GADL : a gate array description language

Research Report

Eindhoven
University of Technology
Netherlands

Department of Electrical Engineering

GADL: A Gate Array Description Language

by
P.E.R. Lippens
and
A.G.J. Slenter

EUT Report 87-E-165
ISBN 90-6144-165-X
ISSN 0167-9708

January 1987

Eindhoven University of Technology Research Reports
EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Eindhoven The Netherlands

GADL
A GATE ARRAY DESCRIPTION LANGUAGE

by

P.E.R. Lippens
and
A.G.J. Slenter

EUT Report 87-E-165
ISBN 90-6144-165-X
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
January 1987

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

(Multiview VLSI-design System ICD)

code: 991

DELIVERABLE

Report on activities:	5.2.C :	Cell generation schemes. Developing or studying cell architectures for various CMOS families.
	5.3.A :	Generating layouts for random logic.
	5.3.B :	Implementation of one cell generation scheme of some CMOS family.

CADL : A gate array description language.

Abstract:
This report describes a new way to map digital networks onto gate array images.
A language is defined to describe gate array images, design rules for the wiring and arbitrary specified macros from image dependent libraries. A data base is generated by compiling the description of a gate array family in terms of the above items.
Place and route software is automatically conditioned to implement arbitrary specifications of a digital system in terms of a netlist.
The performance of the system is demonstrated for various practical situations.

deliverable code: WP 5, *task*: 5.2, *activity*: C;
task: 5.3, *activities*: A and B.

date: 23 - 12 - 1986

partner: Eindhoven University of Technology

authors: P.E.R. Lippens, A.G.J Slenter

This report was accepted as a M.Sc. Thesis of P.E.R. Lippens by Prof. Dr.-Ing. J.A.G. Jess, Automatic System Design Group, Department of Electrical Engineering, Eindhoven University of Technology. The work was performed in the time from November 1985 to December 1986 and was supervised by ir. A.G.J. Slenter.

CIP- GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Lippens, P.E.R.

CADL - a gate array description language / by P.E.R. Lippens and A.G.J. Slenter. - Eindhoven: University of Technology. - Fig., tab. - (Eindhoven University of Technology research reports / Department of Electrical Engineering, ISSN 0167-9708; 87-E-165)
Met lit. opg., reg.
ISBN 90-6144-165-X
SISO 664.3 UDC 621.382:681.3.06 NUGI 832
Trefw.: elektronische schakelingen; computer aided design.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Gate arrays, otherwise known as logic arrays or cellular arrays, are a type of programmable or semi-custom semiconductor components made possible by the rapid advances in integrated circuit design techniques and manufacturing technology. Where an electronic system may be implemented using TTL or CMOS logic, a gate array can typically replace twenty to fifty SSI/MSI packages and effect considerable cost, weight, size and power savings in the end equipment.
The concept of having a fixed base pattern of logic gates on a silicon integrated circuit (IC) which is then programmed or "wired up" by customised metal patterns is not new. It was first proposed in the mid-1960s and used by several companies where the development cost of custom IC design were not justified for the relative low volumes of each variant desired. It is this trade-off of development cost, unit cost and prototyping time that is addressed by the use of gate arrays and other types of programmable components, notably microprocessors. Such components are called semi-custom integrated circuits as they are part standard (the base diffusion) and part custom (usually the interconnect and contacts). The term semi-custom is not usually applied to microprocessors, which are personalised or programmed by the ROM program mask, although conceptually they too are semi-custom. Another class of semi-custom products are cell based designs where blocks of predesigned elements are drawn from a library, and a total design requiring unique masks for all processing levels, instead of just a few, is generated.
This report only regards the design automation of the interconnect masks of gate arrays. Gate arrays are designed in many technologies (TTL, I^2L, CMOS, NMOS). CAD systems for gate arrays have to face new problems caused by the rapid growing of the gate array market. The GAS gate array design system developed at the Eindhoven University of Technology claims to be very flexible with regard to the gate array type. It is designed for small and medium sized companies which have to deal with a large and pluriformly organised set of vendors and foundries, predominantly located in the US. To establish the flexibility an entry in the design system is

created, where structure and properties of a gate array can be described in a special for this purpose developed language: GADL (gate array description language). In GADL it is possible to describe the gate array structure, its design rules, the set of macro stamps and the properties of a cost function (used to guide the Leerouter) in an easy way. The main part of the report consists of a description of GADL (chapter 5) and a description of the GADL compiler (chapter 6).

2. GATE ARRAY FEATURES

When we look at gate arrays, there are a few things they have in common. The most consist of a regular arrangement of basic cells (cellular array). Each cell can be personalised by metalisation to perform logic functions like NAND, OR, FLIPFLOP, etc.. These cells, usually referred to as function boxes, active areas etc., are separated by routing space, usually referred to as channels. Channels can be divided into basic cells called junction boxes, connection boxes or channel cells.
The area formed by regular repetition of function and junction boxes is called the core of a gate array. Around the core, a number of bonding pads and I/O-buffers is located. These I/O cells don't offer much space for routing and therefor are not interesting for us except the terminal positions. Figure 1 shows two typical gate array structures.

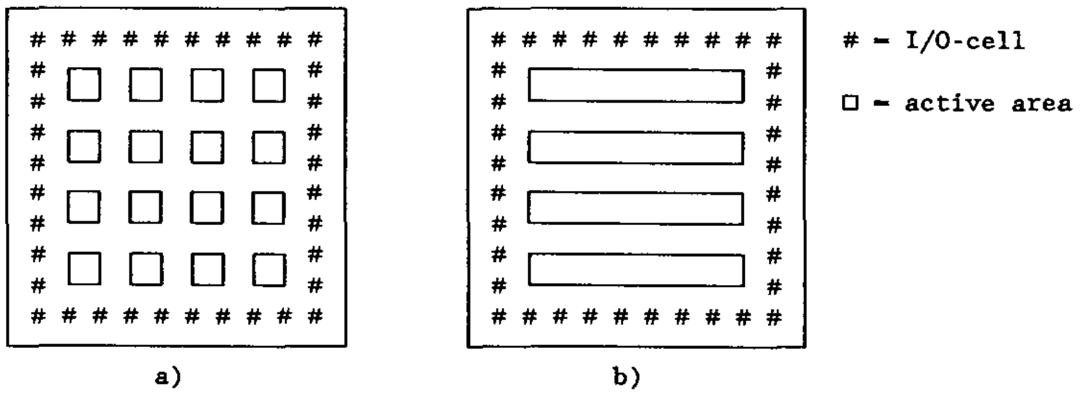


Figure 1. Two typical gate array structures: a) island structure, b) row structure

Let us now take a closer look at the junction and function boxes. A channel cell can vary in complexity from just some orthogonal tracks for intercellular connections to a more complex structure with prefabricated connection paths (cross unders or underpasses) with fixed or programmable contacts to the metal layer(s) above (figure 2). The gate cells of gate arrays are relatively more complex since many trade-offs have been made in these cell design. A gate cell consists of a number of discrete components with which one can build a set of useful functions in a

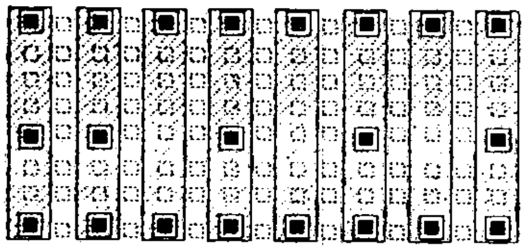


Figure 2. Channel cells.

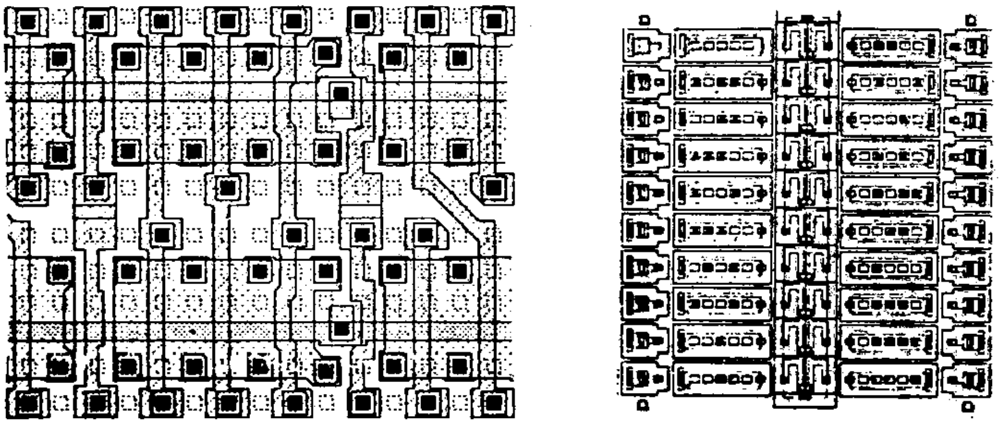


Figure 3. Gate cells.

particular technology. These components enable the function to be built up to give the best compromise in terms of performance. The basic design will support the use of a number of cells to form a more complex function. A cell includes the active and passive components necessary to build any of the required functions with the greatest flexibility in programmability.

Depending on design philosophy, process technology, I/O capacities and, most importantly, routability of the gate array, gate cells may (and do) have quite different structures and properties. However, a gate cell always has to be:

- Programmable (intraconnectable)
- Interconnectable
- Repeatable

A few different gate cells are given in figure 3.
Although the gate arrays have different structures and are designed in different technologies, the following attributes the most have in common, even for those gate arrays which have not gate cells and routing channels explicitly:

- The intercellular connections can only be made vertically and/or horizontally on certain pre-defined tracks.
- When non-orthogonal wires are permitted in gate arrays, they are either provided by gate array foundries or routed manually.

This observation leads to the conclusion that all, for routing interesting, features can be mapped onto a three dimensional grid. Figure 4 gives an example of the "gridding" of a part of a gate array.

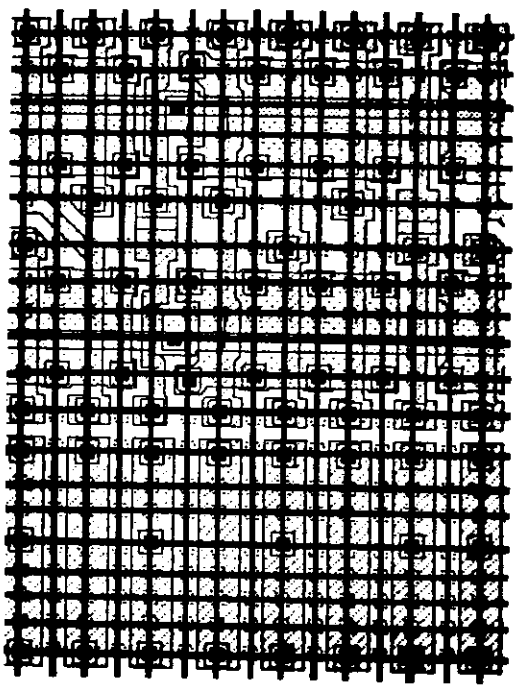


Figure 4. Gridding of a gate array part.

Each coordinate in the third dimension represents a layer, where a layer corresponds (in general) with two physical masks: an interconnection mask

and a contact (hole) mask.
The grid representation of gate arrays will serve as a basis for the routing data structure.

3. SYSTEM OVERVIEW

In figure 5 an overview of the gate array design system is given.

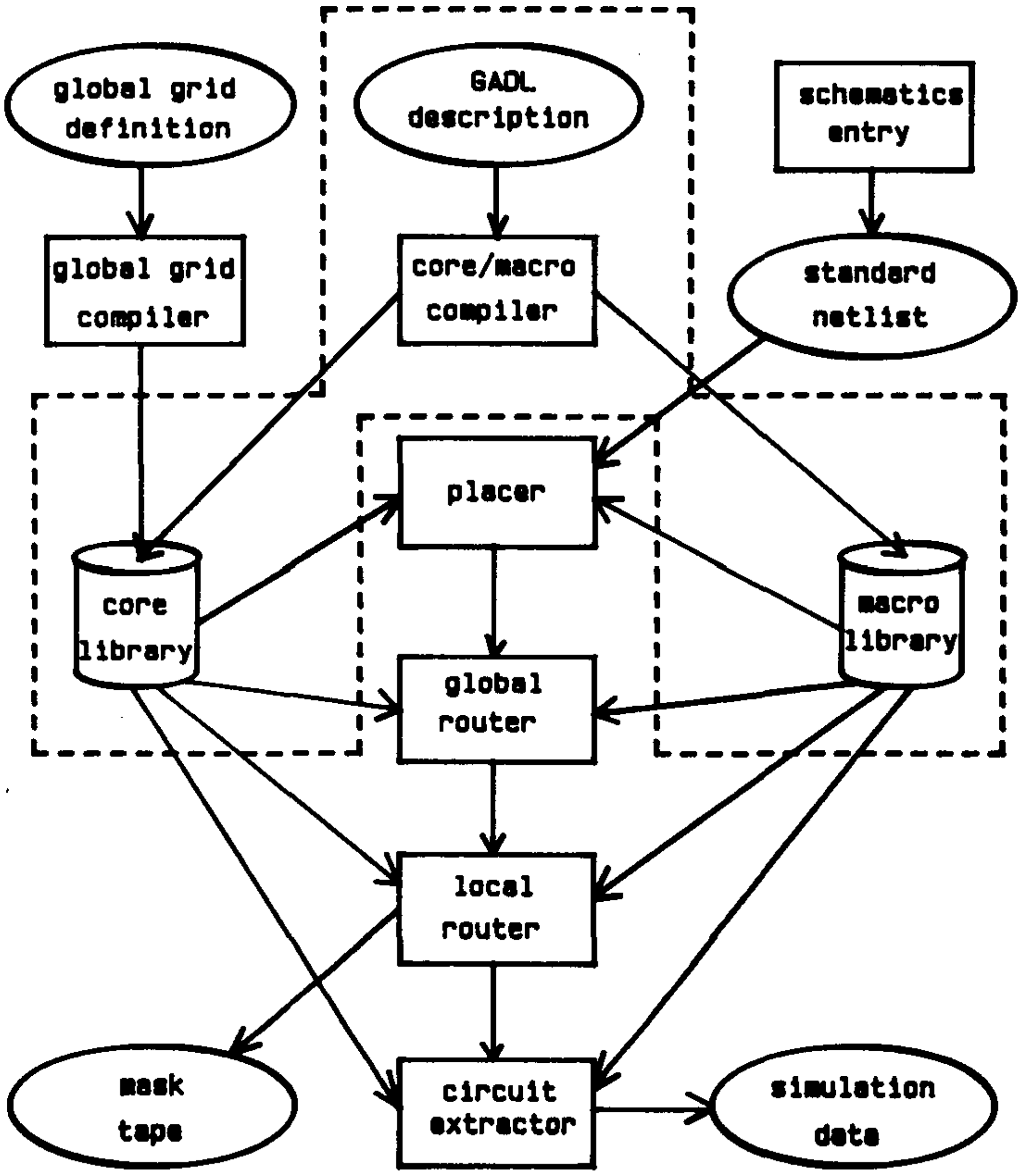


Figure 5. System overview.

First the gate array (structure, designrules, costs) and its macro stamps (structure, terminal positions, legal positions) have to be described in GADL and compiled to a datastructure suitable for the placement and routing programs. A macro is here regarded as a functional element (NAND, NOR, FLIPFLOP, etc.) while a macro stamp is one physical realisation of a functional element. The compilation has to be done just once for each gate array type. The data obtained from the *core/macro compiler* is stored in the *core library* (gate array core data) and in the *macro library* (macro stamp data).
Now netlists can be generated, either by hand (text) or by a *schematics editor*.
The *placer* takes care of the placement of macro stamps (when one macro has several physical realisations, the placer choses one stamp). The currently implemented placer works on the basis of simulated annealing (statistical cooling) [1].

After the placement step, the gate array circuit is routed globally by the *global router*. The predicate "global" is assigned to this router because it operates on a global grid. This grid is usually much rougher than the fine grid proposed in chapter 2 and is defined by the *global grid definition file*. The global grid divides the gate array in global grid cells, where the cells correspond with "natural" routing space and the cell boundaries with "natural" routing blockades (usually power lines or active area). Figure 6 shows an example how a gate array could be divided in global cells.
A complete description of the global router is given by P. Nuijten [2].
With the information of the global router, the results of the placement, the gate array core grid and the contents of the macro library, the *local router* takes care of the final routing of the gate array circuit. For this purpose, an extended Leerouter is developed [3].
In order to check the results (while developing the programs) and to simulate the circuit with regards to electrical performances (node capacities, cross coupling etc.) a *circuit extractor* will be developed.

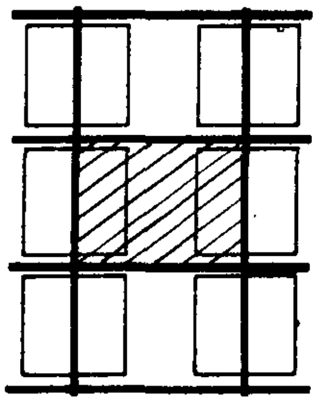


Figure 6. Global cell choice.

This report regards the GADL language, the GADL compiler and the structure of the core and macro libraries (enclosed in dashed lines in figure 5).

4. ROUTING DATA STRUCTURE

The GADL compiler is developed in parallel with the program for local routing. Most of the data generated by the compiler is used by this router. The router operates on a universal data structure (universal with regards to the gate array type) which will be described here. This data structure is the target 'language' for the GADL compiler. As proposed in chapter 2, the information for routing can be mapped onto a three dimensional grid.

First of all, a few notions with regard to grids are stated. A grid G of size XSIZE * YSIZE * ZSIZE is a set of *vertices* (gridpoints) p_i:

$$G = \{ p_i = (x_i, y_i, z_i) \mid 0 \le x_i < \mathrm{XSIZE},\ 0 \le y_i < \mathrm{YSIZE},\ 0 \le z_i < \mathrm{ZSIZE} \}.$$

x_i represents the x-coordinate, y_i the y-coordinate and z_i the z coordinate of the vertex.

The *distance* $D(p_1,p_2)$ between vertices p_1 and p_2 is defined as:

$$D(p_1,p_2) = |x_1-x_2| + |y_1-y_2| + |z_1-z_2|.$$

p_1 and p_2 are called neighbours if $D(p_1,p_2) = 1$.

An *edge* is a pair of neighbour vertices, representing the gridline part between the vertices. The set E of edges is defined by:

$$E = \{ (p_1,p_2) \mid p_1 \in G,\ p_2 \in G,\ D(p_1,p_2) = 1 \}.$$

A *path* P in G is an ordered set of vertices:

$$P = \{ p_1, p_2, \dots, p_m \mid (p_i, p_{i+1}) \in S,\ 1 \le i < m \}.$$

so that,

$$D(p_i,p_{i+1}) = 1, \qquad 1 \le i \le m.$$

The path is said to be constructed of the edges (p_1,p_2), (p_2,p_3), .. , (p_{m-2},p_{m-1}), (p_{m-1},p_m).
A *layer* L_1 is a set of vertices defined by:

$$L_1 = \{ (x_i,y_i,z_i) \in G \mid z_i = 1 \}.$$

Our task now, is to assign those attributes to the vertices and edges so that the router can fulfil its task properly (without an unjustifiable amount of overhead). In this chapter, the information needed by the router is summarised.

4.1 The position of wires, vias, wire blockades and via blockades

To map these properties onto the data structure, each edge is assigned a **signal type**. A signal type can have one of the three values: **INITIAL**, **IMAGE** or **INHIBIT**. A signal type initial indicates that the corresponding edge is free for routing (no blockade, no fixed wire or via). Likewise signal types of fixed and inhibit indicate an edge is occupied by a (prefabricated) wire or via, or blocked for routing respectively.
The router issues a fourth signal type: **ROUTER**, but this is beyond the scope of this report.
It will be obvious to the reader that in this way every wire being a path in G and every via that is positioned at a gridpoint can be modelled in the grid data structure. A careful choice of the positions of gridlines (which don't have to be equidistant) guarantees that a grid representation of each orthogonal wire and each via is possible. Non orthogonal wires however can not be transformed to a proper signal type setting of edges in the grid. Therefor the predicate **equivalent point** is introduced. In this context, two (or more) points are called (electrically) equivalent if there is a galvanic connection between these points and there is no way to model this connection in the grid representation. So a diagonal wire, that starts and ends at gridpoints, can be modelled by assigning an equivalent relation to these points. Care must be token that no other wire can cross the spanning orthogonal region of this diagonal

wire.
Generally each vertex has a set of equivalent vertices associated with it. It will be obvious that if p_2 is an element of the equivalent vertex set of p_1, p_1 will be an element of the equivalent vertex set of p_2.

4.2 Design rules

Design rules are divided into numeric and structural rules. Structural rules tell something about preferred directions, not recommendable configurations etc.. They can be merged into the Leerout cost function and will not be discussed here.
In general numeric design rules take the form of a set of permissible geometries that can be used by the designer to make devices and interconnections within the resolution of the process and without violating the device physics required for their proper operations. They can be reduced to their simplest form: a set of geometrical constraints of the form of minimum allowable values for certain widths, separations, extensions and overlaps of geometrical elements. The length unit is usually in Lambda [4].
Gate array (numeric) design rules are quite different from the general ones. In gate arrays all the devices (transistors) are prediffused and only those processing steps involving the interconnections have to be carried out. Now, the width of wires and the size of via holes are process determined (and can be regarded sizeless) so the gate array design rules only concerns the minimum separations between interconnection layout elements (wires and via holes). The most basic design rules are already mentioned:

- All interconnections have to be made on gridlines
- All vias have to be made on crossing points of gridlines (gridpoints)

These rules are automatically provided by our way of modelling. Another consequence of the gridding process is that spacing rules can be described in terms of grid steps instead of Lambda. Because of the fact

that gridlines don't have to be equidistant, and design rules are expressed in terms of grid steps, design rules are *position dependent*.
In our system, the design rules are modelled by the concept of **shadowing**. In general each edge in the grid shadows other edges in its surrounding area. If an edge s_1 shadows an edge s_2, it means that if s_1 is occupied (i.e. a wire or via runs along the edge), s_2 should be free and vice versa.
In practice this means that, if the router decides to occupy edge s, all edges shadowed by s should be free. Up to now we have been able to model all numeric gate array design rules by this shadowing process.
For example, a horizontal edge can shadow horizontal edges on the adjacent horizontal gridlines (figure 7a).

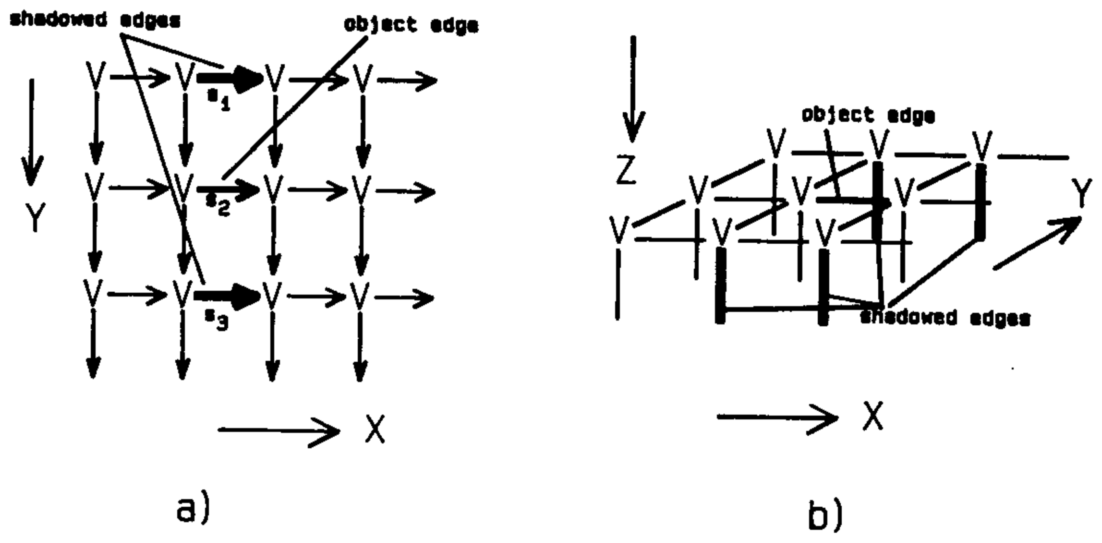


Figure 7. **Shadowing**

This means that if the router decides to rout a wire along edge s_2, its shadowed edges (edge s_1 and s_3) have to be free i.e. not occupied by another net (figure 7a). Likewise a wire can shadow vias on adjacent gridlines (figure 7b).
It will be obvious that if s_1 shadows s_2, then s_2 will shadow s_1, so for every shadow relation a complementary relation exists.

4.3 Leerouter cost function

The Leerouter is guided by a path cost function F. This function defines (for each edge s) the costs F(s) to extend the path along s. The costs F(P) of a path P = $\{p_1,p_2,..,p_m\}$ is now defined as the sum of the costs of the edges the path is constructed of:

$$F(P) = \sum_{i=1}^{i=m-1} F((p_i,p_{i+1})).$$

The router tries to find a minimum cost path between the gridpoints it has to connect.
F has to be consistent to guarantee the proper operation of the router: [5]

Path consistency property:
Let F be a path cost function, P any minimum cost path from vertex A to vertex B, and Q any minimum cost path from B to C. If PQ is a minimum cost path from A to C through B, then F is called consistent with respect to P and Q. If F has this property for all choices of A, B, C, P and Q, then F is called consistent.

This implies that every path cost function F, where the costs to add an edge s to the path P are independent of the edges P is constructed of, is consistent.
Although the cost function could have any exotic shape and could be dependent on anything, we have to keep in mind that the router has to compute the function value every time it extends a path. Therefor we have restricted the properties of the function to the following simple form:

Suppose s = (p_1,p_2), then F(s) is defined as:

$$F(s) = \begin{cases} c_o & \text{if } p_1 \text{ or } p_2 \text{ is occupied} \\ c_n & \text{if } p_1 \text{ or } p_2 \text{ is not occupied} \end{cases}$$

where c_o and c_n are predefined constant costs values. A vertex p is called occupied with respect to a rout R, if p is a start point of R.
In spite of the fact that the above mentioned cost function form is very simple, and may be too simple in some cases, we have been able to model most of the desired costs in this form, while the 'computation' of the edge costs can be done by just testing the occupancy condition.

4.4 Macro stamp information

For macro stamps some extra information has to be stored. First there are the positions of the terminals. The router has to know the exact position of the macro stamp terminals.
Second, the placer needs to know where the legal positions of the macro stamps are. A legal position of a macro stamp is some position in the gate array core, where the stamp performs its required function. For most gate array types the legal positions of macro stamps are sets of equidistant points.
Besides the terminals and legal positions we need to know the position of the internal wires of the macro stamps. Internal wires are those nets of a macro, which are not electrically equivalent to any terminal. These wires can not be used for routing and should be isolated. Isolation can not be performed during compilation because the internal wires can have connections to the 'outside' world. This implies that internal wires should be isolated during the design phase of the circuit. However, when the user specifies the positions of terminals and power lines, internal wires can be extracted automatically. Terminals and power lines are easier to discover than internal wires, so we will perform this automatic extraction.
Finally we need to know the equivalent terminals of a macro. Terminals are called equivalent if the nets connected to these terminals can be interchanged without changing the function of the macro. For example, the inputs A and B of a 2 input 'and-gate' can be interchanged, while the function will remain out = A.B . The router is not yet able to exploit this feature, while routing the circuit, but in future it will be.

4.5 Implementation considerations

The in the previous sections proposed modelling of gate array features should be implemented in a form where we have an easy access to the proposed properties, and where the amount of required memory space is minimal.
The grid is stored as a three dimensional array of vertices. This is a minimum because we need information about each vertex, so for each gridpoint data should be present. The edge descriptions are merged into the vertex records. Every gridpoint contains information about three edges: the edge in the north direction, the edge in the east direction and the edge in the down direction. The compass-card is given in figure 8.

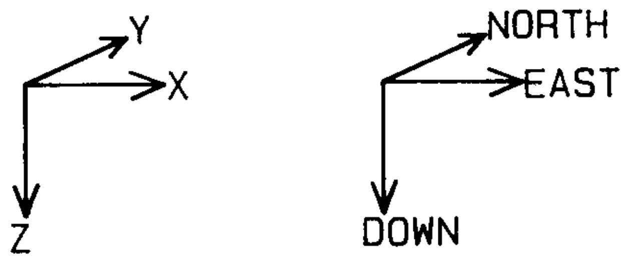


Figure 8. Compass-card

What information has to be stored at each vertex ?

1. *The signal type of the associated edges.* This means that in the vertex data structure minimal 6 bits have to be reserved for the signal types of the three corresponding edges.
2. *Equivalent positions to the vertex.* Equivalent relations are merged into the data structure by using a so called **equivalence table.** Each vertex has a **equivalence index field** and a **equivalence offset field.** The equivalence index is a pointer into the equivalence table. Each entry in the table represents a set of equivalent points. The equivalent points are stored cyclic as offsets to the next point in the set. The equivalence offset addresses a point in the cycle. Figure 9 shows the representation of two sets of equivalent gridpoints. This way of storage makes it possible to represent different sets of

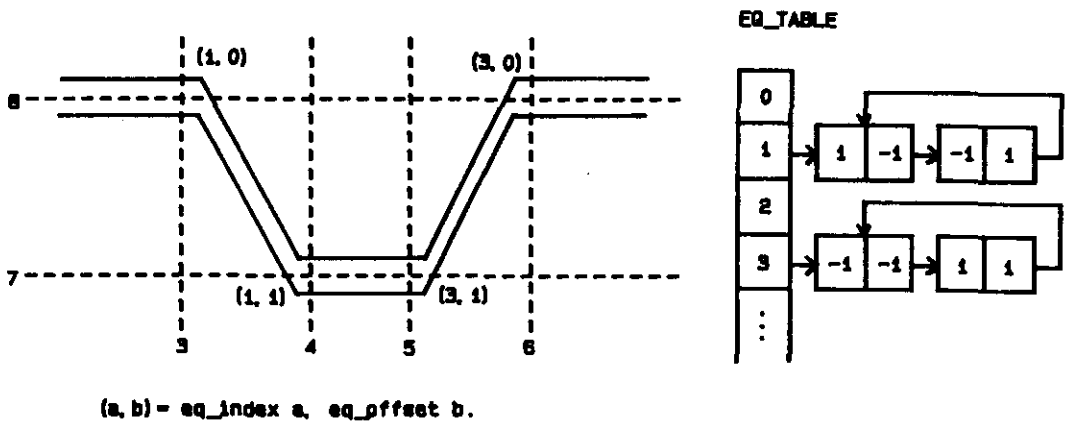


Figure 9. Equivalent points

equivalent points, that have the same geometrical appearance, by just one table entry. Because of the repetitive structure of gate arrays, this means that the number of table entries is usually rather small, while the information stored at each vertex is minimal.

3. *Shadowed edges of the associated edges*. To store these properties, here again an external table, the **design rule table**, is used. Each vertex contains a **design rule index**, which is an entry in the design rule table. Every table entry contains three shadow lists. The first is the list of edges, shadowed by an east edge at the vertex. The second the list of edges, shadowed by a north edge and third the list of edges, shadowed by a down edge. A shadowed edge is fully determined by the position of the associated vertex and the direction of the edge (north, east or down). Therefor we store the shadowed edges as a relative position (relative to the position of the vertex in question) plus an indication about the direction of the edge. The direction indicators are: EAST_EDGE, NORTH_EDGE and DOWN_EDGE. A shadowed edge can now be addressed by two attributes: (direction, rel_position).
For example, if the east shadow list at some gridpoint contains the elements (EAST_EDGE, (0,1,0)) and (EAST_EDGE, (0,-1,0)), it means that an east edge at the vertex shadows horizontal edges on adjacent tracks (figure 10).

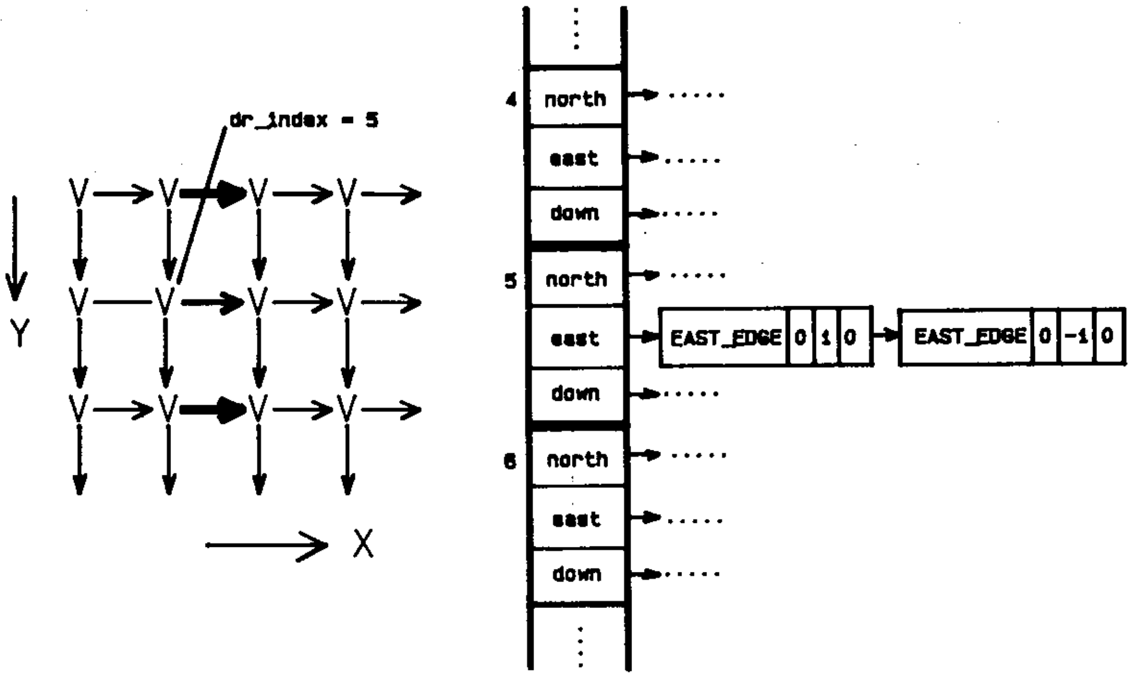


Figure 10. Design rule table.

Again because of the repetitive structure of gate arrays (and their design rules) the number of entries in the design rule table is quite small.

4. *The costs to occupy each of the associated edges.* As could be expected, here again we use an external table: the **cost table**. A vertex is assigned a **cost index** which represents an entry in the cost table. Every entry in the cost table contains six absolute cost values being the c_o's and c_n's of the associated (three) edges. If the router decides to occupy edge $s = (p_1,p_2)$, then dependent whether p_2 is occupied with respect to the path currently expanded, the cost for this occupation is defined by the corresponding c_o or c_n. Figure 11 shows an example of cost definitions using a cost table.

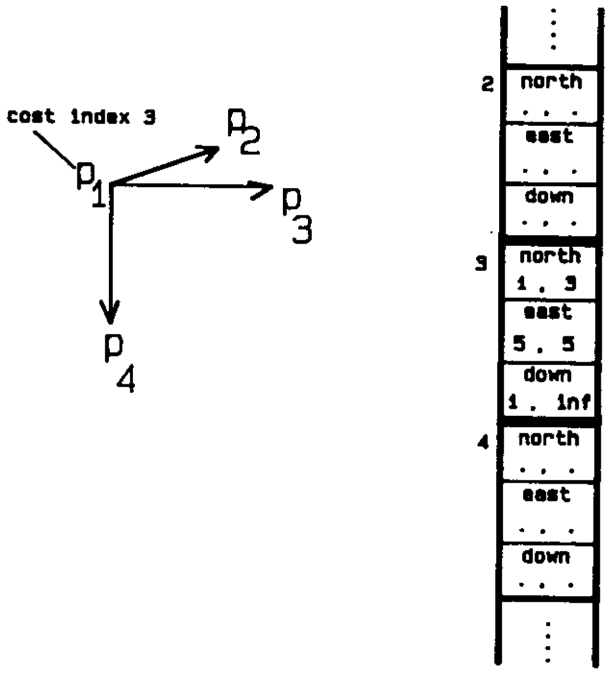


Figure 11. Cost table

4.6 Summary

The for the compiler interesting datafields at each gridpoint (vertex) are:

- North signal type
- East signal type
- Down signal type
- Equivalence table index
- Equivalence table entry offset
- Design rule table index
- Cost table index

Besides this vertex data structure three tables are necessary:

- Equivalence table
- Design rule table

- Cost table

Macro stamps require some extra information:

- Terminal positions.
- Legal positions.
- Internal wire positions.
- Equivalent terminals

The definition of the vertex data structure is given in appendix A.

5. LIBRARY STRUCTURES

In the previous section, the routing data structure was stipulated. The two libraries where the data is stored are described in this section.
For the gate array core, the data is stored in a directory (gate array core library). In here, each item (grid, equivalence table, etc.) corresponds with a special file.

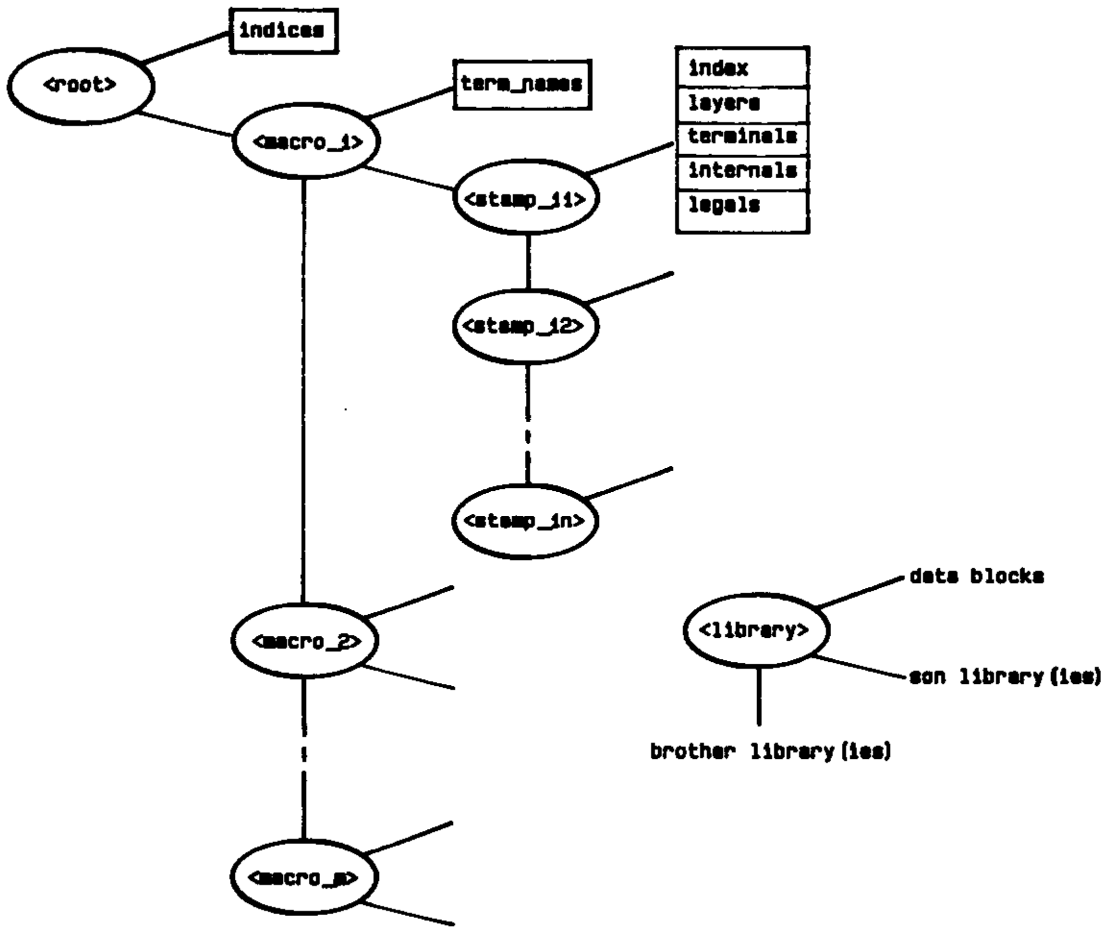


Figure 12. Macro library structure

Besides the gate array core data however, we need information about the macro stamps. Of course, here again a grid must be stored, but we also need information about the stamp's terminal positions, the position of internal wires (these can not be used for routing) and its legal positions (a macro stamp can not be mapped at every arbitrary place in the

core, but has several predefined positions where the stamp performs its required behaviour). For this purpose a directory or directory tree could be used, but we have chosen (for sake of surveyability) for a separate macro library file. The file is a binary dump of the macro library structure given in figure 12. The used library access routines are extensively described in literature [6].
Now that we have defined the target language for the compiler, the source language (GADL) will be described.

6. THE GADL LANGUAGE

6.1 Goals of GADL

The objective of GADL is to describe the features of gate arrays, as mentioned in the previous sections, in an easy, compact and 'natural' way. The frame of GADL is a combination of the languages GADSL and GADRL, as proposed by Jichun Bu [7]. In these languages it is possible to describe the gate array core structure and its design rules. These languages are combined, adjusted and extended so that we can now describe the total gate array (inclusive macro stamps, design rules and cost function properties) in one uniform grammar.
Because of the repetitive structure of gate arrays the operation repetition is supported in GADL. Also transformations (mirror, rotation) on grid structures are legal operations. Besides the repetitive features, the language is hierarchical to an unlimited depth.
The combination of hierarchy and repetition makes it possible to describe most gate array cores in an extremely compact way (1 a 2 pages).

6.2 An outline of GADL

Each legal GADL sentence starts with a library definition followed by a layer declaration. Then an item list with all image- design rule- and cost function descriptions follows.
syntax:
```
<GADL_sentence> ::= "(" <library_def>
                        <layer_decl>
                        <item_list> ")".
```

The library definition defines the location where the libraries should be stored.
syntax:
```
<library_def> ::= "(" "LIBRARY" <dir_name> ")".
```
semantics:

<dir_name> is the name of a (UNIX) directory. In this directory a sub-directory *CORE* is created. In here, all gate array core information is stored (gate array core library). In <dir_name> the file *MLIB* is created too. This file contains the information about the macro stamps (macro library). In the sub directory *MACROS* of <dir_name>, for each macro, a directory <macro_name> is generated, where the names of the macro's terminals and a file flag called *gate_array* are stored. The sub directory MACROS is used by the schematics entry program: ESCHER. The directory structure is visualised in figure 13.

example:

(*LIBRARY /users/paul_1/TAL004*)

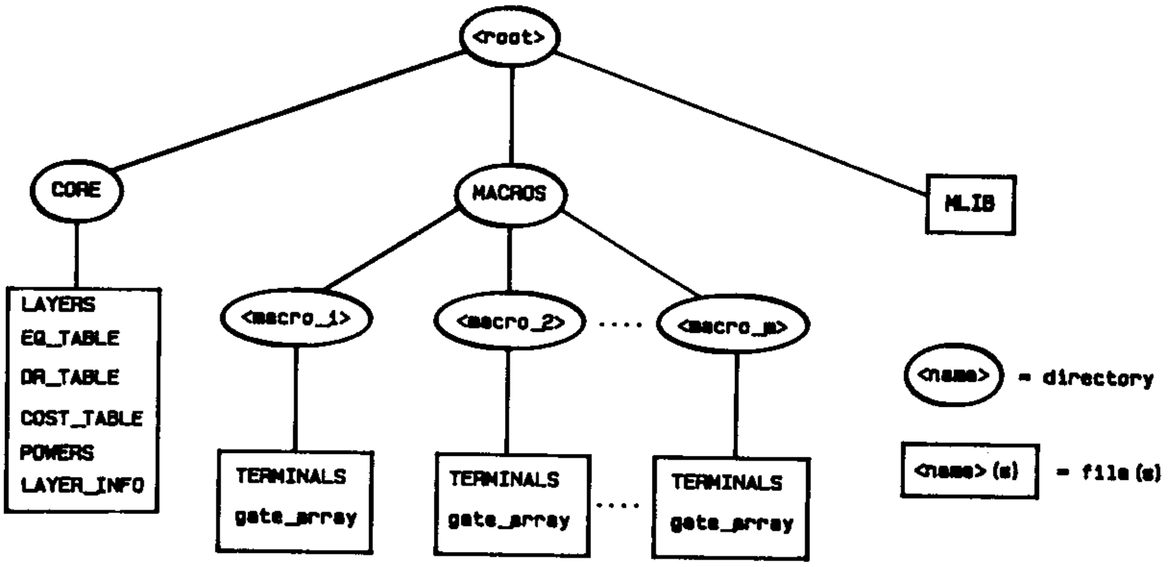


Figure 13. The directory structure

The layer declaration statement declares all layer names used in the GADL description.

syntax:

<layer_decl> ::= "(" "LAYERS" {<layer_name>}+ ")".

semantics:

The layer names have to be declared from the wafer upto the surface.

example:

(*LAYERS diffusion poly metal1 metal2*)

The <item_list> is a list of items:
syntax:
<item_list> ::= {<item>}+.

We distinguish four kinds of items:

- image definition item
- range definition item
- design rule definition item
- cost function definition item

6.2.1 Image definition item

The image definition item defines the geometrical appearance of the core or the macro stamps. These alternatives are mutually exclusive.
As mentioned in section 6.1 the structure description is hierarchical. The basic blocks are *modules*. A module can either be a simple module (type MODULE), a core module (type CORE) or a macro module (type MACRO). A simple module only describes geometry of a grid part. A core module defines the geometry of the total gate array core and a macro module defines the geometrical appearance of a macro (and its associated stamps). Simple modules can be called at certain positions in other modules. Of course, no recursion is allowed because an endless loop would be created.
Now the total image definition item is a list of modules. When a core module is defined, no macro modules are allowed.
syntax:

```
<image_definition> ::= "(" "IMAGE" {<module>}+ ")".
<module> ::= "(" "CORE"   <name>
                     {<power_name>}*
                     <simple_module_body>
                ")"
           | "(" "MODULE" <name> <simple_module_body> ")"
           | "(" "MACRO"  <name> <macro_body> ")".
```

```
<simple_module_body> ::= <dimension_sm> {<image_statement>}*.
<macro_body> := "(" {<term_name>}* ")"
                "(" {<power_name>}* ")"
                {<eq_term_set>}*
                {<stamp>}+.
```

semantics:

In the description of the core, the names of the power lines (and their positions) should be defined, because they are treated as special nets (if they don't appear in the netlist they should be isolated). The body of a simple module is just a dimension specification followed by a list of image statements. Some image statements are only meaningful and legal in macro stamp descriptions. For example the definition of a terminal in a simple module is meaningless and therefor not allowed.

The body of a macro consists of a list of terminal names, a list of power names, a list with equivalent terminal sets and a stamp list. The terminal names and power names of one macro are the same for all stamps. The positions of the terminals and the powers should be defined within the stamp body. The positions of the power lines are important when we want to determine the internal wires of a macro stamp. The equivalent terminal sets are identical for all stamps too.

syntax:

```
<eq_term_set> ::= "(" "EQTERM" <term_name> { <term_name>}+ ")".
```

example:

(EQTERM in1 in2 in3)

A stamp defines one physical realisation of a macro. Different stamps of one macro could have totally different geometric appearances, but they can also have the form of rotations or mirrors of one basic structure.

In the latter case, it is advisable to define a simple module containing the basic structure, and call this module (with the proper transformations) in the macro stamp bodies.

syntax:

```
<stamp> ::= "(" <name>
                <dimension_sm>
```

{<image_statement>}* ")".

All image statements concern positions in grids. For ease of speech, first a few notions with regards to grids are stated. Each simple module, core module or macro stamp has its own *local grid.* The gridlines in this grid have to correspond with gridlines in the fine routing grid. However, sequential gridlines in the local grid can have many fine gridlines between them. Gridlines where no module is called are regarded as the gridlines of the fine routing grid. The gridpoints in the local grid are addressed by a *position.* A position is a pair of integers representing the x gridline and the y gridline in the local grid respectively. Note: the origin of the coordinate-system is (0 0).
Gridpoints in the fine grid are addressed by *coordinates.* It will be obvious to the reader that every position in the local grid corresponds with a coordinate in the fine grid.

The size of the local grid is defined by a <dimension_sm>.
syntax:
<dimension_sm> ::= "(" "DIMENSION" <x_dim> <y_dim> ")".
semantics:
This statement defines the dimension of the module's local grid. <x_dim> defines the number of y-axis-parallel gridlines. Likewise, <y_dim> defines the number of x-axis-parallel lines.
example:

(DIMENSION 14 12)

Now we have to choose our set of image statements in such a way that all properties mentioned in chapter 4 can be described. We should also have the possibility to indicate the router freedom in the different layers to determine reducibility of layers (see section 7.1.1), and to make the description more compact.
syntax:
<image_statement> ::= <layer_sm>
| <call_sm>

```
                    | <wire_sm>
                    | <nwire_sm>
                    | <via_sm>
                    | <nvia_sm>
                    | <pvia_sm>
                    | <equivalence_sm>
                    | <terminal_sm>
                    | <power_sm>
                    | <legal_sm>.
```

The <layer_sm> specifies the default wire and via types.

syntax:

<layer_sm> ::= "(" "LAYER" <name> <wire_type> <via_type> ")"

semantics:

The wire and via types determine the degree of freedom for the router in the layer defined by <name>. <wire_type> or <via_type> can either be "*FIX*(*ED*)" or "*PROG*(*RAMMABLE*)". A type of FIX means that there is no freedom for the router to expand its paths in the layer. Likewise a type of PROG means that the router can create wires (or vias) in the corresponding layer. The layer type definition of the core module is used to determine the reducibility of layers. Layers with both via type and wire type set to FIXED offer no freedom for the router and can be reduced.

examples:

```
        ( LAYER diffusion FIXED FIXED )
        ( LAYER metal     PROG  FIX )
```

In the layer 'diffusion', the router can not create wires or vias (to the layer beneath). In the layer 'metal', wires can be generated, but vias are at (user defined) fixed positions.

A <call_sm> calls a simple module (optionally transformed) at a certain position in the local grid.

syntax:

<call_sm> ::= "(" "AT"

```
                    {<position>}+
                    [<copy_attr>]
                    {<transform>}*
                    <name> ")".
<position> ::= "(" <xpos> <ypos> ")".
<copy_attr> ::= [ "CX" "(" <integer> <integer> ")" ]
                [ "CY" "(" <integer> <integer> ")" ].
<transform> ::= "MX" | "MY" | "R90" | "R180" | "R270".
```

semantics:

This statement defines calls of simple module <name> at all local grid positions defined by {<position>}+ and <copy_attr>. The size of the called module with respect to the local grid is always 1 * 1. The defined positions are the points where the left under corner of the called module is mapped to.

The two integers in each copy attribute represent the delta- and times-value respectively. So CX(2 3) means: copy three times in the x direction with a delta of two. The <copy_attr> operates on all positions of {<position>}+. If a module is called at position, let's say (x y), then between the local gridlines x and x+1 as many fine gridlines as required by the called module are inserted. The same holds for the y direction. The first gridline of the called module covers the local gridline of the calling module at the called position. Care must be token that all modules, called at coordinate x, require an equal number of gridlines to be inserted.

The transformation MX mirrors the called module in the x-axis, MY mirrors in the y-axis. R90 rotates the called module 90 degrees anti clock-wise. Transformation are executed 'from the <name> away'.

example:

(AT (3 3) CX(2 2) CY(1 3) MX R90 gate_cell)

Here, the module 'gate_cell' is called (first rotated and then mirrored) at the positions (3 3), (5 3), (7 3), (3 4), (5 4), (7 4), (3 5), (5 5), (7 5), (3 6), (5 6) and (7 6). These are the positions where the left under corner of the (already transformed) module 'gate_cell' is mapped to.

Wire statements define the galvanic horizontal connections (horizontal means: situated in one layer).

syntax:

```
<wire_sm> ::= "(" "WIRE" <name>
                  {<coordinate>}+
                  [<copy_attr>] ")".
<coordinate> ::= "(" <int> ["." <int>] <int> ["." <int>] ")".
```

semantics:

A wire is here regarded as a horizontal galvanic connection between gridpoints in the layer defined by <name>. The wire sequentially connects the gridpoints defined by their coordinates. When non-orthogonal wire-parts are defined, the spanning region of the wire part is blocked for routing, i.e. when the wire-part runs from (x1 y1) to (x2 y2), the region formed by the cartesian product of (x1,x1+1, .., x2) and (y1, y1+1, .., y2) is blocked for routing (no wires can enter this region). The copy attribute operates on the total wire shape.

Coordinates defined in a wire statement are a little more complex than the positions defined in a call statement. Where the positions in a call statement only address positions in the local grid, in a wire statement it is possible to address coordinates in the fine grid. This is established by adding sub coordinates to the positions in the local grid. These sub coordinates define the number of fine gridlines that have to be added to the fine grid coordinate corresponding with local grid position. For example the coordinate (5.3 8.9) defines a coordinate obtained by stepping 3 fine gridlines in the x direction from the fine gridline corresponding with local gridline 5, and 9 gridlines in the y direction from the fine gridline corresponding with local gridline 8. This feature can be very useful to define powerlines (which are across the whole gate array) and to connect modules. Sub coordinates are only legal at positions where modules are called, and where the sub coordinate does not exceed the called module's fine dimension.

NOTE: The copy attribute only affects the local grid position. It is not legal to add sub coordinates to the copy attributes.

examples:

(WIRE metal1 (0 0) (3 0) (3 4) CX(4 1))
(WIRE poly (0 8.1) (0.13 8.1) CX(2 1))

The first statement defines two wires of equal shape. The second again defines two wires and shows the use of sub coordinates. One wire runs from (0 8.1) to (0.13 8.1), the other from (2 8.1) to (2.13 8.1).

A nwire statement defines the position of wire blockades.
syntax:
```
<nwire_sm> ::= "(" "NWIRE" <name>
                {<coordinate>}+
                <copy_attr> ")".
```
semantics:
A nwire statement defines gridline parts that are blocked for routing. If a diagonal wire segment is detected, the spanning region is blocked as a whole. The same remarks as to wire statements apply to nwire statements.
example:

(NWIRE poly (3.4 0) (5 0) CY(2 3))

Via statements define the position of vias (contact holes).
syntax:
```
<via_sm> ::= "(" "VIA" <name1> [ "." <name2>]
              {<coordinate>}+
              [<copy_attr>] ")".
```
semantics:
A via statement defines vias (holes) from layer <name1> downto layer <name2>. If no <name2> is defined, a via to the underlying layer is assumed. The positions of the vias are defined by the coordinates and the <copy_attr>. If two layer names are defined, it is necessary that <name1> is nearer to the surface of the wafer than <name2>. For the coordinates, the same as for the wire statement holds.
example:

(VIA metal2.diffusion (0 0) (3.2 4.6) CX(2 1))
(VIA poly (0 0) (0 5) (0 11) CX(2 10))

It is advisable to define 'related' vias in just one statement in stead of using a separate one for each individual via.

Nvia statements define the via blockades.
syntax:
```
<nvia_sm> ::= "(" "NVIA" <name1> ["." <name2>]
                  {<coordinate>}+
                  <copy_attr> ")".
```
semantics:
At the defined positions, no via can be created by the router.
example:
        (NVIA metal.poly (1 0) (1 5) CX(2 100) CY(20 8))

Pvia statements define the positions of programmable vias.
syntax:
```
<pvia_sm> ::= "(" "PVIA" <name1> ["." <name2>]
                  {<coordinate>}+
                  <copy_attr> ")".
```
semantics:
This statement is useful in cases where the layer via type is set to FIXED and where at some pre-defined positions vias can be created. The pvia statement defines those positions.
example:
        (PVIA poly (1 0) (1 5) CX(2 100) CY(20 8))

An equivalence statement defines a set of electrically equivalent points.
syntax:
```
<equivalence_sm> ::= "(" "EQ(UIVALENT)?" <name>
                         {<coordinate>}+
                         <copy_attr> ")"
```
semantics:
An equivalent statement defines a set of electrically equivalent points in the layer <name>. The set is defined by the coordinate list. <copy_attr> does not extend the set, but increases the number of sets. In

fact it creates another set with the same shape but located elsewhere.
example:

(EQ poly (3.1 0) (6.8 3.9) CX(5 1) CY(3 7))

A terminal statement defines the position of a terminal.
syntax:

```
<terminal_sm> ::= "(" "TERM(INAL)?" <name1> <name2>
                      {<coordinate>}+ ")".
```

semantics:
This statement is only meaningful and legal in a macro stamp body. <name1> is the name of the terminal. The name has to be declared in the terminal list of the macro module. <name2> defines the name of the layer, the coordinate(s) of the terminal are positioned. Only one coordinate of a terminal tree has to be defined. All galvanically connected coordinates are regarded as terminal positions. Usually the coordinate list exists of one coordinate. The defined layer should be not-reducible.
NOTE: A terminal position definition for each declared terminal should be present.
example:

(TERMINAL in2 metal (0 5.1))

The power statement defines the position of power lines.
syntax:

```
<power_sm> ::= "(" "POWER" <name1> <name2>
                   {<coordinate>}+ ")".
```

semantics:
All remarks concerning a terminal statement apply to a power statement. However, power position definitions are also allowed in the core module description.
example:

(POWER vdd metal (0.9 6))

A legal statement defines the legal positions of a macro stamp.
syntax:

```
<legal_sm> ::= "(" "LEGAL"
                   "(" <start> <delta> <end> ")"
                   "(" <start> <delta> <end> ")"
               ")".
<start> ::= <int>.
<delta> ::= <int>.
<end>     ::= <int> | "END" ["-" <int>].
```

semantics:

Each legal statement defines a set of legal positions. The first <start>-<delta>-<end> sequence concerns the x-direction, the second the y-direction. <start> defines the first fine grid coordinate in the core grid. This position is repeated with delta <delta> until <end>. The legal position set is formed by the cartesian product of the x-direction coordinates and the y-direction coordinates. If <end>is "END", it means that the coordinates should be repeated until the size of the gate array core. <end> does not has to be: a multiple of <delta> plus <start>.

examples:

(LEGAL (0 20 200) (0 10 50))

(LEGAL (0 14 END) (0 8 END-1))

6.2.2 Range definition item

The range definition item defines ranges, and assigns names (identifiers) to them. These ranges may be used in the design rule and cost function part of the description.

syntax:

```
<range_item> ::= "(" "DEFINE" {<range_def>}* ")".
<range_def> ::= "(" <name1> {<interval>}+ ")".
<interval ::=   <integer1> [".." <integer2>] [<copy_op>]
              | "ALL"
              | <name2> ("+"|"-") <integer3>.
```

semantics:

A range definition assigns <name1> to the range defined by the interval list. A range is a sequence of x (y) coordinates. An interval is a closed sequence of coordinates (optionally copied) or a point to the definition of an earlier defined range (optionally offsetted). In the interval definition, just an <integer1> defines the coordinate with that value. <integer1> .. <integer2> defines a range from <integer_1> upto and including <integer2>. Copy operators can be added to these intervals. "ALL" defines the total axis (x or y). Whether the x or y axis is meant should be clear from the context the range is called in. <name2> ("+"|"-") <integer3> defines a range obtained by adding (subtracting) <integer3> coordinates to each element of the range associated with <name>.

example:

```
        ( DEFINE
            ( vdd_x          19 CP(30 5) )
            ( poly_under_x   0  CP(2 94) vdd_x+2 vdd-2 )
            ( x_axis         ALL )
            ( junction_x     10..29 CP(30 5) )
        )
```

The first range defines a sequence 19, 49, .., 169, the second a sequence 0, 2, .., 188, 21, 51, .., 171, 17, 47, .., 167, the third a sequence 0..MAX and the last a sequence 10, 11, .., 29, 40, .., 59 etc..
Now, areas can be defined as the cartesian product of two ranges, where the first is the x coordinate range and the second the y coordinate range.

6.2.3 Design rule definition item

The design rule description knows no hierarchy because of the problems that would occur at the module's boundaries. As stated earlier, design rules are defined by the concept of shadowing. We define three kinds of objects: *HORWIRE*, *VERWIRE* and *VIA*. HORWIRE corresponds with an edge in the east direction, VERWIRE with an edge in the north direction and VIA with an edge in the down direction. With these three objects and area definitions it should be possible to model all of the design rules applying to a gate array type. The design rule part of the core description consists of a list of design rule statements.

syntax:

```
<design_rule_item> ::= "(" "DRL" {<drl_statement>}* ")".
<drl_statement> ::= "(" <object> <name>
                        "(" {<interval>}+ ")"
                        "(" {<interval>}+ ")" "SHADOW"
                        {<shadow>}+
                    ")".
<object> ::= "HORWIRE" | "VERWIRE" | "VIA".
<shadow> ::= "(" <object> <name> <integer1> <integer2> ")".
```

semantics:

A design rule statement says: When <object> is made in the area defined by the cartesian product of the two interval lists and the layer <name>, then it shadows the elements in the shadow list. A shadow element consists of the shadowed object, the layer name of the shadowed object and the position of the shadowed object relative to the position of the "make-object". <integer1> defines the delta-x and <integer2> the delta-y, so a shadow (VIA metal -1 1) defines a shadowed via in layer metal (to the layer beneath) at the position obtained by adding -1 to the x-coordinate of the position of the object made and adding 1 to the y-coordinate. Every ("make-object",shadow) combination has its complement, i.e. if object A at (x,y) shadows object B at (x+dx, y+dy) then object B at (x,y) shadows object A at (x-dx,y-dy). These complements are generated automatically with respect to effects at the area boundaries. This means that the area defined in the input is not automatically copied to the complement relation, but shifted over the delta's as defined in the shadows.

example:

```
        ( DRL

          ( VERWIRE metal2 (in_gate) (ALL) SHADOW
            ( VIA metal2 -1 0 )
            ( VIA metal2 1 0 )
          )
```

```
    ( VERWIRE metal2 (in_channel) (ALL) SHADOW
      ( VERWIRE metal2 -1 0 )
      ( VERWIRE metal2 1 0 )
    )

    ( VERWIRE metal1 (in_channel) (ALL) SHADOW
      ( VIA metal2 -1 0 )
      ( VIA metal2 1 0 )
    )
  )
```

6.2.4 Cost function item

Here again, the cost function description knows no hierarchy because of boundary problems. The same objects: *HORWIRE*, *VERWIRE* and *VIA*. as in the design rule item are appropriate to describe the costs to occupy edges. It is also legal to use defined ranges (defined in the range definition item) in the cost function description.

syntax:

```
<cost_item> ::= "(" "COST" {<cost_statement>}* ")".
<cost_statement> ::= <cost_define> | <cost_assignment>.
<cost_define> ::= "(" "DEFINE" <name> <cost> ")".
<cost_assignment> ::= "(" <object> <name>
                          "(" {<interval>}+ ")"
                          "(" {<interval>}+ ")" "COST"
                          <costn> [ "OCCUP(IED)?" <costo> ]
                      ")".
<object> ::= "HORWIRE" | "VERWIRE" | "VIA".
```

semantics:

A cost definition statement associates the identifier <name> with the cost value <cost>. A cost assignment says: The costs to occupy the edges defined by the layer <name>, the area defined by the interval lists and the direction defined by <object> are equal to <costn>, if one of the gridpoint defining the edge is not a start point of the route, and equal

to <costo> if so. <costo> and <costn> are either values (integers) identifiers (defined in a cost definition) or "INFINITE" to indicate a cost value of infinite.
The costs of edges which are not set in the input default to the unit cost 1 (costs have to be positive).
example:

```
        ( COST
          ( DEFINE function_box_cost 100 )
          ( DEFINE junction_box_cost 200 )

          ( HORWIRE metal (diffusion_x) (diffusion_y)
            COST INFINITE OCCUPIED 1
          )

          ( HORWIRE metal (function_x) (function_y)
            COST function_box_cost
          )

          ( VERWIRE metal (junction_x) (junction_y)
            COST junction_box_cost
          )
        )
```

In appendix E an example of a gate array core description and some macro descriptions are included.

7. THE GADL COMPILER

In this chapter the algorithms and data structures used by the actual compiler are summarised. All routines operate on an internal data structures, independent of the input syntax. It is rather easy to redefine the input syntax (the features however have to be the same) without rewriting the compiler.
We distinguish three main parts in the compilation phase:

- Image compilation.
- Design rule compilation.
- Cost function compilation.

These parts are discussed separately.

7.1 Image compilation

Before we take a look at the internal data structure and the routines, the notion 'layer reduction' is introduced.

7.1.1 Layer reduction

In gate array technology some layers are provided by the foundry and offer no freedom for routing. It would be useless to regard these layers in our system because they only require storage space and don't contribute to the amount of information. These layers can be 'deleted' by the process of **layer reduction**. However, we can not just delete the layer(s) because they generally contain information about electrical equivalences in the layer above. When we discover that a layer can be reduced, all electrical equivalences obtained from the 'wire pattern' in the reducible layer have to be merged in the data structure of the upper layer. These electrical equivalences can be modelled in the same way as the equivalent relations discussed in section 4.5. For example: A fixed poly underpass under a routing channel of some CMOS gate array, with fixed metal poly

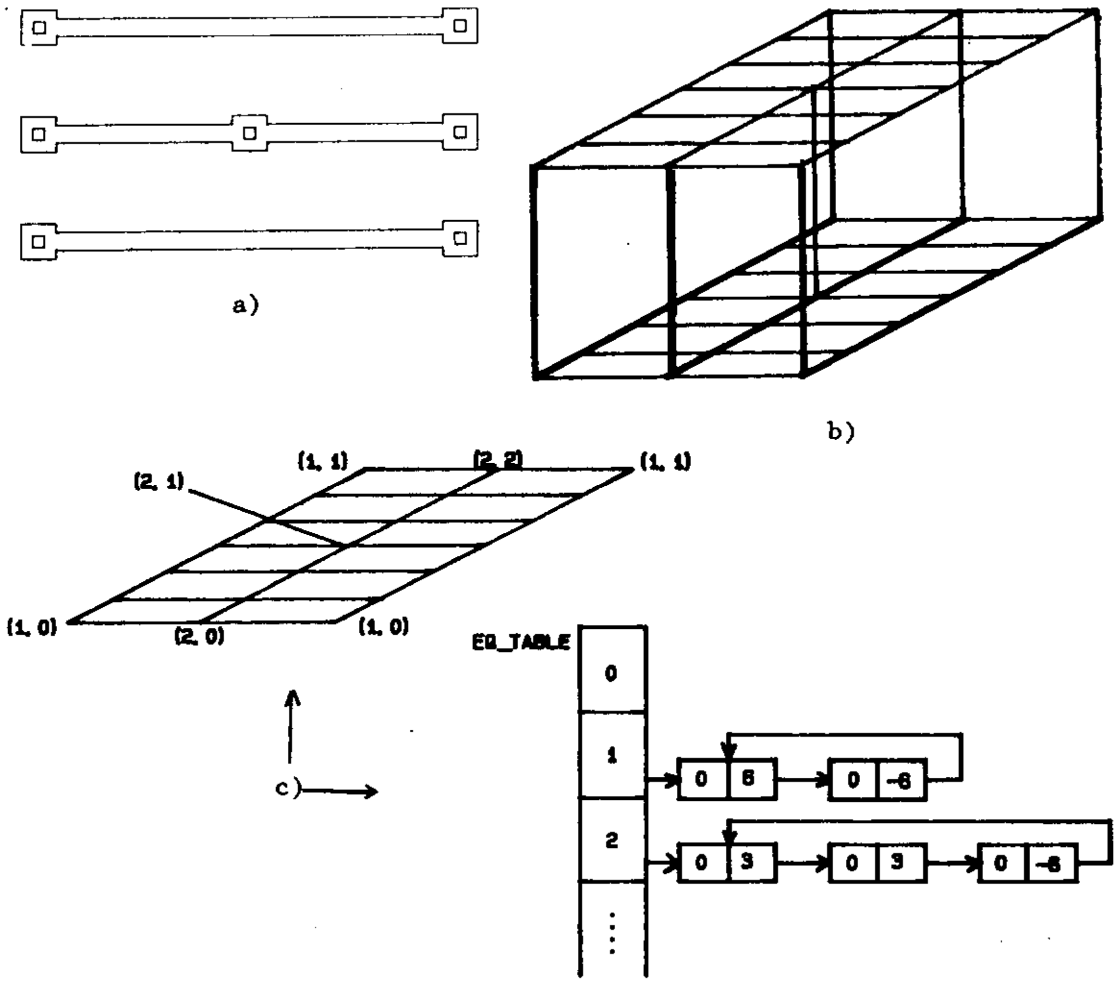


Figure 14. Layer reduction. a) geometry. b) grid model. c) reduced grid model.

vias can be modelled as shown in figure 14.

7.1.2 Data structure

The input description statements are one to one projected to an internal data structure described in this paragraph. As the description is fully hierarchical, the data structure is too. The basic block is the module (figure 15a). The modules which are not a macro stamp are linked in a list called MODULES. The macros (figure 15b) are linked in a list called MACROS. The field 'stamps' of a macro is a list of modules, where each

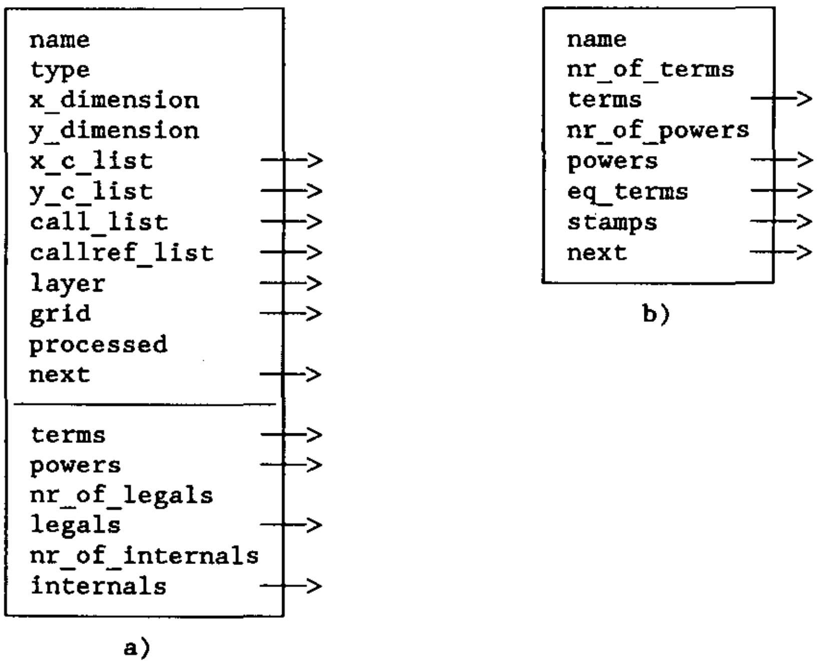


Figure 15. Structures: a) MODULE b) MACRO

module represents a macro stamp. Now, each module structure corresponds with a module-, core- or macro stamp definition in the input description.

Besides these two lists there is an extra pointer called 'CORE_MODULE' pointing to the module, describing the core of the gate array, and a list called 'LEAVES' containing all leaf cells. If a macro stamp is not calling any sub-module, the type will be 'MACRO_STAMP', but it will be added to 'LEAVES'.

Module structure description:

- *'name'* is the name of the module as defined in the input.
- *'type'* can either be MODULE_CELL, CORE_CELL, LEAF_CELL, MACRO_STAMP or NOT_CONNECTED. MODULE_CELL, CORE_CELL and MACRO_STAMP are self explanatory. A module is assigned the type LEAF_CELL if it calls no other module. NOT_CONNECTED means that the module is not called and is not a CORE_CELL or MACRO_STAMP.
- *'x_dimension'* and *'y_dimension'* define the size of the local grid.

'x_dimension' is the number of y-axis parallel local gridlines, 'y_dimension' the x-axis parallel gridlines.

- *'x_c_list'* and *'y_c_list'* are arrays of size x_dimension and y_dimension respectively. They are used during the expansion of the local grid to the fine grid (hierarchy processing) and are discussed in the description of the expansion routines.
- *'call_list'* is the list of called modules, inclusive the positions where the modules are called, and the effective transformations.
- *'callref_list'* is the list of all modules that call this one.
- *'layers'* is an array of size NR_OF_LAYERS, where NR_OF_LAYERS is the number of layers declared in the input description. Each element of layers contains the (n)wires, the (np)vias, the equivalent relations and the default wire and via types (for that layer) as described in the input.
- *'grid'* is a three dimensional array of type CVERTEX (appendix A). Initially this array is empty (filled with nulls) but will be filled during the compilation.
- *'processed'* is a boolean to indicated that the module has been processed during some compilation phase. The initial value of 'processed' is FALSE.
- *'next'* points to the next element of the list.

Beneath the line of figure 15a the extra data for macro stamps is summarised:

- *'terminals'* is an array where the positions of the terminals are stored.
- *'powers'* is an array where the positions of the power lines are stored (is also used in the CORE_MODULE).
- *'nr_of_internals'* is the number of internal wires of the stamp.
- *'internals'* is a list of the positions of the internal wires.
- *'nr_of_legals'* is the number of legal position sets of the stamp.
- *'legals'* describes the legal position sets.

Macro structure description:

- *'name'* is the name of the macro as defined in the input description.
- *'nr_of_terms'* is the number of terminals of the macro.
- *'terms'* is a list with the terminal names of the macro.
- *'nr_of_powers'* is the number of powers defined for the macro.
- *'powers'* is a list with the power names.
- *'eq_terms'* is a list of the equivalent terminal sets.
- *'stamps'* is the list of all stamps. The elements in the list are of the type MODULE.
- *'next'* points to the next macro in the list (MACROS).

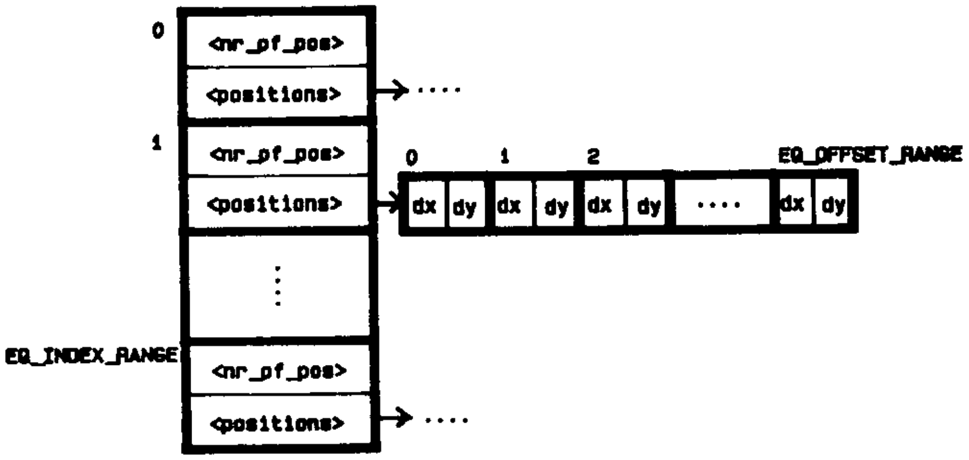


Figure 16. The EQ_ARRAY

Besides these two structures we need a special array to store the equivalent relations: the EQ_ARRAY (figure 16). The size of the array is EQ_INDEX_RANGE + 1, where EQ_INDEX_RANGE = $2^{\text{eq\_bits}}$-1. 'eq_bits' is the number of bits reserved for the equivalence index in the VERTEX data structure (appendix A). Each entry in the array represents an equivalence set. The elements in the array contains two fields: 'nr_of_pos' and 'positions'. 'positions' is an array of size EQ_OFFSET_RANGE + 1, where EQ_OFFSET_RANGE = $2^{\text{offset\_bits}}$-1. The actively used number of elements in this array is 'nr_of_pos'. The equivalent positions in the array are stored as offsets to the first element of 'positions'.

7.1.3 Routines

The routines in the image compilation part can be divided in three sections:

- Grid expansion routines
- Grid filling routines
- Layer reduction routines

NOTE: The algorithms are described in terms of a meta-language, presented in appendix C.

Grid expansion:
During grid expansion, the local grid is expanded to the fine grid. This means that between the gridlines of the local grid as many fine grids as required by the called modules are inserted. The grid expansion routines use the 'x_c_list' and 'y_c_list' fields of the modules. A 'c_list' contains three fields: 'nr_of_pos', 'nr_of_coord' and 'coord_array'. 'nr_of_pos' is the number of fine gridlines, 'nr_of_coord' is the number of local gridlines and 'coord_array' is an array of size 'nr_of_coord' Each element in 'coord_array' contains two fields: 'position' and 'expansion_list'. 'position' is the position of the local gridline in the fine grid (relative to the origin of the module), and 'expansion_list' is a pointer to the 'c_list' of the module called at that coordinate. Figure 17 shows an example of how to interpret these lists. Expand_grids is the routine that builds this part of the data structure.

- **expand_grids ()**

  synopsis:

  ```
  void expand_grids ()
  ```

  description:

  The objective of this routine is to define the fields in the 'c_list's as described above.

  algorithm:

  ```
  BEGIN
  ```

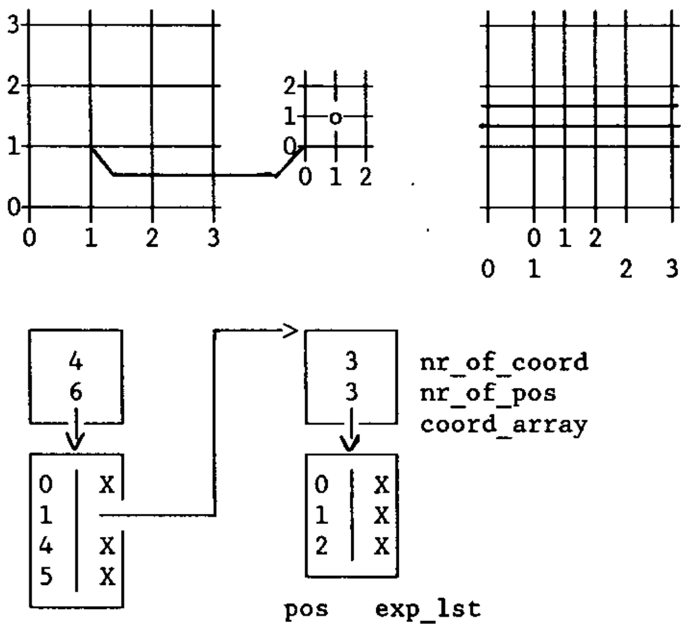


Figure 17. Coordinate lists.

```
    FOR all leafs DO
      { set processed flag in leaf          }
      { (they don't have to be expanded): }
      leaf is processed;
    ROF;
    FOR all leafs DO
      { expand (recursively) the axes }
      { of the calling modules        }
      expand super axes of leaf;
    ROF;
  END;
```

subroutine:

```
  void  expand_super_axes ();
```

- **expand_super_axes ()**

synopsis:

```
     void expand_super_axes ( module )
          MODULE_PTR          module;
```

description:

This routine recursively checks whether the grids of the modules, that call <module>, can be expanded and expands them.

algorithm:

```
BEGIN
  FOR all calling modules DO
    IF calling module not yet processed THEN
      IF all sub modules of calling module processed THEN
        expand axes of calling module;
        now calling module is processed;
        expand super axes of calling module;
      FI;
    FI;
  ROF;
END;
```

subroutines:

```
BOOLEAN  sub_modules_processed ();
void     expand_axes ();
void     expand_super_axes ();
```

- **sub_modules_processed ()**

synopsis:

```
BOOLEAN sub_modules_processed ( module )
        MODULE_PTR            module;
```

description:

Returns TRUE if all sub-modules of <module> are processed, i.e. if the 'processed' field of all sub-modules is TRUE. Else returns FALSE.

- **expand_axes ()**

synopsis:

```
void expand_axes ( module )
     MODULE_PTR   module;
```

description:

Actually expands the axes of <module>.

algorithm:

```
  BEGIN
    FOR all called modules DO
      IF at called position already expansion list THEN
        IF expansion lists not equal THEN
          error;
        FI;
      ELSE
        set expansion list to c_list of called module;
      FI;
    ROF;
    update positions in c_list of module;
  END;
```

subroutines:

```
  BOOLEAN  exp_lists_equal ();
  void     update_positions ();
```

- **exp_lists_equal ()**

  synopsis:

  ```
       void exp_lists_equal ( list1, list2 )
            C_LIST_PTR        list1, list2;
  ```

  description:

  Returns TRUE if the lists <list1> and <list2> are equal concerning number of gridlines. Else returns FALSE.

- **update_positions ()**

  synopsis:

  ```
       void update_positions ( c_list )
            C_LIST_PTR         c_list;
  ```

  description:

  Updates the position fields in the 'coord_array' of <c_list>.

Grid filling:

After the expansion of the grids, the filling can start.

- **image_cmp ()**

  synopsis:

  ```
  void image_cmp ()
  ```

  description:

  Compiles the description as defined by the statements in the input to a proper signaltype and equivalence index and offset setting in the grid.

  algorithm:

  ```
  BEGIN
    IF macros defined THEN
      determine core pattern under macro stamps;
      { fill all grids: }
      fill grids;
      find internal wires of macros;
    ELSE
      IF core module defined THEN
        fill grids; { fill all grids except the core grid }
        fill core grid; { fill the core grid }
      FI;
    FI;
  END;
  ```

  subroutines:

  ```
  CVERTEX_PTR_3 read_image_grid ();
  void          read_eq_table ();
  void          read_pattern ();
  void          free_grid ();
  void          fill_grids ();
  void          find_internals ();
  void          fill_core_grid ();
  ```

- **fill_grids ()**

  synopsis:

  ```
  void fill_grids ()
  ```

  description:

  Fills the grids of all modules except the core module (special case) according to the statements in the input definition.

algorithm:

```
BEGIN
  { processed field of the core module is set to TRUE }
  { to avoid grid filling during this routine:        }
  all modules except core module not processed;
  FOR all leafs DO
    IF leaf not processed THEN
      fill leaf layers;
      now leaf is processed;
      fill super modules (calling modules) of leaf;
    FI;
  ROF;
END;
```

subroutines:

```
CVERTEX_PTR_3 allocate_grid ();
void          fill_layer ();
void          fill_super_modules ();
```

- **fill_core_grid ()**

synopsis:

```
    void fill_core_grid ()
```

description:

Fills the grid of the core module. The core module is processed separately because in the grid of this module we can (generally) reduce layers. The core module usually is the biggest of all and requires an enormous amount of storage space. Therefor, we don't allocate the hole grid but in first instance only one layer, the nearest to the wafer. After the filling of that layer, for every other layer we check whether the layer beneath can be reduced and merged into the current layer. If so, the actual reduction takes place. The layer reduction routine returns whether a layer was really reducible. If it was, the space of the reduced layer is used for the layer above the current one.

algorithm:

```
BEGIN
  allocate bottom layer;
  fill bottom layer;
  FOR all other layers from wafer upto surface DO
    IF no space for layer reserved THEN
      allocate layer;
    FI;
    fill layer;
    IF not a not-reducible layer detected THEN
      IF layer under current one reducible THEN
        reduce layer under current one;
        IF really reduced THEN
          reserve space of layer under current one for
            layer above current one;
        ELSE
          not-reducible layer detected;
        FI;
      ELSE
        not-reducible layer detected;
      FI;
    FI;
  ROF;
END;
```

subroutines:

```
CVERTEX_PTR_3 allocate_layer ();
void          fill_layer ();
BOOLEAN       reduce_layer ();
void          init_layer ();
```

- **fill_layer ()**

synopsis:

```
void fill_layer ( module, layer_number )
     MODULE_PTR   module;
     int          layer_number;
```

description:
Fills layer <layer_number> of <module>. The layers of all called modules should be filled before this routine is called.
algorithm:
```
  BEGIN
    { fill the grid with the grids of the called modules: }
    fill calls;
    { fill the grid with all other objects: }
    fill the layer with the in the input defined wires
      for the module;
    fill nwires;
    fill vias;
    fill nvias;
    fill pvias;
    fill equivalences;
  END;
```
subroutines:
```
  void  fill_wires ();
  void  fill_vias ();
  void  fill_equs ();
  void  fill_calls ();
```

- **fill_super_modules ()**

synopsis:
```
     void fill_super_module ( module )
          MODULE_PTR            module;
```
description:
Recursively fills the super modules (calling modules) of <module>.
algorithm:
```
  BEGIN
    FOR all calling modules DO
      IF calling module not processed THEN
        IF all sub modules of calling module processed THEN
          fill layers of calling module;
          calling module is processed;
          fill super modules of calling module;
        FI;
      FI;
    ROF;
  END;
```

subroutines:

```
  BOOLEAN        sub_modules_processed ();
  CVERTEX_PTR_3  allocate_grid ();
  void           fill_layer ();
  void           fill_super_modules ();
```

- **fill_wires ()**

synopsis:

```
     void fill_wires ( module, layer_nr, list, type )
          MODULE_PTR  module;
          int         layer_nr;
          WIRE_PTR    list;
          int         type;
```

description:

Fills the grid of <module> with the information in <list>. <list> is either the wire- or the nwire list of the module. <type> is the signal type the signal type fields of the vertices have to be set to (IMAGE or INHIBIT). If <type> is IMAGE and a diagonal wire segment is defined, the ends of the wire segment are included in the EQ_ARRAY and the spanning region of the wire segment is blocked for routing.

algorithm:

```
  BEGIN
    FOR all wires in the list DO
      FOR all coordinates of the wire DO
        get next_coordinate;
        IF vertical or horizontal wire segment THEN
          set signal types;
        ELSE  { diagonal wire segment: }
```

```
          IF previous wire segment was not diagonal THEN
            add coordinate to equivalence bucket;
          FI;
          add next_coordinate to equivalence bucket;
          block spanning region for routing;
        FI;
      ROF;
      empty bucket in EQ_ARRAY and set indices in grid;
    ROF;
  END;
```

subroutines:

```
  void   set_north_type ();
  void   set_east_type ();
  void   add_eq_position ();
  void   update_eq_array ();
```

- **fill_vias ()**

synopsis:

```
     void fill_vias ( module, layer_nr, list, type )
          MODULE_PTR  module;
          int         layer_nr;
          VIA_PTR     list;
          int         type;
```

description:

Fills the grid of <module> with down signal types according to the information in <list>.

subroutines:

```
  void   set_down_type ();
```

- **fill_equs ()**

synopsis:

```
     void fill_equs ( module, layer_nr, list)
          MODULE_PTR  module;
          int         layer_nr;
          WIRE_PTR    list;
```

description:

Fills the grid of <module> with equivalence indices and offsets according the information in <list>.

subroutines:

```
  void   add_eq_position ();
  void   update_eq_array ();
```

- **fill_calls ()**

synopsis:

```
     void fill_calls ( module, layer_nr )
          MODULE_PTR   module;
          int          layer_nr;
```

description:

The layer <layer_nr> of the called modules (called by <module>) are copied (with respect to the transformations) to the grid of the calling module.

algorithm:

```
  BEGIN
    FOR all called modules DO
      compute the absolute position of the called origin;
      transform the grid and copy it to the grid of the
        calling module;
    ROF;
  END;
```

subroutines:

```
  void transform ();
```

- **transform ()**

synopsis:

```
     void transform ( target, position, call )
          MODULE_PTR  target;
          int         *position;
          CALL_PTR    call;
```

description:
Copies one layer of the grid of the module represented by <call>, with respect to the transformations, to the grid of <target> at <position>. The copied layer is the layer field of <position>. During the transformations, a vector containing: x-value, y-value, north-signal-type, east-signal-type, south-signal-type and west-signal-type is used. This vector is passed to the transformation primitives which return the vector after transformation.

algorithm:
```
BEGIN
  IF there are transformations THEN
    FOR all positions in the layer DO
      initiate the transformation vector;
      copy the equivalence row associated with the
        current position from the EQ_ARRAY to bucket;
      FOR all transformations DO
        transform transformation vector;
        transform equivalence bucket;
      ROF;
      set the vertex fields in the grid of the calling
        module according to the data in the
          transformation vector;
      empty bucket in EQ_ARRAY and set eq indices in grid;
    ROF;
  ELSE
    no transformations, just copy the grid;
  FI;
END;
```
subroutines:
```
void   mirror_x ();
void   mirror_eq_x ();
void   mirror_y ();
void   mirror_eq_y ();
void   rotate_90 ();
```

```
    void   rotate_eq_90 ();
    void   update_eq_array ();
```

The called routines of transform () are the transformation primitives. For example, rotate_90 () rotates a vector 90 degrees anti-clockwise:

- **rotate_90 ()**

  synopsis:

```
     void rotate_90 ( vector, max_x, max_y )
          int         vector [];
          int         *max_x, *max_y;
```

  description:

  Transforms <vector> so that the result represents the same point after 90 degrees rotation (anti-clockwise). Exchanges values of <max_x> and <max_y>.

  algorithm:

```
    BEGIN
      new_x     := max_y - old_y;
      new_y     := old_x;
      new_north := old_east;
      new_east  := old_south;
      new_south := old_west;
      new_west  := old_north;
      exchange max_x and max_y;
    END;
```

Layer reduction:

During layer reduction, the data in the 'fixed' layer has to be merged in the data structure of the layer above. A layer is reduced in two passes.
First all positions where a programmable via (type INITIAL) to the upper layer exists are expanded, i.e. a depth first search of the wire tree is performed and all positions in the tree where a via to the layer above exists are made equivalent.
Second all positions where a fixed via (type IMAGE) to the upper layer exists are expanded. All positions above the wire tree where a fixed via

exists are made equivalent. If there are only second pass equivalences, all information of the bottom layer is merged into the layer above. However, if there are first pass equivalences the layer con not be deleted because it contains information of the reduced layer structure. The horizontal wire types (north type and east type) however, can be set to INHIBIT because all wire information is enclosed in the equivalence settings in the reducible layer. This will limit the search space during the local routing phase.

- **reduce_layer ()**

  synopsis:
```
    BOOLEAN reduce_layer ( module, layer_index )
            MODULE_PTR     module;
            int            layer_index;
```
  description:

  According to the above mentioned procedure the layer <layer_index> of <module> will be reduced. Reduce_layer () will return TRUE if the layer is really reduced, and FALSE if second pass equivalences are detected. During the first pass, the routine add-lower_eq () is passed to the find_equs () procedure. add_lower_eq () adds a position of the reducible layer to the equivalence bucket. During the second pass, add_upper_eq () is passed to find_equs (). add_upper_eq () adds a position of the layer above the reducible one to the equivalence bucket.

  algorithm:
```
BEGIN
  { pass 1 }
  FOR all positions in the reducible layer DO
    IF programmable via to the layer above THEN
      find the equivalent points in reducible layer;
      update equivalence array with data from bucket
        (bucket is filled by find equivalences routine);
    FI;
  ROF:
  { pass 2 }
```

```
    FOR all positions in the reducible layer DO
      IF fixed via to layer above THEN
        find equivalent points in layer above;
        update equivalence array with data from bucket
          (bucket is filled by find equivalences routine);
      FI;
    ROF;
    IF equivalences during first pass detected THEN
      set horizontal signal types to INHIBIT;
    FI;
  END;
```

subroutines:

```
  void   find_equs ();
  void   update_eq_array ();
  void   set_inhibits ();
```

- **find_equs ()**

synopsis:

```
     void find_equs ( module, position, direction, routine )
          MODULE_PTR  module;
          int         position [];
          int         direction;
          void        (*routine) ();
```

description:

Recursively expands <position> in the layer (DFS). <direction> indicates which direction to omit (incoming direction). <routine> is a pointer to the routine to be called each time a via to the layer above is detected.

algorithm:

```
  BEGIN
    IF position not yet reduced THEN
      now position is reduced;
      IF there are equivalent positions to this one THEN
        FOR all equivalent positions DO
```

```
            find equivalences, direction = NONE;
          ROF;
        FI;
        FOR all directions DO
          IF this direction not equal to 'direction' THEN
            CASE direction OF
              UP:
                IF prog or fixed via to layer above THEN
                  call 'routine';
                FI;
              DOWN:
                { layer under this already reduced }
                no action;
              NORTH:
                IF connection to north THEN
                  find equs of position north, dir = SOUTH;
                FI;

              etc. for SOUTH, EAST and WEST.
            ESAC;
          FI
        ROF;
      FI:
    END;
  subroutines:
    void   find_equs ();
```

In the previous the EQ_ARRAY was used by several routines. The name equivalence bucket was mentioned too. The bucket was used to collect all elements of an equivalent set, whereafter the bucket was 'emptied' in the EQ_ARRAY. Because equivalence indices in the vertex data structure are only meaningful if they are positive (equivalence index 0 means no equivalent points), the entry 0 in the EQ_ARRAY is used as the bucket. The routines that operate on the EQ_ARRAY and equivalence bucket

(EQ_BUCKET) are described now:

- **add_eq_position ()**

  synopsis:
```
     void add_eq_position ( module, position, bucket )
          MODULE_PTR         module;
          int                position [];
          EQ_ARRAY_PTR       bucket;
```
  description:

  Adds <position> to <bucket>. Takes care that all already existing equivalent positions are copied to <bucket> too.

  algorithm:
```
  BEGIN
    IF position has already equivalent positions THEN
      add absolute coordinates of the equivalent set
        to the bucket;
    ELSE
      add absolute coordinates of 'position' to bucket;
    FI;
  END;
```
  subroutines:
```
  void   add_eq_element ();
```
- **update_eq_array ()**

  synopsis:
```
     void update_eq_array ( module, bucket )
          MODULE_PTR         module;
          EQ_ARRAY_PTR       bucket;
```
  description:

  Updates the EQ_ARRAY with the information in <bucket>. Takes care of the proper setting of equivalence indices and offsets in the grid structure.

  NOTE: The coordinates in the bucket are stored as absolute coordinates. In the EQ_ARRAY, they are stored as offsets to the first element. The coordinate of the first element in an EQ_ARRAY entry is always (0,0).

  algorithm:

```
BEGIN
  IF there are at least two elements in the bucket THEN
    IF there is a row in EQ_ARRAY equal to the bucket THEN
      set the indices and offsets in the grid;
    ELSE
      add a row to the EQ_ARRAY;
      set indices and offsets in the grid;
    FI;
  FI;
END;
```

subroutines:

```
int     check_array_row ();
void    add_array_row ();
void    set_eq_indices ();
```

7.2 Design rule compilation

7.2.1 Data structure

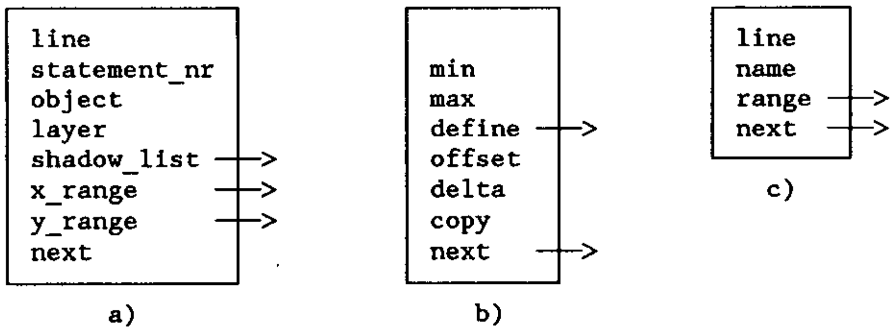


Figure 18. Structures: a) DRL_STATEMENT b) RANGE c) DEFINE.

The area description parts in the design rule statements offer the possibility to call defined ranges (section 6.2.2). The DEFINE and the RANGE are basic blocks in the design rule data structure. Besides the defines and the ranges, for each design rule statement a structure DRL_STATEMENT (figure 18) is present. All design rules are linked in a list called DRL_STATEMENTS. The range definitions are linked in a list called

DEFINES.

Design rule structure description:

- *'line'* is the number of the line, the statement was defined in the input. It is used to facilitate the communication with the user.
- *'object'* is the 'make-object' of the design rule statement.
- *'layer'* is the number of the layer the statement applies to.
- *'shadow_list'* is the list of shadowed objects (and their positions).
- *'x_range'* and *'y_range'* define the affected area. They are pointers to lists of structures of type RANGE. The area is formed by the cartesian product of the coordinates in the 'x_range' and the coordinates in the 'y_range'.
- *'next'* is a pointer to the next element of the list (DRL_STATEMENTS).

Range structure:

- *'min'* and *'max'* define the interval of the range.
- *'define'* points to a range definition. A range definition (structure DEFINE) is a range with a name assigned to it. The range is now defined by the range of the define and *'offset'*. To each element of the defined range, 'offset' is added. A range is either defined by 'min'-'max' or by 'offset'-'define'.
- *'copy'* and *'delta'* define how often and how spaced the range defined by 'min'-'max' or by 'offset'-'define' has to be repeated.
- *'next'* is a pointer to the next element in the list (of ranges).

Define structure description:

- *'name'* is the name of the range definition.
- *'range'* is a list of structures RANGE, associated with the defined name.
- *'next'* points to the next element of the list (DEFINES).

7.2.2 **Routines**

In the following considerations, a one layer case is assumed. It is rather easy to extend this contemplation to a multi layer case.

Denote G as the set off all gridpoints in the core.
The, in the input defined, set C_i of designrules takes the form of a set of **constraints** c applying to a rectangle denoted by

$$\square(c), \quad c \in C_i,\ \square(c) \subset G, \text{ where}$$
$$\square(c) = \square_x(c) * \square_y(c).$$

$\square_x(c)$ is the x-axis projection of $\square(c)$ and $\square_y(c)$ the y-axis projection.
A constraint is here regarded as a the set of shadow relations associated with the vertex in consideration.
It is allowed that associated rectangles overlap, i.e. for some $c_1 \in C_i$ and $c_2 \in C_i$ $(c_1 \neq c_2)$ it is allowed that

$$\square(c_1) \cap \square(c_2) \neq \emptyset.$$

Now, the object of the design rule compilation is to determine, for each $(x,y) \in G$, the set of constraints $D(x,y) \subset C_i$, that (x,y) has to obey. Note that D(x,y) is a constraint by itself. In fact, D(x,y) is the entry in the design rule table.
The most straight forward algorithm A1 to implement this compilation is:
Algorithm A1:

```
BEGIN
  FOR all $(x,y) \in G$ DO
    set := $\emptyset$;
    FOR all $c \in C_i$ DO
      IF $(x,y) \in \square(c)$ THEN
        set := set $\cup$ c;
      FI;
    ROF;
    D(x,y) := get_index(set);
  ROF;
END;
```

get_index(set) is a function that checks whether there already exists an

entry in the design rule table corresponding with 'set'. If so, the index of the entry is returned. If not, a new entry in the table is created and the index of that entry is returned.
The complexity of A1 is:

• $|G| \cdot |C_i|$	tests,
• $|G| \cdot |C_i|$	unions,
• $|G|$	D-assignments,
• $|G|$	design rule table checks.

Let us now suppose that we have a set C_e of **elementary** constraints, where the associated rectangles do not overlap, i.e.

$$\square(c_1) \cap \square(c_2) = \emptyset,$$
$$c_1 \in C_e,\ c_2 \in C_e,\ c_1 \neq c_2.$$

The set of elementary rectangles R_e is now defined by

$$R_e = \{\ \square(c)\ |\ c \in C_e\ \}$$

Note that in this case we can define an inverse function of $\square$. This inverse function: Γ, returns for each $r \in R_e$ the constraint(s) affecting r,

$$\Gamma(r) = c \text{ if and only if } \square(c) = r.$$

The algorithm A2 to determine D(x,y) for all $(x,y) \in G$ is now:
Algorithm A2:

```
BEGIN
  FOR all $c \in C_e$ DO
    i := get_index(c);
    FOR all $(x,y) \in \square(c)$ DO
      D(x,y) := i;
    ROF;
  ROF;
END;
```

The complexity of A2 is:

• $|G|$	D-assignments,
• $|C_e|$	design rule table checks.

It will be obvious that A2 is much less complex A1. So if we are able to transform the set C_i to an equivalent set C_e, the determination of D(x,y) would be more simple.
A transformation algorithm will now be discussed.
Define I_x as the smallest set of non-overlapping intervals along the x-axis, so that every x-projection $\square_x(c)$, $c \in C_i$ can be represented by the union of a number of elements of I_x, i.e.

$$\forall_{i_1 \in I_x,\, i_2 \in I_x} \quad (i_1 \cap i_2 = \emptyset) \qquad (1)$$

$$\forall_{c \in C_i} \quad (\square_x(c) \subset I_x) \qquad (2)$$

Likewise we define I_y:

$$\forall_{i_1 \in I_y,\, i_2 \in I_y} \quad (i_1 \cap i_2 = \emptyset) \qquad (3)$$

$$\forall_{c \in C_i} \quad (\square_y(c) \subset I_y) \qquad (4)$$

Define the constraint x-projection $P_x(i)$ on interval i as

$$P_x(i) = \{ c \in C_i \mid i \subset \square_x(c) \},\ i \in I_x. \qquad (5)$$

Likewise:

$$P_y(i) = \{ c \in C_i \mid i \subset \square_y(c) \},\ i \in I_y. \qquad (6)$$

Proposition:
The set of constraints C_e defined by

$$C_e = \{ C(i_x,i_y) \mid i_x \in I_x,\ i_y \in I_y,\ \square(C(i_x,i_y)) = i_x * i_y \} \qquad (7)$$

where,

$$C(i_x,i_y) = \{ c \in C_i \mid c \in (P_x(i_x) \cap P_y(i_y)) \} \qquad (8)$$

so,

$$\Gamma(i_x * i_y) = C(i_x,\ i_y)$$

is a set of elementary constraints equivalent to C_i. The equivalence holds that the result of algorithm A1 and the result of algorithm A2 are the same.

Proof:
From (1) yields that intervals do not overlap, so $i_{x_1} * i_{y_1}$ ($i_{x_1} \in I_x$, $i_{y_1} \in I_y$) does not overlap any other rectangle $i_{x_2} * i_{y_2}$ ($i_{x_2} \neq i_{x_1}$ or $i_{y_2} \neq i_{y_1}$). This means that the rectangles are elementary, so C_e is too.
The other part of the proof (equivalency) is divided into two steps:

a. If $(x,y) \in \square(c)$, $c \in C_i$ then according to C_e, c applies to (x,y) too.
b. If $(x,y) \notin \square(c)$, $c \in C_i$ then according to C_e, c does not apply to (x,y) too.

a. Let $c^* \in C_i$, $x^* \in \square_x(c^*)$ and $y^* \in \square_y(c^*)$. We proof that c^* applies to (x,y) according to C_e too.
From (2) yields:
$$\exists_{i_x^* \in I_x} \; (x^* \in i_x^*),\ i_x^* \subset \square_x(c^*) \;) \qquad (9)$$
From (5):
$$c^* \in P_x(i_x^*) \qquad (10)$$
Likewise:
$$c^* \in P_y(i_y^*) \qquad (11)$$
From (10), (11), (8) and (9):
$$c^* \in C(i_x^*, i_y^*) \qquad (12)$$
$$(x,y) \in i_x^* * i_y^* \qquad (13)$$
From (12) and (13) yields that c^* applies to (x,y) according to C_e.
b. Let $c^* \in C_i$, $x^* \notin \square_x(c^*)$. We proof that c^* is not an element of D(x,y) according to C_e too. If x^* is not an element of one of the intervals in I_x, there will no constraints be defined for x^* according to C_e ((7) and (8)).
If x^* is an element of i_x^* ($i_x^* \cap \square_x(c^*) = \emptyset$), then:
From (5):
$$c^* \notin P_x(i_x^*)$$
and from (8):
$$c^* \notin C(i_x^*,\ i_y),\quad i_y \in I_y.$$

so:

$$c^* \notin \Gamma(i_x^* * i_y)$$

This means that c* does not apply to (x^*,y) for some y. The same holds for $y^* \in i_y^*$ $(i_y^* \cap \square_y(c^*) = \emptyset)$.

The algorithm A2 now looks like:

Algorithm A2:

```
BEGIN
  { Determine $I_x$, $I_y$, $P_x(i_x)$, $P_y(i_y)$: }
  $I_x$ := x-axis;
  $I_y$ := y-axis;
  FOR all $c \in C_i$ DO
    { x-direction: }
    FOR all $i \in I_x$ DO
      IF $\square_x(c) \cap i \neq \emptyset$ THEN
        $I_x := I_x \setminus i$;
        split(i,$\square_x(c)$);      (*)
        $P_x(i_1) := P_x(i)$;
        $P_x(i_2) := P_x(i) \cup c$;
        $P_x(i_3) := P_x(i)$;
        $I_x := I_x \cup i_1 \cup i_2 \cup i_3$;
      FI;
    ROF;
    { same for y-direction: }
              .
              .
  ROF;
  { Determine $C(i_x,i_y)$: }
  FOR all $i_y \in I_y$ DO
    FOR all $i_y \in I_y$ DO
      i := get_index ( $P_x(i_x) \cap P_y(i_y)$ );
      FOR all $(x,y) \in i_x * i_y$ DO
        D(x,y) := i;
      ROF;
      ROF;
    ROF;
  END;
```

(*): split(i,$\square_x$(c)) needs some explanation. The interval i is divided into three intervals: i_1, i_2 and i_3 according to figure 19.

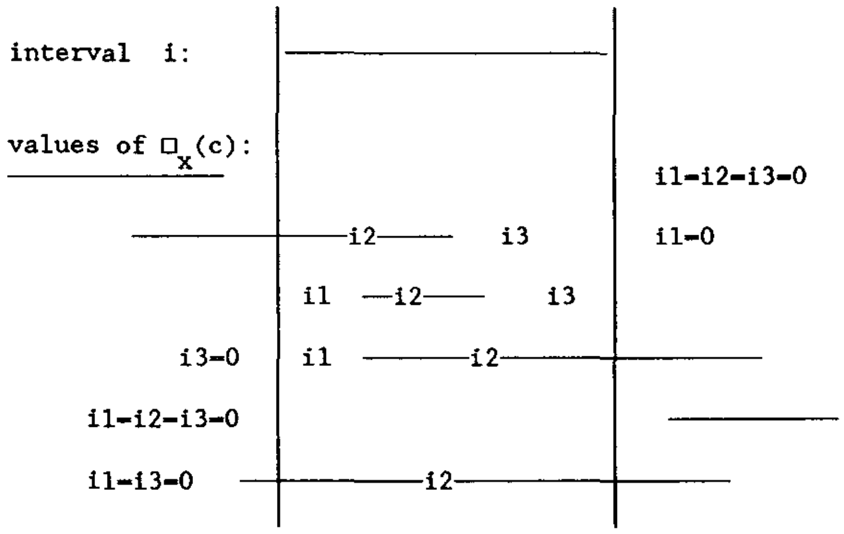


Figure 19. Interval splitting.

Figure 20 visualises the result after the determination of I_x, I_y, $P_x(i_x)$ and $P_y(i_y)$. To state the complexity of this transformation algorithm let's take a closer look at it (with implementation in mind). The determination of I_x starts with one interval (the complete axis) and ends with $|I_x|$ intervals. The mean value of the number of intervals in I_x is $\frac{1}{2}\cdot(|I_x|+1) \simeq \frac{1}{2}\cdot|I_x|$. So, the number of splittings along the x-axis is $\pm$ $\frac{1}{2}\cdot|I_x|\cdot|C_i|$. Likewise the number of splittings along the y-axis is $\pm$ $\frac{1}{2}\cdot|I_y|\cdot|C_i|$. The total number of splittings is $\pm$ $\frac{1}{2}\cdot(|I_x|+|I_y|)\cdot|C_i|$. In the worst case: $|I_x|+|I_y| = |C_e|+1 \simeq |C_e|$ $(|C_e| = |I_x|\cdot|I_y|)$. So the 'maximum' number of splittings is: $\frac{1}{2}\cdot|C_e|\cdot|C_i|$. Per splitting at most two interval tests and two unions have to be executed, so the determination of I_x, I_y, $P_x(i_x)$ and $P_y(i_y)$ requires:

- $|C_e|\cdot|C_i|$ interval tests
- $|C_e|\cdot|C_i|$ unions

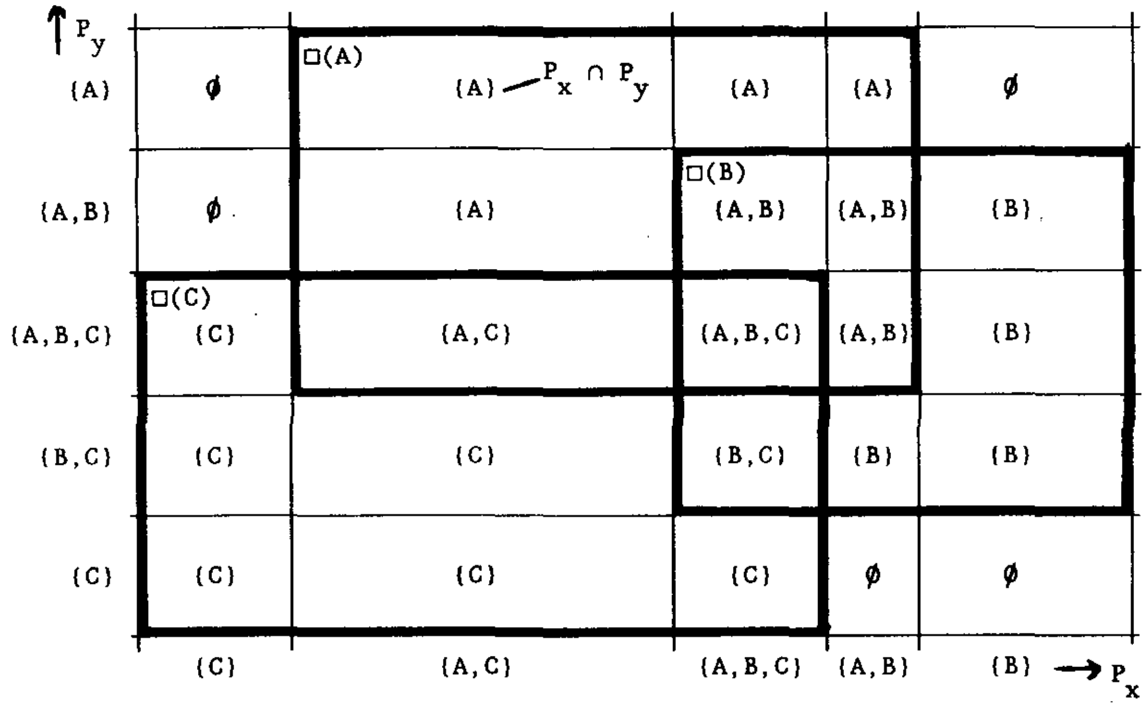


Figure 20. Result of first part of algorithm A2.

The intersection of $P_x(i_x)$ and $P_y(i_y)$, $i_x \in I_x$, $i_y \in I_y$ at most requires $|P_x(i_x)| + |P_y(i_y)|$ comparisons (P_x and P_y are sorted). Because $P_x(i_x) \subset C_i$, $P_y(i_y) \subset C_i$ yields: $|P_x(i_x)| + |P_y(i_y)| \le 2 \cdot |C_i|$. So the determination of $C(i_x,i_y)$ for all $i_x \in I_x$ and $i_y \in I_y$ takes $2 \cdot |C_e| \cdot |C_i|$ comparisons.

An interval test takes two comparisons. Now if we express the complexity of A1 and A2 in the 'primitives': number of tests, unions, D-assignments and design rule table checks (which is the most expensive one) we obtain:

item	algorithm A1	algorithm A2
comparisons	$4 \cdot \lvert G\rvert \cdot \lvert C_i\rvert$	$4 \cdot \lvert C_e\rvert \cdot \lvert C_i\rvert$
unions	$\lvert G\rvert \cdot \lvert C_i\rvert$	$\lvert C_e\rvert \cdot \lvert C_i\rvert$
drl table checks	$\lvert G\rvert$	$\lvert C_e\rvert$
D-assignments	$\lvert G\rvert$	$\lvert G\rvert$

$|C_e|$ is usually a lot smaller than $|G|$ (in the TAL004 gate array: $|G|$ = 81,918 , $|C_e|$ = 881). Therefor we may conclude that algorithm A2 will be a lot faster than algorithm A1.

7.3 Cost function compilation

During the cost function compilation the same algorithms as during design rule compilation are used. The only difference is that, in stead of shadow lists, cost values are computed. This means that cost table checks are less complex than design rule table checks (values in stead of lists should be compared).

7.3.1 Data structure

Here again the DEFINE and RANGE are basic parts of the data structure. Besides these, the STRUCTURE COST_STATEMENT is defined as in figure .

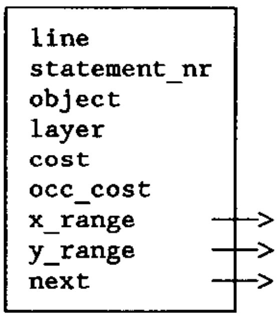


Figure 21. Structure COST_STATEMENT.

The fields in a cost statement are self explanatory (see section 4.5 and section 7.2.1).

7.3.2 **Routines**

As mentioned, the same algorithm as in the design rule compilation part is used (section 7.2.2). During cost table checks, the union of shadowed edges in the design rule table check should be replaced by summing corresponding costs.

8. DIAGNOSTICS AND PERFORMANCE

Syntax errors:
The parser is generated by 'yacc'. This implies that a syntax error is a fatal error and no recovery can be performed.
Semantics errors:
Semantics errors are caught as much as possible and reported as early as possible. Errors detectable during the scan through the input are reported within the input text at the corresponding line. These kinds of errors are declarative errors and boundary exceeding errors.
Other error messages are suspended until the end of the compilation, where the message inclusive the line number in the input text (if determinable) are reported. Error messages are sorted according to their line numbers. If no line number can be determined, the error message is generated in such a way that the error is locatable in an other way (e.g. the position in the grid or the module the error occurred in).
A special effort is made to suppress multiple reporting of declarative errors. Also a limit for the total number of errors is defined (currently set to 30).
The messages themself seem to be self explanatory.

Performance:

gate array	item	time (s)	space (kbytes)
AMI	core	249	574
	macros	76	51
TAL004	core	104	409
	macros	233	288

Figure 22. Performance of the program.

The criterions considered are:

- Response time (core compilation, macro compilation).
- Required storage space (core grid, macro library).

These entities are determined for two gate array types:

- AMI CMOS gate array (1000 gate equivalents, 1 routing layer).
- TAL004 low power shottky gate array (500 gate equivalents, 2 routing layers).

The results are stipulated in figure 22.

9. CONCLUSIONS AND ACKNOWLEDGEMENT

• In GADL it is possible to describe gate array core and macro stamp features in a compact way. The language offers the possibility to define the geometrical appearance, the design rules and the Leerout cost function.
The gridding of gate arrays seems to be very practical process to describe the features and the grid can serve as a frame for the routing data structure.
The concept of shadowing to model the gate array numeric design rules seems to cover all present-day known constraints.
Altough the form of the Leerout cost function that can be entered by GADL is quite simple, it seems to meet most of the modeller's needs. The cost function can also be used to model structural design rules.

• The GADL compiler generates (given a gate array description in GADL) most of the data required by the other modules in the GAS gate array design system in an easy accessible way. The compiler seems to be user friendly concerning use, response time and error messages.

• To generate a mask tape (goal of the system), a layout item should (and will) be added to GADL.

• At this place I would like to thank professor J.A.G. Jess, my direct coach ir. A. Slenter and the other menbers of the design automation group for their coorporation and the pleasant time I had during my graduation period.

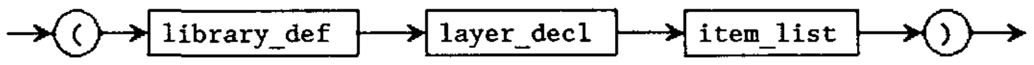
Appendix A: Vertex Data Structure

Data structure CVERTEX:

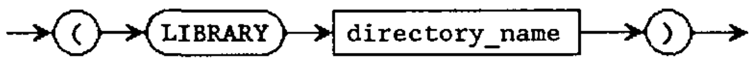
field:	*bits:*	*comment:*
east_signal_type	2	signal type of edge
north_signal_type	2	signal type of edge
down_signal_type	2	signal type of edge
equivalence_index	6	index in equivalence table
equivalence_offset	8	address in equivalent set cycle
design_rule_index	8	index in design rule table
cost_index	8	index in cost function table
reduced	1	flag (vertex is reduced)
traced	1	flag (vertex is traced)
macro_wire	1	flag (vertex element of macro wire)
total:	39 = 5 bytes/vertex	

Appendix B: Syntax of GADL

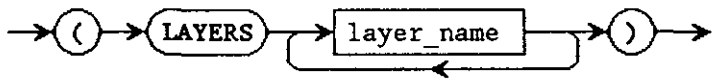
GADL_sentence :



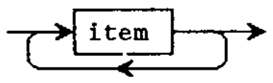
library_def :



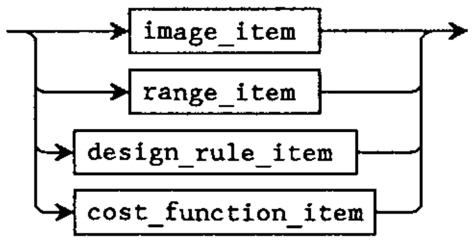
layer_decl :



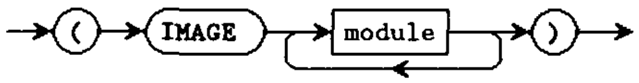
item_list :



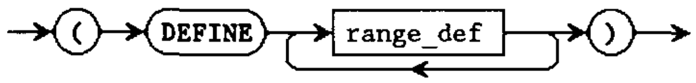
item :



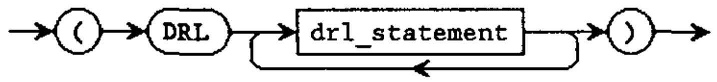
image_item :



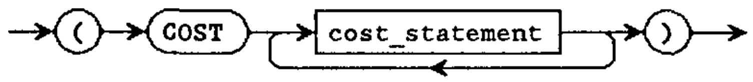
range_item :



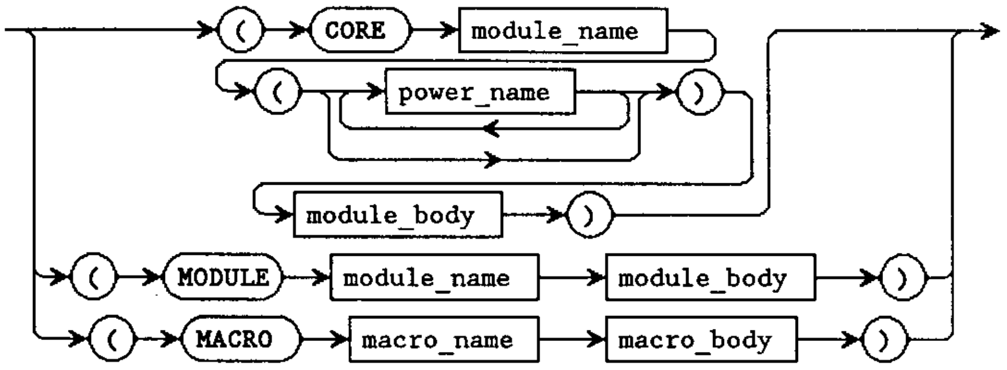
design_rule_item :



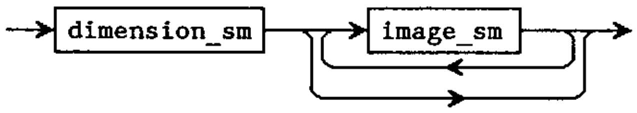
cost_function_item :



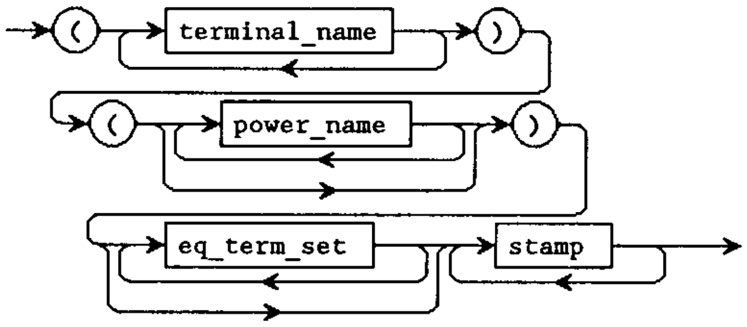
module :



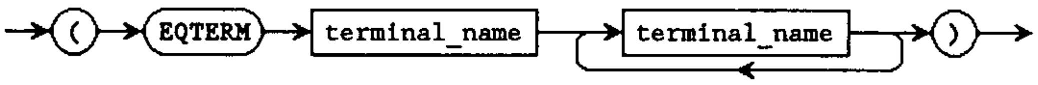
module_body :



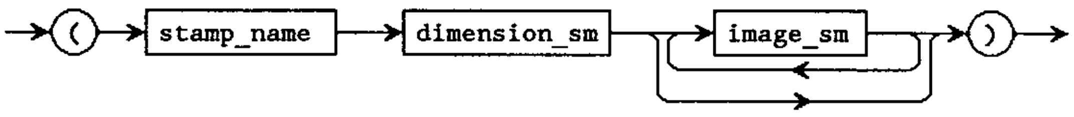
macro_body :



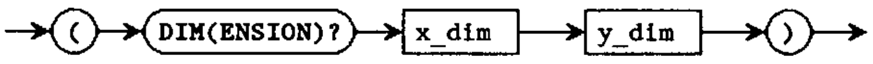
eq_term_set :



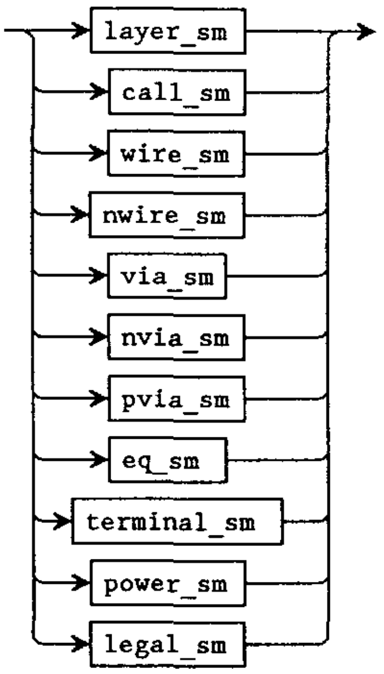
stamp :



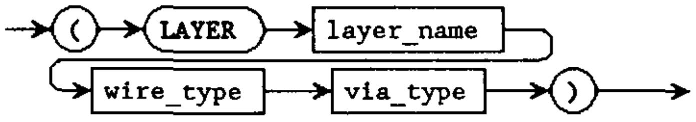
dimension_sm :



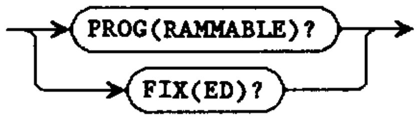
image_sm :



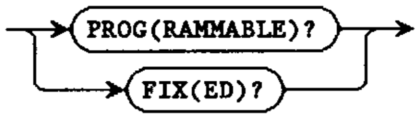
layer_sm :



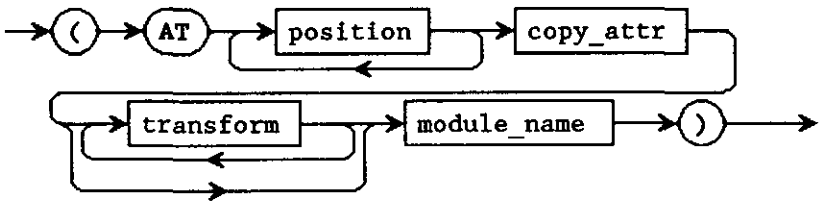
wire_type :



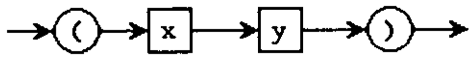
via_type :



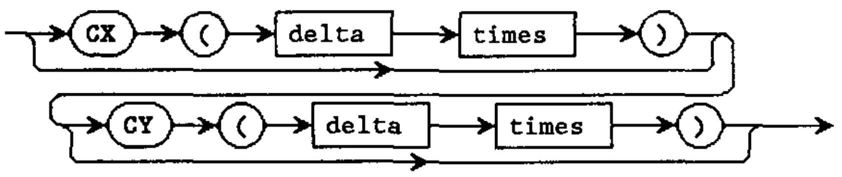
call_sm :



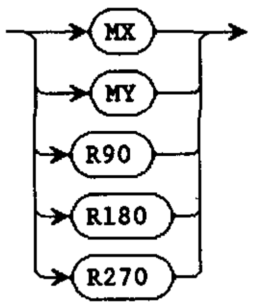
position :



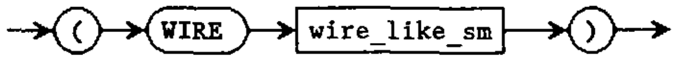
copy_attr :



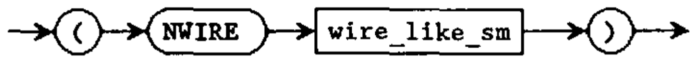
transform :



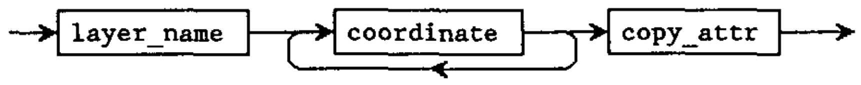
wire_sm :



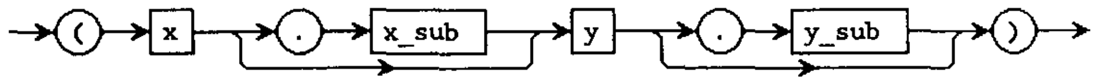
nwire_sm :



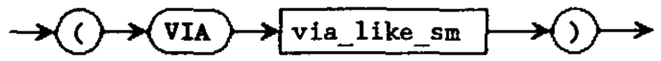
wire_like_sm :



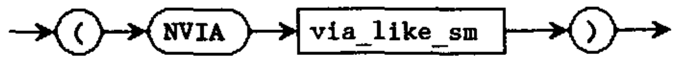
coordinate :



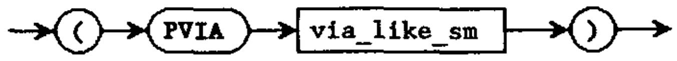
via_sm :



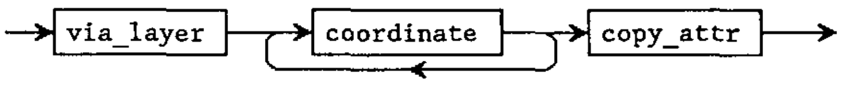
nvia_sm :



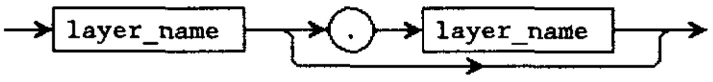
pvia_sm :



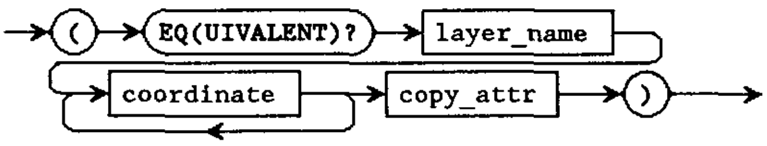
via_like_sm :



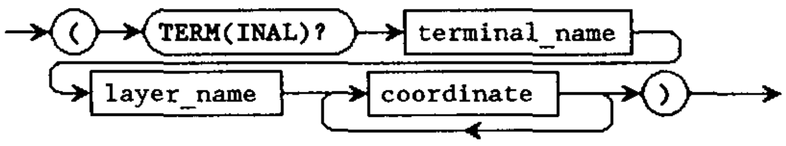
via_layer :



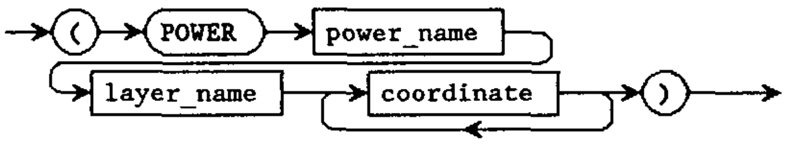
eq_sm :



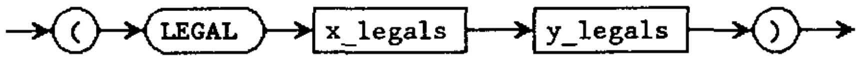
terminal_sm :



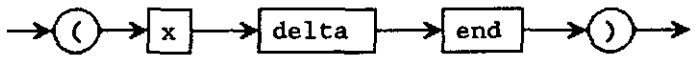
power_sm :



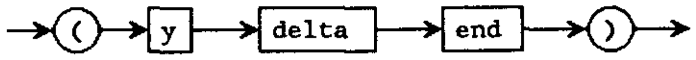
legal_sm :



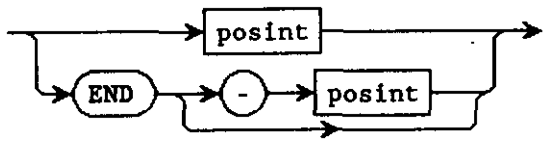
x_legals :



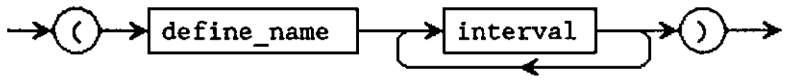
y_legals :



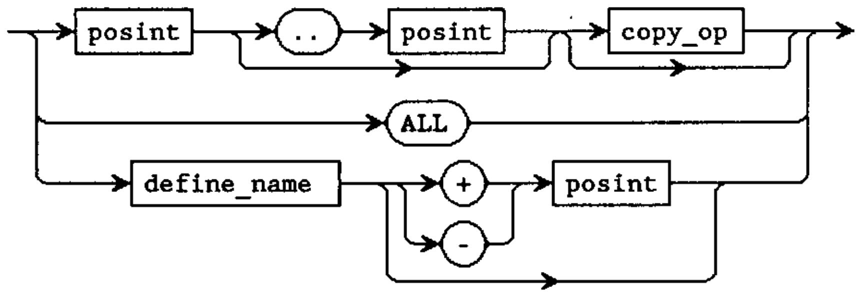
end :



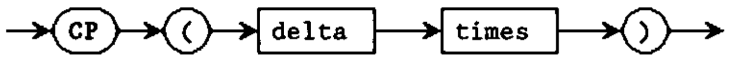
range_def :



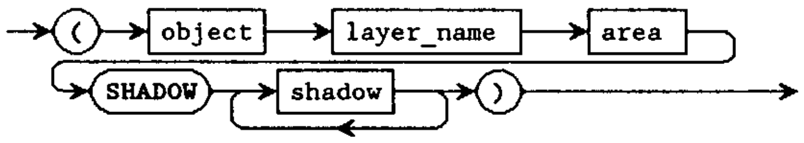
interval :



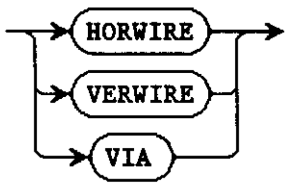
copy_op :



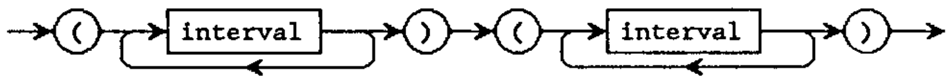
drl_statement :



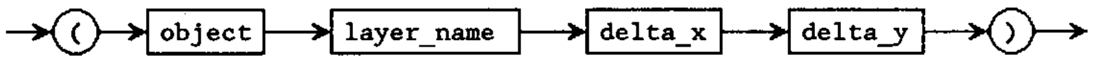
object :



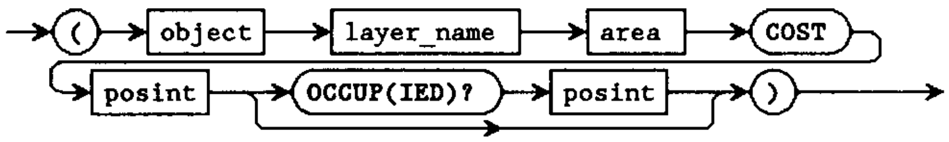
area :



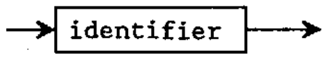
shadow :



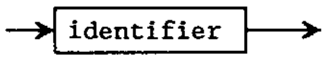
cost_statement :



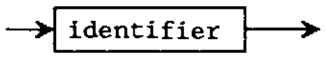
directory_name :



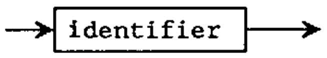
layer_name :



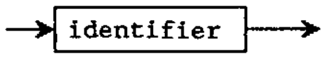
terminal_name :



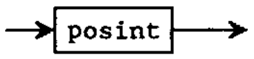
power_name :



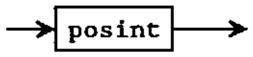
define_name :



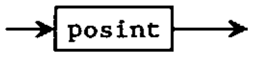
x_dim :



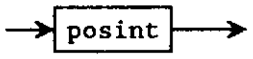
y_dim :



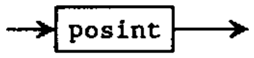
line :



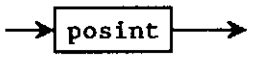
distance :



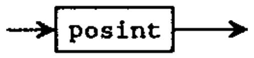
delta :



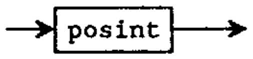
times :



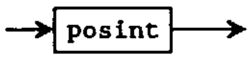
x :



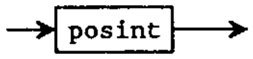
y :



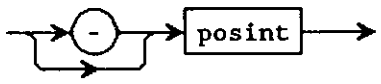
x_sub :



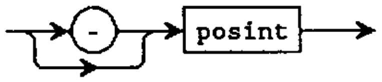
y_sub :



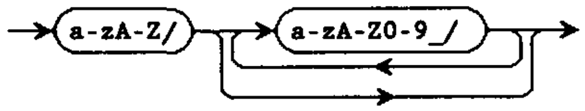
delta_x :



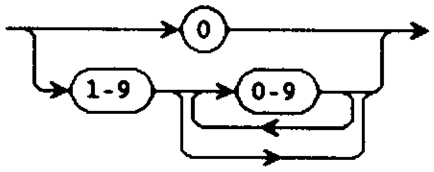
delta_y :



identifier :



posint :



Appendix C: Algorithm Description Meta Language

The language used in the descriptions of the routines is a high level 'meta'-language. The language only knows five statement kinds:

Action specifier:

```
<text>;
```

<text> specifies the action to be executed. An action is formulated in a natural language.

IF-statement:

```
IF <test> THEN
  <statement(s)>
FI;
```

<test> is an, in a natural language, formulated test. E.g. "all elements are processed" or "an element exists" are valid expressions for tests. If the test is obeyed, the action(s) defined by <statement(s)> is (are) executed.

IF-ELSE-statement:

```
IF <test> THEN
  <statement(s)_1>
ELSE
  <statement(s)_2>
FI;
```

If <test> is obeyed, <statement(s)_1> are executed. Else <statement(s)_2> are executed.

FOR-statement:

```
FOR <set_description> DO
  <statement(s)>
ROF;
```

<set_description> describes a set of elements (could be anything). E.g. "all modules" or "all elements not in A" are valid set descriptions. The statement executes <statement(s)> for each element in the set.

CASE-statement:

```
CASE <variable> OF
  <value_1> :
    <statement(s)_1>
  <value_2> :
    <statement(s)_2>
  .
  .
  .
  <value_n> :
    <statement(s)_n>
ESAC;
```

<variable> is a variable, e.g. "element" or "counter". If the variable has value <value_i>, <statement(s)_i> (0<i<n+1) are executed.

Comment is enclosed in "{" "}" and is not regarded as a part of the algorithm. An algorithm description is a list of action specifiers enclosed in BEGIN - END;.

Example:

```
{ Example of an algorithm description:             }
{ Marks the elements in set A that are processed.  }
{ and unmarks the elements that are not.           }

BEGIN
  FOR all elements of A DO
    IF element is processed THEN
      mark element;
    ELSE
      unmark element;
    FI;
  ROF;
END;
```

Appendix D: Ouput Formats

The output format of each file in CORE is separately discussed.

LAYERS:

LAYERS is a so called 'binary' file. Use fread() and fwrite() to access LAYERS.

format:

```
        LAYERS ::= <nr_of_layers> <x_size> <y_size> <vertices>
```

<nr_of_layers>, <x_size> and <y_size> are of type GRID_DIM (= int). The <vertices> are stored as records of type CVERTEX (character array of 5 elements). The layer-direction is the most significant one, the x-direction the least significant one.

EQ_TABLE:

EQ_TABLE is an ASCII file (text).

format:

```
        EQ_TABLE ::= <nr_of_entries> "\n" { <entry> }*.
        <entry> ::= <nr_of_elements> "\n" { <element> }*.
        <element> ::= <x_offset> <y_offset>.
```

The length of the list { <entry> }* should correspond with <nr_of_entries>. Likewise, the length of the list { <element> }* should correspond with <nr_of_elements>.

DR_TABLE:

DR_TABLE is an ASCII file.

format:

```
        DR_TABLE ::= <nr_of_entries> "\n" { <entry> }*.
        <entry> ::= <nr_of_down_shadows> { <shadow> }* "\n"
                    <nr_of_east_shadows> { <shadow> }* "\n"
                    <nr_of_north_shadows> { <shadow> }* "\n".
        <shadow> ::= "\t" <direction> <delta_layer> <delta_y> <delta_x>.
```

<direction> could either be DOWN_EDGE, EAST_EDGE or NORTH_EDGE.

COST_TABLE:

COST_TABLE is an ASCII file.

format:

```
        COST_TABLE ::= <nr_of_entries> "\n" { <entry> }*.
        <entry> ::= <north_cost> "\t" <east_cost> "\t" <down_cost> "\t"
                    <north_occ> "\t" <east_occ> "\t" <down_occ> "\n".
```

LAYER_INFO:

LAYER_INFO is used to store the layer names and information about their reducibility. The file is read during macro image compilation.

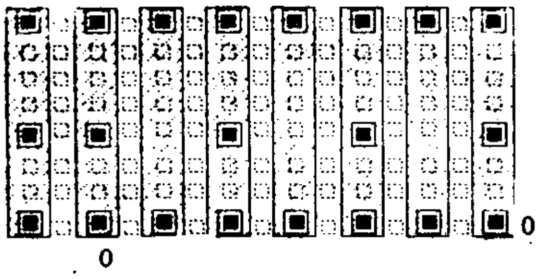
format:

```
        LAYER_INFO ::= { <layer> }*.
        <layer> ::= <layer_name> <reduce_info> "\n".
        <reduce_info> ::= "reducible" | "not_reducible".
```

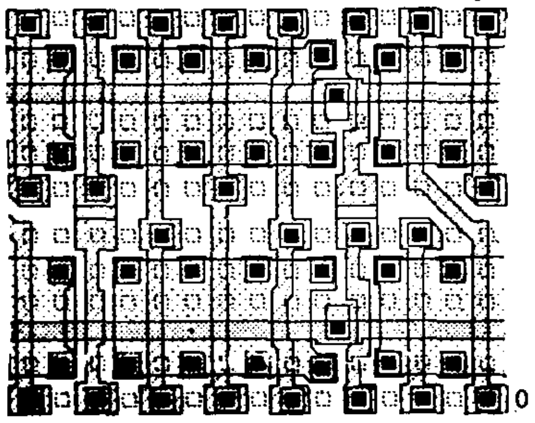
Appendix E: Examples

AMI CMOS GATE ARRAY CORE INPUT:

```
(
   ( LIBRARY  /users/paul_l/develop/AMI )

   ( LAYERS metal )

   ( IMAGE

      ( CORE ami_core (vdd vss)
         ( DIMENSION 25 33 )

         ( LAYER metal     PROG FIX )

         ( AT (0 0) CX(1 24) CY(2 16) channel_cell )
         ( AT (0 1) CX(1 24) CY(2 15) gate_cell )

         (* power lines: *)
         ( WIRE metal (0 1.2) (24.13 1.2) CY(2 15) )
         ( POWER vss metal (0 1.2) CY(2 15) )
         ( WIRE metal (0 1.9) (24.13 1.9) CY(2 15) )
         ( POWER vdd metal (0 1.9) CY(2 15) )
      )

      ( MODULE channel_cell
(*
```



```
*)
         ( DIMENSION  14  8 )

         ( VIA metal (0 0) (0 3) (0 7) CX(4 3) )
         ( VIA metal (2 0) (2 7) CX(4 2) )
         ( EQ  metal (0 0) (0 3) (0 7) CX(4 3) )
         ( EQ  metal (2 0) (2 7) CX(4 2) )
      )

      ( MODULE gate_cell
(*
```

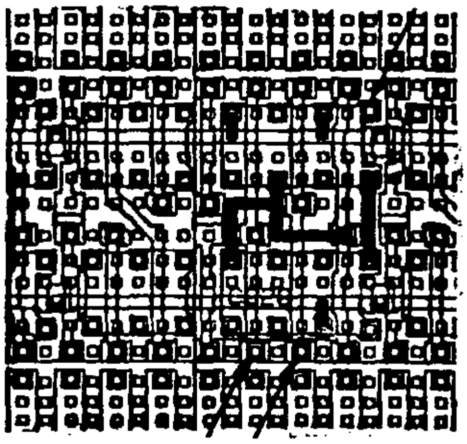


```
    *)

      ( DIMENSION  14  12 )

      (* poly gates and underpasses: *)
      ( VIA metal (0 0) (0 6) (0 11) CX(4 1) )
      ( VIA metal (2 0) (2 5) (2 11) CX(4 1) )
      ( VIA metal (8 0) (8 5) (8 11) )
      ( VIA metal (10 0) (10 5) )
      ( VIA metal (12 0) (10 11) )
      ( VIA metal (12 6) (12 11) )
      ( EQ  metal (0 0) (0 6) (0 11) CX(4 1) )
      ( EQ  metal (2 0) (2 5) (2 11) CX(4 1) )
      ( EQ  metal (8 0) (8 5) (8 11) )
      ( EQ  metal (10 0) (10 5) )
      ( EQ  metal (12 0) (10 11) )
      ( EQ  metal (12 6) (12 11) )

      (* diffusion vias: *)
      ( VIA metal (1 1) (1 4) CX(2 6) )
      ( VIA metal (1 7) (1 10) CX(2 6) )
      ( EQ  metal (1 1) (1 4) CX(2 6) )
      ( EQ  metal (1 7) (1 10) CX(2 6) )
   )
)

( DEFINE

   ( diffusion_x       1 CP(2 174) )
   ( diffusion_y       9  CP(20 15)
                       12 CP(20 15)
                       15 CP(20 15)
                       18 CP(20 15)
   )
   ( difis_y           8..9 CP(20 15)
                       11..12 CP(20 15)
                       14..15 CP(20 15)
                       17..18 CP(20 15)
   )
   ( channel_y         0..7 CP(20 16) )
```

```
        ( channel_bot_y   0..6 CP(20 16) )
        ( channel_via_y   0 CP(20 16)
                          3 CP(20 16)
                          7 CP(20 16)
        )
        ( poly_ft_x       0 CP(14 24)
                          8 CP(14 24)
        )
        ( poly_ft_y       8  CP(20 15)
                          19 CP(10 15)
        )
        ( poly_ft_is_x    0
                          13..14 CP(20 14)
                          7..8   CP(20 15)
        )
        ( poly_ft_is_y    7..8   CP(20 15)
                          18..19 CP(20 15)
        )
    )

    ( COST

        ( HORWIRE metal (ALL) (ALL)                  COST 1 )
        ( VERWIRE metal (ALL) (ALL)                  COST 1 )
        ( VIA     metal (ALL) (ALL)                  COST 1 )

        ( HORWIRE metal (ALL) (diffusion_y)          COST INFINITE OCCUP 1 )
        ( VERWIRE metal (diffusion_x) (difis_y)      COST INFINITE OCCUP 1 )

        ( VERWIRE metal (ALL) (channel_bot_y)        COST 50 )
        ( HORWIRE metal (ALL) (channel_via_y)        COST 50 )
        ( HORWIRE metal (ALL) (channel_y)            COST 10 )
        ( VERWIRE metal (ALL) (channel_bot_y)        COST 10 )
        ( HORWIRE metal (poly_ft_is_x) (poly_ft_y) COST 250 )
        ( VERWIRE metal (poly_ft_x) (poly_ft_is_y) COST 250 )
    )
)
```

AMI CMOS GATE ARRAY MACRO INPUT:

```
(
  ( LIBRARY /users/paul_l/develop/AMI )
  ( LAYERS metal )

  ( IMAGE

    ( MACRO and2 (in1 in2 out) (vdd vss)

      ( cc28
(*
```



```
*)
        ( DIMENSION 7 12 )

        ( TERMINAL in1 metal (1 0) )
        ( TERMINAL in2 metal (3 0) )
        ( TERMINAL out metal (6 10) )

        ( POWER vdd metal (0 9) )
        ( POWER vss metal (0 2) )

        ( LEGAL (1 14 END) (8 20 END) )

        (* wire definitions for macro and2 cc28 *)
        (* wire 1 *)
        ( WIRE metal (0 4) (0 6) (2 6) (2 5) (5 5) )
        ( WIRE metal (2 6) (2 7) )
        (* wire 2 *)
        ( WIRE metal (6 4) (6 7) )
        (* POWER connections *)
        ( WIRE metal (4 1) (4 2) )
        ( WIRE metal (0 10) (0 9) )
        ( WIRE metal (4 10) (4 9) )

      ) (* end of stamp cc28 *)

      ( cc29
  ( DIMENSION 7 12 )

  ( TERMINAL in1 metal (3 0) )
  ( TERMINAL in2 metal (5 0) )
  ( TERMINAL out metal (0 10) )

  ( POWER vdd metal (0 9) )
  ( POWER vss metal (0 2) )

  ( LEGAL (1 14 END) (8 20 END) )

  (* wire definitions for macro and2 cc29 *)
  (* wire 1 *)
  ( WIRE metal (1 5) (4 5) (4 6) (6 6) (6 4) )
  ( WIRE metal (4 6) (4 7) )
  (* wire 2 *)
  ( WIRE metal (0 4) (0 7) )

  (* POWER connections *)

  ( WIRE metal (2 1) (2 2) )
  ( WIRE metal (2 10) (2 9) )
  ( WIRE metal (6 10) (6 9) )

) (* end of stamp cc29 *)

( cc30

  ( DIMENSION 13 12 )

  ( TERMINAL in1 metal (1 0) )
  ( TERMINAL in2 metal (3 0) )
  ( TERMINAL out metal (6 10) )

  ( POWER vdd metal (0 9) )
  ( POWER vss metal (0 2) )

  ( LEGAL (9 14 END) (8 20 END) )

  (* wire definitions for macro and2 cc30 *)

  (* wire 1 *)
  ( WIRE metal (1 5) (1 8) (3 8) (3 6) )
  (* wire 2 *)
  ( WIRE metal (2 7) (2 5) (4 5) (4 3) (7 3) (7 5) )
  (* wire 3 *)
  ( WIRE metal (6 4) (6 7) )

  (* power connections *)

  ( WIRE metal (0 1) (0 2) )
```

```
      ( WIRE metal (8 1) (8 2) )
      ( WIRE metal (0 10) (0 9) )
      ( WIRE metal (4 10) (4 9) )
      ( WIRE metal (8 10) (8 9) )

    ) (* end of stamp cc30 *)

    ( cc31
      ( DIMENSION 13 12 )

      ( TERMINAL in1 metal (9 0) )
      ( TERMINAL in2 metal (11 0) )
      ( TERMINAL out metal (6 10) )

      ( POWER vdd metal (0 9) )
      ( POWER vss metal (0 2) )

      ( LEGAL (1 14 END) (8 20 END) )

      (* wire 1 *)
      ( WIRE metal (5 5) (5 3) (8 3) (8 6) (10 6) (10 7) )
      (* wire 2 *)
      ( WIRE metal (6 4) (6 7) )
      (* wire 3 *)
      ( WIRE metal (9 5) (11 5) (11 6) )

      (* power connections *)

      ( WIRE metal (4 1) (4 2) )
      ( WIRE metal (12 1) (12 2) )
      ( WIRE metal (4 10) (4 9) )
      ( WIRE metal (8 10) (8 9) )
      ( WIRE metal (12 10) (12 9) )

    ) (* end of stamp cc31 *)
  ) (* end of macro addl *)

  (* other macros *)
  (*       .      *)
  (*       .      *)
 )
)
```

REFERENCES

[1] Jongen, R.J.
GATE ARRAY PLACEMENT BY SIMULATED ANNEALING.
M.Sc. Thesis. Automatic System Design Group, Department of Electrical Engineering, Eindhoven University of Technology, 1984.

[2] Nuijten, P.A.C.M.
HIERARCHICAL WIRE ROUTING OF GATE ARRAYS.
M.Sc. Thesis. Automatic System Design Group, Department of Electrical Engineering, Eindhoven University of Technology, 1985.

[3] Slenter, A.G.J.
LOCAL ROUTING OF GATE ARRAYS.
M.Sc. Thesis. Automatic System Design Group, Department of Electrical Engineering, Eindhoven University of Technology, 1985.

[4] GATE ARRAYS: Design and applications. Ed. by J.W. Read.
London: Collins, 1985.

[5] Rubin, F.
THE LEE PATH CONNECTION ALGORITHM.
IEEE Trans. Comput., Vol. C-23(1974), p. 907-914.

[6] Woudenberg, J.G.A. van
A LIBRARY PROGRAM.
Report of practical work. Automatic System Design Group, Department of Electrical Engineering, Eindhoven University of Technology, 1986.

[7] Bu Jichun
GADSL: A gate array image description language. The implementation of the program DRLCMP describing the numerical gate array design rules.
M.Sc. Thesis. Circuit Theory Group (Prof. De Wilde), Department of Electrical Engineering, Delft University of Technology, 1985.

Eindhoven University of Technology Research Reports
Department of Electrical Engineering

ISSN 0167-9708
Coden: TEUEDE