if the nets connected to this gate are placed, place another gate, out of the set of gates that have to be placed into that column, into the column; again place the nets connected to this column......
In the left edge algorithm we used for net placement in the unfolded gate matrix, it was always allowed to place a certain net. This is not true any more. How do we know whether we are allowed to place a net?

First we define the array ACTIVE:

> ACTIVE[*col*] = *gat* : in column *col*, the gate *gat* is said to be active if it is allowed to place nets in the gate matrix connected to this gate.

Several facts about ACTIVE can be stated:

> Out of every column, only one gate can be active at the same time.
>
> Starting the net placement, all the gates that have to be connected with the upper outside of the gate matrix are active because they will have to be the top gates of a column anyway.
>
> A gate, not connected to the bottom side of the gate matrix, can be made active if no other gate folded to the same column is active. If the gate has to be connected to the bottom side of the gate matrix, every other gate folded to the same column has to be placed already: every other gate folded to that column has to be placed above the gate that has to be connected to the bottom side of the gate matrix.
>
> A gate is made inactive if all the nets connected to the gate are placed.
>
> In the columns of the power supply, situated at the left and right side of the gate matrix, the gates containing the power supply are always active.

Looking at ACTIVE, we know what gates we are allowed to use. What we want to know is what nets we are allowed to place. We need to know on what gates the placement of a certain net *depends*; what gates have to be ACTIVE if we want to place a certain net.
The placement of a net depends on the gates the net is connected to. If those gates are inactive, the net can not be placed. If we use the second Heuristic of the last paragraph, we will want to place the nets of a net path without having to place other nets in between, thus increasing the probability of generating a realizable layout.
If we state, the placement of a net depends on the gates connected to the net and on the gates connected to the same net path, we will always be able to place the nets of the net path as a whole. So we state:

> **A net depends on all the gates connected to the net path, the net is part of.**

At this stage the reader should be able to understand why we did not want to fold gates connected to the same net path in the paragraph on column folding. We can not place net paths connected to more than one gate out of a column because the gates connected to the net path will never all be active at the same time.

In the table below, an example is shown:

	1	2	3	4	5	6
a	-	T1	.	.	.	.
b		T1	-	-	T2	.
c	.	.	.	-	T2	.

Net a,b and c, all part of the same net path, all depend on the gates 1,2,4 and 5.

One last Heuristic is used during the net placement. It should assure that the gates in the columns do not use an unnecessary number of rows.

Heuristic 4:

> **Do not make any new gates active if there are still nets left, depending only on gates already active.**

The net placement algorithm becomes:

```
BEGIN

    WHILE ( not all the nets are placed )
    {
        search from the left to the right, for a placeable net;
                ( a net only depending on active gates )
        IF ( no placeable net is found )
        {
            try, again from the left to the right, to make a net placeable
                        by making the gates it depends on active;
            IF ( still no placeable net is found )
                    exit; ( the placement failed )
        }
        place_net();
        update_active();
    }

END
```

The column folding determines whether the exit will be reached or not. If the folding was allowed by the new graph G_fold, as discussed in the paragraph on column folding, the placement will never fail because of not finding any placeable net.

9. The final steps of the placement

Having ordered the gates and placed the nets, all that remains to be done is the determination of the functions *k* and *l*. The next two paragraphs will discuss those functions. The last paragraph discusses what has to be done if an unrealizable layout is generated.

9.1 The function l

The rules for the generation *l* are quite simple:

- If the two nets, containing a transistor piece of a certain transistor, are situated in the same row of the gate matrix, we also place the transistor in this row.
- If there is only one net containing a piece of a certain transistor, the transistor will be placed in the same row.
- The situation becomes a bit more complicated if we have two nets containing a transistor piece of a certain transistor and they are situated in different gate matrix rows. If we allow one diffusion run for each transistor, only the two rows containing one of the nets are candidates for the transistor placement.
  The choice between those two rows is based on the width of the transistors. We choose that gate matrix row that contains the transistor with the largest width already.

9.2 The function k

The function k has to be determined next. Look at the following example:

		col1		col2		col3		col4		col5	
1	-	-	-	T1		.		t3	-	-	-
2		.		.		T2	-	T3		.	
3	-	-	-	t1		.		.		T4	
4		.		.		t2	-	-	-	-	-

Syntax:

. = empty
- = net
tx = metal/diffusion contact for transistor x
Tx = transistor x.

We have to realize diffusions contacting Tx and tx. They can be placed in one of the columns from C', adjacent to the C column in which Tx and tx are placed. Working our way through the matrix from the left to the right, we will try to place each diffusion at the C' column at the left of the C column the transistor is placed in. If this is not possible, we will place the diffusion at the right side.
We will explain the determination of *k* using the example shown above. We start at the left side of the matrix, at the left of the first C column. We move from top to bottom and look at each row if a diffusion is needed. At the left of column 1, no diffusion has to be placed. At the left of column 2, we can place the diffusion for transistor 1 and at the left of column 3, the diffusion run for transistor 2 can be placed.
We have to be alert arriving at transistor 3. Transistors 2 and 3 will be connected by a

net implemented in the diffusion layer. Therefore, the diffusion of transistor two will have to be situated at the left of T2, and the diffusion of transistor three will have to be situated at the right of T3. In these cases, there is only one C' column available.

We continue to work our way through the columns until we have reached the last row in the last C' column. If by then, all diffusions are placed, we succeeded in generating a layout. If not, we failed and we will have to proceed as described in the next paragraph.
If it is possible to place the diffusions using the given functions *f* and *h*, this algorithm will find a diffusion placement. If the placement of diffusion *a* is disabled by diffusion *b* at the left C' column, there was no alternative placement for diffusion *b*. If diffusion *a* cannot be placed in the right C' column, this cannot be caused by a diffusion run, but must be the result of a transistor placement. In that case, the net placement is to blame.

9.3 Realizability of the layout

The layout generated will not always be realizable. If the layout is not realizable, we have to find out between which diffusions the collision takes place. At least one of the transistors connected to these diffusions is part of a net of the type TT. If that TT net did not exist, the collision would not take place. The TT nets were generated making the structure of the gate matrix. If a gate, connected with two nets of type 0T, was deleted, a net of type TT was born. If the gate is not deleted, the TT net is not generated.
The thing to do, if a collision occurs, is to find out which of the deleted gates was connected to the two nets now included in the TT net causing the collision. If, during the generation of the structure of the gate matrix, this gate is not deleted, the problem causing TT net will not be created and the same collision can not occur.
In the worst case, all the TT nets will be rejected in subsequent runs. Even in this case, the program will still be able to generate a realizable layout since the result of the algorithms discussed in the previous chapters is always realizable if no TT nets are present.

10. The power lines

In chapter 4, the generation of the layout structure was discussed. It was mentioned that all the nets are two terminal nets. The disadvantages of this approach became clear dealing with signals having a lot of contacts. They were subdivided in so many non overlapping gate matrix nets, that the result could not be efficient any more. Signals with many terminals are e.g.:

- a clock signal in clocked logic
- power lines

Because of this effect power nets are treated differently from all other nets. It is not unusual to treat the power nets differently; in many articles discussing gate matrix layout generation, the power nets are situated in a different layer (extra metal layer), and are not really placed in the gate matrix.
In chapter 4 it was also mentioned that the signals connected to the power supply, are not represented by gate matrix nets. Only after the other nets are placed, these power nets are placed. The horizontal part of the power lines is placed in the metal layer, the vertical part in the diffusion layer. The metal part of the power lines can be situated between two tracks of the layout matrix we generated until now. We will call the tracks from the layout matrix made so far "signal tracks", while the possible situations for the power lines will be called "power tracks".

In this chapter, the determination of the position of the power lines will be discussed. First, a matrix POW_MAT is generated, and using this matrix, the positions of the power lines are calculated.

10.1 The Power Matrix

The matrix POW_MAT has a row for each transistor *trns*, and a column for every power track *trck*. If POW_MAT(*trns*,*trck*) = 1 , this means that it is possible for a vertical diffusion to go from the power line *trck* to the transistor *trns*. So one could say that if POW_MAT(*trns*,*trck*) = 1, *trck* is a legal position for the power line connected to transistor *trns*.
The matrix is constructed as follows:
The power track beneath and above the signal track in which the transistor is placed will, apart from the signal tracks at the outside of the gate matrix, always be legal because a diffusion run is always possible. Next, move away from the transistor in vertical direction, at each signal track checking for a collision, e.g. caused by another transistor.[10] We could have the user specify how far we are allowed to move away from the transistor by an input variable, e.g. MAX_TRA_CROS. MAX_TRA_CROS specifies the MAXimum number of signal TRAcks the power line is allowed to CROSs.

Apart from this matrix, we need two other pieces of information: what is the polarity of the power line needed and between what columns is it going to be situated. The

10. There will not be a collision every time another transistor is met; e.g. if the transistor is placed in the same column and also needs a connection to a power line of the right polarity at the right side, there will be no collision, and the diffusion may be continued.

procedure is illustrated by the following example:

trans	track1	track2	track3	track4	track5	track6	track7	pol	col
1	1	1	1	0	0	0	0	+	2
2	0	1	1	1	1	0	0	-	2
3	1	1	1	1	1	1	0	+	3
4	0	0	0	1	1	0	0	+	4
5	1	1	0	0	0	0	0	-	1

In the first line it says that, as far as transistor 1 is concerned, it's power supply will be well taken care of if there is a power line with positive polarity situated in power track1, power track2 or power track3.

It seems that this is all the information we need to determine where the power lines should be situated.

10.2 Calculation of the power line positions

Our goal is to realize the power supply with the smallest number of necessary power lines. We applied the following algorithm:

```
BEGIN

   WHILE (not all power transistors are connected to a power line)
   {
      Walk through POW_MAT from the left to the right, and if at one row
         there is a 1 to 0 transition, a power line has to be placed at
         this position, having the polarity of the transistor;

      Connect as much transistors to this power line as possible, using the
         constraints set by POW_MAT;

      Update POW_MAT;
   }

END
```

Updating the POW_MAT matrix is more than just the deletion of power line requests of transistors just connected. This updating is performed by the algorithm shown on the next page[11].

We will demonstrate the algorithms, using the example of the previous paragraph:
The first power tracks where there is a 1 to 0 transition, are tracks 2 and 3. So at track 2, a power line will have to be placed and transistor 5 can be connected. But if a

11. To understand exactly why the $<$, $>$ or the $=$ signs are used in the algorithm one should know that if two power lines are placed at the same power track, the second power line is placed above the first one.

```
BEGIN
 DO ( for all power transistors trns connected to the power line placed latest)
   FOR ( trck = 1 ; trck <= numb_tracks ; trck++)
           POW_MAT[trns][trck] = 0;

 DO ( for all power transistors trns, demanding a diffusion in a column where
     a diffusion run is just placed)
   IF ( power[trns].track <= pow_end )
      FOR ( trck=pow_end ; trck <= power[trns].track + MAX_TRA_CROS;
          trck++)
           POW_MAT[trns][trck] = 0;
   ELSE
      FOR ( trck=power[trns].track - MAX_TRA_CROS ; trck < pow_end ;
          trck++)
           POW_MAT[trns][trck] = 0;

END
   pow_end = position of the power line placed latest.
   power[trns].track = signal track of transistor trns.
```

Algorithm for POW_MAT updating.

power line is placed at power track 2, transistor 2 can also be connected. (The other transistors cannot be connected, because their polarity is incorrect .) Fill the rows of transistors with a 1 at the power track where the power line is going to be placed with 0's, because they do not need a power line any more. The matrix of our example now becomes:

trans	track1	track2	track3	track4	track5	track6	track7	pol	col
1	1	1	1	0	0	0	0	+	2
2	0	0	0	0	0	0	0	-	2
3	1	1	1	1	1	1	0	+	3
4	0	0	0	1	1	0	0	+	4
5	0	0	0	0	0	0	0	-	1

Suppose we do not perform the rest of the updating.
The next column will have to be placed in track 3, because of transistor 1. Transistor 3 can also be connected . Everything seems to go all right, but just now we made an error and the generated layout will have an interconnection between the positive and negative power supply.
Chapter 2 dealt with the layout configuration of a gate matrix: only 1 diffusion run was allowed between two columns. More diffusion runs will result in a short circuit. In column 2 a collision is created between two power diffusions: if transistor 2 is situated in signal track 4, there will be a diffusion from signal track 4, to the power line in power track 2. If transistor 1 is situated in signal track 1, we placed a diffusion from track 1 to the power line in power track 3. The two diffusions will be short circuited at track 3.

The example shows the use of the updating algorithm. If a diffusion is placed in a certain column, all the diffusion entries in POW_MAT in rows with the same column label, will have to be checked and updated depending on where the power line and the transistors are situated. So, during the power line placement the positions of the transistors have to be known also.

In the table below, the updated POW_MAT matrix is shown, present after the placement of the first power line.

trans	track1	track2	track3	track4	track5	track6	track7	pol	col
1	1	0	0	0	0	0	0	+	2
2	0	0	0	0	0	0	0	-	2
3	1	1	1	1	1	1	0	+	3
4	0	0	0	1	1	0	0	+	4
5	0	0	0	0	0	0	0	-	1

Another power line will have to be placed at power track 1 connecting transistors 1 and 3. Again the matrix is updated :

trans	track1	track2	track3	track4	track5	track6	track7	pol	col
1	0	0	0	0	0	0	0	+	2
2	0	0	0	0	0	0	0	-	2
3	0	0	0	0	0	0	0	+	3
4	0	0	0	1	1	0	0	+	4
5	0	0	0	0	0	0	0	-	1

One last power line is needed at track 5. This time, updating the matrix leaves no POWMAT elements equal to one. This completes the power line placement, resulting in three power lines:

Power line	position	polarity	connected transistors
1	2	-	2 & 5
2	1	+	1 & 3
3	5	+	4

The algorithm discussed are implemented in GM and guarantee that no collision occurs between power diffusions.

11. Compacting the layout

Having placed the nets into the gate matrix, we should be able to generate a symbolic layout without any problem. Next, we will have to work our way towards the "real " layout. We have to compute the coordinates of the different parts of the gate matrix. For this purpose, we think of the gate matrix as being projected on a grid; the rows of the gate matrix are the rows of the grid and the columns of the gate matrix are the columns of the grid. We use a grid compaction: we determine the necessary separation between rows and columns and out of this information, we calculate the different coordinates.
There is more than one possible way to perform a grid compaction:

method 1) A first implementation is to let the compactor work his way through each column (row) computing the maximum width needed for a certain column (row). Columns a and b are placed at a separation distance of 0.5 * width(col a) + 0.5 * width(col b).

This method has one serious disadvantage: if we calculate the necessary distance between two columns and the first column has a large transistor at track 1 and the second column has a large transistor at track 2, the computed separation distance will be very large: to large because this method does not use the information that the transistors are placed into two different tracks. A better result can be obtained if we use the next approach:

method 2) Look at every track (column) what the separation distance should be in order to have no design rule errors. The final separation distance between two columns (tracks), will be the maximum of these distances over all tracks (columns).

If we would use method two like this, the generated layout would probably contain design rule errors. Look at the next situation:

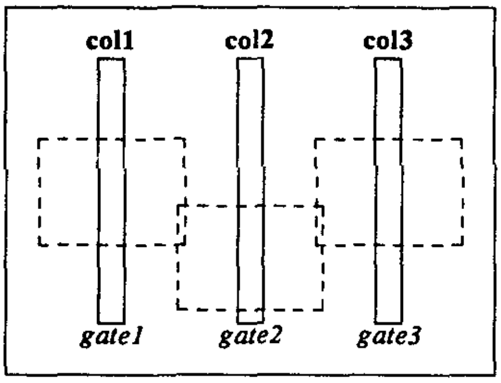


Figure 24. Example of a cross conflicts

Neither one of the two compacting steps of method two will detect the combination of transistors and overlapping may be the result. For this reason one of the two steps should not only look at one track (column) at a time. In GM, the row compactor computes the distance between signal tracks by looking at one element of track1 and looking at three elements situated in track 2. This way, not only conflicts in the the same column are detected, but also cross conflicts are seen. The idea is demonstrated in the figure below:

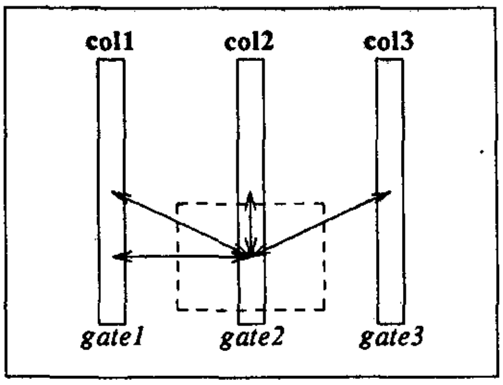


Figure 25. Directions of design rule checking done by the row compactor

We choose to let the row compaction solve the cross conflicts because most of the time a gate matrix will have far more columns than rows. So a worsening of the column compaction would have a larger impact on the area needed than a worsening of the row compaction.

The two methods described are both used in GM. Method 1 is used:

-at the borders of the gate matrix.

-to compute separation distances between power tracks and other parts of the gate matrix. This can be done because of the nearly constant width used by a power track.

The distance between two signal tracks and the distance between columns is computed using method 2.

The necessary separation distance between the gate matrix elements depends on the design rules of the technology used. In order to keep a program as technology independent as possible, we could have the user specify the distance between all possible elements on a column / track; e.g. the user has to specify what the distance between two columns should be as a result of two metal/poly contacts. All this technology dependent information is brought into the layout generator by a technology file. Details about this file for GM, can be found in the appendix on implementation.

12. Some results

In this chapter, the layouts generated by GM will be discussed on 2 aspects:

-efficiency.

-shape transformability.

1) It is hard to say something about the efficiency of the layout produced in general. We should compare GM with other gate matrix layout generators on a fair basis. GM does not use a second metal or polysilicon layer. Many IC generating technologies have this layer nowadays and all the gate matrix layout generators we were able to lay our hands on, used this extra layer. So we were not in a position to compare GM with another gate matrix layout generator in a fair way. to
We did compare GM with a Standard cell layout generator, called *logic*, for generating random logic. The area needed for small layout examples (up to 100 modules) is about equal (60-120%)[12]. An example is given in appendix 2. For larger layout examples, we were only able to generate unfolded gate matrices because of the cycles spoiling the folding. For larger layout examples the area needed by GM increased more rapidly than for Standard Layout generator; the disadvantage of one small signal demanding a total column became dominating (see appendix 3).

   GM can generate layout using any size of transistors, but if the size difference between the transistors becomes to large, the layout can hardly be efficient any more. Look at the example in appendix 4. The three large transistors screw up the result; the tracks they are placed in have the same height over the entire width.

2) It depends on the layout structure, but most of the time the shape can be changed relatively easy. There are several possibilities to correct the shape of the layout:

   a) If we do not use folding, the width of the gate matrix depends on the number of signals, present in the circuit. If we vary this number, we will vary the width of the gate matrix as well. This approach can be used while generating logic: manipulating with the logic expressions, we can change the number of signals in the circuit. If we want to have many signals, we should do as many kernel and cube substitutions as possible. On the other hand, if we want to have a small number of signals, corresponding to a high gate matrix layout, we do not want to perform any substitutions. In general, the most efficient results were obtained performing as many substitutions as possible.

   b) In GM, the user can increase the height of a layout by increasing BACK_COL and decreasing ESTIMATION_CO, thus allowing more signals to be folded. Examples of a result one can get with varying

12. Interpreting these results, one should keep in mind that the Standard cell Layout generator was specially developed for generating logic, and did not have the freedom of using different sized transistors as we have with GM.

these two variables are shown in appendices 5 and 6. The flat layouts are generated without column folding. There is no real need to be so flexible: it would have been sufficient if we could have changed from the flat layout into a square[13] , but nevertheless it illustrates the shape flexibility of the gate matrix.

13. If GM would have be able to go from flat to square, a thick layout could be obtained by turning the flat layout.

13. Suggestions for continuation

GM in its present form, is nearly complete: Except for the cycle check,the program does not need any alterations.
This does not mean that there are no further positive modifications possible. In this chapter I will give some suggestions as to how I think the performance of GM could be improved.

1) While generating the initial gate matrix, we could allow *more terminal nets*. Although this alteration might improve the efficiency of the layout considerably, it will also have many consequences for the rest of the program. It will be difficult to maintain the guarantee of being able to generate a realizable layout for all possible circuits.

2) The signal ordering and folding could be integrated into one single step[14].

3) During the placement of the nets into the gate matrix, we could look more carefully if it is allowed to place a net into a certain track. Now, GM only allows the overlap of different nets if they do not have components in the same layout layer. We could extend the overlap allowance to nets having components at the same layout layer e.g. if two nets are connected to the same signal, in many cases overlap of these two nets will be allowed.
This alteration should not be to hard to implement and could partly undo the disadvantage created by using *two terminal nets* as discussed in Chapter 4.

4) The level of compaction could be increased. We could make graphs for the horizontal and vertical directions. For the horizontal graph, the different nets would be the nodes. The labeled edges would indicate the existence of a necessary separation distance between the signals represented by the nodes. For the vertical direction, the different signals would be the nodes and the edges would again indicate a necessary separation distance.
We could calculate the coordinates of a certain net (signal) by taking the longest possible path to get at a net (signal) and add the distances shown by the edges along the way.

14. At the time this report was written, far developed plans for integrating a 2-dimensional Min Cut approach into GM, were already present.

14. Conclusions

At the end of this report, I want to make some concluding remarks:

- We showed that it is possible to obtain promising layout results with a gate matrix layout generator only using quite simple algorithms.

- The gate matrix layout style,used with column folding, has indeed quite flexible shape constraints. With some manipulation, large variations in the aspect ratio can be reached.

- The usage of different sized transistors, in analog or digital circuits, does not decrease the efficiency of the gate matrix layout very much as long as the size difference does no exceed a factor 5/6.

- Some alterations could increase the efficiency of the layout to some extent. Do not expect miracles; I guess an improvement of over 10/20% cannot be made as long as the same gate matrix structure is being used.

Appendix 1: Literature

[ASAN82] Asano, T.
AN OPTIMUM GATE PLACEMENT ALGORITHM FOR MOS ONE-DIMENSIONAL ARRAYS.
J. Digital Syst., Vol. 6(1982), p. 1-27.

[DEKR87] Deo, N. and M.S. Krishnamoorthy, M.A. Langston
EXACT AND APPROXIMATE SOLUTIONS FOR THE GATE MATRIX LAYOUT PROBLEM.
IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., Vol. CAD-6(1987), p. 79-84.

[DEVA86] Devadas, S. and A.R. Newton
GENIE: A generalized array optimizer for VLSI synthesis.
In: Proc. ACM/IEEE 23rd Design Automation Conf., Las Vegas, Nev., 29 June-2 July 1986.
New York: IEEE, 1986. P. 631-637.

[GINN84] Ginneken, L.P.P.P. van and R.H.J.M. Otten
STEPWISE LAYOUT REFINEMENT.
In: Proc. 2nd IEEE Int. Conf. on Computer Design: VLSI in Computers (ICCD'84), Port Chester, N.Y., 8-11 Oct. 1984.
New York: IEEE, 1984. P. 30-36.

[HUWI86] Huang, S. and O. Wing
IMPROVED GATE MATRIX LAYOUT.
In: Proc. 4th IEEE Int. Conf. on Computer-Aided Design (ICCAD'86), Santa Clara, Cal., 11-13 Nov. 1986.
New York: IEEE, 1986. P. 320-323.

[HWAN86] Hwang, D.K. and W.K. Fuchs, S.M. Kang
AN EFFICIENT APPROACH TO GATE MATRIX LAYOUT.
In: Proc. 4th IEEE Int. Conf. on Computer-Aided Design (ICCAD'86), Santa Clara, Cal., 11-13 Nov. 1986.
New York: IEEE, 1986. P. 312-315.

[JTLI83] Li, J.-T.
ALGORITHMS FOR GATE MATRIX LAYOUT.
In: Proc. 16th IEEE Int. Symp. on Circuits and Systems, Newport Beach, Cal., 2-4 May 1983.
New York: IEEE, 1983. P. 1013-1016.

[KANG83] Kang, S.
LINEAR ORDERING AND APPLICATION TO PLACEMENT.
In: Proc. ACM/IEEE 20th Design Automation Conf., Miami Beach, Fla., 27-29 June 1983.
New York: IEEE, 1983. P. 457-464.

[KAFU79] Kashiwabara, T. and T. Fujisawa
AN NP-COMPLETE PROBLEM ON INTERVAL GRAPHS.
In: Proc. 12th Int. Symp. on Circuits and Systems, Tokyo, 17-19 July 1979.
New York: IEEE, 1979. P. 82-83.

[KERN78] Kernighan, B.W. and D.M. Ritchie
THE C PROGRAMMING LANGUAGE.
Englewood Cliffs, N.J.: Prentice-Hall, 1978.
Prentice-Hall software series

[LEON86] Leong, H.W.
A NEW ALGORITHM FOR GATE MATRIX LAYOUT.
In: Proc. 4th IEEE Int. Conf. on Computer-Aided Design (ICCAD'86), Santa Clara, Cal., 11-13 Nov. 1986.
New York: IEEE, 1986. P. 316-319.

[LOLA80] Lopez, A.D. and Hung-Fai S. Law
A DENSE GATE MATRIX LAYOUT METHOD FOR MOS VLSI.
IEEE Trans. Electron Devices, Vol. ED-27(1980), p. 1671-1675.

[NAKA86] Nakatani, K. and T. Fujii, T. Kikuno, N. Yoshida
A HEURISTIC ALGORITHM FOR GATE MATRIX LAYOUT.
In: Proc. 4th IEEE Int. Conf. on Computer-Aided Design (ICCAD'86), Santa Clara, Cal., 11-13 Nov. 1986.
New York: IEEE, 1986. P. 324-327.

[OHMO79] Ohtsuki, T. and H. Mori, E.S. Kuh, T. Kashiwabara, T. Fujisawa
ONE-DIMENSIONAL LOGIC GATE ASSIGNMENT AND INTERVAL GRAPHS.
IEEE Trans. Circuits & Syst., Vol. CAS-26(1979), p. 675-684.

[WEIN67] Weinberger, A.
LARGE SCALE INTEGRATION OF MOS COMPLEX LOGIC: A layout method.
IEEE J. Solid-State Circuits, Vol. SC-2(1967), p. 182-190.

[WEST85] Weste, N.H.E. and K. Eshraghian
PRINCIPLES OF CMOS VLSI DESIGN: A systems perspective.
Reading, Mass.: Addison-Wesley, 1985.
Addison-Wesley VLSI systems series

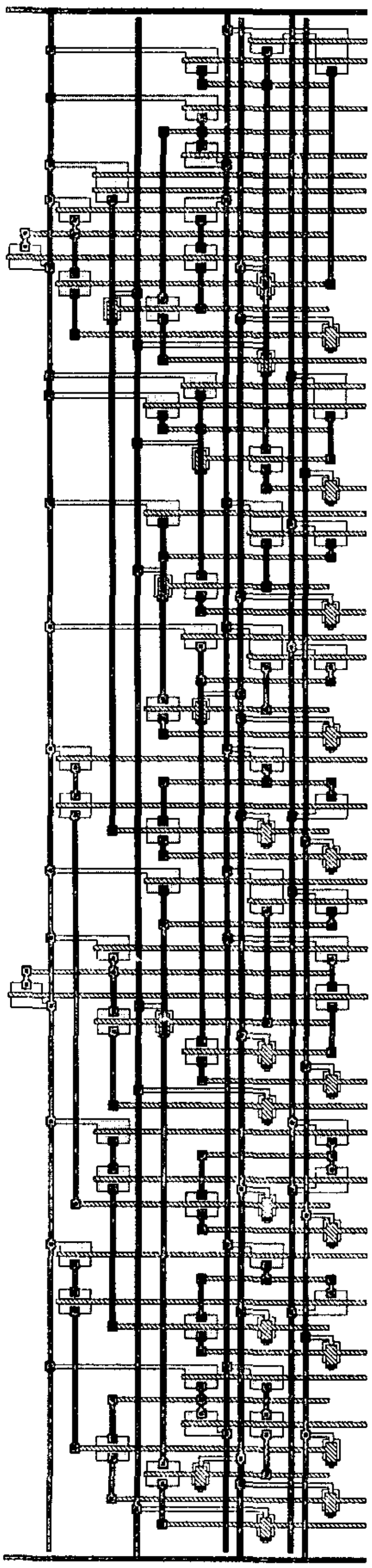
[WIHU85] Wing, O. and S. Huang, R. Wang
GATE MATRIX LAYOUT.
IEEE Trans. Comput.-Aided Des. Integrated Circuits & Syst., Vol. CAD-4(1985), p. 220-231.

[WING82] Wing, O.
AUTOMATED GATE MATRIX LAYOUT.
In: Proc. 15th Int. Symp. on Circuits and Systems, Rome, 10-12 May 1982.
New York: IEEE, 1982. P. 681-685.

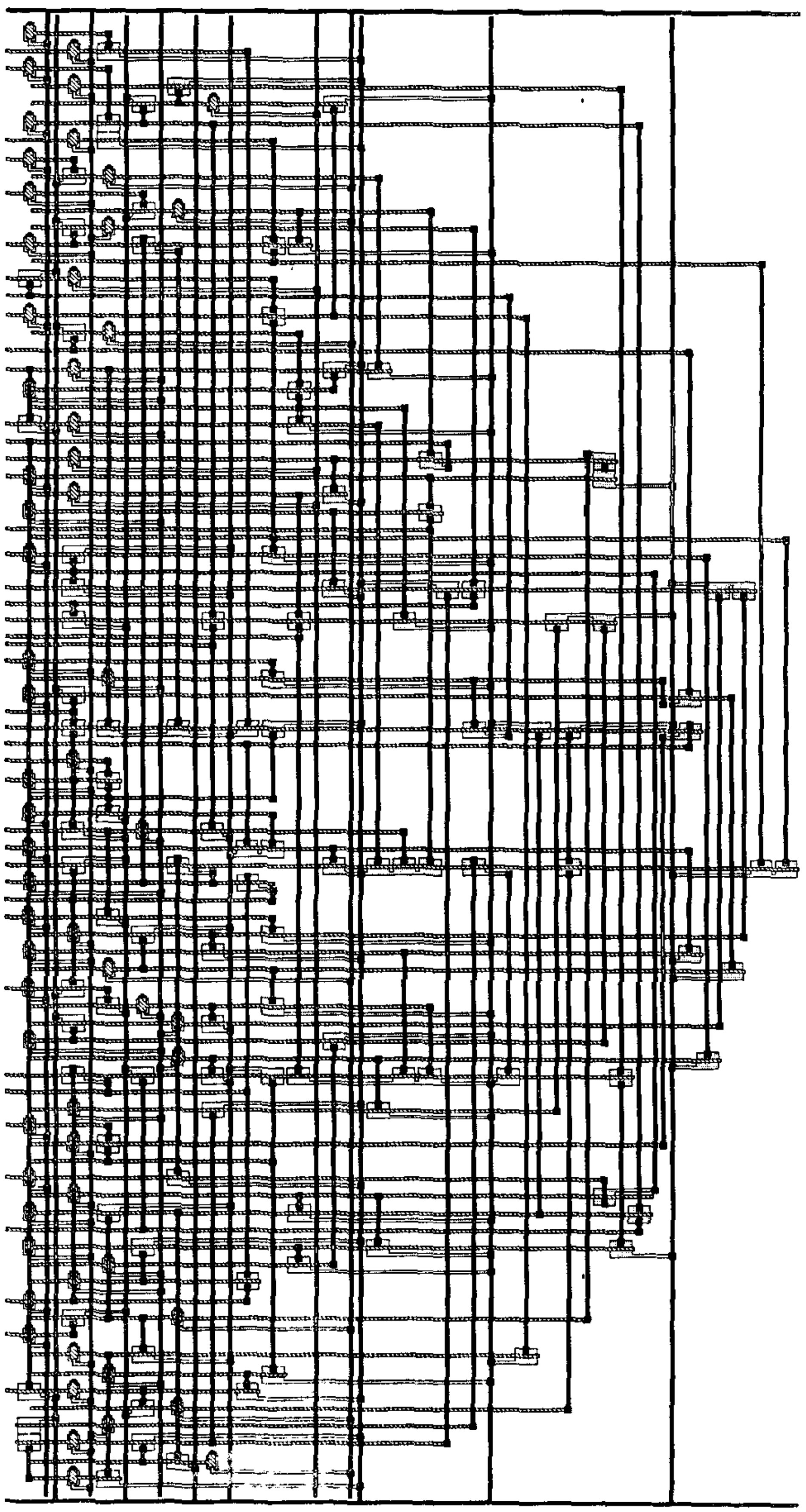
[WING83] Wing, O.
INTERVAL-GRAPH-BASED CIRCUIT LAYOUT.
In: Proc. 1st IEEE Int. Conf. on Computer-Aided Design (ICCAD'83), Santa Clara, Cal., 12-15 Sept. 1983.
New York: IEEE, 1983. P. 84-85.

[WIRT71] Wirth, N.
PROGRAM DEVELOPMENT BY STEP-WISE REFINEMENT.
Commun. ACM, Vol. 14(1971), p. 221-227.

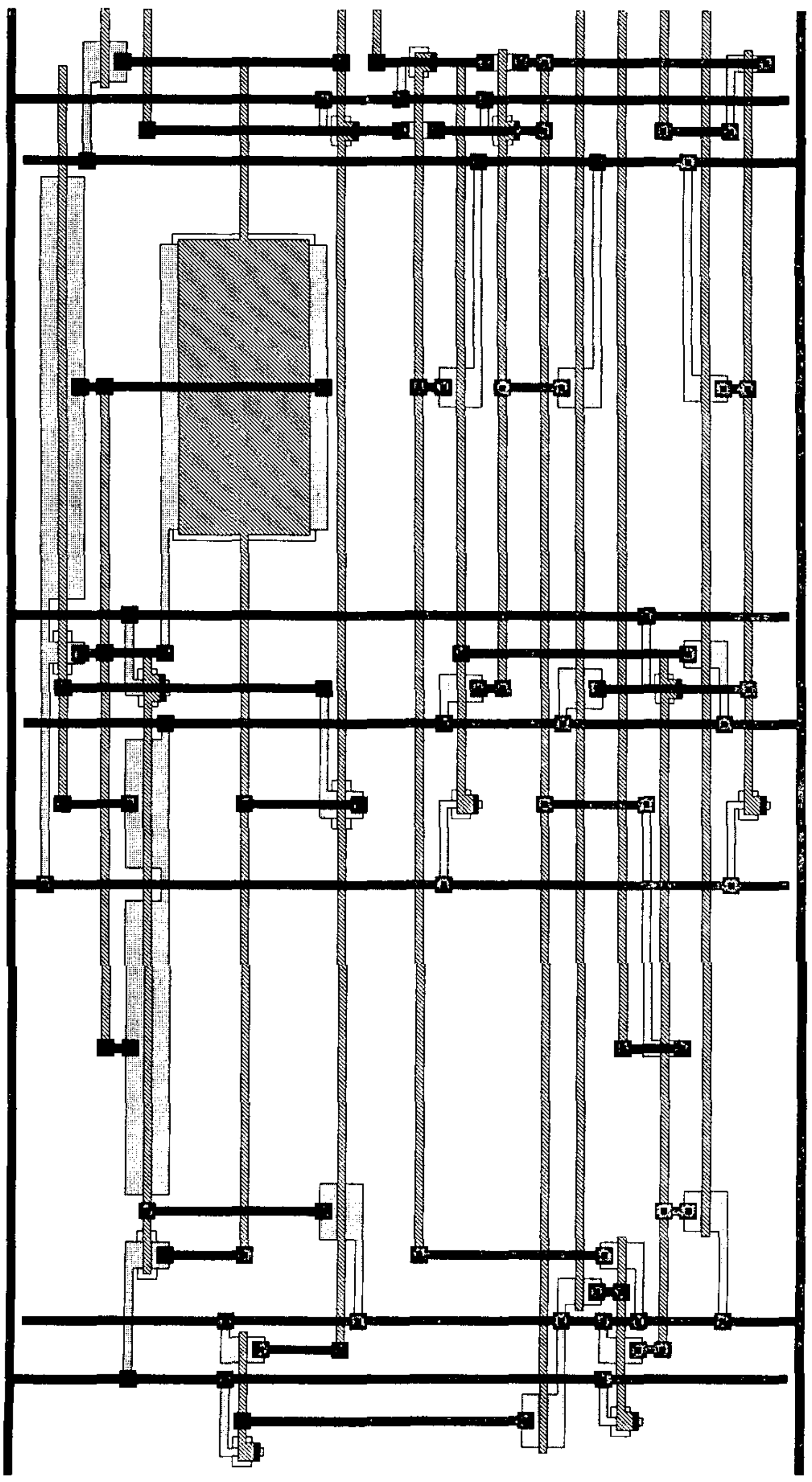
Appendix 2: Layout example (83 transistors)



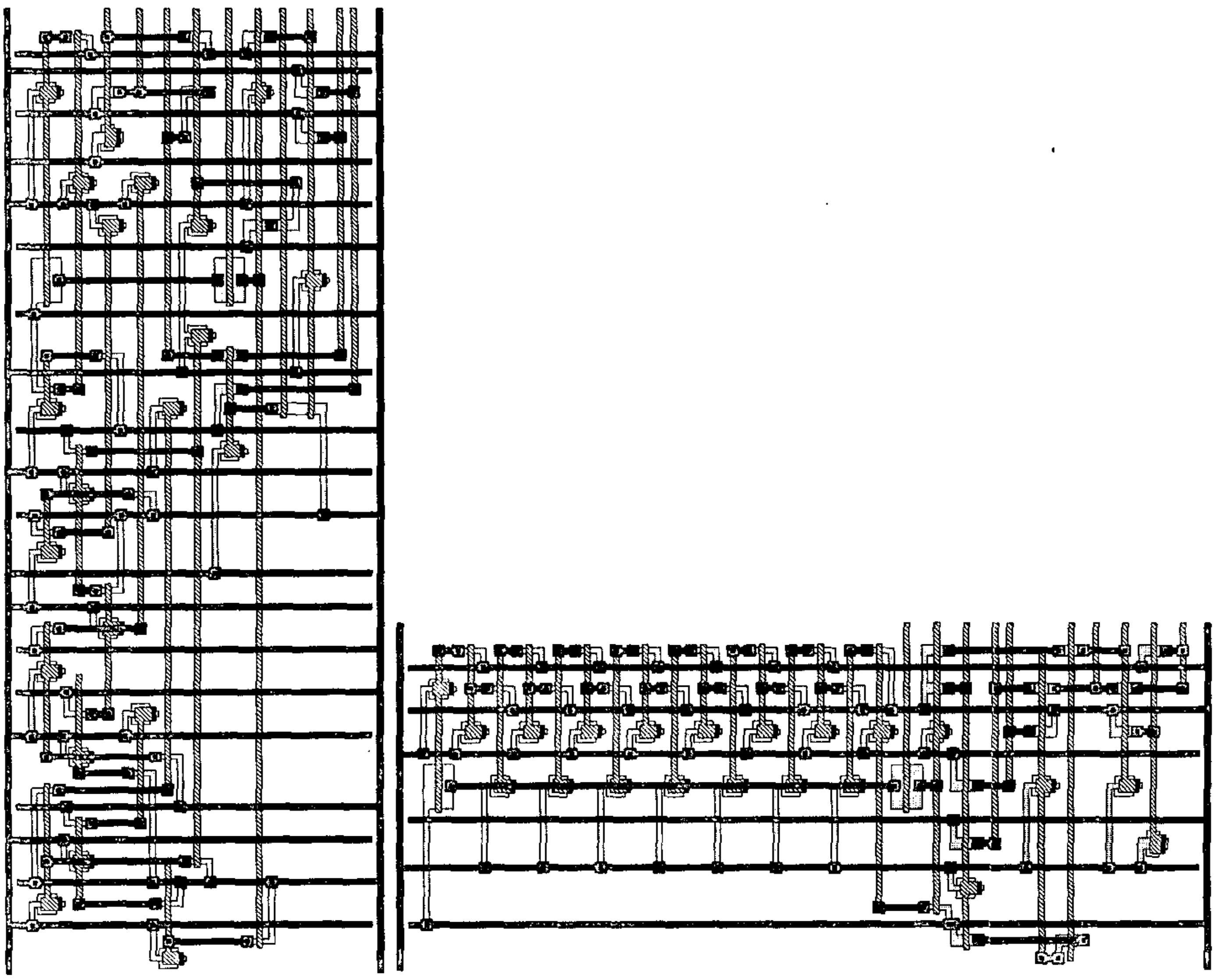
Appendix 3: Larger layout example (180 transistors)



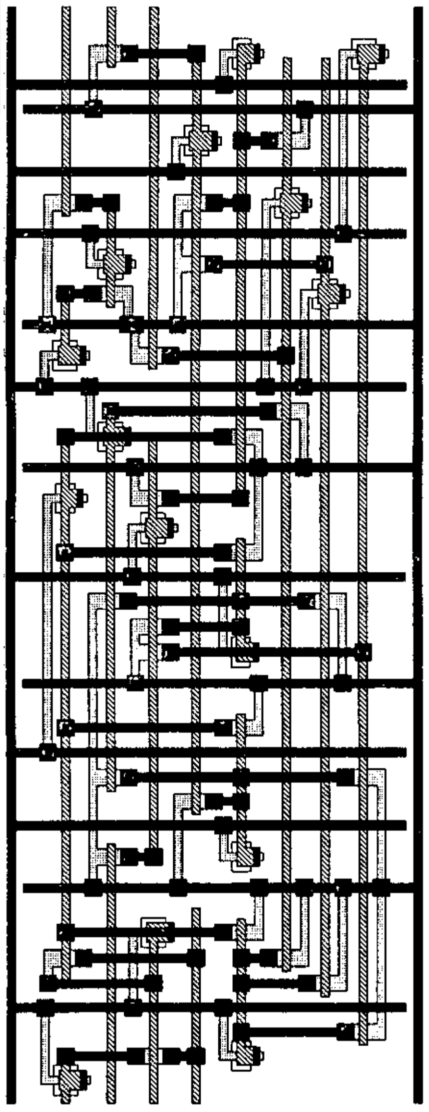
Appendix 4: An example of layout inefficien cy

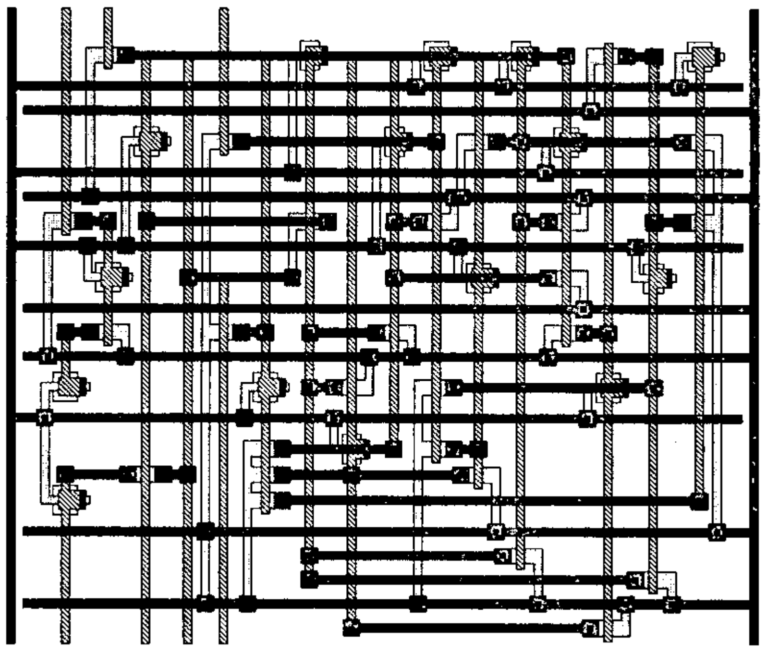


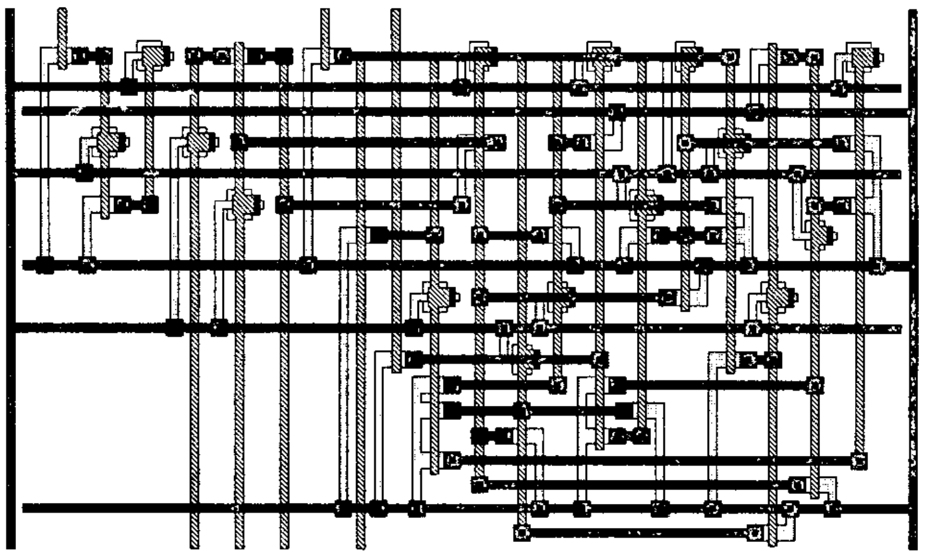
Appendix 5: An example of shape flexibility



Appendix 6: Another example of shape flexibility







Appendix 7: Implementation details

In this Appendix, we present details about the implementation of GM. We will start with a description of the different input files. In the second part, several tables, specifying relations between different functions in GM are presented.

7.1: Input description

The input for GM consists out of 4 different parts:

- 1) The input file **modules**.
- 2) The input file **terminals**.
- 3) The input file **exits**.
- 4) Specification of design rules.

In the file **modules**, all transistors have to be specified. The syntax:

< module name > < module type > < Width > < Length >

The *module name* has to be unique for every module. *Module type* specifies the type of transistor. For NMOS we have *nenh*, for enhancement transistors, and *ndep* for depletion transistors.

The connections between the different transistors are specified in the file **terminals**. The syntax:

< signal name > < module name > < module terminal >

One line describes one connection. e.g. the line " *13 nenh_40 g*", specifies that the *g* of module *nenh_40*, is connected to signal 13. An in- or output signal, demanding a terminal in the gate matrix, should have an extra line in this file:

< signal name > < "root" > < signal name >

The third input input file is **exits**. This file specifies on what side of the gate matrix a terminal should be present, connected to a certain signal. The syntax:

< "root" > < signal name > < side specification >

Possible *side specifications* are *u* for up, *d* for down, *l* for left and *r* for right. GM wants one line in the file *exits* for every terminal. It will give an error message if this is not the case.

The last input for GM specifies the different design rules. For NMOS, we have the function *design_rules_nmos*, in which all necessary variables can be specified.
The variables can be divided into three classes:

- Variables specifying horizontal distances.
- Variables specifying vertical distances.
- General variables.

An example of a variable of the first class is *POLY_CON_POLY*. It should specify the necessary separation distance between two columns if there is no gate matrix element placed in the columns[15], only poly is present, and there is a poly/metal contact situated in between the two columns.

In the second class, we find the same kind of variables but now specifying vertical distances.

In the third class we find all kinds of general variables like width specifications for certain layers or minimal overlap specifications. Here we also specify the names of the different layers in the final layout. If a layer is given the name *cancel*, this layer will not be present in the final layout. This layer will be removed out of the layout by a special C-function, called *del-cancel*.

7.2: Three tables specifying GM's functions

In the following tables, details about the structure of GM are presented. The first table specifies the file in which a certain function can be found:

FUNCTION:	IS DEFINED IN FILE:
Min_cost1	optimal.c
Min_cost2	optimal.c
Min_cost3	optimal.c
adapt1	adapt1.c
adapt2	adapt2.c
add_track	matr_lay.c
addnetl	mkstr.c
addord	mkstr.c
after_place	place1.c
allow	matr_lay.c
and_bitbit	bitmat.c
and_bitmat	bitmat.c
cal_fold	col_fold.c
calcul_coord	coord.c
case_T	gen_lay.c
case_ps_t	coord.c
case_t	gen_lay.c
case_t_ps	coord.c
check_dif	matr_lay.c

15. If the reader has studied Chapter 11 on layout compaction, he will understand that all these variables only specify the necessary separation distance on one track. The compactor calculates the maximum of all these variables.

check_dif_placed	matr_lay.c
choose_trans	matr_lay.c
chose_net	place1.c
col_fold	col_fold.c
coordinates	coord.c
create_bitmat	bitmat.c
delexc	mkstr.c
design_rules_cmos	cmos_rules.c
design_rules_nmos	nmos_rules.c
det_hor	coord.c
det_ver1	coord.c
det_ver2	coord.c
deter_best_track	matr_lay.c
deter_pow_pos	power.c
deter_track_pair	matr_lay.c
determination	optimal.c
detr_pow	power.c
disapprove	power.c
end_generation	assist_lay.c
exnet	mkstr.c
firstpla	mkstr.c
forbidden	place1.c
gateas	mkstr.c
get_bit	bitmat.c
improvement	optimal.c
init	mkstr.c
init_active	place1.c
init_coord	coord.c
init_generation	gen_lay.c
init_ldmfile	assist_lay.c
init_pow	power.c
initial	place1.c
innit	col_fold.c
inv_net	bitmat.c
kang	optimal.c
layout_generation	gen_lay.c
left_right_sets	adapt1.c
load	load.c
loadexit	load.c
loadmod	load.c
loadter	load.c
main	lead.c
matr_init	matr_lay.c
mkstr	mkstr.c
negpow	mkstr.c
net_contacts	adapt1.c
net_len	optimal.c
net_placement	place1.c
netnam	load.c
new_left_right_s	adapt2.c
new_net_contacts	adapt2.c
new_new_s_net	adapt2.c
new_priint	adapt2.c
new_s_net	adapt1.c
new_update_exits	adapt2.c

nnplace	place2.c
noplace	place2.c
onplace	place2.c
ooplace	place2.c
opplace	place2.c
optimal	optimal.c
or_bitmap	bitmat.c
or_bitmatr	bitmat.c
place_col	col_fold.c
place_con_trans	gen_lay.c
place_contact	assist_lay.c
place_dif	matr_lay.c
place_exit	assist_lay.c
place_gasodr	gen_lay.c
place_grid	gen_lay.c
place_net	matr_lay.c
place_nets	gen_lay.c
placed	optimal.c
ploce_net	place2.c
poplace	place2.c
pow_print	power.c
pplace_dif	gen_lay.c
ppplace	place2.c
pre_update_dif	matr_lay.c
priint	adapt1.c
prins	col_fold.c
print_coord	coord.c
print_pos	power.c
prrint	optimal.c
rect	assist_lay.c
set_bit	bitmat.c
sodras	mkstr.c
sprinnt	matr_lay.c
sprint	mkstr.c
start	optimal.c
switch_columns	optimal.c
test_split	matr_lay.c
trans	assist_lay.c
two_track	optimal.c
upd_del_col	mkstr.c
upd_del_columns	matr_lay.c
upd_split	matr_lay.c
upd_tr_pieces	mkstr.c
update_dif	matr_lay.c

The second table describes which functions are called, by functions located in a certain file, present outside this file. Due to the modular structure of GM, this table is not to extensive.

FILE:	CALLS FOR EXTERNAL FUNCTIONS:
adapt1.c	
adapt2.c	
assist_lay.c	
bitmat.c	
cmos_rules.c	
col_fold.c	
coord.c	design_rules_cmos, design_rules_nmos
gen_lay.c	end_generation, init_ldmfile, place_contact, place_exit, rect, trans
lead.c	adapt1, adapt2, col_fold, coordinates, detr_pow, layout_generation, load, mkstr, net_placement, optimal
load.c	
matr_lay.c	
mkstr.c	
nmos_rules.c	
optimal.c	and_bitmat, create_bitmat, get_bit, inv_net, or_bitmap, set_bit
place1.c	check_dif_placed, choose_trans, matr_init, place_dif, ploce_net, pre_update_dif, sprinnt, upd_split, update_dif
place2.c	add_track, allow, check_dif, deter_best_track, deter_track_pair, place_net, test_split

In the third and final table specifies which functions are called by a certain function.

FUNCTION:	CALLS:
Min_cost1	get_bit
	two_track
Min_cost2	create_bitmat
	get_bit
	or_bitmap
	two_track
Min_cost3	get_bit
	kang
adapt1	left_right_sets
	net_contacts
	new_s_net
	priint
	update_exits
adapt2	new_left_right_sets
	new_net_contacts
	new_new_s_net
	new_priint
	new_update_exits
and_bitmat	get_bit
	set_bit
case_T	place_contact
	trans
case_t	place_contact
chose_net	forbidden

col_fold	place_col
	cal_fold
	innit
coordinates	calcul_coord
	design_rules_cmos
	design_rules_nmos
	det_hor
	det_ver1
	det_ver2
	init_coord
	print_coord
delexc	upd_del_col
det_hor	case_ps_t
	case_t_ps
det_ver1	strncmp
det_ver2	strncmp
deter_best_track	add_track
	allow
	check_dif
deter_track_pair	add_track
	check_dif
	deter_track_pair
detr_pow	deter_pow_pos
	init_pow
exnet	addnetl
firstpla	addnetl
	addord
	gateas
	sodras
init_generation	init_ldmfile
init_pow	disapprove
initial	init_active
	matr_init
kang	get_bit
	net_len
	set_bit
	switch_columns
	two_track
layout_generation	end_generation
	init_generation
	place_con_trans
	place_gasodr
	place_grid
	place_nets
	pplace_dif
load	loadexit
	loadmod
	loadter
	netnam
main	adapt1
	adapt2
	col_fold
	coordinates
	detr_pow
	layout_generation

	load
	mkstr
	net_placement
	optimal
mkstr	delexc
	exnet
	firstpla
	init
	negpow
	upd_tr_pieces
net_len	get_bit
net_placement	after_place
	check_dif_placed
	choose_trans
	chose_net
	initial
	place_dif
	ploce_net
	pre_update_dif
	sprinnt
	upd_split
	update_dif
nnplace	add_track
	allow
	check_dif
	deter_best_track
	noplace
	onplace
	place_net
	test_split
noplace	add_track
	allow
	check_dif
	deter_best_track
	deter_track_pair
	place_net
	test_split
onplace	add_track
	allow
	check_dif
	deter_best_track
	deter_track_pair
	place_net
	test_split
ooplace	add_track
	allow
	place_net
	add_track
opplace	allow
	check_dif
	deter_best_track
	deter_track_pair
	place_net
	test_split
optimal	Min_cost1

	Min_cost2
	Min_cost3
	create_bitmat
	improvement
	inv_net
	or_bitmap
	placed
	prrint
	set_bit
	start
	and_bitmat
	determination
or_bitmatr	get_bit
	set_bit
place_con_trans	case_T
	case_t
	place_contact
	place_exit
	rect
place_grid	place_exit
	rect
place_nets	rect
ploce_net	nnplace
	noplace
	onplace
	ooplace
	opplace
	poplace
	ppplace
poplace	add_track
	allow
	check_dif
	deter_best_track
	deter_track_pair
	place_net
	test_split
pplace_dif	rect
ppplace	add_track
	allow
	check_dif
	deter_best_track
	opplace
	place_net
	poplace
	test_split
start	get_bit
	or_bitmap
	set_bit
	main
switch_columns	improvement
two_track	placed
update_dif	upd_del_columns

Eindhoven University of Technology Research Reports
Department of Electrical Engineering

ISSN 0167-9708
Coden: TEUEDE