EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Eindhoven University of Technology

MASTER

Doubly folded transistor matrix layout

Brouwers, A.H.C.M.

Award date:
1988

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

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

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

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
DESIGN AUTOMATION SECTION

DOUBLY FOLDED TRANSISTOR MATRIX LAYOUT

by

A.H.C.M. Brouwers

Master thesis
reporting on graduation work
performed from 04-11-87 to 24-08-88
by order of prof. dr. ing. J.A.G. Jess
and supervised by ir. L.P.P.P. van Ginneken

The Eindhoven University of Technology is not responsible
for the contents of training and thesis reports.

ABSTRACT

We present a flexible module generator for transistor net lists is presented to be used in a top-down VLSI-design environment. The module generator is automatically interfaced with a floorplanner and offers accurate control over aspect ratio and pin positions, while the area remains approximately constant.
The generator is based on a doubly folded transistor matrix. The folding algorithm uses an elegant hierarchical divide and conquer technique, to control the aspect ratio while minimising the area.
A small library of adaptable transistors is used to lay out the module. This library offers a simple but very efficient compaction method.
Steps to optimise the foldresult are discussed as well as an automatically controlled module partitioning to extend the aspect ratio range.
Compared to other automated layout styles, the presented module generator makes smaller modules, that have a great flexibility. The layout of the modules can be customised with respect to all major design parameters: function, speed, design rules, aspect ratio and pin positions.

CONTENTS

1. INTRODUCTION 1

2. THE LAYOUT PROBLEM 3
   2.1 A module generator 3
   2.2 Make intervals 3
   2.3 Folding the transistor matrix 7
   2.4 Laying out the folded matrix 9

3. COMPACTION OF ADAPTABLE TRANSISTORS 11
   3.1 Compaction steps 11
   3.2 Surrounding boxes with overlapping vias 11
   3.3 Using a standard library 12
   3.4 Stretching transistor models 13
   3.5 Selecting and placing a model 14
   3.6 Connecting wires 17

4. IMPROVING THE FOLDRESULT 18
   4.1 Folding-groups 18
   4.2 Re-ordering columns 18
   4.3 A heuristic for re-ordering the columns 19
   4.4 Re-ordering transistors 22

5. PARTITIONING THE TRANSISTOR MATRIX 23
   5.1 Heavily connected signals 23
   5.2 Controlling the aspect ratio 23
   5.3 A linear-time heuristic for the mincut-algorithm 25
   5.4 Shorter diffusion wires 27
   5.5 Single poly-silicon wires 27
   5.6 Folding partitioned modules 27
   5.7 Partitioning power and ground 28
   5.8 Second metal layer 28

6. EXPERIMENTAL RESULTS 30
   6.1 Foldresult to layout 30
   6.2 Re-ordering columns and rows 31
   6.3 Controlling the aspect ratio 31
   6.4 Comparison of layout methods 32
   6.5 Computation time 35

7. CONCLUSIONS AND RECOMMENDATIONS 37
   7.1 Conclusions 37
   7.2 Recommendations 37

8. REFERENCES 39

APPENDICES 40

1. INTRODUCTION

A complex system usually can be divided in several more or less individual subsystems. The subsystems can be connected to each other and make up the complete system. Dividing systems into smaller subsystems is a well known approach in laying out a complex circuit on one chip and is known as stepwise refinement [WIRT71].
To get good results with stepwise refinement it is important that the smaller subsystems can easily match the requirements of a global structure defined on a higher level. In VLSI-design this approach leads to the use of a *floorplanner* [LAUT79].
A floorplanner divides the circuit in a number of connected layout parts, it assigns positions to the layout parts and reserves area for the connecting wires. The floorplanner uses a mincut-algorithm to divide the system in a set of smaller subsystems called *slices* and these slices in even smaller slices. This top-down approach results in a hierarchical ordering of the slices. Each slice consists of one or more smaller slices, called child slices. This division continues until the slices are small enough. At that stage the slices will be called *modules*, functional layout parts with a flexible shape.
After all divisions are made, the floorplanner uses the *shape constraints* to select the best shape of a module. These shape constraints define the the possible shapes of the slices and the corresponding minimal area. The shape constraints of a slice are the sum of all shape constraints of the child slices. The stepwise refinement approach is also used in selecting the final shapes of all slices and modules. Within the shape of one slice, the best configuration and shape of all child slices are chosen.

After all these steps, the system will be represented by a set of modules and their interconnections. All modules will have a relative position and the floorplan is nearly complete. Making a plan for the connecting wires is the final step. The best pin positions are determined, and from these pin positions the number of wires can be derived. The floorplanner then makes room for these wires by creating channels between the modules. This leads to the final floorplan of the circuit.

A module generator then lays out the modules. Of course this module generator has to take the desired shape and pin positions in account and try to match these requirements as well as possible. On the other hand the floorplanner should only ask for shapes and pin positions that can be generated by the module generator. The final shape of the module has also to be predictable.
A good module generator gives complete freedom of pin positions and the final area occupied by the module is independent of these pin positions and of the aspect ratio (width / height). A wide range of aspect ratio's must also be possible. Finally the generated layout must be compact, i.e. the total area must be small.

One of the current projects at the Design Automation Section of the Eindhoven University of Technology is the construction of a silicon compiler based on the stepwise refinement approach. Part of this project is the construction of a flexible module generator.
This report is a master thesis of a graduation project and it reflects the implementation and results of such a module generator. The presented module generator is based on a doubly folded transistor matrix. The generated layouts can be customised with respect to all major design parameters: function, speed, design rules, aspect ratio and pin positions.

A general introduction to the module generator and the layout problem derived from the folding of the transistor matrix are described in chapter 2. Chapter 3 presents the actual module generator. In chapter 4 some improvements to the folding result are presented. Chapter 5 will discuss a partitioning of the modules to get more flexibility and chapter 6 will show the results from various tests.
To make a layout a small library of transistor models is used. This library can be designed by using a interactive layout editor. A manual to design these standard models, as well as other manuals, is given in the appendices.

2. THE LAYOUT PROBLEM

2.1 A module generator

As stated in chapter 1 a module generator generates compact layouts of a module. The description of the module is given in three different files:

- *netlist:* The netlist describes the nets that make the connections between the transistors and I/O-pins within the module. All transistors, I/O-pins and nets have unique names.
- *module-file:* For each transistor in the netlist, the module-file describes its parameters, like channel length, channel width, channel type (n-channel or p-channel) and optional diffusion implant (enhancement- or depletion type transistor).
- *interface-file:* The interface-file is the interface between the floorplanner and the module generator. It specifies the desired aspect ratio (width / height) and desired (relative) pin positions.

From these three inputfiles a *transistor matrix* is extracted. In the transistor matrix each net is represented by a column and each transistor by a row. The coordinates of the connections are the same as given in the netlist.
The transistor matrix is represented by an *interval-file*. The interval-file will be folded in two directions and the resulting transistor matrix, the *foldresult*, will be used to finally generate the layout of the module.
Figure 2.1 gives an overview of the steps to be taken. In the next paragraphs these steps will be explained.

2.2 Make intervals

The netlist

A netlist of transistors allows total freedom for the design of transistor networks. There are no constraints to the number of connections to be made to one signal and the gate, drain and source of a transistor can be connected to any signal. We therefore use a netlist to describe the circuit.

The netlist contains all connections to be made. Each line in the netlist describes one connection between a signal and a transistor. It also states to which terminal of the transistor (gate, drain or source) the connection is made.

The syntax for all lines of the netlist is:

<netname> <transistorname> <terminal>

The netlist of a simple depletion load inverter is given in figure 2.2.

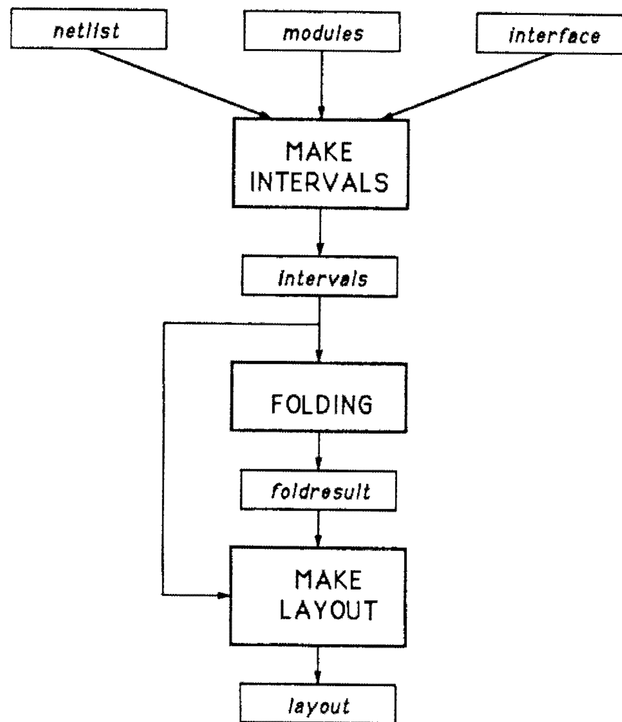
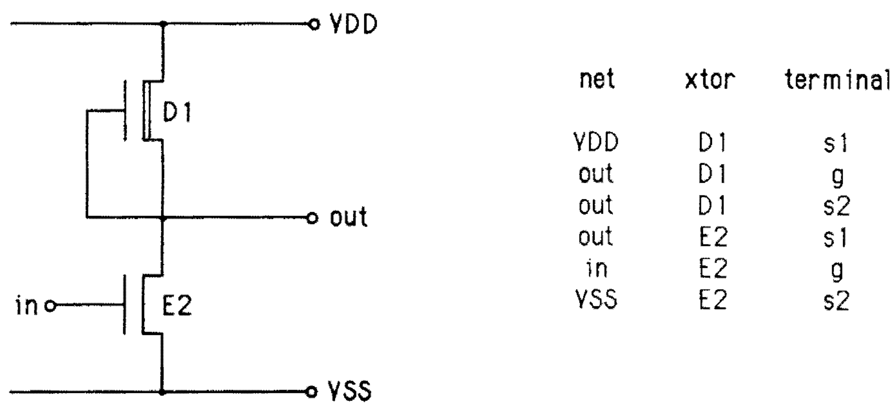


Figure 2.1. An overview of the steps to generate a layout.



net	xtor	terminal
VDD	D1	s1
out	D1	g
out	D1	s2
out	E2	s1
in	E2	g
VSS	E2	s2

Figure 2.2. The netlist for a simple inverter.

There is no actual difference in the layout between source and drain, so they are called s1 and s2.

The module-file

The module-file describes the type of transistor and the size of its channel. There are four different types:

nenh : n-channel enhancement transistor;
ndep : n-channel depletion transistor;
penh : p-channel enhancement transistor;
pdep : p-channel depletion transistor;

The lines in the module_file have the following syntax:

<transistorname> <type> <channel length> <channel width>

The interface-file

The interface-file describes the aspect ratio and pin positions as desired by the floorplanner. The syntax of the interface-file is as follows:

"module" <module name>
"shape" <width> <height>
"pin" <pin name> <coord> <coord>
...
"end"

The first line must always contain the keyword "module", followed by the name of the module. The second line always starts with "shape" followed by the desired width and height of the module. The file has to be terminated with the keyword "end" on the last line.
All other lines start with the keyword "pin" followed by the pinname and the interval of allowed positions of the pin. Figure 2.3 shows the mapping of the intervals. The coordinates are floating point numbers, so only a part of a side can be chosen. This offers the possibility to define relative pin positions on the same side.

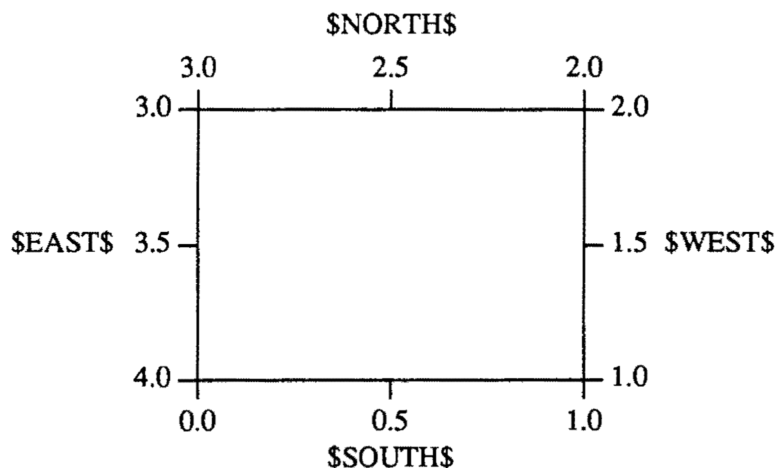


Figure 2.3. The side numbering of a module.

The interval-file

The netlist can be mapped to a two-dimensional transistor matrix. In this matrix all transistors are mapped to the rows and all nets to the columns. The I/O-pins connected to the north-side of the module are all combined on one row called '$NORTH$'. The 'south'-I/O-pins are all combined in the row '$SOUTH$'. The 'west'- and 'east'-I/O-pins are all combined in two columns called '$WEST$' and '$SOUTH$'. These rows and columns are always fixed to the four sides of the transistor-matrix. In this matrix all connections can easily be represented.
To demonstrate this mapping we use the simple latch shown in figure 2.4

Figure 2.4. A simple latch example.

After the mapping of all nets to columns and all transistors to rows, the transistor matrix mapping of the circuit is given in figure 2.5. Where 'S' is a s1 or s2 contact. 'G' is a gate contact and 'X' is a combined gate and s1 contact.

To describe the circuit one could use the whole matrix. The matrix however is very sparse and therefore it is more useful to describe the circuit only by the connections in the matrix. This leads to the interval-file. The interval-file represents the connections in the matrix by only specifying the coordinates of the connections.

The interval-file is also used in the block that makes the layout of the module. At that point the parameters of the used transistors are needed, so they are added to the intervals-file.

The syntax of the interval-file is as follows:

<module name>
<width> <height>
<column> <row> <blocknr> <typenr> <chl> <chw> [<name>]
...

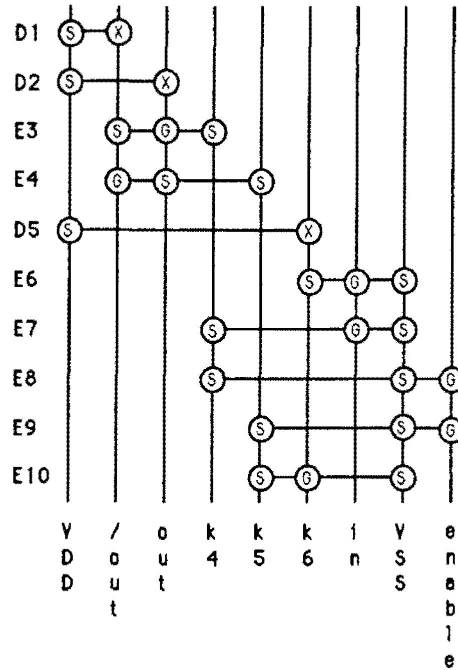


Figure 2.5. The transistor matrix of the latch circuit.

The block number is used by the folding to fold partitioned modules (see chapter 5).

2.3 Folding the transistor matrix

The transistor matrix can be transformed directly to a layout of the module, however, it would give a very inefficient layout with no control over the aspect ratio.
Using a two-dimensional folding algorithm [TEEF88] some nets are mapped to the same column and transistors are mapped to the same row, resulting in a smaller transistor matrix and control over the aspect ratio.

The folding problem can be formally stated as follows: The circuit is specified as a bipartite graph $B(G,N,E)$, with the nodes G representing the gates and N representing the nets. The edges $E \subset G \times N$ represent the gate/net incidences. The circuit is to be realized on a grid of *rows* and *columns*. The set of grid points is represented by $Z \times Z$. The layout of a circuit is determined by a *gate assignment function* ϕ: $G \rightarrow Z$ which assigns gates to columns and a *net assignment function* ψ: $N \rightarrow Z$ which assigns nets to rows. Let $v(n)$ denote the set of neighbors of n: $v(n) = \{g \in G \mid (g,n) \in E\}$. The *span* σ of a net $n \in N$ is an interval of columns defined as $\sigma(n) = [\min_{g \in v(n)} \phi(g), \max_{g \in v(n)} \phi(g)]$. The spans of gates that are assigned to the same column are not allowed to overlap:

$$\forall_{g_i, g_j \in G} [\phi(g_i)=\phi(g_j) \Rightarrow \sigma(g_i) \cap \sigma(g_j) = \emptyset]$$

Since the problem is symmetric the same goes, for the nets. The objective of the folding algorithm is to find the best valid ϕ and ψ subject to some cost function, for instance area.

Using straight orthogonal cutting lines, the matrix is repeatedly divided in *gategroups* (rows) and *netgroups* (columns). After the kth horizontal cut the transistors are partitioned in k+1 gategroups.

$$G = \bigcup_{i=0}^{k} G_i \qquad \forall_{G_i, G_j} [G_i \cap G_j = \emptyset]$$

Similarly the nets are partitioned into l+1 netgroups.

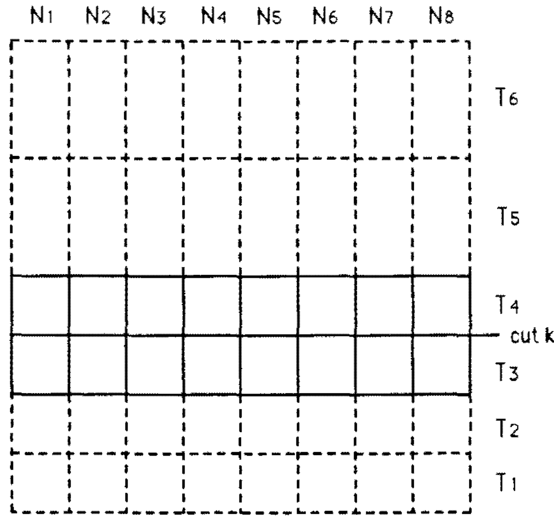


Figure 2.6. The partiotioning after some cuts.

As the exact assignment has not yet been determined the span of a gate will be defined as

$$\bar{\sigma}(g) = [\ \min\{i \mid N_i \cap v(g) <> \emptyset\},\ \max\{i \mid N_i \cap v(g) <> \emptyset\}\ ]$$

The maximum number of rows needed for a set of gates is given by

$$\mu(G_i) = \max_j \#\{g \in G_i \mid j \in \bar{\sigma}(g)\}$$

Notice that this is the exact number of rows if $\sigma = \bar{\sigma}$. A lower bound for the number of rows is determined by the number of transistors that cross a vertical cutting line:

$$\delta(G_i) = \max_j \#\{g \in G_i \mid j \in \bar{\sigma}(g) \wedge j+1 \in \bar{\sigma}(g)\}$$

Since the terminals of the transistors are not allowed to overlap there is another lower bound:

$$\gamma(G_i) = \max_{n \in N} \#\{v(n) \cap G_i\}$$

The mean of these upper and lower bounds can be used to estimate the size of the array:

$$\left(\sum_{i=0}^{l}\frac{\max(\delta(G_i),\gamma(G_i))+\mu(G_i)}{2}\right)\cdot\left(\sum_{i=0}^{k}\frac{\max(\delta(N_i),\gamma(N_i))+\mu(N_i)}{2}\right)$$

However, if the folding reaches the final partitions, this mean of the upper and lower bounds is not a good estimation, because the size of a group usually is the same as the upper bound. Therefore only the upper bounds have to be used to estimate the size of the array:

$$\left(\sum_{i=0}^{k}\mu(G_i)\right)\cdot\left(\sum_{i=0}^{l}\mu(N_i)\right)$$

These estimations can also be used in estimating the aspect ratio of the matrix. As horizontal cuts tend to make the matrix lower and vertical cuts tend to make the matrix narrower, the direction of the cutting can be used to control the aspect ratio and make the estimated aspect ratio to convert to the desired aspect ratio (if possible). Figure 2.7 shows the result of the folding.

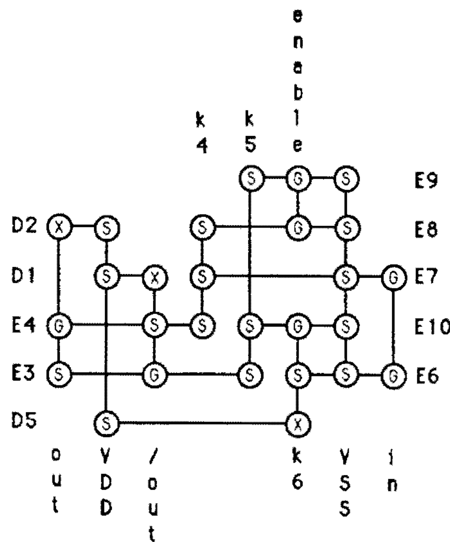


Figure 2.7. Result of folding the transistor matrix of figure 2.5.

This foldresult can be used as a 'floorplan' for the final layout of the module. All relative positions of the transistors, nets and pins are now known. The absolute positions, however, are still to be generated.

2.4 Laying out the folded matrix

The vertical nets of the foldresult are implemented in metal wires, while the horizontal transistors are implemented in diffusion and polysilicon. The foldresult assumes a fixed grid in horizontal and vertical direction. In horizontal direction we can use a fixed grid between the vertical wires. In vertical direction we can use a fixed grid if we assume that all transistors have the same size. Figure 2.7 shows that the gate-connection may be on the left or

right of both source and drain, it may be in between them or it may be on one of them. This can only be realised with different transistor models. Figure 2.8 shows the smallest models that can be used for three different situations.

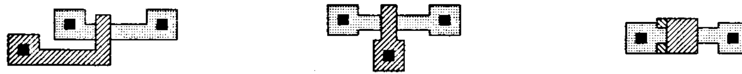


Figure 2.8. Smallest transistor models

If the length of the transistor channel is larger than the room between drain and source the models in figure 2.8 can not be used and we have to use the models given in figure 2.9.

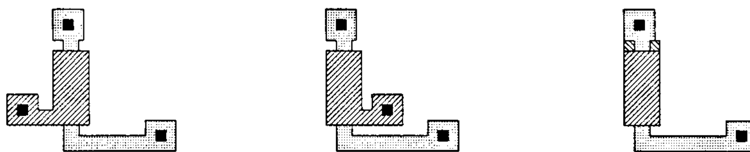


Figure 2.9. Models for long channel transistors

This shows that the assumption of uniform transistor sizes is not realistic. To use a fixed grid in vertical direction we must first find the size of the highest transistor, and then set the grid to this size. This way we act as if all transistors have the same height. Of course, this will cause a great waste of chip area. A floating grid per row can reduce this waste, but a compact layout can only be achieved if no grid is used at all.
In the next chapter the compaction of the layout will be discussed.

3. COMPACTION OF ADAPTABLE TRANSISTORS

3.1 Compaction steps

Using the fixed grid of the foldresult to lay out the module produces a great waste of chip area, as can be seen in figure 3.1a.
A better solution could be the use of a floating grid. For every row the highest transistor on that row determines the gridsize. Figure 3.1b shows only little improvement by this step.
Dropping the idea of a grid offers more freedom to place the transistors in a compact manner. Figure 3.1c shows how transistors can be placed if we only use the contour of the transistors below as a border for a transistor on the next row. Some transistors are now mirrored in the X-axis to give a better fit. Still there is one more step to be made to improve the result. If two diffusion-strips are connected to the same metal strip, their vias are allowed to overlap. The same goes for two poly-strips. A diffusion-metal via however is not allowed to overlap with a poly-metal via. Figure 3.1d shows how this may result in a final layout. On several locations two vias of the same kind partly or completely overlap. There's even one spot where three vias overlap.
In this example the height of the module drops from 246 via 210 and 152 to 106, which means a final reduction to 43 %.

So, to get a compact layout of a folded transistor matrix, the module generator must be able to keep track of the transistors-contours, determine what transistor model gives the best fit, and allow vias of the same kind to overlap, if they are connected to the same metal strip.
The next paragraphs will demonstrate how all these objectives have been reached. In fact, figure 3.1d is the result of automatically generating the layout of the module from its discribtion in the netlist, module-file and interface-file (the I/O-pins are removed).

3.2 Surrounding boxes with overlapping vias

The design rules of a certain technology state the minimal distance between two unrelated tracks of poly and/or diffusion. If we assume that each transistor model has a surrounding box, that is large enough to prevent violating the design rules we won't have to worry about these design rules any more. This surrounding box is half the size of the largest design rule wider than the actual transistor. Surrounding boxes allow us to forget about the actual structure of the transistor and leave us with the problem to place them in a compact manner.
This implementation of the surrounding boxes leads to a layout as shown in figure 3.1c. No overlap of vias of the same kind is possible. To allow vias to overlap they should be left out of the surrounding boxes, but this can also result in an overlap of vias of different types. To avoid this we introduce two types of surrounding boxes. One that prevents everything but poly-metal vias to overlap and one that prevents everything but diffusion-metal vias to overlap. Combining both boxes leaves at the poly-metal via position the possibility to overlap with another poly-metal via and on the diffusion-metal via a possible overlap with another diffusion-metal via. These surrounding boxes are shown in figure 3.2. The two contours we

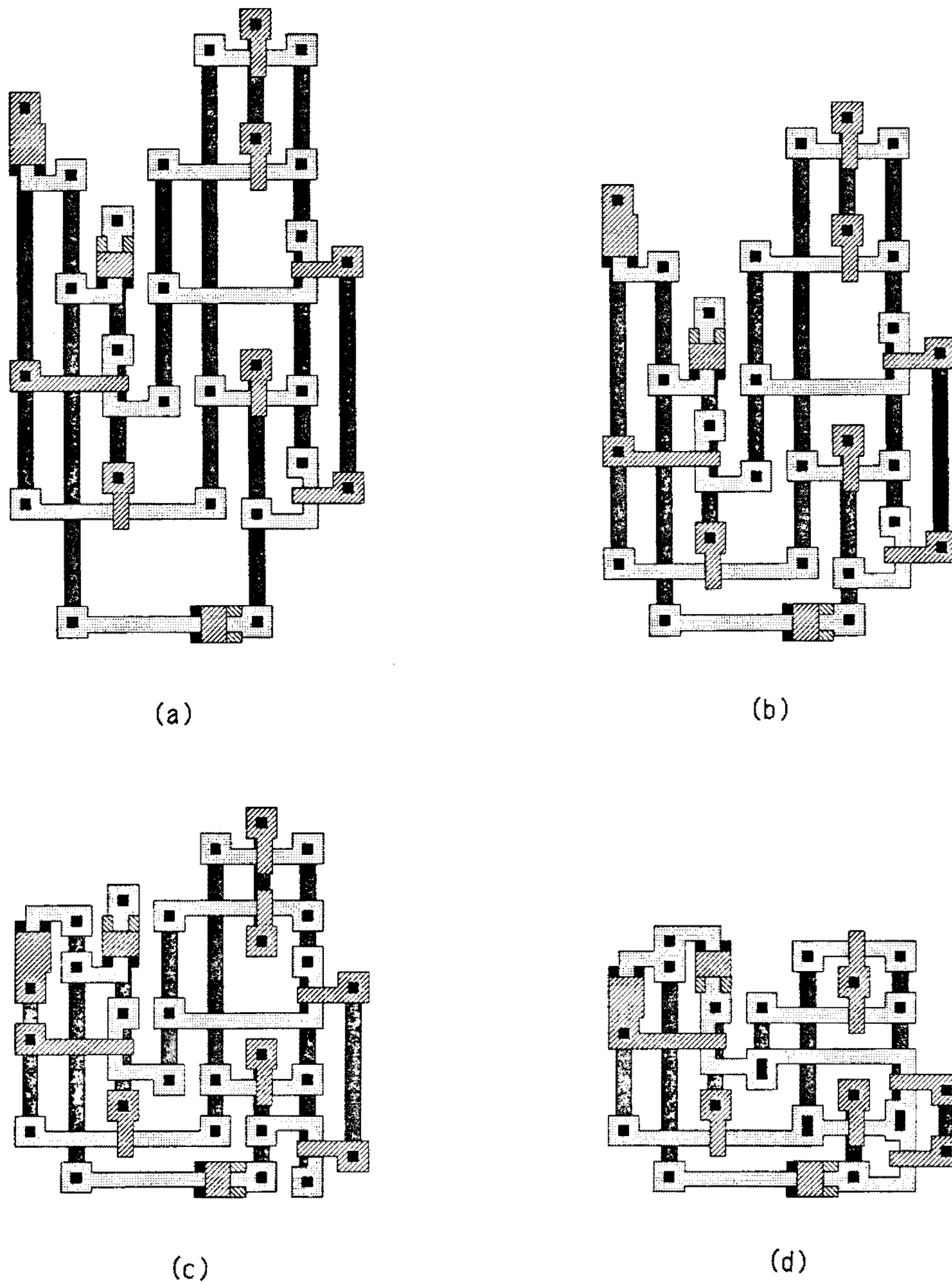


Figure 3.1. Layout compaction steps.

use now allow overlap of the same kind, but prevent each other to overlap.

3.3 Using a standard library

Using an interactive layout editor we can construct a library that contains all transistor models. This library offers for each model a list of boxes of layout elements, described in the layout description language LDM. The surrounding boxes can be simply derived from these boxes by incrementing their size in each direction by half the size of the largest design rule.

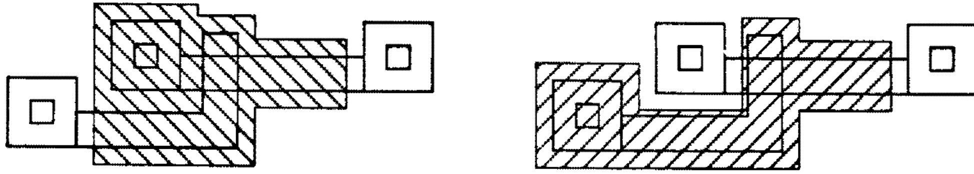


Figure 3.2. Surrounding boxes for diffusion- and poly-overlap.

The surrounding boxes are determined separately and placed in a standard library. To distinguish between poly-via overlap, diffusion-via overlap and no overlap at all, three different 'layers' are used called px, dx and xx.

Figure 3.3 shows the three different boxes for a transistor model. The xx-boxes are generated direct from the poly and diffusion strips. This means that they overlap with the px- and dx-boxes. These overlapping areas are removed from the xx-boxes.

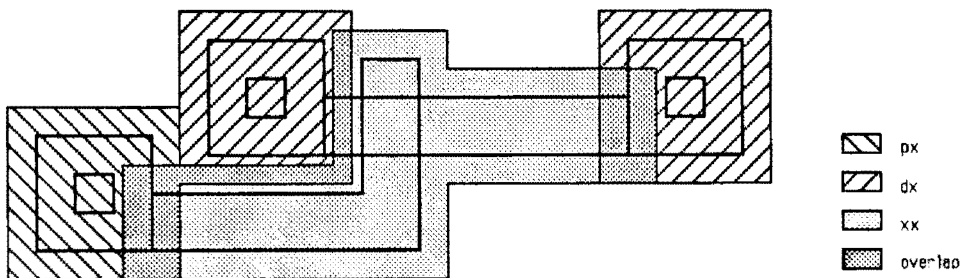


Figure 3.3. Three surrounding box layers and their overlap.

3.4 Stretching transistor models

To avoid a great number of transistor models in the library, it is possible to stretch the models. This way one transistor model can be used to span different numbers of columns. Figure 3.4 shows how one model can be used if two stretch points are introduced. One stretch point to the right of the leftmost via and one to the left of the rightmost via. The foldresult tells to which column a signal of the interval-file is mapped. So combining both interval-file and foldresult the actual column-positions for gate, drain and source can be determined. This is done by a program called 'ctm_stretch'. Row by row, starting at the bottom of the module, it places the characteristics of the transistors in a file with extension .tor. These characteristics are s1-column, s2-column, gate-column, channellength, channelwidth, transistor type (enhancement or depletion) and an optional name. Using the M4-preprocessor, a string substitution leads to the final model with the appropriate stretch.

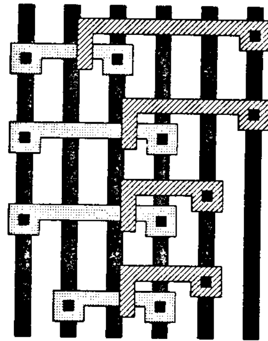


Figure 3.4. Stretching a transistor model.

Not only stretch can be modified using the M4-preprocessor, also channelwidth and -length of the transistor and an optional diffusion implant can be specified. M4 can also be used to select a model from the library. It can select on the order of gate, drain and source, and determine if a transistor is too large to fit between drain and source, so these models can't be selected.

3.5 Selecting and placing a model

For every transistor in the circuit M4 puts all the possible models in a file with extension .c2in. This file is a list of transistors with the following syntax:

<transistorlist> ::= <transistor> <transistorlist> | <transistor>

<transistor> ::= <instance> <definitionlist>

<instance> ::= "instance" <int> <int> <int> <int> <int> <int> [<name>] <eol>

> Instance is the header of a list of definitions. The integers stand for s1-column, s2-column, gate-column, channellength, channelwidth and type of the transistor.

<definitionlist> ::= <definition> <defintionlist> | <definition>

<definition> ::= <newdef> <elementlist>

<newdef> ::= "newdef" <int>

> Newdef is the header of a list of elements, that define one transistor model. The integer gives the modelnumber.

<elementlist> ::= <element> <elementlist> | <element>

<element> ::= <box> | <module-call> | <terminal>

<box> ::= "box" <layer> <int> <int> <int> <int> <eol>

<module-call> ::= "mc" <name> <int> <int> <eol>

<terminal> ::= "term" <layer> <int> <int> <int> <int> <name> <eol>

<int> ::= {<digit>}+

<name> ::= <letter> {<letter> | <digit>}*

<layer> ::= <name>

All lines following the newdef-key are the LDM-description of a model until another newdef or instance-key is encountered. This way each transistor can have an arbitrary number of models to choose from.

All models passed on by M4 can be used to construct a valid circuit. By placing the boxes one by one, starting at the lowest row, we can easily keep track of the outline of the boxes that are already placed. This allows us to select from the possible models the one thats fits best.
To select a model, the extra size that this model will occupy, the increase in maximum height or a combination of both can be taken in account. Experiments showed the extra space times the increase in height to be a good criterion. Extra space here means not only the size of the surrounding box, but also the wasted space beneath it after placement.
Figure 3.5 shows that even a shape belonging to a transistor with larger height can be selected because it gives a better fit.

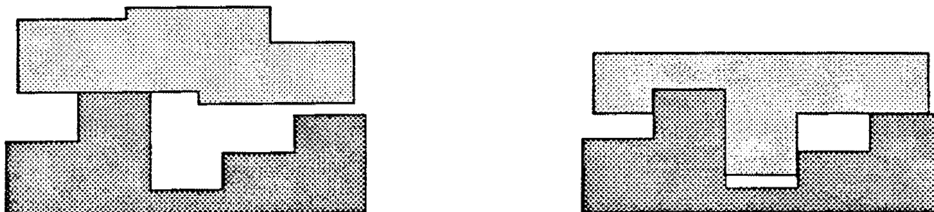


Figure 3.5. Selecting the model that fits best.

The vertical positions of all boxes of a model use the same offset. To place a transistor it is only necessary to compute this offset, called the base.
For both surrounding boxes two separate layers are defined. Both layers are handled in exactly the same way, so we will not make any distinction between them. To represent each box two arrays are used. One to represent the top contour of the box and out to represent the bottom contour. In the following these array will be called top[] and bottom[]. A third array, size[], represents the top-contour of the total of all placed transistors.
To compute the base of a transistor we start at the left of bottom[]. Base will be set to the lowest value that doesn't cause a forbidden overlap. Then we go step by step to the right and every time we come across a forbidden overlap, the base will be adjusted to this overlap. This way the lowest base that doesn't cause forbidden overlaps is found.

While computing the contours the left and right side of the model are also determined. This allows us to calculate the extra area used by the model as follows:

$$\text{extra area} = \sum_{i=\text{'left'}}^{\text{'right'}} \text{base} + \text{top}[i] - \text{size}[i]$$

This of course for both contour-layers.
The difference in maximal height is also easy to get, and so the extra size times the increase in height is easily computed. This criterion is used to select a model. The model with the lowest value is chosen and added to size[].
This model is placed in the outputfile with extension .c2out. It gives the definitionnumber, s1-column, s2-column, gate-column, channellength, channelwidth and the computed base, optionally followed by a name. This file is used to select with M4 to final layout of the module, which is placed in an LDM-file.
Figure 3.6 gives an overview of the steps that lead to the final layout.

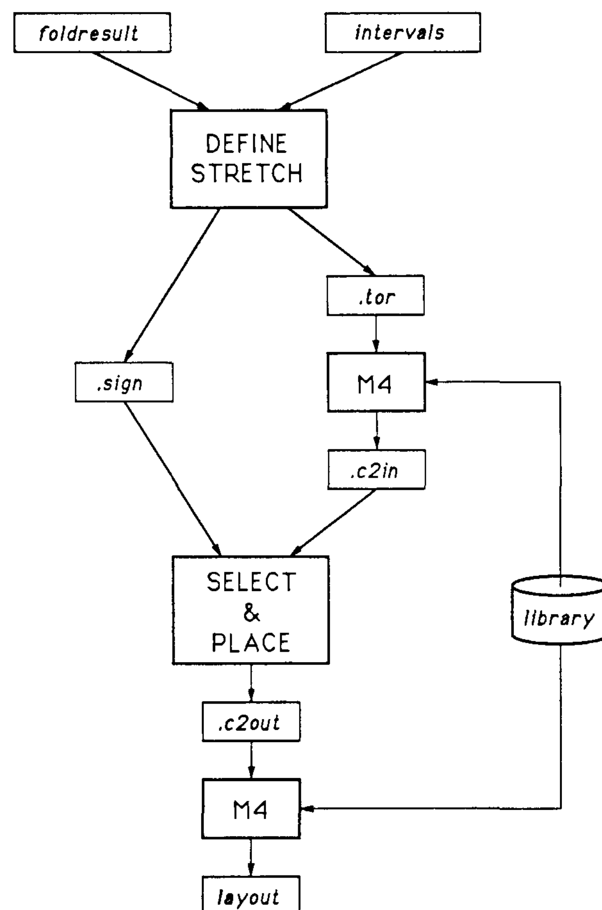


Figure 3.6. Using a model library to make a layout.

3.6 Connecting wires

The folding maps more signals to one column. To make the connections in the layout this causes some problems. First of all, because we don't use a fixed grid, the length of the wires no longer depends on the number of rows it has to span, but of the positions of the highest and lowest connected transistors. So only after all connected transistors are placed we can determine the size and position of a wire. Secondly, vias may only overlap if they are connected to the same wire.

The 'ctm-stretch'-program offers a second outputfile with extension .sign. This file contains the actual columns that wires are mapped to, the number of connections to that wire and the relative position of the wire in the column. This allows the wires to be placed at the same time as the transistors.
All wires that are mapped to the same column are linked in a list. The relative positions of the wires in the same column is used to sort this list from bottom to top. All linked lists are placed in a hashtable for easy access.

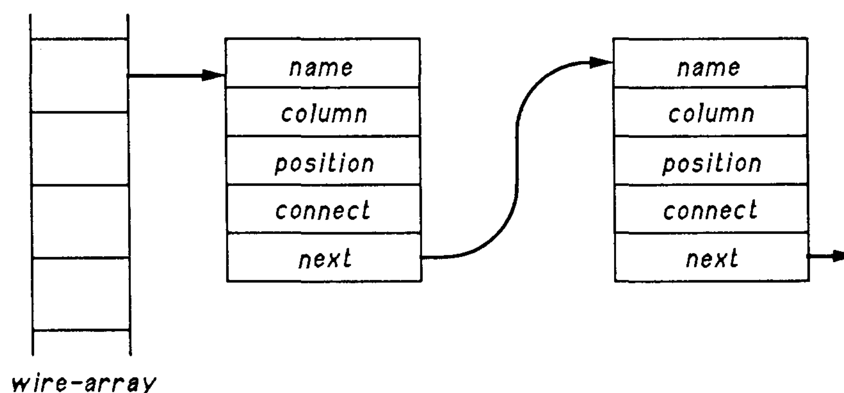


Figure 3.7. Datastructure for wires.

By addressing the first wire in the linked list this wire can be updated after a transistor is connected to it. A wire is updated by setting its bottom-coordinate to the centre of the lowest connected via and setting the top-coordinate to the highest connected via. After these coordinates have been calculated, the number of connections to the wire is checked. If all connections to that wire are now made, the final size and position of the wire is known, so the box-description is placed in the .c2out file. (M4 will not find any string substitutions for this box-description and transfers the description direct to the LDM-layout-file.)
All other connections to a wire in the same column can not belong to this wire anymore, so it is removed from the list. The next wire will now become the head of the list.

4. IMPROVING THE FOLDRESULT

4.1 Folding-groups

The folding-program looks at the transistor matrix in a purely symbolic manner. Due to its general-purpose character, it does not take in account any technology dependencies. The only objective of the folding-algorithm is to compress the size of the transistor matrix by reducing the number of columns and rows. If several mappings all lead to the same result the folding algorithm just picks one of them. It does not look if the area between drain and source is large enough to fit the transistor. In fact it doesn't even distinguish gate, drain and source. It is also incapable of allowing overlap of vias of the same kind.
The folding-algorithm divides all columns in groups. The columns in the same group may be switched around, without causing any overlapping nets or transistors. To the folding algorithm it makes no difference what the order of the columns within a group is, so the ordering of the columns in a group is chosen arbitrary. The only important ordering is the ordering of the groups. The same goes for the rows.
All groups have a unique *groupnumber*, which gives the ordering of the groups. For net-groups the groupnumbers increase from left to right, for the gategroups the groupnumbers increase from top to bottom. Running the folding-program with the '-g' option, this groupnumber is placed after every column and row of the foldresult. This allows us to change, within the groups, the order of columns and rows, without really changing anything to the foldresult.

4.2 Re-ordering columns

In chapter 2 we found that, if the size of a transistor is to large to fit between drain and source, a model with a greater height has to be used. These models usually occupy more space in the module.

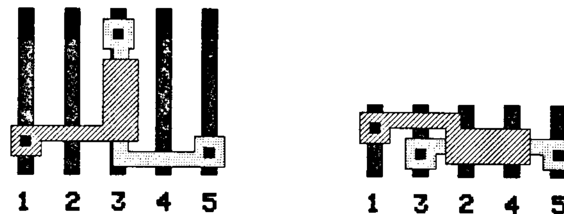


Figure 4.1. Re-ordering columns to use smaller transitor models.

To avoid these situations, we change the order of the columns a bit. Figure 4.1a shows a configuration of a transistor with a large channel length. Because s1 and s2 are to close, the transistor can not lie flat between them. If, however, we change the order of columns 2 and 3, there is enough space between s1 and s2. This way, the transistor can lie flat, resulting in the

situation of figure 4.1b.

Of course we may only swap columns 2 and 3 if they belong to the same folding group. Swapping columns 2 and 3 may, however, also result in obstructing other transistors, that originally were flat, to stay flat. Therefore, we must consider all transistors at the same time. For each transistor we can find what the minimal distance between drain and source should be. This results in a graph, that states for each column the minimal distance it should have to other columns. The nodes represent the columns, and the branches give the desired minimal distances.
The objective of the re-ordering algorithm is to find a mapping that satisfies all desired distances of the graph. If such ordering of the columns is found, we say the graph is solved.

Branches between nodes of different groups may be useless, because the minimal distance between a column of the first group and a column of the second group is larger than the desired distance of the branch. Also, the desired distance of a branch can be to large for any of the possible positions of the columns. Both branches can be discarded, the first can never be violated, the second will always be violated.
Still, the resulting graph may not be completely solvable. If the graph can't be solved, we want the best possible result. The larger the transistor, the more extra height it will have if it can not lie flat. Therefore it is more important that large transistors can lie flat than small ones. Also, if between two columns there are several smaller transistors, the extra height may be added, to make the branch representing the minimal distance between these columns more important.
After the columns are ordered, the importances of the branches whose minimal distances were violated, are added together. The ordering that gives the lowest value is considered to be the best ordering of the columns.

Figure 4.2 shows the graph for the example of the latch in chapter 2. The numbers at the nodes are column number, minimal column and maximal column. The numbers at the branches represent minimal distance and priority. The redundant branches are dashed.

4.3 A heuristic for re-ordering the columns

The minimal distance graph can't be solved in polynomial time. A dynamic programming strategy could solve the graph, or at least find the best solution, but if the size of the groups becomes to large, the number of possible solutions, that will have to be remembered during the computation, grows too large.
A greedy algorithm can be used to place the columns one by one. Using a branch and bound strategy, the greedy algorithm can look a fixed number of placements ahead. In most cases this results in a good placement, but it can not guarantee to find the best possible solution. This paragraph will discuss this heuristic.

The folding divides the columns for the latch-example in 4 groups. Group 1 contains columns 1, 2 and 3, group 2 only contains column 4, group 3 contains column 5 and group 4 contains columns 6, 7 and 8. Using the graph of figure 4.2, the heuristic will be explained, with a lookahead of 4 placements.

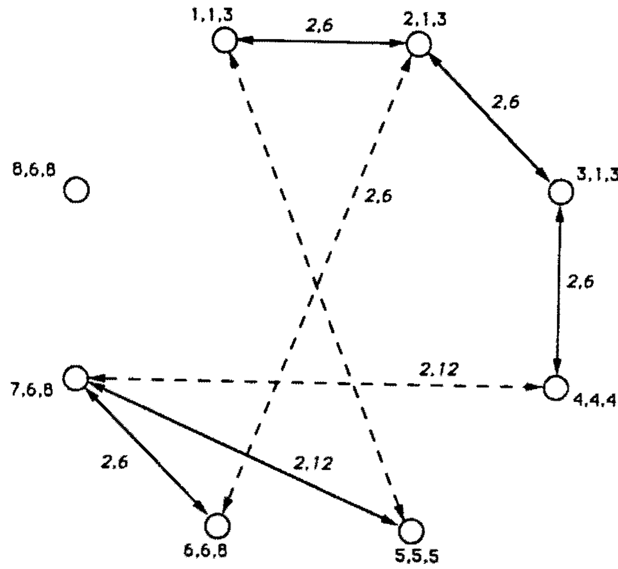


Figure 4.2. Desired distances between the columns.

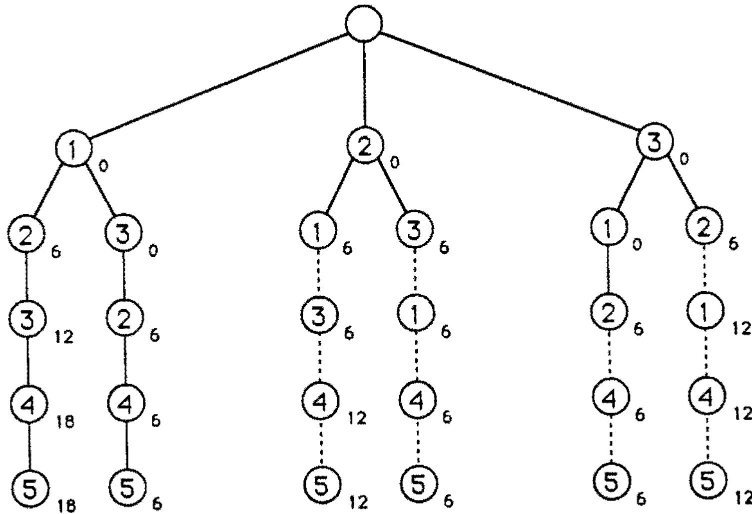


Figure 4.3. Search tree for placement on position 1.

Starting at position 1 there are three columns that can be placed. Figure 4.3 gives the tree that will be checked if the lookahead is 4. In the circle the placed column is written. The index to the left of a circle gives the violation of that particular placement (summed from the top), according to the graph of figure 4.2.

The tree shows that all three columns give the same violation in the following placements. The branch and bound criterion now states, that, if in a branch of the tree a violation is encountered that is no smaller than the best violation so far, the branch will not be examined

further. The best violation can, of course, only be calculated at the end of a branch. This way the dashed branches will not be examined. Using this strategy, column 1 will be placed on position 1.
Continuing this for the other positions, the final placement will be:

position	1	2	3	4	5	6	7	8
column	1	3	2	4	5	6	8	7

Starting at the left, this strategy tends to push the problems to the right of a group. As long as a column can be placed without violating any desired distance, this will be done, leaving all columns that may cause a violation to be placed last.
Just picking the columns to be placed one by one, a column, that has no branches to others, can be placed at a point, where also a column, that has branches could be placed. At that point it makes no difference, but in future positions it can cause unnecessary problems. The branchless column can be placed on any position, but the one with branches can only be placed on some positions. Therefore it is better to place the column, that may give the most problems, i.e. has the most branches, as soon as possible.
Before a placement on a position starts, a priority list of the placement order of the columns is determined. This is done by adding the priorities of all branches to one column, that are not connected to a placed column. This sum is a good criterion for the problems a column can give in future placements. By applying this list, the columns that may give the most problems will always be considered first and be placed as soon as possible, while columns that never will cause any problems will only be placed if all other columns failed.
This strategy still causes problems to be pushed to the right of a group, but now it will try to save the easy columns for this part of the group. Less accumulation of problems at the right side of a group may occur.

Using a lookahead of k, the number of steps to be taken at the first position of a group is:

$$\frac{n!}{(n-k-1)!}$$

for selecting a single column at a position, where n is the number of columns in that group. Summing this over all positions, we find the order of the algorithm to be $O(n^{k+2})$ where n is the number of columns in the largest group. The lookahead should be large enough to span the largest minimal distance between two groups. This can be determined by the largest transistors. Normally this lookahead will not be larger than 4, so we used a fixed lookahead of 4, resulting in an order of $O(n^6)$. Usually groups are small (3 to 10 columns). These small groupsizes make the algorithm to run fast, despite its high order.
However, the speed can be increased. This is done by assuming, that a placement without any violation is possible. The branch and bound lookahead originally starts with a very high best violation, which ensures the computation of the violation of the first branch. Now we start with a best violation of zero. This causes the branch and bound criterion to discard all branches that cause a violation, and, if possible, find a violation-free placement. If no violation-free placement can be found, the branch and bound set is preformed once again,

now with a very high initial best violation.
At the start of a new group, it usually is possible to place columns without any violation. Here, the placement is almost linear. At the end of the group, violations may have to occur, but at that time, the groupsize is very small and the algorithm still works fast.

If a situation occurs, where a violation is inevitable, this violation may as well be made as soon as possible. This prevents accumulation at the end of the group and the high transistors will not be grouped in the same columns. This can easily be achieved by starting the placement after a violation in the lookahead is found, with this violation as the best violation.

A great number of tests showed that the algorithm runs in only a few seconds for up to 150 columns, with highest groupsize of 15.

4.4 Re-ordering transistors

A second limitation of the folding is the inability to allow overlapping vias. Using a simple left-edge algorithm that allows overlap of vias of the same kind, the result of the folding can be improved.
Within one row-group, all transistors of that group can be moved to any row in that group, without causing nets to overlap. By placing all the transistors in a group again, using this left-edge algorithm, all possible overlaps are taken into account. The left-edge algorithm is linear and it can easily be proven to give an optimal result.

Figure 4.4 shows a simple demonstration of the way a row can be won by letting vias overlap. In various examples it showed that up to 8 rows were won, using this re-ordering of the transistors.

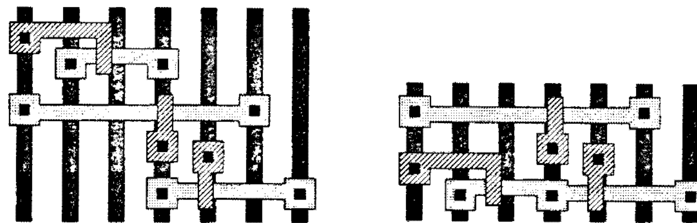


Figure 4.4. Re-ordering transistors with left-edge allowing vias to overlap.

Re-ordering the transistors may not be done before the columns are re-ordered. Due to the fact that within a netgroup there can be no situation where a transistor 'ends' in the group and another transistor 'begins' in that group on the same row, the columns may be switched freely within the group, without ever causing any overlap. The left-edge algorithm however may result in a situation, which might give overlapping transistors if the columns are switched.

5. PARTITIONING THE TRANSISTOR MATRIX

5.1 Heavily connected signals

Power- and ground-signals, as well as clock-signals, can be connected to a great number of transistors. These long wires usually prevent the folding to produce a transistor matrix with more columns than rows. For instance, if one signal has 40 transistors connected, the minimal height of the transistor matrix must at least be 40. In general, the folding will not be able to bring it below 50, due to transistors, that cross the signal wire.
This produces a strong limitation to the range of aspect ratio's that can be reached. In practice, the aspect ratio hardly ever comes above 1.0 and for large modules it might not even reach 0.5. A very important requirement of a good module-generator, however, is a wide range of aspect ratio's, that can be very well controlled. Hence, we have to avoid these long wires in the module to get higher aspect ratio's.

One way to achieve this is cutting the wires in two and connect half of the transistor to one wire and half to the other. The transistors can be divided random, but a mincut algorithm offers a more elegant way to deal with this problem, as will be seen in the next paragraphs.

5.2 Controlling the aspect ratio

The mincut algorithm divides all transistors in two groups in such way, that the number of wires connecting the two groups is minimal. If we put all transistors of one group to the left of the module and the others to the right, we have partitioned the module in two blocks. Some signals now are used in both blocks. Because the signals are vertical wires, they have to be connected by horizontal strips. These horizontal strips are the only elements that cross the border between the two blocks.
Now we have two blocks, that each can reach a highest aspect ratio between 0.5 and 1.0. The combined module can therefore reach an aspect ratio between 1.0 and 2.0. To get even higher aspect ratio's we might partition both blocks and get for the total of 4 blocks a maximal aspect ratio between 2.0 and 4.0. This partitioning of blocks can be continued, but for smaller circuits the number of extra elements that are introduced (for each cut wire we introduce a strip and an extra wire) may dominate the number of original elements. This will cause an increase of total size of the module, which should be avoided.
Partitioning the module increases the maximal aspect ratio that can be obtained, it also may increase the minimal aspect ratio. Both size of the transistor matrix and the desired aspect ratio influence the number of partitions that give an optimal result in aspect ratio and minimum area of the module. Experiments showed that a good result will be obtained if the number of columns after folding is about 15 for each block. This rule can be applied to determine the number of blocks by the desired aspect ratio and the size of the transistor matrix. The size of the transistor matrix is the number of columns times the number of rows of the matrix before folding.
A great number of tests resulted in the empirical relation between the size of the matrix and

the size of the foldresult to be:

$$\text{foldresult} = 2 \cdot \text{size}^{0.75}$$

Also:

$$\text{foldresult} = \text{width} \cdot \text{height}$$

and:

$$\text{aspect ratio} = \frac{\text{width}}{\text{height}}$$

gives the width of the foldresult to be:

$$\text{width} = \sqrt{2 \cdot \text{size}^{0.75} \cdot \text{aspect ratio}}$$

This will result in an optimal number of blocks:

$$\#\text{blocks} = \frac{1}{15}\sqrt{2 \cdot \text{size}^{0.75} \cdot \text{aspect ratio}}$$

This number of blocks will be rounded to the nearest power of 2 to determine the actual number of blocks.

Though an extra number of elements and wires are added to the transistor matrix by the partitioning, the result in total area may still be better, because now two separate blocks have to be folded at the same time. If a module is partitioned, the size of an individual block is about 1/4 of the original module. The addition of a few extra strips and wires makes them only a little larger. Using the above relation between size of the foldresult and size of the transistor matrix it is easy to see, that the sum of the individual blocks is smaller than the original result. Of course, the connecting strips between the two blocks put some constraints on the folding, but in general, partitioning the module in a few blocks doesn't increase the area of the foldresult significantly.
So, depending on the desired aspect ratio and the size of the transistor matrix, we have a good criterion to control the aspect ratio and at the same time keep the total area of the module constant. Table 5.1 give the results for a test on the module 'five' (177 transistors and 95 signals).

blocks	minimal ratio	maximal ratio	minimal area	maximal area
1	0.35	0.55	5500	6345
2	0.68	0.85	3630	3848
4	0.75	1.86	3066	4200
8	1.17	3.50	3255	4002

TABLE 5.1. Aspect ratio for partitioning module 'five'.

This module was folded with all 45 I/O-pins in the upper side of the module. Aspect ratio's

smaller than 0.68 were for that reason not achievable if the module was partitioned. Without the partitioning an aspect ratio of 0.35 was reached, but the area of the foldresult was 6345, so the smaller aspect ratio was mostly the result of making the module higher, not of making it narrower. For this example, the automatic control of the number of blocks gives only an unpartitioned module if the desired aspect ratio was 0.2 or smaller.

5.3 A linear-time heuristic for the mincut-algorithm

The mincut partitioning problem consists of finding a partition of a set of cells into two blocks, such that the number of nets which have cells in both blocks is minimal. In general, this process is subject to a *balancing condition*. Allowing a certain deviation, this condition keeps the sizes of both blocks in balance, preventing all cells to move to one block.
No polynomial-time algorithm is known to compute the exact and optimal solution to this problem. Since networks may be very large, a practical algorithm must employ heuristics. At the 19th Design Automation Conference in 1982 Fiduccia and Matteyses presented a linear-time heuristic [FIDU82].

The basic idea of the algorithm is to move one cell at a time from one block to the other and compute the number of nets that will be cut after the move. The balancing condition is used to determine the block from which a cell has to be selected for a move. A moved cell will be locked to prevent moving it back. After all cells have been moved, the best partition encountered during the pass is taken as the new partition. This partition can usually be optimised by a second pass. All locked cells are made free again and moving some of them back can give a better result. Additional passes may be performed until no further improvements are obtained (in practice this occurs after just a few passes).
The best cell to select from a block is the one that gives the most gain by moving it. The cellgain can be defined by the decrease in the number of cut nets if it is moved from its current block to its complimentary block. This can also be a negative number.

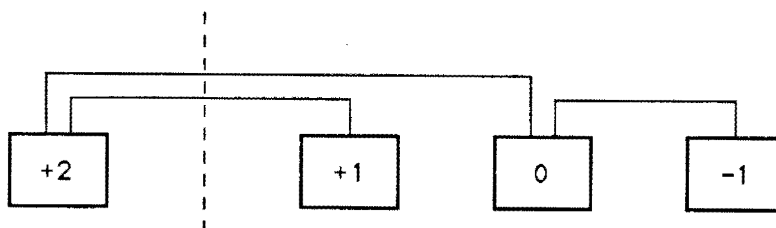


Figure 5.1. Examples of cellgains.

To select the cell with the best gain from a block, "bucket" sorting is used. This is done using an array whose k-th entry contains a linked list of all cells with cellgain k. Direct access is achieved by using the maxgain index.
Using the cellgain also simplifies the work to compute the number of cut nets after a move, by just subtracting the cellgain from the previous number of cut nets.

After a cell is moved it is taken out of the bucket array and placed in a free-cell-list which will be used to fill the buckets again at the start of a new pass. This makes it possible to use cells that are always locked, like for instance I/O-pins, that are on the east or west side of the module. These cells are not allowed to move under any circumstance.
If a cell is moved, the gains of the cells connected to the same nets, can change. Therefore these cellgains will be updated and the cells will be taken from their bucket and placed in another bucket. To update the cells, only the 'critical' nets have to be considered. A net is said to be critical, if there exists a cell connected to it, which would change the net's cutstate if it is moved. Non-critical nets can never change the gains of the connected cells and therefore need not to be considered.
Before the move of a cell the connected nets have to be checked. If the net has no cells in the complimentary block, the gains of all connected free cells in the current block have to be incremented, because moving them can not change the cutstate of the net anymore (figure 5.2a). If the net has only one cell in the complimentary block, the gain of this cell has to be decremented, because moving this cell can no longer change the cutstate of the net (figure 5.2b).
After the move all nets connected to the cell have to be checked again. If there are no cells left in the current block, the gains of all connected free cells in the complimentary block have to be decremented, because they may cause an uncut net to be cut (figure 5.2c). If there is just one connected cell left in the current block, the gain of this cell has to be incremented, because now a move of this cell can change the cutstate of the net (figure 5.2d).

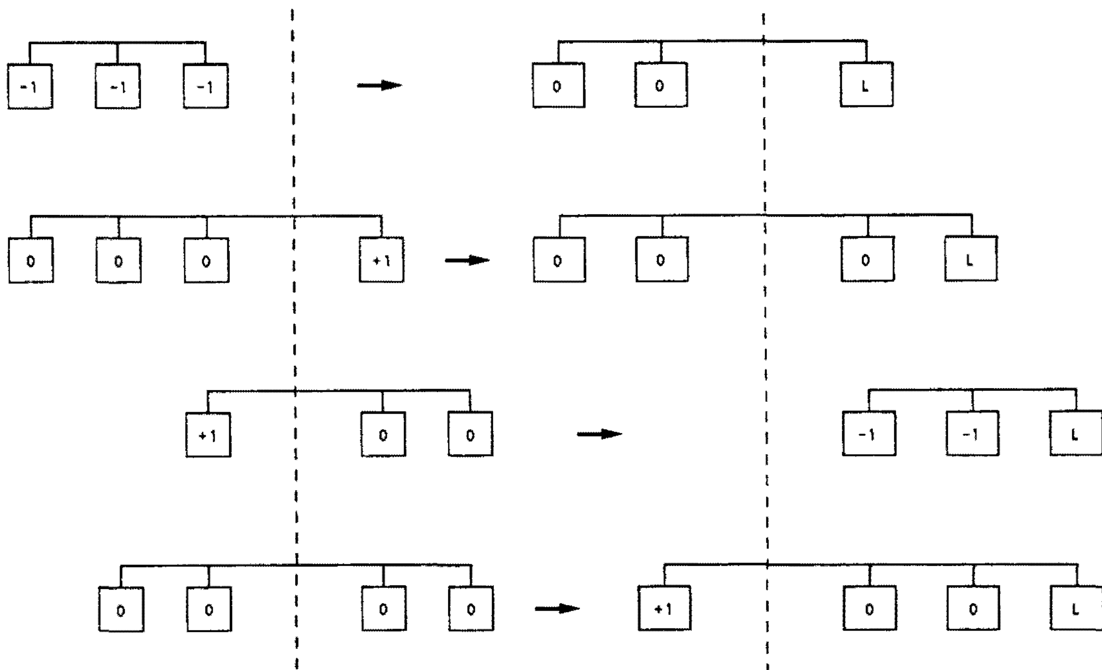


Figure 5.2. Updating critical nets.

No more than four update operations per net are performed during one pass of the algorithm. In [FIDU82] this is proven, as well as the time-linearity of the heuristic.

5.4 Shorter diffusion wires

Due to the p-n junction between diffusion and substrate, a diffusion wire has a high capacitance. This capacitance slows down the circuit and is therefor undesired. The value of the capacitance is proportional to the size of the diffusion wire, hence, long diffusion wires should be avoided.
Unpartitioned module can have diffusion wires running over the total width of that module. The partitioning of the module offers a great opportunity to bound the length of diffusion wires, because the only elements allowed to cross the partition-lines are poly strips. An upper limit for the diffusion wire length is also the maximal width of a block.

So, not only the control over the aspect ratio benefits from the partioning, the performance of the circuit is also improved by it. Two important factors to the decision of partitioning modules.

5.5 Single poly-silicon wires

A net that is cut by the mincut algorithm may have on one side of the cut just one connected cell. Splitting these nets in two and adding a connecting strip only increases the area of the transistor matrix. This leads to an unnecessary extra increase in the foldresult, so these cuts must be ommitted.

However, as we have seen in the last paragraph, the circuit may have a better performance if the diffusion wires a kept within the bounds of the partition. Therefore before cutting a net we look if the net has only one connection on either side of the cutting line. If so, the net will only be cut if this connection will not be layed out in poly-silicon. Tests showed that, due to this action, the area of the foldresult reduced by 2 %.

5.6 Folding partitioned modules

The folding uses a mincut algorithm that applies a different heuristic [TEEF88]. Without further action, this may result in a different partitioning. The strips may not be the only elements that cross the partition-line and some strips won't cross it at all. Experiments showed that this gives poor results, because the wrong signals were cut and others that should be cut were uncut. The folding didn't treat the partitioned module as two (or more) connected blocks, but as one undivided module. The partitioning usually resulted in a better control over the aspect ratio, but in most cases it also gave a larger foldresult, due to the extra wires and strips.

To force the folding to use the same partition, we added the '-b' option to the folding. With this option, the folding reads the blocknumber of an element from the interval-file. Before the folding uses its own mincut-algorithm, it first partitions the module in the previously defined blocks.

This partitioning is done step by step, each time taking the largest block that can be partitioned. After each step the aspect ratio of the module is calculated. As soon as this aspect ratio rises above the desired aspect ratio, the partitioning is stopped and the actual folding is started. This approach to control the aspect ratio is similar to the one used in the folding, where a partitioning of the netgroups tends to decrease the height of the module and a partitioning of the gategroups (cells) tends to decrease the width of the module.
Partitioning the module in all blocks might result in an aspect ratio that is too high and can't be reduced sufficiently by the partioning of the gategroups. Moreover, too many partitions in one direction may obstruct any partition in the other direction.

After the folding can't find a better result by partitioning groups, a second stage of the folding starts. In this stage two adjacent groups are merged and the folding tries to find a better partitioning for them. This is done for all pairs of adjacent groups.
This optimisation step may result in saving a few columns or rows. However, it also destroys the original partitioning of the module. This means, that the poly strips are no longer the only elements that cross the partition-lines, but diffusion wires may also cross these lines. In fact, this usually occurs several times, resulting in much longer diffusion wires.

A solution to this problem is to skip in the second stage only those netgroup pairs, that have nets that belong to a different block in the original partition. This avoids skipping the second stage completely and still keeps the upper bound for the diffusion wires.

5.7 Partitioning power and ground

The global router, that connects all modules of the floorplan to one circuit, demands the power and ground wires to run through the module, i.e. the wire has on both sides an I/O-pin. It also connects all I/O-pins in the same channel, and uses only metal-wires, because of the relative high currents.
In the module it is also best not to make connections between two power or two ground wires in poly. Therefor, if we cut a power or ground wire, the two parts are not connected by a poly-strip, but they get two I/O-pins each. These I/O-pins will only have a metal-terminal, while all other I/O-pins can only be connected via a poly terminal.

5.8 Second metal layer

The partitioning of the module offers an excellent possibility to use a second metal layer. All connecting strips can use this second metal layer. It has no design rules to any other layer and can be layed on top of all other layers. This offers a much more compact layout of the module.
The only restriction to the second metal layer is that it can only be connected to the first metal layer and this connection may not be on top of a poly-metal or diffusion-metal via. Hence, only the vias between first and second metal layers are 'visible' to the rest of the layout.

To implement the second metal layer, a third contour type has to be used to separate the strips in this layer. It also requires a different approach to the overlapping of the vias. The vias may overlap anything but the contour of the new type and the poly-metal and diffusion-metal vias. To achieve this, the metal-metal2 vias may not be placed on a position where only the px-layer or only the dx-layer is used.
A more difficult situation occurs at placing a transistor after a strip in second metal is placed. Now both px- and dx-contours have to be adjusted at the same time to be able to let the metal-metal2 via to overlap anything but an other via.

Due to the fact that the second metal layer may overlap all other layers, the aspect ratio will be hard to control. It depends on the number of strips used and on the positions of the vias how much space can be saved. The folding however can not take the overlapping into account and uses for each strip the same space as for a transistor.
The second metal layer strips usually are grouped at the partition-line between two blocks. This implies a number of rows can be save at these positions. However, at the left or right side of the module there will hardly ever be any strips, so no rows can be saved there. This will cause the module to remain high at the edges and drop lower in the centre. This can not be avoided by the folding or adjusted after the folding.
The strips can not be left out of the folding and after the folding be introduced again, because this may lead to two wires that have to be connected by a strip that have no common vertical position and therefore can't be connected by a horizontal strip.

A solution to this problem has not been found yet and the second metal layer has not yet been implemented.

6. EXPERIMENTAL RESULTS

6.1 Foldresult to layout

The main goal of the project was to transform the foldresult into a layout of the module. This transformation must be linear with respect to aspect ratio and chip area.
A great number of tests showed that the linearity was independent of the aspect ratio, area or number of partitioning blocks. The results are presented in table 6.1:

		foldresult		*layout*		*factor*	
module	*size*	*area*	*ratio*	*area*	*ratio*	*area*	*ratio*
hell84	192	108	0.75	49920	0.49	462	0.65
data	414	187	0.65	87936	0.42	470	0.65
adc	1170	351	0.48	159600	0.33	454	0.69
mp5	1536	378	1.17	176328	0.78	466	0.67
logic	1632	495	2.20	229320	1.51	463	0.69
four	3952	990	1.10	500388	0.69	505	0.63
cnt4.4	12936	1736	1.81	700398	1.43	403	0.79
loc	12056	2236	0.83	967680	0.61	432	0.73
five.2	20223	3850	0.79	1801704	0.54	467	0.68
five.4	25875	3066	1.74	1506816	1.14	491	0.66

TABLE 6.1. Area and aspect ratio of foldresult and layout.

Table 6.1 shows an almost constant factor of both aspect ratio (mean factor is 0.68) and area (mean factor is 461). The size of the transistors is of some importance to the area and aspect ratio of the layout. If a lot of transistors have a large channel-length, they probably can't all lie flat, so the height of the module will increase. Wide transistors may also increase the height of the module, because they will need a model that has a greater height. Module 'four' has only long or wide transistors, so it can't be compacted as much as other modules. Module 'cnt4.4' has no long or wide transistors at all, so the compaction is much better, resulting in a lower area factor and a lower ratio factor.

This result makes it possible to predict the area and aspect ratio of the layout as:

$$\text{area} = 450 \cdot \text{foldresult}$$

$$\text{aspect ratio} = 0.7 \cdot \text{desired ratio}$$

if we assume, that the desired aspect ratio can be reached by the folding.

6.2 Re-ordering columns and rows

The deviations in table 6.1 can also be explained by the fact that the tests were run with the re-ordering of columns and rows. In some examples more rows were saved than in others, so the height of the layout was a little harder to predict. Skipping the re-ordering of the modules might give a better control over the aspect ratio as will be demonstrated in table 6.2, but it also results in larger modules.

desired	*factor without re-ordering*		*factor with re-ordering*		*rows*
ratio	*area*	*ratio*	*area*	*ratio*	*saved*
0.25	390	0.80	348	0.90	5
0.33	359	0.86	324	0.97	8
0.50	409	0.76	393	0.82	1
0.67	408	0.78	393	0.81	2
1.00	422	0.75	387	0.82	0
2.00	389	0.82	394	0.81	1

TABLE 6.2. The effect of re-ordering columns and rows for module cnt4.

6.3 Controlling the aspect ratio

The final shape of the module highly depends on the results of the folding. Both aspect ratio and area are a linear function of the results of the folding. Therefore it is important to have a good control over the folding. In chapter 5 we found that a good control over the aspect ratio can only be obtained by partitioning the module. Table 6.3 shows the results of folding the module 'five' unpartitioned and partitioned in respectively 2, 4, 8 and automatic number of blocks. Module 'five' has 45 I/O-pins, which were all placed on the upper side of the module. For this reason, modules narrower than 45 columns could not be formed. Small aspect ratio's were only reached by making the module higher, causing a great increase in area. Table 6.3 clearly shows, that the control over the aspect ratio of an unpartitioned module can be very poor. Even partitioning in two blocks gives little control over the aspect ratio.
The automatic control of the number of partitioning blocks gives a good control over the aspect ratio in a wide range, where the area of the module remains approximately constant, as demonstrated in figure 6.1 and 6.2.

Figure 6.2 gives the obtained area as function of the desired aspect ratio. A more interesting picture is the obtained area as function of the obtained aspect ratio. This is given in figure 6.3. If we forget the large deformation due to the very low desired aspect ratio, figure 6.3 shows, that the area of the foldresult remains more or less constant within the aspect ratio

desired	*1 block*		*2 blocks*		*4 blocks*		*8 blocks*		*automatic*	
ratio	*area*	*ratio*	*area*	*ratio*	*area*	*ratio*	*area*	*ratio*	*area*	*ratio*
0.2	5684	0.42	-	-	-	-	-	-	5684	0.42
0.3	5684	0.42	3848	0.70	-	-	-	-	3848	0.70
0.4	6345	0.35	3848	0.70	5720	0.47	-	-	3848	0.70
0.5	5865	0.44	3848	0.70	-	-	7375	0.47	3848	0.70
0.7	5500	0.55	3795	0.80	4200	0.75	-	-	3795	0.80
1.0	5564	0.49	3819	0.85	4012	0.87	4002	1.19	4012	0.87
1.5	5564	0.49	3630	0.83	3102	1.40	3450	1.63	3102	1.40
2.0	5564	0.49	3630	0.83	3066	1.74	3680	1.74	3066	1.74
3.0	-	-	3630	0.83	3276	1.86	3255	2.66	3255	2.66
4.0	-	-	-	-	3150	1.79	3584	3.50	3584	3.50
5.0	-	-	-	-	-	-	3432	3.15	3432	3.15

TABLE 6.3. Controlling the aspect ratio for various partitions of module five.

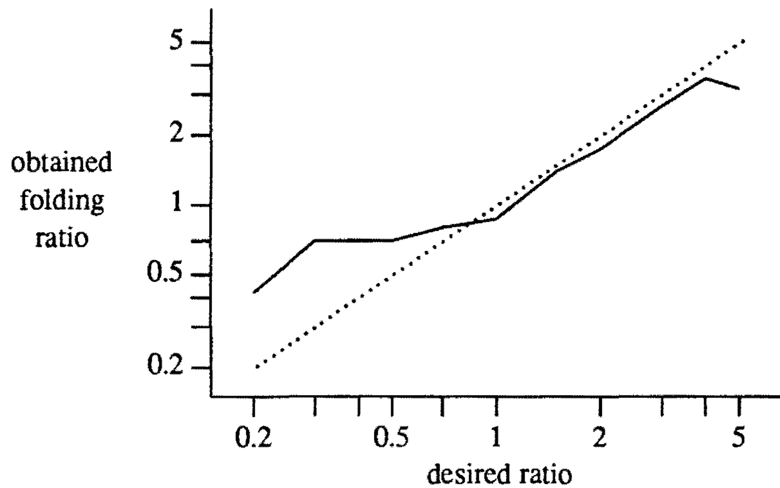


Figure 6.1. Obtained ratio versus desired ratio.

range from 0.7 to 3.50. This constant area is also found in the actual layouts of the module within the aspect ratio range from 0.55. to 2.66 as shown in figure 6.4.

Figure 6.5 shows three layouts for the circuit 'five' at different aspect ratio's.

6.4 Comparison of layout methods

To compare this layout style with other automatic layout styles, a number of experiments

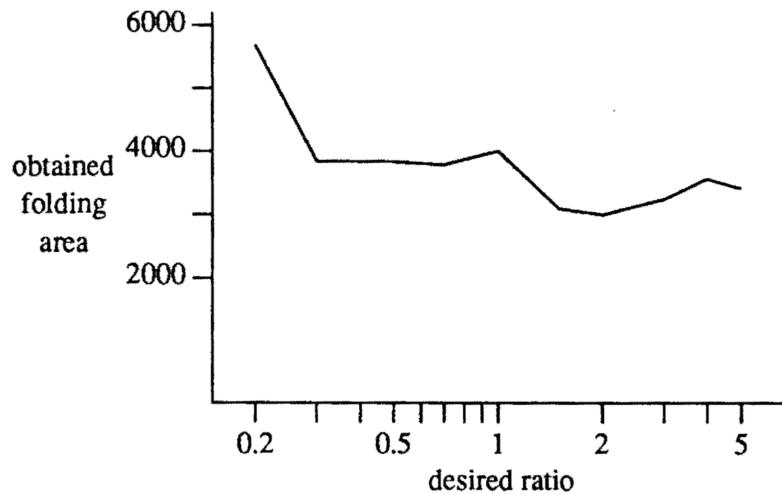


Figure 6.2. Obtained folding area versus desired aspect ratio.

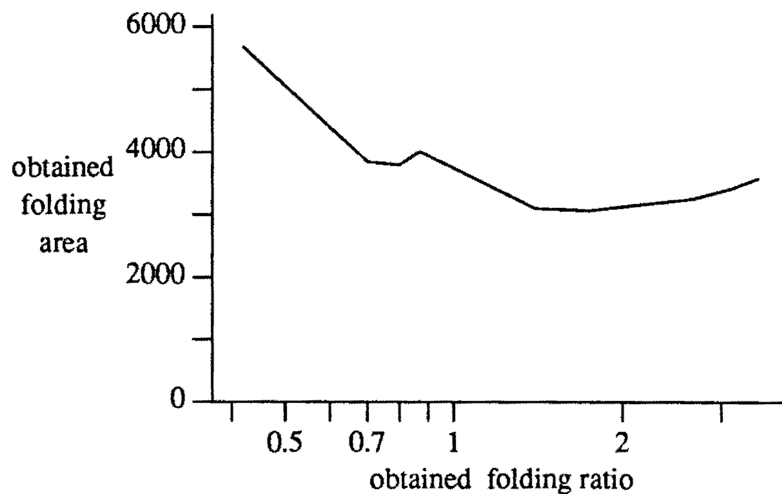


Figure 6.3. Obtained folding area versus obtained folding ratio.

were done. All systems designed for the same nMOS process, with the same design rules. We compared the results with a conventional gate matrix generator [LIES87] and a standard cell place and route system [THEE85]. Table 6.4 shows the results of these methods. This table shows, that the two-dimensional folded transistor matrix reduced the area of the gate matrix layout to 42%-73% and the standard cell to 37%-66%. A manual designed layout for the module cnt4 has an area of 0.183. This is only 3.11 times smaller than the doubly folded transistor matrix layout.

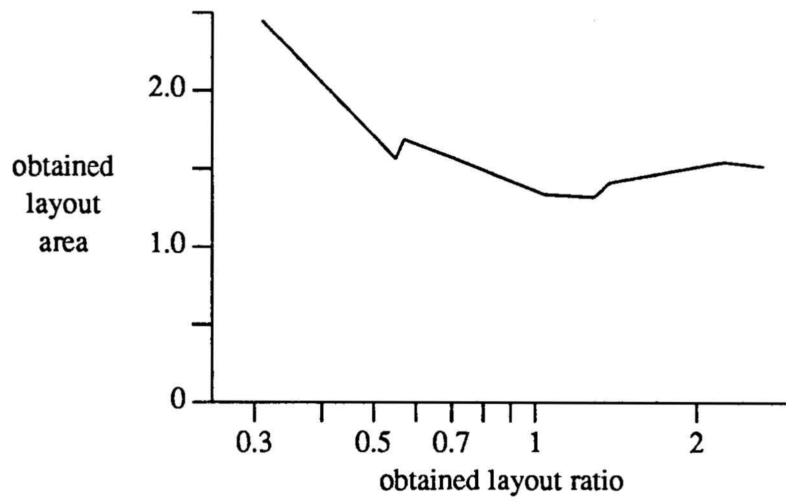


Figure 6.4. Obtained layout area versus obtained layout ratio.

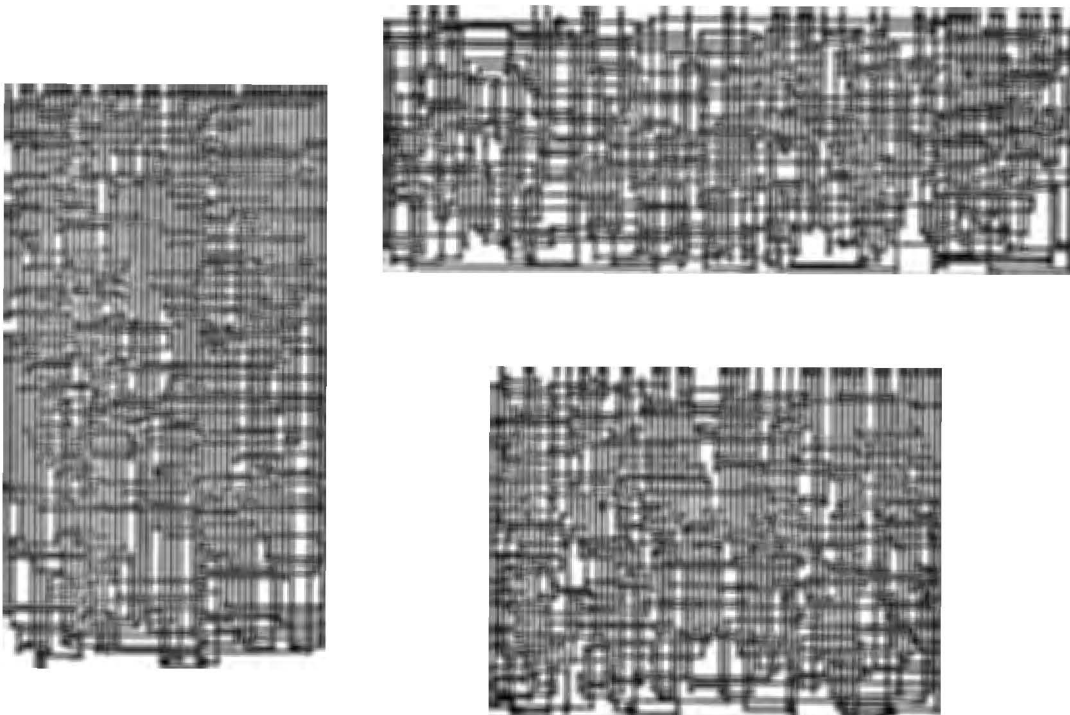


Figure 6.5. Three different aspect ratio's for the module five.

module	*nr of xtors*	*2 dim folding*	*gate matrix*	*standard cell*
hell84	12	0.043	0.072	-
data	24	0.088	0.166	-
adc	42	0.175	0.411	0.472
mp5	46	0.176	0.436	0.465
four	68	0.38	0.52	0.58
cnt4	96	0.57	1.35	-
loc	130	0.97	-	2.13
five	177	1.33	2.99	2.29
six	332	3.85	-	6.99

TABLE 6.4. Comparing the ctm-layout with gate matrix and standard cell.

6.5 Computation time

Another important aspect to automatic module generation is the time consumption of the algorithms. In table 6.5 the computation time is given. The six steps to generate a module are given separate as well as the total of all six steps. M4 (1) is the step to get the surrounding-box models from the library, where M4 (2) is the step to get the final models from the library.

module	*ctm_ivs*	*folding*	*ctm_stretch*	*M4 (1)*	*ctm_place*	*M4 (2)*	*total*
hell84	0.3	6.8	0.8	19.1	59.5	3.8	90.3
data	0.6	25.4	1.1	32.3	101	4.9	165
adc	1.7	68.5	1.8	64.7	226	7.0	370
mp5	1.7	124	2.0	71.4	260	7.4	466
four	2.6	418	2.6	95.0	371	9.6	898
cnt4	4.8	1110	4.1	158	588	13.6	1870
loc	6.4	2720	4.7	200	792	16.5	3740
five	11.2	4910	6.6	305	1270	22.3	6530
six	31.8	39800	12.2	537	2150	39.1	42600

TABLE 6.5. Computation time in seconds for various examples.

Table 6.5 clearly shows, that ctm_place takes the most time for small modules, while the total time for large modules mostly depends on the folding. The other steps can usually be neglected.

The table also shows the linearity of the ctm_place algorithm. The placement takes between 4.21 seconds (for 'data') and 7.18 seconds (for 'five') per transistor. The larger modules are split in more blocks, so the number of layout-elements is larger than just the number of

transistors. The folding algorithm shows a third-order relation between time and number of transistors ($time = 0.0012 \cdot \#xtors^3$).
The re-ordering of columns is part of the ctm_stretch program. The table clearly shows, that this re-ordering of columns runs very fast. This is probably due to the small folding-groups.

Of course these results are not absolute. The program was run on a HP9000 in a multitasking environment. The table only demonstrates the relation between time consumption and the number of transistors. The table also demonstrates the relation between the time consumption of the different steps.

7. CONCLUSIONS AND RECOMMENDATIONS

7.1 Conclusions

The doubly folded transistor matrix module generator offers great design flexibility. The module can be customised with respect to function, speed, design rules, aspect ratio and pin positions. Both aspect ratio and pin positions can be accurately controlled, while the area remains more or less constant. A wide range of aspect ratio's could only be obtained if the module was partitioned in blocks. An automatic control over the number of partitions proved to be very effective if both desired aspect ratio and size of the transistor matrix were taken into account.

Compared to conventional methods of automatic module generation, the presented generator gives a drastic improvement in area usage. The folding offers a uniform distribution of transistors and wires over the complete rectangle, leaving no empty spaces or corners. This keeps the enclosing rectangle small. Because of the library of adaptable transistor models an efficient compaction is obtained, in spite of the greedy approach.

The module generator is automatically interfaced with a floorplanner. The module generator adapts the module from a global floorplan and uses an elegant hierarchical divide and conquer algorithm to refine the two dimensional folding. Finally the transistor layouts are adapted to the wire plan designed by the folding.

7.2 Recommendations

For larger modules the time consumption of the module generator increases drastically. This is mainly due to the third order 'lin' algorithm used in the folding. A quadratic 'fast' algorithm is also supplied by the folding program, but the results of the 'lin' algorithm are up to a factor two better. An algorithm that can get the same results as the 'lin' algorithm, but is also quadratic is highly desired. At the moment steps are taken to combine the linear mincut algorithm as described in chapter 5 with the 'fast' algorithm. Experiments so far showed a quadratic computation time, with results better than 'fast' but not as good as 'lin'.

The ctm_place program uses arrays to represent the contours of the transistor models. For transistors that span a great number of columns, this means that a great number of positions have to be updated in the select and place functions. The use of a special database structure, that only contains the points were the contour steps up or down, can speed up the program.

The contour-array can not distinguish vias connected to wires on different columns. For this reason, we use vias that are centered on the wires and make sure the distance between the wires is large enough to satisfy the design rules. In our case, the size of a via is 12 μm and the minimal distance is 6 μm, which demands a pitch of 18 μm. Because the wiresize is 6 μm, uncentred vias can offer a pitch of only 16 μm. This results in a 20 % reduction of the width of the module. The database for the contours should be able to make the distinction

between the vias.

The M4-preprocessor adapts transistor models from the library, using string substitutions. This is not a very efficient procedure as can be seen in table 6.5. A more dedicated program probably can speed up this step drastically.

The design of the model library is only partly supported by a tool. The models can be designed using an interactive layout editor and be transformed to the library syntax automatically. However, this transformation program does not check for errors. The selection criterion, used by M4 to get a model from the library have to be generated manually. Both design steps can cause errors in the library. It is up to the designer to verify the correctness of the models in the library and selection criterions. A more sophisticated tool could check the models for errors and automatically generate the selection criterion.

8. REFERENCES

[FIDU82]: **C.M. Fiduccia and R.M. Mattheyses**: "A Linear Time Heuristic for Improving Network Partitions", *Proc. 19th Design Automation Conf.*, Las Vegas, 1982, pp.175-181.

[GINN84]: **L.P.P.P.van Ginneken and R.H.J.M.Otten**: "Stepwise Layout Refinement", *Proc. Int. Conf. on Computer Design*, Port Chester NY, October 8-11 1984, pp.30-36.

[GINN88]: **L.P.P.P. van Ginneken, J.T.J. van Eindhoven, P.R.M. van Teeffelen and Th. J. Deckers**: "Soft Macro Cell Generation by Two Dimensional Folding", *Proc. Int. Conf. on Circuits and Systems*, Helsinki, june 1988, pp.727-730.

[KERN70]: **B.W.Kernighan and S.Lin**: "An Efficient Heuristic Procedure for Partitioning Graphs", *The Bell System Technical Journal*, February 1970, pp.291-307.

[LAUT79]: **U. Lauther** "A Min-cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation", *Proc. 16th Design Automation Conference*, San Diego, 1979, pp.1-10.

[LIES87]: **G.J.P. van Lieshout and L.P.P.P. van Ginneken**: "GM: A Gate Matrix Layout Generator", *Master Thesis*, Eindhoven University of Technology, Department of Electrical Engineering, Design Automation Section, august 1987.

[OTTE83]: **R.H.J.M.Otten**: "Efficient Floorplan Optimization", *Proc. Int. Conf. Computer Design*, Port Chester NY, October 1983.

[TEEF88]: **P.R.M. van Teeffelen**: "Two Dimensional Folding for Soft Macro Cell Generation", *Master thesis*, Eindhoven University of Technology, Department of Electrical Engineering, Design Automation Section, April 1988.

[THEE85]: **J.F.M. Theeuwen and P.T.H.M. van Paassen**: "Automatic Generation of Boolean Expressions in nMOS Technology", *Proc. Int. Conf. on Computer-Aided Design*, Santa Clara, November 18-21 1985, pp.332-334.

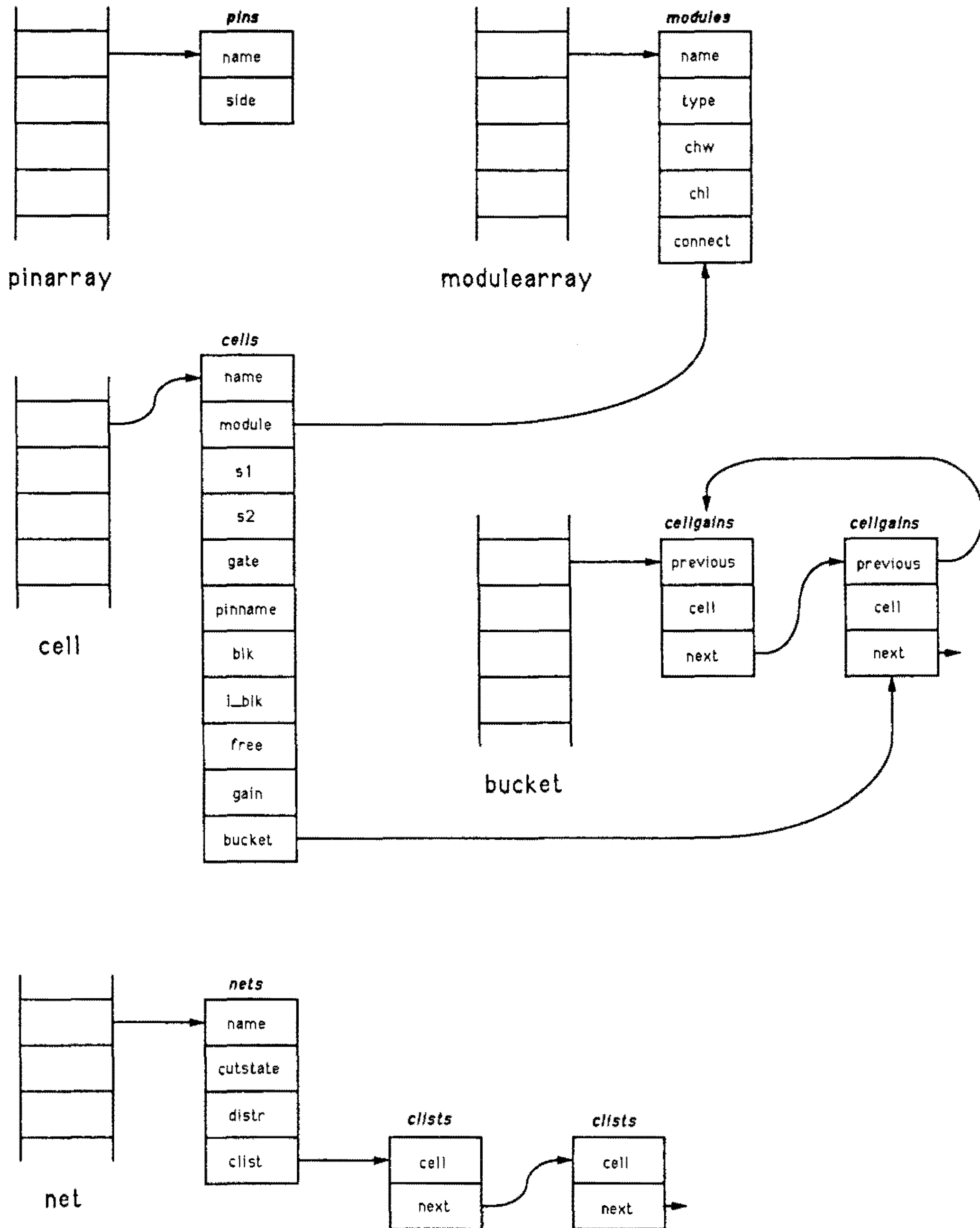
[WIRT71]: **N. Wirth**: "Program Development by Step-wise Refinement", *Commun. ACM*, vol. 14 (1971), pp.221-227.

APPENDICES

In the appendices are given:

- ctm_ivs global data structure
- ctm_stretch global data structure
- ctm_place global data structure
- ctm_ivs manual
- ctm_stretch manual
- ctm_place manual
- ctm_mklib manual

ctm_ivs global data structure



database structure of the program ctm_ivs.

PINS (describe I/O-pins.)

name : name of the I/O-pin
side : side of the I/O-pin

MODULES (describe the transistor parameters)

name : name of the transistor
type : transistor type
chw : width of transistor channel
chl : length of transistor channel
connect : number of connections made to the transistor, to check the input file (not implemented yet)

CELLS (describe the transistor position and terminals)

name : name of the transistor
module : pointer to the definition of transistor parameters
s1 : index of 'net'-array for s1-net
s2 : index of 'net'-array for s2-net
gate : index of 'net'-array for gate-net
pinname : name for I/O-pin
blk : global block after partitioning
l_blk : local block during partitioning
free : free to move or locked during partitioning
gain : cellgain if cell is moved during partitioning
bucket : pointer to item in bucket

CELLGAINS (item of doubly linked list of cells with the same cellgain)

previous : pointer to previous item in linked list
cell : index of 'cell'-array
next : pointer to next item in linked list

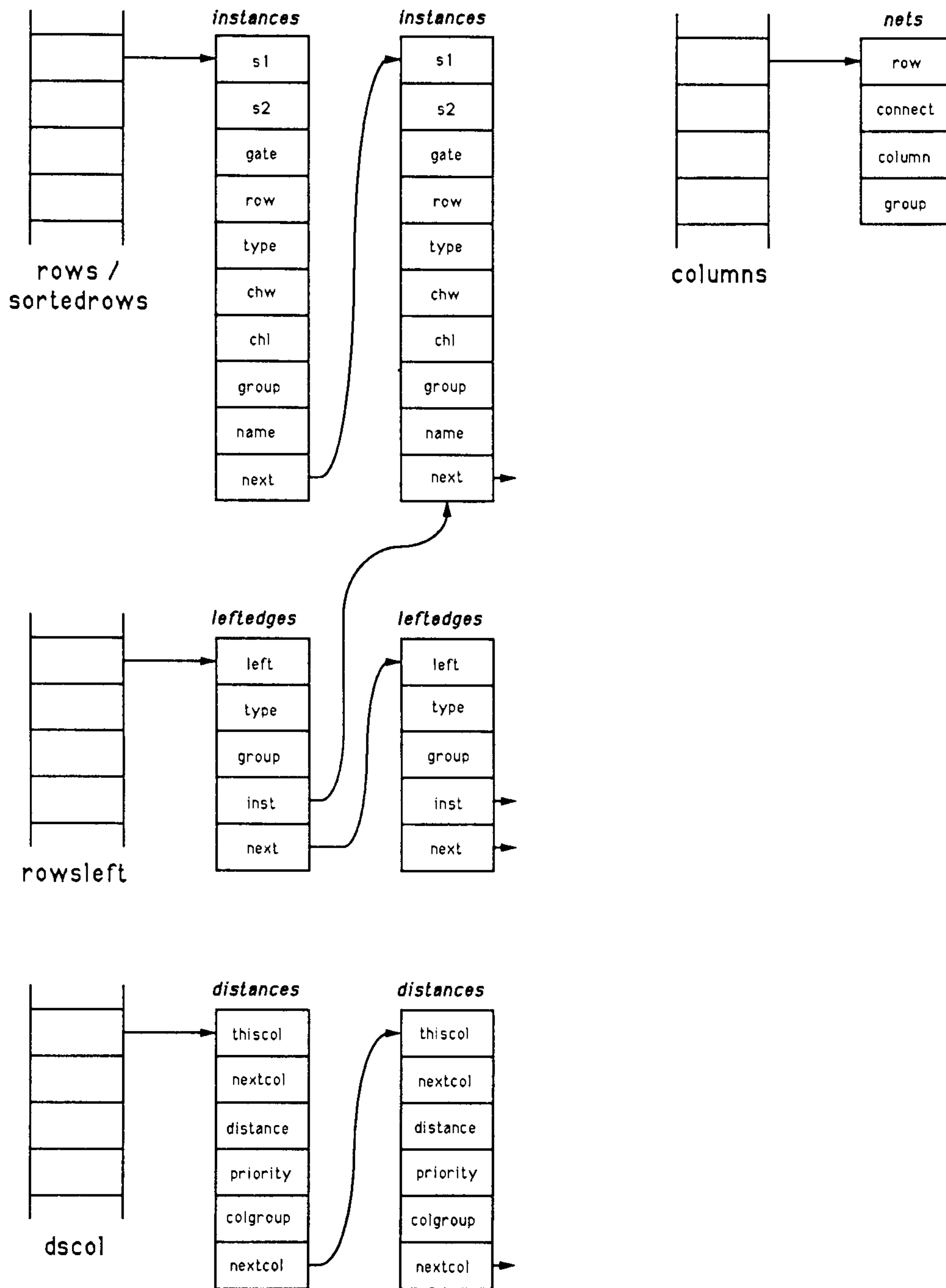
NETS (describe net)

name : name of the net
cutstate : state of net after partitioning (cut / uncut)
distr : distribution of locked and free cells connected to the net over both blocks during partitioning
clist : pointer to a linked list of cells connected to the net

CLISTS (item of a linked list of cells connected to the same net)

cell : index of 'cell'-array
next : pointer to the next item in the linked list

ctm_stretch global data structure



global data structure of the program ctm_stretch.

INSTANCES (describe parameters and position of a cell)

s1 : index to 'columns'-array for s1-column
s2 : index to 'columns'-array for s2-column
gate : index to 'columns'-array for gate-column
row : final row position after re-ordering
chw : width of the transistor channel
chl : length of the transistor channel
group : folding group number
name : name of the cell
next : pointer to the next item of the linked list of cells on the same row

NETS (describe the position and connections to a net)

row : relative y-coordinate of the net
connect : number of connected cells
column : column of the net after re-ordering
group : folding group number

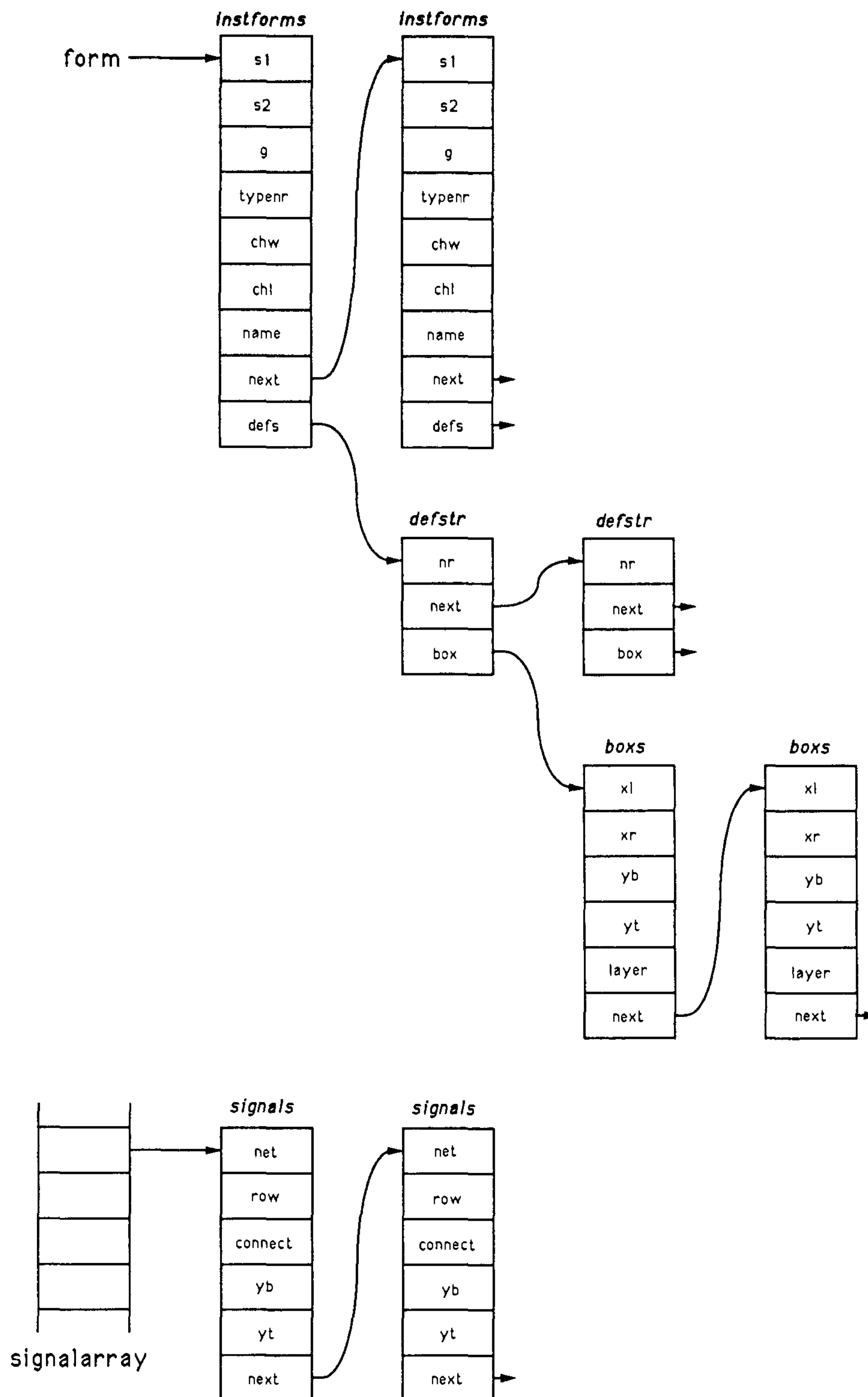
LEFTEDGES (describe cells during left-edge re-ordering of cells)

left : column of leftmost via
type : type of leftmost via
group : folding group number
inst : pointer to cell description
next : pointer to the next item in the linked list of cells

DISTANCES (describe desired distances during re-ordering of columns)

thiscol : position of this column
othercol : position of other column
distance : desired distance between thiscol and othercol
priority : distance priority
colgroup : folding group number of thiscol
nextcol : pointer to the next item in the linked list

ctm_place global data structure



global data structure of the program ctm_place.

INSTFORMS (describe form of instance to be placed)

s1 : column of s1
s2 : column of s2
gate : column of gate
typenr : instance type number
chw : transistor channel width
chl : transistor channel length
name : name of the instance
next : pointer to the next instance structure (not used yet)
defs : pointer to an item in the linked list of model definitions

DEFSTR (item of linked list of model definitions for the same instance)

nr : number of the definition
next : pointer to the next item in the linked list of model definitions
box : pointer to an item in the linked list of boxes that make up a model definition

BOXS (item of linked list of boxes that make up a model definition)

xl : left coordinate of the box
xr : right coordinate of the box
yb : bottom coordinate of the box
yt : top coordinate of the box
layer : layer of the box (xx, px or dx)
next : pointer to the next item in the linked list of boxes

SIGNALS (item of linked list of signals on the same column)

net : netname
row : relative y-coordinate of the net on the column
connect : the number of connections that still have to be made to the net
yb : the y-coordinate of the center of the lowest connected via to the net
yt : the y-coordinate of the center of the highest connected via to the net
next : pointer to the next item in the linked list of signals on the same column

NAME

ctm_ivs – make transistor matrix intervals for folding

SYNOPSIS

ctm_ivs [-b] <interface-file> <module-file> <netlist> <interval-file>

DESCRIPTION

A circuit can be described in a combination of three files, the interface-file, the module-file and the netlist. The netlist describes the connections between the modules used in the circuit. The module-file describes the used transistors and the interface-file describes the I/O-pins, their positions and the shape of the layout. This description is transformed by ctm_ivs to a transistor matrix, where each net is represented by a column and each transistor represented by a row. The connections between the transistors and nets are represented by the interval-file.

OPTIONS

-b The transistor matrix is partitioned in a number of blocks. The number of blocks depends on the size of the transistor matrix and the desired aspect ratio. The partitioning is preformed by a mincut algorithm which minimises the number of nets that are used in two blocks. These nets will be split in two nets, connected by a strip. This option allows the folding algorithm to have more control over the aspect ratio, so chances of reaching the desired aspect ratio increase.

INTERFACES

The program uses three input-files and one output-file

netlist

A netlist of transistors allows total freedom for the design of transistor networks. There are no constraints to the number of connections to be made to one signal and the gate, drain and source of a transistor can be connected to any signal. The netlist contains all connections to be made. Each line in the netlist describes one connection between a signal and a transistor. It also states to which terminal of the transistor (gate, drain or source) the connection is made. The syntax for all lines of the netlist is: <netname> <transistor-name> <terminal>

module-file

The module-file describes the type of transistor and the size of its channel. The module_file has the following syntax: <transistorname> <type> <channel length> <channel width>

interface-file

The interface-file describes the aspect ratio and pin positions as desired by the floorplanner.
The syntax of the interface-file is as follows: "module" <module name>
"shape" <width> <height>
"pin" <pin name> <coord> <coord>
...
"end"
The first line must always contain the keyword "module", followed by the name of the module. The second line always starts with "shape" followed by the desired width and height of the module. The file has to be terminated with the keyword "end" on the last line.
All other lines start with the keyword "pin" followed by the pinname and the interval of allowed positions of the pin. Figure 2.3 shows the mapping of the intervals. The coordinates are floating point numbers, so only a part of a side can be chosen. This offers the possibility to define relative pin positions on the same side.

interval-file

The netlist can be mapped to a two-dimensional transistor matrix. In this matrix all transistors are mapped to the rows and all nets to the columns. The I/O-pins connected to the north-side of the module are all combined on one row called '$NORTH$'. The 'south'-I/O-pins are all combined in the row called '$WEST$' and '$SOUTH$'. These rows and columns are always fixed to the four sides of the transistor-matrix. In this matrix all connections can easily be represented.

To describe the circuit one could use the whole matrix. The matrix is however very sparse and therefore it is more useful to describe the circuit only by the connections in the matrix. This leads to the interval-file. The interval-file represents the connections in the matrix by only specifying the coordinates of the connections. The syntax of the interval-file is as follows: <module name>
<width> <height>
<column> <row> <blocknr> <module typenr> <channel length> <channel width> [<pin name>]
...

CONTRIBUTED BY

Jos Brouwers

STATUS

In development

SEE ALSO

folding
ctm_stretch
ctm_place

NAME

ctm_stretch – define stretch for transistor models used in compact transistor matrix module generator

SYNOPSIS

ctm_stretch [-r] <interval-file> <foldresult> <transistor-file> <signal-file>

DESCRIPTION

The interval-file and the foldresult are combined to get the doubly folded transistor matrix. This results in the relative positions of the transistors and the signal-wires. The columns to which the transistors are connected define the stretch of the models that can be used. This will be written to the transistor-file. The relative positions of the signals as well as the number of connections to the signals are written to the signal-file.

OPTIONS

-r The columns of the transistor matrix are re-ordered to allow transistors to lie flat as much as possible. This usually results in a compacter layout. Also the transistors are re-ordered in the rows, allowing vias of the same kind to overlap. This can usually save some rows of the transistor matrix, resulting also in a compacter layout.

INTERFACES

The program uses two input-files and two output-files

interval-file

see ctm_ivs.

foldresult

The folding-program generates a foldresult with the following syntax: <module name>
<width> <height>
<vert-int> <x-coordinate> [<groupnumber>]
...
<blank line>
<hor-int> <y-coordinate> [<groupnumber>]
... If the -r option is used, the groupnumber must be specified (by running the folding with the -g option).

transistor-file

The transistor-file defines the stretch and terminal-positions of all used transistors. Also the channel length and -width are specified. The syntax of one line is: "TOR(" <s1> "," <s2> "," <gate> "," <chl> "," <chw> "," <type> "," <name> ")" No distinction is made between drain and source. They can be connected to column <s1> or column <s2>, where <s1> <= <s2>. The gate-connection is made at column <gate>. The channel length and -width are specified by <chl> and <chw>. <type> gives the typenumber of the transistor. If the transistor is not named, "~" will be substituted for <name>.

signal-file

The signal-file states the relative position of a signal-wire and the number of connections made to it. The syntax is: <signal> <column> <y-coordinate> <connect> <signal> is the original column of the signal in the interval-file, <column> is the column the signal has been mapped to (after folding and re-ordering). The <y-coordinate> is a relative coordinate, only to be used together with other signals on the same column. <connect> gives the number of connections made to the signal.

CONTRIBUTED BY

Jos Brouwers

STATUS

In development

SEE ALSO

ctm_ivs
folding
ctm_place

NAME

ctm_place – select and place transistors and wires used in compact transistor matrix module generator.

SYNOPSIS

ctm_place <definitions-file> <signal_file> <placement-file>

DESCRIPTION

For each transistor, the definition-file gives a list of definition for the models that can be used. The model that fits best will be selected and placed in the placement-file. The selecting criterion is the extra area multiplied by the increase in height of the module. The length and positions of the signal-wires are calculated by the positions of the transistors connected to it and they are placed in the placement-file as well.

INTERFACES

The program uses two input-files and one output-file.

definitions-file

This file is a list of transistors with the following syntax:

<transistorlist> ::= <transistor><transistorlist> | <transistor>

<transistor> ::= <instance><definitionlist>

<instance> ::= "instance" <int><int><int><int><int><int>[<name>]<eol>

> Instance is the header of a list of definitions. The integers stand for s1-column, s2-column, gate-column, channellength, channelwidth and type of the transistor.

<definitionlist> ::= <definition><definitionlist> | <definition>

<definition> ::= <newdef><elementlist>

<newdef> ::= "newdef" <int>

> Newdef is the header of a list of elements, that define one transistor model. The integer gives the modelnumber.

<elementlist> ::= <element><elementlist> | <element>

<element> ::= <box> | <module-call> | <terminal>

<box> ::= "box" <layer><int><int><int><int><eol>

<module-call> ::= "mc" <name><int><int><eol>

<terminal> ::= "term" <layer><int><int><int><int><name><eol>

<int> ::= {<digit>}+

<name> ::= <letter> {<letter> | <digit>}*

<layer> ::= <name>

> All lines following the newdef-key are the LDM-description of a model until another newdef or instance-key is encountered. This way each transistor can have an arbitrary number of models to choose from.

signal-file

see ctm_stretch

placement-file

This file is a list of transistor- and wire definitions with the following syntax:

<definition> ::= <transistor def> | <wire def>

<transistor def> ::= "INST(" <int> "," <int> "," <int> "," <int> "," <int> "," <int> "," [<name>] ")"
The integers stand for s1-column, s2-column, gate-column, channellength, channelwidth and type of the transistor, name gives the - optional - transistorname.

<wire def> ::= "box nm" <int> <int> <int> <int>
The integers give the coordinates of the left bottom and right top corners of the box.

<int> ::= {<digit>}+

<name> ::= <letter> {<letter> | <digit>}*

BUGS

The I/O-pins on the south-side must have typenumber 9003 (for poly-terminals) or 10003 (for metal-terminals). The I/O-pins on the north-side must have typenumber 9004 (for poly-terminals) of 10004 (for metal-terminals).

CONTRIBUTED BY

Jos Brouwers

STATUS

In development

SEE ALSO

ctm_ivs
ctm_stretch
folding

NAME

ctm_mklib – ctm_mklib makes a library for the ctm-program from an ldm-description of the models.

SYNOPSIS

ctm_mklib <ldm-file> <lib-file> [<tech-file>]

DESCRIPTION

The last module in the ldm-file will be transformed to a model in the lib-file. Also the module is mirrored in the X-axis, in the Y-axis and both to give three models more in the lib-file. These models are placed in 'lib-file'.n1 to 'lib-file'.n4. Automatically the surrounding boxes of all models are also computed and placed in 'lib-file'.x1 to 'lib-file'.x4.
The program will ask the name and corresponding number for the module. The names will be converted to 'name', 'name'_x, 'name'_y and 'name'_x_y while the number will be incremented by respectively 0, 1, 2 and 3.
The tech-file can be used to set the technology-dependent design rules.

INTERFACES

ldm-file

The input is a standard ldm-file (for syntax see ldm) from a layout made with the layout editor 'euler'. To design a good model for ctm_mklib, some important rules must be satisfied:

- The transistor may not be constructed by simply overlapping diffusion by poly, but a compound-model must be used. These compound-models may not be rotated or mirrored (if needed, a special compound has to be constructed).
- If, due to a larger channel, the transistor-compound becomes higher, only the layout-elements that lie above the centre of the transistor-compound will be shifted accordingly.
- The channel of the transistor can only grow in the upward or right direction, the model must allow this growth.
- The library is constructed around the positions of the vias. These vias must be layed apart at least three columns. The left and right sides of the layout-elements will be computed from the position of the nearest via. They will have a fixed offset to the position of that via. Usually three columns between the vias gives enough distance to assure the selection of the proper via. If in doubt, the distance may be enlarged, this does not affect the actual shape or stretch-points of the model.
- While constructing a module no rotating or mirroring of any layout-element, or the whole module are allowed.
- All wires must be drawn at the predefined pitch (see tech-file).

lib-files

There are two types of lib-files, the nlib-files and the xlib-files. The M4-preprocessor is able to convert these files to definition-files (see ctm_place) respectively ldm-files (see ldm). The syntax for the xlib-files is:

<xmodel> ::= <xhead> {<xdef>}* <xtail>

<xhead> ::= "define(" <name> "," <eol> "‘" <eol> "newdef" <number> <eol>

<xdef> ::= <xbox> | <xcompound>

<xtail> ::= "’)"

<xbox> ::= "box" <xlayer> <xcoord> <ycoord> <xcoord> <ycoord> <eol>

<xcompound> ::= "mc" <name> <xcoord> <ycoord> <eol>

<xlayer> ::= "px" | "dx" | "xx"

<xcoord> ::= "eval(" <int> "+" <via> "+" <int> | { ["-"] "$4" | "$5" } ")"

<ycoord> ::= :eval(" <int> "+$6+" <int> | { ["-"] "$4" | "$5" } ")"

The syntax for the nlib is:

<nmodel> ::= <nhead> {<ndef>}* <ntail>

<nhead> ::= "define(INST"<number> "," <eol> "‘" <eol>

<ndef> ::= <nbox> <ncompound>

<nbox> ::= "box" <nlayer> <xcoord> <ycoord> <xcoord> <ycoord> <eol>

<ncompound> ::= "mc" <name> <xcoord> <ycoord> <eol>

<nlayer> ::= "nm" | "np" | "nd" | "ni" | "nb" | "nx" | "na"

<xcoord> ::= "eval(" <int> "+" <via> "+" <int> | { ["-"] "$4" | "$5" } ")"

<ycoord> ::= :eval(" <int> "+$6+" <int> | { ["-"] "$4" | "$5" } ")"

tech-file

The tech-file can be used to set the technology parameters. 7 keywords are recognised: "pitch" <int> : pitch of the wires (default 18). "wiresize" <int> : width of a wire (default 6). "viasize" <int> : width and height of the vias (default 12). "overlap" <int> : extra poly-diffusion overlap to assure a good transistor (default 4). "safety" <int> : minimal surrounding box, half the size of largest design rule (default 3). "pmvia" <name> : name of the poly–metal via, used in the layout (default pm004004). "dmvia" <name> : name of the diffusion–metal via, used in the layout (default dm004004). If a keyword is not found in the tech-file, the default-value will be used. If no tech-file is specified, all default-values will be used.

USAGE

The program is developed to construct a library to the ctm-module generator. The nlib- and xlib-files can be joined and appended to standard library-files. These standard library-files must contain a definition of all used transistor-compounds and vias. These definitions can not be generated automatically, so they must be made by hand. The syntax of a definition is:

<definition> ::= <header> {<box>* | <copy>} <tail>

<header> ::= "define("<name> "," <eol> "‘" <eol>

<box> ::= "box" <layer> <xcoord> <xcoord> <ycoord> <ycoord> <eol>

<copy> ::= <name>"($1,$2,$3,$4)"

name is the name of the compound to be copied.

<xcoord> ::= "eval($1" [{"+" | "-" "$3" | "$4"}] "+" | "-" <int> ")"

<ycoord> ::= "eval($2" [{"+" | "-" "$3" | "$4"}] "+" | "-" <int> ")"

The definitions of strips and I/O-pins must also be made by hand in the same way. The library should be used in combination with a selection-file. This file must be written conform the m4-syntax. Depending on the ordering of gate, drain and source, and on the size of the channel, the m4-preprocessor can with this file make a selection between all models in the library. To extract the surrounding boxes from the library, M4 is used twice. First it is run on the concatenation of pitch-file, select-file and .tor-file. The pitch-file only contains one line, stating the wire-pitch. The syntax is:

"define(P," <pitch> ")"

This first step selects the models and stretch. In the second step, M4 is run on the concatenation of pitch-file, x-library and the output of the first step. To extract the final layout from the library, M4

is run on the concatenation of pitch-file, n-library and .c2out-file. To run the program, a shell-script called 'mklib' is provided. A menu can be used to select steps to be taken. It also provides the automatic updating of the library-files and ctm_mklib can be run in background for a sequence of compounds. Running in background needs a special file, called 'tech'.add. The syntax of this file is:

{<name> <eol> <number> <eol>}*

The models are added to the library, not substituted, so name and number may not have been used before.

BUGS

The program can only be used for transistor definitions with one or two diffusion contacts and zero or one poly contact. The one diffusion and zero poly contact combination can't be used either.

CONTRIBUTED BY

Jos Brouwers

STATUS

In development

SEE ALSO

ldm
m4
euler
ctm_stretch
ctm_place

NAME

ctm_mklib – ctm_mklib makes a library for the ctm-program from an ldm-description of the models.

SYNOPSIS

ctm_mklib <ldm-file> <lib-file> [<tech-file>]

DESCRIPTION

The last module in the ldm-file will be transformed to a model in the lib-file. Also the module is mirrored in the X-axis, in the Y-axis and both to give three models more in the lib-file. These models are placed in 'lib-file'.n1 to 'lib-file'.n4. Automatically the surrounding boxes of all models are also computed and placed in 'lib-file'.x1 to 'lib-file'.x4.
The program will ask the name and corresponding number for the module. The names will be converted to 'name', 'name'_x, 'name'_y and 'name'_x_y while the number will be incremented by respectively 0, 1, 2 and 3.
The tech-file can be used to set the technology-dependent design rules.

INTERFACES

ldm-file

The input is a standard ldm-file (for syntax see ldm) from a layout made with the layout editor 'euler'. To design a good model for ctm_mklib, some important rules must be satisfied:

- The transistor may not be constructed by simply overlapping diffusion by poly, but a compound-model must be used. These compound-models may not be rotated or mirrored (if needed, a special compound has to be constructed).
- If, due to a larger channel, the transistor-compound becomes higher, only the layout-elements that lie above the centre of the transistor-compound will be shifted accordingly.
- The channel of the transistor can only grow in the upward or right direction, the model must allow this growth.
- The library is constructed around the positions of the vias. These vias must be layed apart at least three columns. The left and right sides of the layout-elements will be computed from the position of the nearest via. They will have a fixed offset to the position of that via. Usually three columns between the vias gives enough distance to assure the selection of the proper via. If in doubt, the distance may be enlarged, this does not affect the actual shape or stretch-points of the model.
- While constructing a module no rotating or mirroring of any layout-element, or the whole module are allowed.
- All wires must be drawn at the predefined pitch (see tech-file).

lib-files

There are two types of lib-files, the nlib-files and the xlib-files. The M4-preprocessor is able to convert these files to definition-files (see ctm_place) respectively ldm-files (see ldm). The syntax for the xlib-files is:

<xmodel> ::= <xhead> {<xdef>}* <xtail>

<xhead> ::= "define(" <name> "," <eol> "`" <eol> "newdef" <number> <eol>

<xdef> ::= <xbox> | <xcompound>

<xtail> ::= "')"

<xbox> ::= "box" <xlayer> <xcoord> <ycoord> <xcoord> <ycoord> <eol>

<xcompound> ::= "mc" <name> <xcoord> <ycoord> <eol>

<xlayer> ::= "px" | "dx" | "xx"

<xcoord> ::= "eval(" <int> "+" <via> "+" <int> | { ["-"] "$4" | "$5" } ")"

<ycoord> ::= :eval(" <int> "+$6+" <int> | { ["-"] "$4" | "$5" } ")"

The syntax for the nlib is:

<nmodel> ::= <nhead> {<ndef>}* <ntail>

<nhead> ::= "define(INST"<number> "," <eol> "`" <eol>

<ndef> ::= <nbox> <ncompound>

<nbox> ::= "box" <nlayer> <xcoord> <ycoord> <xcoord> <ycoord> <eol>

<ncompound> ::= "mc" <name> <xcoord> <ycoord> <eol>

<nlayer> ::= "nm" | "np" | "nd" | "ni" | "nb" | "nx" | "na"

<xcoord> ::= "eval(" <int> "+" <via> "+" <int> | { ["-"] "$4" | "$5" } ")"

<ycoord> ::= :eval(" <int> "+$6+" <int> | { ["-"] "$4" | "$5" } ")"

tech-file

The tech-file can be used to set the technology parameters. 7 keywords are recognised:

"pitch" <int> : pitch of the wires (default 18).

"wiresize" <int> : width of a wire (default 6).

"viasize" <int> : width and height of the vias (default 12).

"overlap" <int> : extra poly-diffusion overlap to assure a
good transistor (default 4).

"safety" <int> : minimal surrounding box, half the size of
largest design rule (default 3).

"pmvia" <name> : name of the poly–metal via, used in the layout (default pm004004).

"dmvia" <name> : name of the diffusion–metal via, used in the layout (default dm004004).

If a keyword is not found in the tech-file, the default-value will be used. If no tech-file is specified, all default-values will be used.

USAGE

The program is developed to construct a library to the ctm-module generator. The nlib- and xlib-files can be joined and appended to standard library-files. These standard library-files must contain a definition of all used transistor-compounds and vias. These definitions can not be generated automatically, so they must be made by hand. The syntax of a definition is:

<definition> ::= <header> {<box>* | <copy>} <tail>

<header> ::= "define("<name> "," <eol> "`" <eol>

<box> ::= "box" <layer> <xcoord> <xcoord> <ycoord> <ycoord> <eol>

<copy> ::= <name>"($1,$2,$3,$4)"
name is the name of the compound to be copied.

<xcoord> ::= "eval($1" [{"+" | "-" "$3" | "$4"}] "+" | "-" <int> ")"

<ycoord> ::= "eval($2" [{"+" | "-" "$3" | "$4"}] "+" | "-" <int> ")"

The definitions of strips and I/O-pins must also be made by hand in the same way.
The library should be used in combination with a selection-file. This file must be written conform the m4-syntax. Depending on the ordering of gate, drain and source, and on the size of the channel, the m4-preprocessor can with this file make a selection between all models in the library.
To extract the surrounding boxes from the library, M4 is used twice. First it is run on the concatenation of pitch-file, select-file and .tor-file. The pitch-file only contains one line, stating the wire-pitch. The syntax is:

"define(P," <pitch> ")"

This first step selects the models and stretch. In the second step, M4 is run on the concatenation of pitch-file, x-library and the output of the first step.
To extract the final layout from the library, M4 is run on the concatenation of pitch-file, n-library and .c2out-file.
To run the program, a shell-script called 'mklib' is provided. A menu can be used to select steps to be taken. It also provides the automatic updating of the library-files and ctm_mklib can be run in background for a sequence of compounds. Running in background needs a special file, called 'tech'.add. The syntax of this file is:

{<name> <eol> <number> <eol>}*

The models are added to the library, not substituted, so name and number may not have been used before.

BUGS

The program can only be used for transistor definitions with one or two diffusion contacts and zero or one poly contact. The one diffusion and zero poly contact combination can't be used either.

CONTRIBUTED BY

Jos Brouwers

STATUS

In development

SEE ALSO

ldm
m4
euler
ctm_stretch
ctm_place