The predictor-adaptor paradigm : automation of custom layout by flexible design

DOI:
10.6100/IR303176

Document status and date:
Published: 01/01/1989

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

The Predictor - Adaptor Paradigm

Automation of Custom Layout
by Flexible Design

Lukas P.P.P. van Ginneken

The Predictor - Adaptor Paradigm

Automation of Custom Layout
by Flexible Design

Lukas P.P.P. van Ginneken

Dit proefschrift is goedgekeurd door de promotoren

Prof.dr.-ing. J.A.G. Jess en Prof.dr.ir. R.H.J.M. Otten

Druk: Dissertatiedrukkerij Wibro, Helmond

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Ginneken, Lukas Paul Pieter Pepijn van

The predictor-adaptor paradigm: automation of
custom layout by flexible design / Lukas Paul Pieter
Pepijn van Ginneken. - [S.l. : s.n.]. - Fig., tab.
Proefschrift Eindhoven. - Met lit.opg., reg., index.
ISBN 90-9002703-3
SISO 663.43 UDC 621.382:681.3.06 NUGI 832
Trefw.: geïntegreerde schakelingen; computer aided
design / algorithmen.

The Predictor - Adaptor Paradigm

Automation of Custom Layout
by Flexible Design

PROEFSCHRIFT

Ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus, prof. ir. M. Tels, voor een commissie aangewezen door het college van dekanen in het openbaar te verdedigen op dinsdag 11 April 1989 om 16.00 uur

door

Lukas Paul Pieter Pepijn van Ginneken

geboren te Tilburg

Contents

Abstract

This book is about the automation of the design of masks for custom integrated circuits. The predictor-adaptor paradigm is the general pattern of collecting information followed by taking design decisions. This pattern is the result of top-down design or design by stepwise refinement. The book explains how this general paradigm can be applied to design problems of custom IC's. Several algorithms are given.

The blocks in a floor plan are designed according to parameters. First the freedom of these parameters must be determined by the predictor. Then the important design decisions can be taken. Taking the global design decisions has been called floor planning. The parameters are then determined and the blocks are adapted to their environment.

A new polynomial algorithm for the optimal slicing of point configurations is presented. Given the shape functions of the blocks and a point configuration, the algorithm finds the slicing structure that has the smallest total area. Although there are an exponential number of possible slicing structures, the number of possible slices is polynomial. This makes a dynamic programming approach feasible. The complexity of the algorithm is $O(n^6)$ where n is the number of blocks. The dimensions of the blocks must be expressed as small integers.

A heuristic for the shortest steiner tree problem is presented. The heuristic starts with a shortest spanning tree from which the topology is derived. The optimal positions of this topology can then be determined using a polynomial algorithm. This algorithm first determines the freedom to choose steiner points. Then the points are chosen in a top-down order.

An algorithm is presented is for the unconstrained two-dimensional folding problem. In this problem horizontal and vertical strips must be assigned to rows and columns. Strips in the same row or column should not overlap, while the connection pattern of the strips must be realized. The algorithm uses an elegant hierarchical divide and conquer technique to gradually refine the folding. Each refinement step consists of partitioning a group of strips. A cutting line separates the groups of strips in horizontal or vertical direction.

Two new layout styles based on this algorithm are presented. They are called 'nor matrix layout' and 'transistor matrix layout'. The nor matrix style is similar to the weinberger array. The nor matrix consists of multilevel nor gates. Several rows of pull-ups are used to allow for folding in both dimensions. The transistor matrix style can handle any circuit of transistors. The W/L ratio of each transistor can be specified. Although the implementation handles only nMOS circuits, an extension to CMOS is straightforward.

The folding algorithm allows for easy adaptation of the pin positions and the shape. This is important in combination with floor planning. It is shown that the area of the blocks can be accurately predicted and the aspect ratio can be accurately controlled. The orientation of the cutting line influences the aspect ratio. The new layout styles give much smaller layouts than conventional layout styles.

Using the flexibility of the sub-designs design problems can be solved more easily. Specialized sub-designs allow the design to be more critical.

Samenvatting

Dit proefschrift gaat over het ontwerpen van geïntegreerde schakelingen, ook wel *chips*. Het einddoel van het ontwerpen bestaat uit een aantal maskers, die met fotografische en scheikundige processen worden afgebeeld op het silicium. Omdat deze maskers erg ingewikkeld kunnen zijn willen we het ontwerpen hiervan zoveel mogelijk automatiseren. Dit spaart tijd en verminderd het aantal fouten. Ook moeten we de ontwerpmethode zeer precies beschrijven om hem door een computer te kunnen laten uitvoeren. Een methode die geschikt is voor een computer wordt een algoritme genoemd.

Het ontwerpen is vooral moeilijk omdat de schakelingen erg groot zijn. Daarom wordt het ontwerp gesplitst in een aantal deelontwerpen. Ieder van die deelontwerpen wordt ook weer gesplitst tot de delen wel hanteerbaar zijn.

Tijdens het ontwerpen van een chip moet er een groot aantal beslissingen worden genomen. Sommige beslissingen hebben een grote invloed op het resultaat, andere een kleinere. Beslissen welke schakelingen links op de chip worden geplaatst en welke rechts heeft veel meer invloed dan het trekken van een enkel draadje. Omdat de grote beslissingen veel meer invloed hebben willen we die het eerst nemen. De kleinere beslissingen kunnen we dan aanpassen aan de grotere.

De titel slaat op het algemene patroon dat we gebruiken in onze ontwerpmethoden. Dat patroon bestaat uit het voorspellen van de eigenschappen van nog niet ontworpen deelontwerpen, en dan het aanpassen van die deelontwerpen aan elkaar en aan de eerdere

beslissingen. In dit proefschrift wordt dit patroon toegepast op een aantal problemen die we tegenkomen bij het ontwerpen van chips.

De ruimte op de chip wordt verdeeld in rechthoekige stukken, de blokken. Een deelontwerp kan bestaan uit een blok, maar afhankelijk van het probleem ook uit een groepje blokken of een stuk van een blok. Het kan ook een stuk van een draadje zijn.

Voor een aantal problemen worden in dit proefschrift nieuwe methoden beschreven. Dit zijn het verdelen van de ruimte in blokken, het trekken van draadjes tussen de blokken en het ontwerpen van de blokken zelf.

Het verdelen in blokken wordt behandeld in hoofdstuk 3. Ieder blok heeft een aantal mogelijke vormen, die worden weergegeven door een vorm functie. Eerst krijgen alle blokken een punt op de chip. Deze puntenwolk wordt daarna doorsneden met rechte lijnen, de *slicing* lijnen. Het nieuwe algoritme kiest deze lijnen zo dat de chip zo klein mogelijk wordt. Het vindt altijd de beste oplossing. Het is ook redelijk snel: de tijd die nodig is is kleiner dan $a.n^6+b$, waarbij n het aantal blokken is, en a en b constanten zijn.

Het trekken van de draadjes wordt behandeld in hoofdstuk 4. Dit ontwerpprobleem is bekend als het steiner probleem in grafen. Het algoritme dat wordt gegeven vindt niet altijd de beste oplossing, maar het is wel snel.

De blokken zelf kunnen worden ontworpen met een methode die 'vouwen' wordt genoemd. Een blok is bijvoorbeeld opgebouwd uit draadjes en transistoren. Voor de draadjes worden lange dunne verticale strips gebruikt. De transistoren zijn ook lang en dun, en lopen horizontaal. Het vouwalgoritme kent draadjes aan kolommen toe, en transistoren aan rijen. Het algoritme probeert zoveel mogelijk draadjes in dezelfde kolom of dezelfde rij te stoppen en ook zoveel mogelijk transistoren in dezelfde rij. Natuurlijk mogen ze niet overlappen en moeten de juiste aansluitingen worden gemaakt. Door meer in de ene of in de andere richting te vouwen kunnen we de vorm van het blok aanpassen aan zijn buren.

Door het gebruiken van de flexibiliteit van de deelontwerpen kunnen we veel ontwerpproblemen gemakkelijker oplossen. Door deelontwerpen aan te passen wordt het hele ontwerp beter.

Preface

The first time I met Ralph Otten was September 1982 when I was doing an assignment for him as an undergraduate student at Eindhoven University. The assignment was to make a channel router that used simulated annealing. These two subjects, simulated annealing and layout design, have dominated the research we did in the following years. Later we used simulated annealing for floor planning, but after that the two subjects were investigated independently. Both have lead to a number of publications over the years. Our simulated annealing research has lead to a book called 'The Annealing Algorithm' [53] that is to appear shortly.

This thesis was the result of the work I did on layout design. It is about top-down layout design by stepwise refinement. The predictor-adaptor paradigm refers to the general pattern of first collecting information and then taking decisions. The predicted design freedom is used to adapt sub-designs to the global specifications.

Of course, many researchers have pointed out the advantages of a top-down approach to layout design. In this book, however, this principle has been applied more universally; it has been applied to many sub problems as well as to the global chip layout problem. The work presented here is an application and extension of the ideas of Ralph Otten on automatic floor planning to all aspects of layout design. He explained his ideas in several papers, among others the paper called 'Stepwise Layout Refinement' [79]. In this book these ideas were also applied to several problems that arise in the design of the sub circuits of a floor plan.

The major contribution of this book is the consistent application of stepwise refinement to many aspects of the automatic layout design of integrated circuits. It features a new algorithm for the Steiner tree problem in graphs and a new algorithm for optimal slicing of point configurations. The hierarchical folding algorithm was due to Jos van Eijndhoven. Using this algorithm some new layout styles have been developed. The transistor matrix style applies principles of stepwise refinement to the design of the smallest possible circuits: the transistors.

The book consists of eight chapters, which each, except for the first and last one, describe an application of stepwise refinement. Since the reader may be interested in only one of the algorithms, each chapter is more or less independent. As a whole, the book forms a consistent application of stepwise refinement to layout design. The book describes solutions to most problems arising in the design of custom layouts for integrated circuits.

In chapters 2-4 the conventional method is described in which the circuits occupy two dimensional rectangular areas. Chapters 5-7 assume unidimensional circuits, in an entirely new approach to matrix style layout.

The first chapter looks at layout design from an abstract point of view. This chapter explains the philosophy of this book in terms of adaptive design using flexible parameterized circuit generators. This is the background against which the rest of the chapters should be seen.

The second chapter describes the usual floor planning - circuit design - routing approach to building block layout. It serves as the framework into which the algorithms in the next two chapters fit.

In chapter 3 the aspect ratio trade-off of the circuits in a floor plan is addressed. An algorithm for slicing of point configurations is presented. This algorithm finds the slicing structure with the smallest possible area in polynomial time.

Chapter 4 describes a heuristic for the steiner tree problem. In this heuristic the steiner tree is viewed as a hierarchy tree. In this tree the positions of the steiner points can be chosen optimally and polynomially in a top-down sequence.

The next three chapters, 5-7 describe a less conventional top-down approach to layout design. In this approach the circuits are not blocks

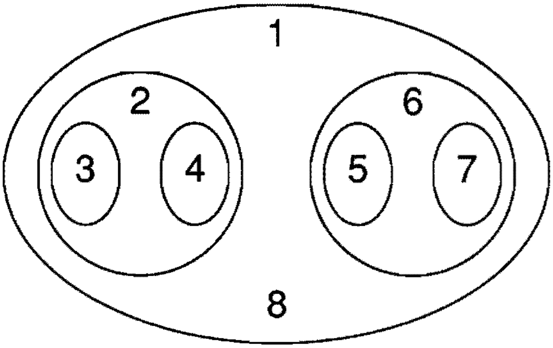


Figure 3.1. The organization of the chapters in this book. Chapters 2 and 6 describe two different approaches to structured layout by stepwise refinement. Chapters 3, 4, 5 and 7 describe specific sub problems. Chapter 1 gives an introduction and 8 gives a review of the methods used.

but unidimensional transistors or gates. The layout problem is formulated as an unconstrained folding problem. In chapter 5 the folding problem is solved by a stepwise refinement heuristic.

Chapter 6 describes two new layout styles that use this folding algorithm. One is a variation on the weinberger array style, the other is called 'transistor matrix layout'. In transistor matrix layout the adaptable circuits are the smallest possible circuits: single transistors.

Chapter 7 describes the shape adaptation of these transistor matrices. Also an prediction is made of their shape function for the floor planner. Some comparisons with conventional layout styles show the very small areas that these new circuit styles need.

The last chapter, chapter 8, a review is given of the algorithms and how they are consistent with the principles. We also look the requirements for a complete layout design package. Some possible areas of further research are indicated.

Finally, I would like to mention those who contributed to this work. I'm grateful to Jochen Jess and Bob Brayton for the opportunity to work in their research groups. My advisors, Jochen Jess and Ralph Otten did the proofreading of the thesis. Jos van Eijndhoven came up with the idea for the folding algorithm just when the need for such an algorithm arose. I would also like to mention the students who developed parts of the package: Paul van Teeffelen, Gert-Jan van Lieshout, Tony Brans, Jos Brouwers and Theo Deckers. My work was supported financially by the foundation FOM under project number EEL 36.0417 and IBM.

I'm especially indebted to Ralph Otten for his continuous support of my work. Our numerous discussions have been major source of inspiration to me. Without his efforts and his insight this book would not have been possible.

Lukas P.P.P. van Ginneken
Eindhoven, January 6, 1989

1. The predictor-adaptor paradigm

In this chapter we will introduce the predictor-adaptor paradigm as an approach to hierarchical design. First we will look at the complexity of design methods and at hierarchy as a means to deal with this complexity.

Then we will introduce the stepwise refinement as a strategy aimed at global optimality. Global optimality is facilitated by taking the first design decisions at a high level. Subsequently, the design is further developed by taking design decisions at the lower levels. Hence the names 'top-down design' or 'stepwise refinement'.

The design decisions must be based on some information. Since the sub-designs have not yet been designed, their properties must be predicted. The *predictor* derives the global design freedom from the properties of the sub-designs. The *adaptor* then adapts the sub-designs to the global design specifications.

The sub-designs are designed according to parameters that have been determined by the top level decisions and the global specifications. These parameters should preferably be optimized simultaneously.

Finally, an example algorithm will be presented that incorporates many principles of the predictor-adaptor paradigm in a simple and concrete manner.

1.1 Design theory of complex systems

When we study the process of design, we study the design methods. For developing a theory of design, we would like to describe these methods formally. A formally described method is an *algorithm*. When such an algorithm is formulated in an algorithmic language, we speak of a *program*.

In this book we will study formal design methods for integrated electronic circuits. These design methods formulated in an algorithmic language are software. In short, we will study software to design hardware. In this book the design problem will be limited to the design of the layout of an integrated circuit.

The practical limits to the design of hardware and software systems are set by the complexity of the systems, and our ability to deal with this complexity. Much of theory of the design of hardware and software systems is the theory of the design of complex systems.

The *time complexity* of an algorithm describes how the time needed to execute the algorithm grows asymptotically with the size of the problem. The size of a problem can be measured as the number of bits to describe a problem instance. Suppose we need n bits to describe the problem instance, and the time needed to execute the algorithm is at most $t(n)$ seconds. If there are two constants a and b such that $t(n) \leq a.f(n)+b$ for all n then the complexity of the algorithm is said to be $O(f(n))$. Mostly the simplest function which is a sharp bound for large n is taken for $f(n)$. If $f(n)$ can be expressed as n^k then the algorithm is called *polynomial*. If this is not possible, the algorithm is called *exponential* [26]. Many of the combinatorial problems posed in this book cannot be solved by a polynomial algorithm.

There is a large class of problems, called the *NP-complete* problems [26]. These problems are formulated as decision problems, for instance: is there a solution with a score better then a given k? It is generally believed that no algorithm exists that can solve these problems in polynomial time, but so far no one has been able to prove or disprove that. The related optimization problems (what is the best solution?) are in the class of *NP-hard* problems.

To reduce the complexity of the algorithms several methods have been devised. Perhaps the most general method to cope with any kind of complexity, is the use of hierarchy [61,54,83]. A *hierarchy* is a recursive tree structure of modules within modules: A *module* M is a set of modules $M=\{m_1,m_2,\cdots,m_{\#M}\}$. The modules m_i are the sub-modules of M. Each sub-module may be decomposed itself. Decomposed modules are called *compound modules*. *Primitive modules* are modules without sub-modules. M is the *super module* of m_i. There is exactly one module without a super module called the *root* module.

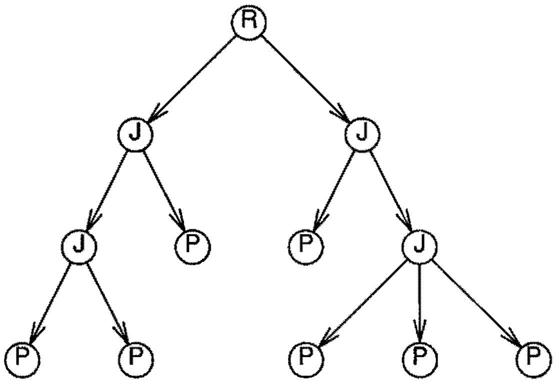


Figure 1.1. A hierarchy of modules. R = root module, J = compound module, P = primitive module.

The hierarchy can be represented as a rooted tree. The modules are represented by the nodes: the root represents the system, the leaves of the tree represent the primitive modules and the internal nodes represent the compound modules. The arcs point from each module to its sub-modules.

The complexity of the design problem can be reduced by representing the design as a hierarchy. The design problem is decomposed into sub designs, called modules. The root module represents the whole design. In stead of solving the entire design problem at once, the design problem is solved in steps. In each step a sub-problem is solved. In algorithms this principle is known as 'divide and conquer' [4].

An example of a hierarchy is the decomposition of an electronic circuit into sub-circuits. The circuits $C=\{c_1,c_2,\cdots\}$ are the modules of the hierarchy. The circuits are connected by nets. A *net list* is a bipartite graph $\mathcal{N}(C\cup\{C\},N,P)$, in which N is the set of *nets* and $P\subset(C\cup\{C\})\times N$ is the set of *pins*. The pins are the edges between the circuit and the nets.

The pins $P \cap \{C\} \times N$ are the pins of circuit C itself, that is, the pins to the outside. In this book we will assume that the design is given as a hierarchy of circuits to the layout algorithms. The primitive circuits are the circuit elements like transistors and resistors. Note that they are the simplest indivisible circuits since they cannot be represented by a net list.

Hierarchy can help to reduce the complexity, by restricting the view to the immediate sub-modules. When looking at a compound module, we are in fact looking at all the leaf nodes in the sub-tree. However, in stead of looking at the leaf nodes and their interactions directly, we look at a *model* of this sub-tree. We look at the compound module as a 'black box' of which we can see only its outside shape, but not its internal structure. Such a model is the *interface* of the module to its super module.

The use of hierarchy can have advantages other than the decrease in complexity by modeling compound modules. Some of the modules in the hierarchy may be identical. The design may be repeated, so the circuit needs to be designed only once. Sometimes this property is used to design an entire *library* of circuits, ready for use.

The design problems of the sub-modules become mutually independent when their inter-relations are explicitly known. The designs can therefore be done simultaneously: Several sub-modules may be designed in parallel. Finally, no single design method may be suitable for the entire design. A different design method can be used on different modules of the hierarchy.

There are two requirements for the hierarchy concept to work. First, the complexity is only reduced if the interface description of a compound module is simpler than a structural description of all the component modules together with their interrelations. Secondly, this interface description must be a sufficiently accurate model for the compound module.

Hierarchy is no panacea, nor is it always obvious how it should be used. Introducing hierarchy almost always means a longer algorithm. It is essential that the interfaces are chosen such that they form a good model, while they provide sufficient simplification. An oversimplification will lead to a 'short sighted' algorithm, while an exact and concise model may lead to a polynomial and optimal algorithm. The better models we have, the better is our 'understanding' of the problem.

1.2 Stepwise refinement

High level decisions are likely to have the most influence on the quality of the design. Therefore the high level decisions should be taken first, when the freedom to take decisions is still large. The first decisions can guide the lower level decisions towards global optimality. The design starts at the root node and progresses down the hierarchy. Hence the name *top-down design*. When the design starts with the leaf nodes, it is called *bottom-up design*.

Stepwise refinement has been widely advocated for the design of complex systems by human designers [47]. Its application to software development was first proposed by Niklaus Wirth in his paper 'Program development by stepwise refinement' [72]. Its application to automatic layout design was described in [64, 79].

Before a module can be designed, some information is needed to base design decisions on. This can be done by preliminary designs, or by predictions derived from key parameters. The *predictor* is an algorithm that makes a such predictions. The information of the predictions can be propagated up the design hierarchy. Of course the quality of the high level decisions depends on the accuracy of the predictions.

The second phase is the top-down *adaptor*. During this phase important design decisions are taken. The first decisions are taken on a high level of the hierarchy. Therefore they will have a large influence on the whole design. The design is then refined, and the sub-modules adapted to the earlier design decisions. The decisions taken in the sub-modules have a smaller influence, and are guided by the decisions taken earlier. This way the design strategy is targeted towards global optimality.

The design decisions taken at a high level must be represented by an intermediate representation of the design. This intermediate representation is a *plan*. Examples of such plans are a floor plan, a wire plan or a data flow graph. A floor plan of a chip for instance spatially orders the circuits on the chip. It relates the shape and positions of the circuits to each other.

Another approach to design which uses hierarchy is bottom-up design. In bottom-up design, the modules that are lowest in the hierarchy are designed first. These modules are then combined to create more complex modules, until the system design has been realized.

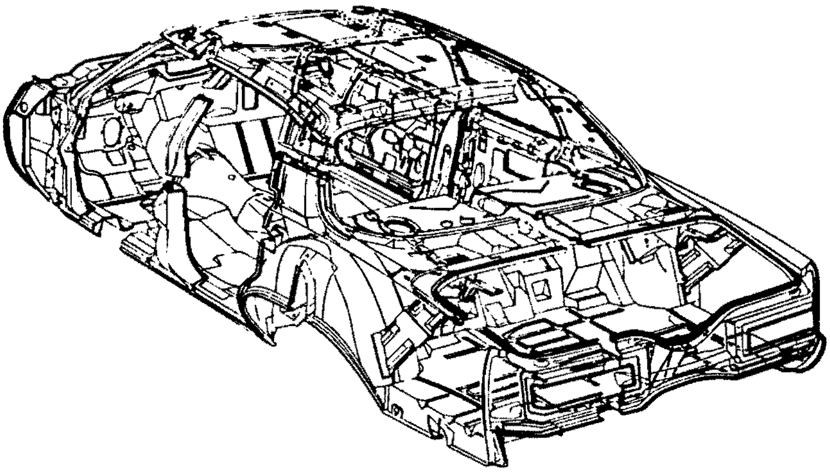


Figure 1.2. This car was designed top-down. The parts are especially designed for this particular car. This requires a lot of time and craftsmanship.

The most important advantage of bottom-up design is that the design of a module, ignoring environment information, is simpler. Only the function has to be specified. For an automatic design algorithm, this means a simpler interface. Also, the design is usually general enough to allow repeated usage. Furthermore, the sub-modules are exactly specified when designing a compound module. The design can be done using exact data, not just predictions.

The price to be paid however, is that the modules are unadapted. The more complex the modules are the more variation is possible. In the layout design of circuits, bottom-up design is often done up to a certain level, for instance the standard cell level. For each specific design top-down design is then used down to the standard cell level. This compromise is also known as a 'meet in the middle' design strategy.

Most design methods use such a compromise. The approach that is presented in this book however aims at a completely top-down design

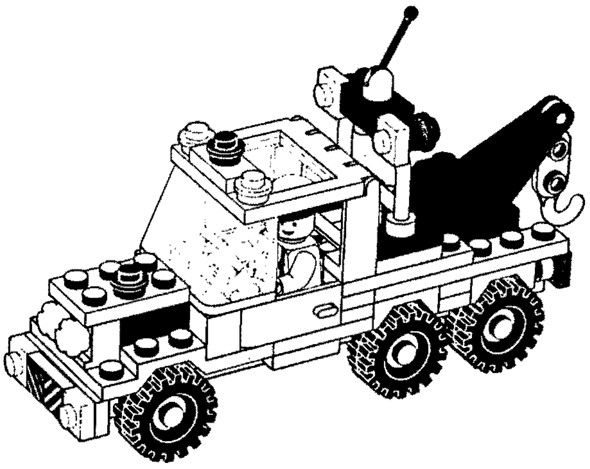


Figure 1.3. This car was designed bottom-up. The building blocks have been designed beforehand, and are not committed to a particular function. The quality of the car is poor, but the job can be done by a child.

flow down to the transistors. The transistors are the primitive circuits because they cannot be represented by a decomposition with a net list.

Top down design can also be looked at as postponing design decisions. It is important that sufficient information is available when taking a decision. Decisions reduce the freedom for further decisions. Taking decisions leads to a gradual stiffening of the design. Decisions should be based on reliable information, limit the freedom of further decisions as little as possible, and result in a maximal amount of new information.

In top-down design, design criteria can often be met by propagating the design criteria towards the sub-modules. The design criteria are merely translated to criteria for the sub-modules. In the sub modules, the problem of meeting these criteria must be solved. Of course, unless the sub-module is a primitive module, the design problem can be delayed further, and moved down the hierarchy again.

The decisions that are taken in a top-down design process are 'self fulfilling'. That is, for the modules that are finally used, the design that has been made with them is likely to be the best possible with these modules.

When the modules have not yet been designed, there is no good reason to assume rigid constraints on the freedom of the modules in advance. We may assume that the modules are completely flexible in all their parameters. Of course not every combination of parameters is possible: parameters can be traded against each other. But within certain bounds each parameter can take any value.

1.3 Design parameters

For a top-down design method, the modules must be adaptable to their environment. The design decisions taken at a high level are presented to the designers of the sub-modules as *parameters*. Such parameters are used in the hierarchy of circuits and in other hierarchies yet to be introduced.

In the design of the layout of a circuit, the design decisions are translated to parameters for the design of the sub circuits. The parameters form the specifications of the circuit; they form the interface through which the circuit generators receive their information. For an electronic circuit on a chip we can distinguish the following important classes of parameters:

1. Function
2. Testability
3. Delay
4. Power consumption
5. Technology
6. Design rules
7. Shape
8. Pin positions
9. Design effort

With so many parameters, it becomes infeasible to maintain a library. Some parameters, like function, would need a very large set. Even if the number of choices per parameter is small, the number of combinations may be enormous. An application specific design must be done for most circuits. Such a design for each circuit is too expensive to do by hand. However if this could be done by an automatic program, a different design for each circuit becomes feasible.

In stead of having a large number of different standard cells, the design software should have a small number of automatic *circuit generators*. The circuit generators should be able to generate circuits according to parametric specifications. Preferably, the circuit generator should take as many parameters into account as possible.

It seems obvious that the *function* of a circuit should be parameterized. The circuit should be able to do any function within the class of functions of the circuit generator. The PLA generator was the first general function generator that could generate any boolean function. Generators may be relatively uncommitted, and generate random logic or finite state machines. For certain classes such as multipliers, memories or register files dedicated circuit generators may be necessary. We will assume that the circuit is given as a hierarchy of circuits with net lists.

Testability can also be built into a circuit. For instance, depending on a parameter, level sensitive scan paths may be built in. Also self test may be built in or spare parts may be included, to replace faulty parts. Finally, extra pins may be added to make internal signals visible.

The *delay* of a circuit should be a parameter. Because timing constraints are often an important and rigid design parameter at the top level, the delay should be a parameter that is set, while the power is minimized. The delay can be shortened by increasing transistor widths, adding buffers, or it may be decreased by adding extra hardware, by duplicating part of the logic to reduce the logic depth, or by introducing parallel processing.

When the delay of the circuit is decreased the *power consumption* of the circuit is likely to increase. Figure 1.4 shows the relation between the delay and power parameters of an inverter. The power consumption also depends on the switching behavior of the circuit, and can therefore only be determined exactly by a realistic simulation.

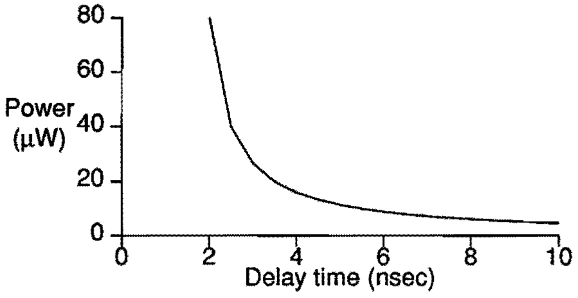


Figure 1.4. Static power dissipation of an nMOS inverter as a function of the rise time.

The power consumption, and its trade-off with speed depends on the *technology*. With technology we mean the kind of process: for instance bipolar, nMOS or CMOS. This is strongly related to the kind of gate circuits that can be used. In general, this will be a difficult parameter to take into account. Certainly on a lower level, the design method depends heavily on the technology.

The *design rules* are structural and numeric constraints on the geometry of the layout. As far as the design rules are numeric rather than structural, they are often easy to parameterize. The numeric design rules are taken into account during the mask generation. In this phase, a topological structure is translated to mask data. The numeric rules determine the minimum width and spacing of the mask elements. Structural rules are difficult to implement, since they often influence design strategy that can be used. Such rules can not easily be parameterized, they must often be built into the algorithms.

The *shape* of the circuit follows from its environment and the shapes of the circuits surrounding it. The shapes of the larger circuits will be restricted to rectangles, but the aspect ratio can be chosen freely. The use of rectangles with variable aspect ratio makes a dense packing of the circuits easy.

The *pin positions* adapt to the environment. The positions of the pins are determined by the neighbors of the circuit. Sometimes through the cell routing is desired. In this case several pins for the same net are necessary.

Finally, the *effort* needed to design the circuit, or the cost to design it could be taken into account. In general, a better design may be expected when the effort that went into the design was greater. For instance certain critical circuits may be redesigned by hand. Less critical circuits can be designed automatically. For an automatic design tool such a parameter may also be useful. For instance, a simulated annealing algorithm may invest more time to achieve a better solution for a critical circuit, while a less critical circuit may use a fast and non-equilibrium schedule.

1.4 Joint parameter optimization

The parameters of the modules must be optimized to match the environment of the modules. The requirements of the super module are translated to requirements of the sub-modules. The optimal parameters of the modules depend on each other. Therefore the parameters of the modules should be chosen simultaneously. Hence the name 'joint parameter optimization'.

Pin positions are an example of a parameter that should be optimized for all circuits simultaneously. When the floor plan is known, an approximate direction for the wire can be derived, and the pin position can be determined. However, when the positions of pins of some circuits are fixed, this optimization problem can become difficult. When two circuits are abutting, the pin position must be chosen for both circuits simultaneously. Preferably, the positions of the two pins should be facing each other, giving a minimum length connection.

Several parameters can be viewed as valuable resources. These parameters are related to the time, space and energy needed to perform the function and perhaps the effort to design the function. For each of these there is a price to be paid in terms of another parameter. These parameters can be traded against one another.

Well known is the trade-off between speed and power. More power can be used to make a gate or a macro faster (see figure 1.4). Such trade-off's can be represented by a *trade relation*. This is a relation between the resources that can be traded against each other. In a trade relation, a parameter cannot increase because another increases.

The parameters are related to each other, often by a partial order. The partial order relates the parameters of the sub-modules to the parameters of the super module. In timing such a partial order is a data flow graph. The data flow graph is a partial order of the computations that are performed by the sub-modules. Each sub-module needs a certain amount of time to perform its computation. The longest path determines the total time for the whole module.

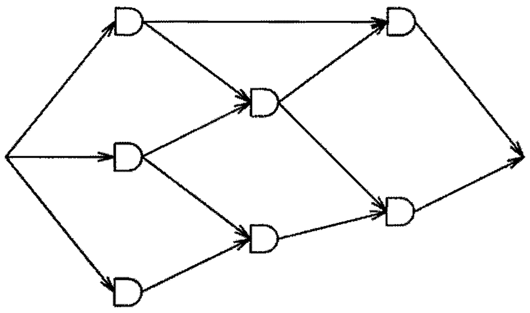


Figure 1.5. Data flow graph of a circuit. The longest path determines the delay of the circuit.

To do the optimization, the trade relations need to be known. Since the design of the sub-module is not yet known, a prediction of the trade relations has to be made. Such a prediction can be made by making a number of sample designs, and interpolating the trade relations between the known values of the parameters. Alternatively, information about old designs can be stored or a prediction can be derived from some characteristic parameters.

Delay can be traded against area. The trade-off between the area and the delay, by using more hardware, is the subject of high level functional synthesis [63]. Timing constraints set by the data flow graph relate the delays of the circuits to each other. Delay can be traded against hardware by introducing parallelism at a high level or by duplicating logic to reduce the length of the critical path.

In this book we will not address the optimization and design problems arising from such an approach to timing. The possibility of giving a macro cell generator parameters for speed, power and testability has not been investigated. We will only consider the function of the circuit and the layout parameters. We believe however, that the methods indicated in this book are well suited for introducing such parameters.

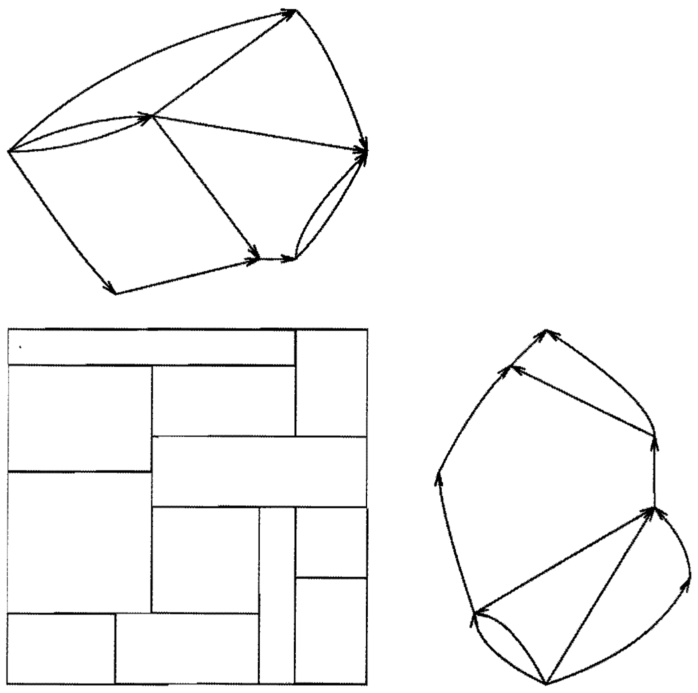


Figure 1.6. A rectangle dissection with its corresponding polar graphs.

The area of a chip is dissected into rectangular areas; a rectangular area is allocated for each circuit. A rectangle which is dissected by straight orthogonal lines is called a *rectangle dissection.* In a floor plan with rectangular circuits the width and the height of the circuits can be traded against each other. A partial order of the coordinates of the circuits is used to represent the relations between the width the different circuits. The graph representing this partial order is called the horizontal *polar graph* and denoted by $\mathcal{P}_h(J_v,C)$. A second, dual, (vertical) polar graph $\mathcal{P}_v(J_h,C)$. represents the relations between the heights. The topology of the rectangle dissection is determined by the polar graphs (see figure 1.6). The nodes J_v of the horizontal polar graph correspond to the vertical dissecting lines. The faces of the horizontal polar graph correspond to the horizontal dissecting lines. For the dual polar graph it is the other way around: the faces are the vertical lines and the nodes are the horizontal lines J_h. The edges of each polar graph represent the rectangles of the rectangle dissection.

A trade relation, called *shape function*, specifies the trade-off of length and width for each circuit. No polynomial algorithm is known to determine the optimal trade-off in the general case [62]. When some restrictions are applied to either the polar graph or to the shape functions, polynomial algorithms become possible. In the next section and in chapter 3 some shape optimization algorithms will be described.

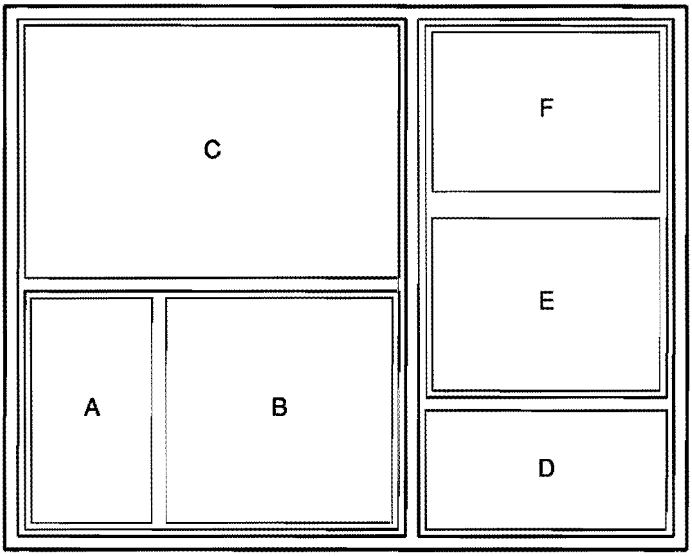
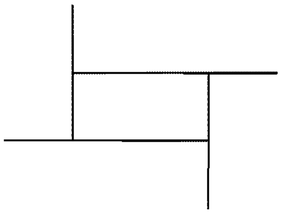


Figure 1.7. A slicing structure. The boxes within boxes indicate the hierarchy of slices.

The restriction to series parallel polar graphs is very effective. The rectangle dissections that have a series parallel polar graphs are called *slicing structures*. A slicing structure is a rectangle dissection that can be obtained by recursively dissecting rectangles into smaller rectangles by vertical and horizontal *slicing lines*. A *slice* is either an undissected rectangle or a rectangle dissected into two slices by a vertical or horizontal slicing line. This recursive definition indicates the hierarchical nature of the slicing structure. The slices are the modules of this hierarchy. The undissected rectangles represent the circuits, the primitive modules of the slicing hierarchy. A slicing structure can be recognized by the property that all dissecting lines dissect a slice from one side to the other. Therefore it does not contain the 'wind mill' pattern:



1.5 Example: shape optimization

In this section we will look at a simple algorithm for shape optimization for rectangular circuits [50]. It serves as an example of how the general predictor-adaptor paradigm can be translated into a concrete and straightforward algorithm. In this algorithm we will use some concepts that will play an important role in chapter 3.

The problem arises in the optimization of the aspect ratio of the circuits in a floor plan. We assume that the circuits have not yet been designed, so that they can take any aspect ratio, but have a fixed area. This assumption is consistent with top-down layout design. The rectangle dissection is already designed, the circuits are designed later. Only the area of the circuits is known, for instance by making a prediction based on the number of transistors or the number of nets in each circuit.

The flexibility of the circuits makes it possible to fill the available area completely. The problem is to find the optimal aspect ratios for the circuits, such that there is no wasted space between the circuits. The aspect ratio of the floor plan is given. The rectangle dissection is represented by a slicing structure. Each undissected slice is a circuit which has two parameters, w and h. The circuits are assumed to be completely flexible, only the product of $w \times h$ is fixed. The trade relation for the length and the width is a hyperbola. Because of the hierarchy and the flexibility of the circuits, an optimal solution is easy to find.

First, the areas of the compound slices are computed in a bottom-up order. This process is simple. The areas of the circuits are known. The

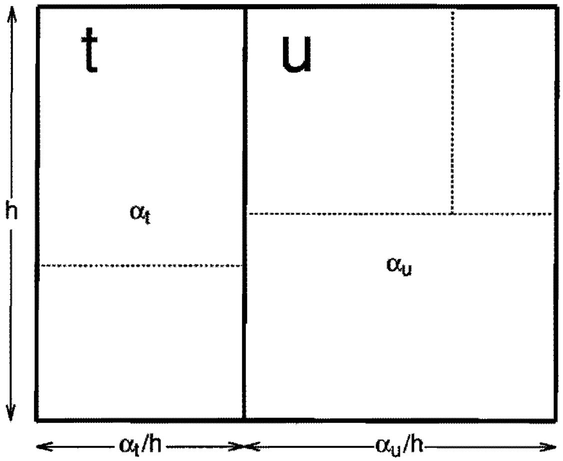


Figure 1.8. Shape assignment in a slicing floor plan. t and u are are slices that are separated by a slicing line *(drawn)* and they are decomposed themselves into sub-slices by further slicing lines *(dashed)*. The optimal aspect ratio of each slice can simply be computed from the area of the slices and the dimensions of the floor plan.

area of a slice is the sum of the areas of its sub-slices. Finally the area of the floor plan is known. This process represents the bottom-up predictor. Information of the circuits is propagated up the hierarchy. This way global information about the design is gathered.

Once all slice areas are known the first global design decisions can be taken. The adaptor adapts the shapes of the slices in a top-down order. The first decision taken is the position of the first slicing line. This position follows directly from the shape of the floor plan and the area of the sub-slices.

Let the area of slice s be α_s. Let slice s consist of two sub-slices t and u. The shape of slice s is $w_s \times h_s$. The shape of the sub-slices can be computed by $h_t = h_s$ and $w_t = \alpha_t / h_t$ if the slicing line is vertical. The shape of u can be computed in the same way. When the slicing line is horizontal h_t can be computed similarly.

By applying this method to all sub-slices, the shape assignment is done in a top-down order. The position of the slicing line determines the shapes of the sub-slices. The shape requirements are realized by translating them to requirements for the sub-slices, thereby moving the problems down the hierarchy. Since we assumed that the circuits are completely flexible, the problems can be solved here easiest.

In this algorithm we can see how the general concepts can be used explicitly in an algorithm for a specific problem. The hierarchy of slices and the flexibility of the rectangles made the problem much simpler. We saw how each slice was modeled by the height and width parameters and how they were related by a hyperbolic shape function.

2. Building block layout

The conventional method of structured layout is building block layout. In this method the circuits are allocated a rectangular area. Because of this restriction they are also called *blocks*. This shape restriction is very common, and rarely a problem. Remember that in top-down design the blocks are designed after the construction of the floor plan, and that the shape of the blocks is to be adapted to the floor plan. There is no reason to make a flexible circuit any other shape than rectangular - rectangular is as good as any. The available area is dissected into rectangles. This dissection is represented by the polar graphs.

The layout problem is reduced to arranging the blocks and realizing the connections between them. In chapter 6 we will introduce another method for top-down layout design in which the primitive circuits are represented by unidimensional 'strips'. The building block method is more suitable for the layout of circuits that consist of complex sub-circuits.

The circuits are connected by nets as specified by the net list. The building block circuit generator designs the layout of a circuit of rectangular blocks. It creates a rectangular layout, which can be used on a higher level in turn. We will assume that the area of the blocks and the area of the interconnections is disjunct: the interconnections are realized between the blocks. This assumption is reasonable if a small number of routing layers is available. When four or more layers are available, this assumption may not be reasonable any more.

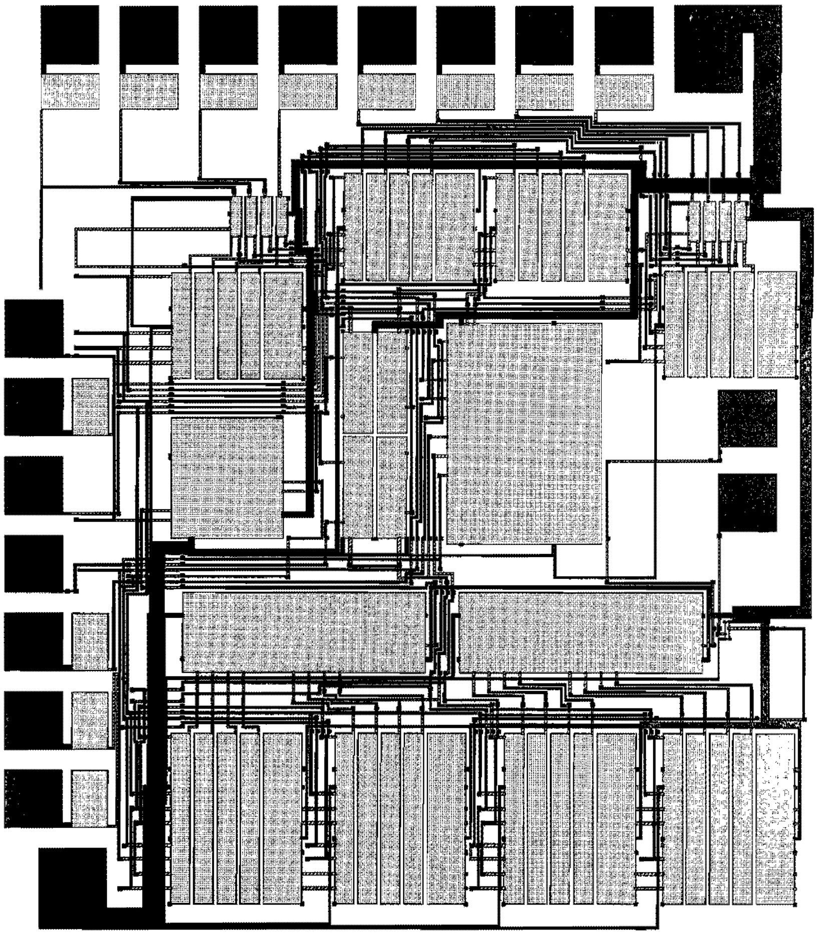


Figure 2.1. A building block layout.

The area for the interconnections is decomposed into *channels*. Each channel corresponds to a line $j \in J$ in the rectangle dissection. The blocks on both sides of this line are moved apart to allow for the routing of the interconnections: each line is replaced by a rectangle that represents the area of the channel. A channel can be seen as an 'interconnection block'.

The layout of the blocks is generated by other circuit generators. The building block circuit generator needs only to generate the layout of the interconnections.

This chapter serves as an example of the use of the predictor-adaptor paradigm to layout design and as a background for the remainder of the book. The generation of a building block layout proceeds in the following steps. First the relative positions of the blocks are determined. Then the nets are routed. From these first design decisions some parameters for the circuit generators are derived. Together these steps are called floor planning. Guided by the parameters, the circuit generators design the layout of the blocks. Finally the layout of the interconnections is generated and combined with the layout of the blocks.

2.1 Floor planning

Floor planning was introduced by Preas and vanCleemput in [56]. They used a polar graph which was constructed by a branch and bound algorithm. In [32] a rectangle dissection was found by planarization of the interconnection graph and making a rectangular dual of this graph. Slicing was introduced in [64]. The approach followed here is based on the ideas of [51] in which a point configuration was sliced to get a slicing structure. Several software packages for floor planning have been developed, for instance [57,41,67].

The *floor plan* consists of a polar graph and its dual and a channel assignment which assigns nets to channels. The design decisions that are taken during the floor planning phase are the design of the polar graphs and the channel assignment.

The design of the floor plan is done in several steps, which can be seen as a gradual refinement of the floor plan. The polar graphs, which give partial orders on the positions of the blocks, are determined first. Once the polar graph is known, a first prediction of the area allocation of the chip can be made, using the predictions for the shape functions of the blocks. This geometric information is used to determine the shortest route of each net.

The polar graph is restricted to a series parallel graph. In that case the rectangle dissection is a slicing structure. Using slicing structures has several advantages due to its hierarchical nature, such as: The use of a slicing structure avoids channel routing order conflicts [50]. A slicing structure facilitates the shape optimization [62, 52]. A slicing structure can be constructed by repeated slicing of a point configuration [51].

Since the shapes of the blocks have not been determined yet, the shape fitting problem does not play such an important role in floor planning. Therefore floor planning aims more at reducing the interconnection length. Shape fitting should be seen as a secondary objective. The slicing structure is designed in two steps [51]. In the first step the interconnections are taken into account. The decisions taken in the first step are refined further in the second step which takes the shape functions of the blocks into account.

The first step in designing a floor plan is the design of a point configuration. In a point configuration, each circuit is assigned to a position in the plane. The circuits are represented by points, and the shape of the circuits is ignored. The point configuration should be regarded as an ideal placement that should be approximated as close as possible. The point configuration is usually designed with disregard of any shape information, often even with disregard of area information. The only kind of information that is used is the net list information.

Once the point configuration has been determined a slicing structure must be found that is consistent with this point configuration. The slicing structure is made by slicing the point configuration by straight orthogonal slicing lines. Any two circuits are separated by a slicing line during some stage of the slicing process. Exactly one of the two ordering relations in both dimensions is enforced by the slicing tree. So, for the purpose of slicing, the point configuration can be seen as two sequences of circuits. Slicing the point configuration means enforcing one of the two ordering between the groups of circuits that are separated by the slicing line.

The distance information in the point configuration may be interpreted as additional connectivity information. However, it is not obvious how this information can be used in the slicing process.

During the slicing the shapes of the circuits can be taken into account. An algorithm that finds the best slicing tree, given a point configuration and the shape functions of the circuits is described in

§ 3.3.

When the topology of the blocks is known, the routes of the nets can be determined. This is called global routing or channel assignment. The channel assignment assigns each net to a set of channel intersections.

After the slicing structure has been determined, the important design decisions have been made and the parameters for the circuits can be derived. From the slicing structure and the trade relations of the length and width parameters, we can derive the optimal shapes of the blocks. This problem is addressed in chapter 3. Also the optimal pin positions can be determined from this floor plan.

2.2 The point configuration

A point configuration is the assignment of circuits to positions in the plane. Each circuit has two coordinates x_c and y_c. The distance d_{ij} between two circuits is

$$d_{ij} = \sqrt{(x_i-x_j)^2+(y_i-y_j)^2} \tag{2.1}$$

The circuits C are connected by the nets N. The connections are represented by a bipartite graph $\mathcal{N}(C \cup \{C\}, N, P)$, in which the edges P are the circuit - net incidences, the pins. Each net has a so-called *net weight* w_n which indicates the relative importance of the net to be kept short. This weight is used in the various score functions used by the different algorithms. A net with weight 2 has the same influence as two nets with weight 1.

A number of numeric methods to construct the point configuration have been described in the literature. The *dutch metric* [51] translates net list information to distances between the circuits. In the dutch metric the distance between two circuits is defined to be

$$m_{ij} = 1 - \frac{\sum\{w_n \mid (i,n) \in P \wedge (j,n) \in P\}}{\sum\{w_n \mid (i,n) \in P \vee (j,n) \in P\}} \tag{2.2}$$

Two circuits that do not share any nets are placed at the maximum possible distance 1. Circuits that share all nets are placed at the minimum distance 0. This distance space is embeddable in a #C–1 dimensional euclidean space.

To reduce the number of dimensions to two, the method of [51] uses a Schoenberg projection [60] to project this multi dimensional point

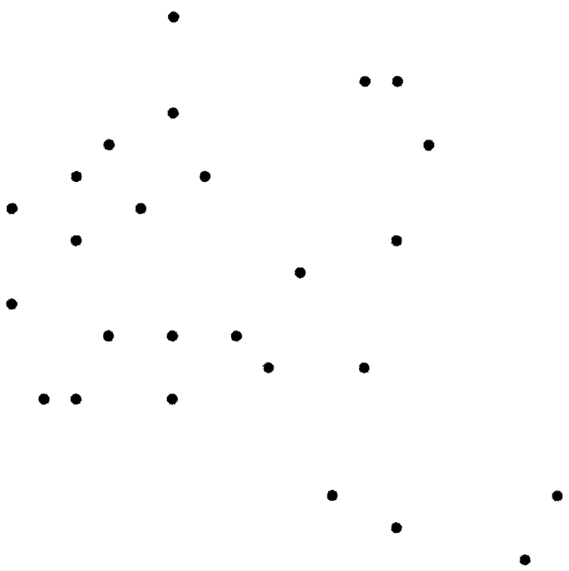


Figure 2.2. The point configuration is a placement of points which represent the circuits in the plane.

configuration onto a plane. This method minimizes the distortion of the distances caused by the projection: it minimizes

$$\sum_{i,j \in C} (m_{ij} - d_{ij})^2 \tag{2.3}$$

The method of [30] tries to minimize the net length by minimizing

$$\sum_{i,j \in C} d_{ij}^2 \sum \{ w_n \mid (i,n) \in P \wedge (j,n) \in P \} \tag{2.4}$$

The minimization of this object function involves the calculation of the eigenvalues of a matrix.

The same problem is posed in [39], except that pin positions on the boundary are given. When given pin positions taken into account, the problem is reduced to solving a number of linear equations.

Equation (2.4) gives a bias for nets with many pins. Of course this problem could be alleviated by adjusting the net weights for multi-pin nets. But preferably, the connections should consist of two pin nets only. We already noted that connectivity is a more important consideration in top-down design than shape fitting. It is therefore advisable to do the routing as early as possible. The ideal is to do routing before placement has been done.

Routing can be done before the construction of the point configuration by using the dutch metric. Using this distance metric, a shortest spanning tree can be determined for every net. The edges of the spanning tree represent two pin nets. The decomposition in two pin nets depends only on the structure of the net list, and not on the placement. Such a decomposed net structure is likely to give less deformation in the schoenberg projection method.

2.3 Channel assignment

After the floor plan has been designed by calculating a point configuration and slicing, the topologies of the interconnections have to be determined. The channel assignment determines which channels are used by a net to make a connection. It is sometimes called global routing because it finds the rough routes of wires. After it is determined which wire uses which channel, the routing within the channels can be compacted. Using contour compaction track assignment is unnecessary.

The *channel intersections* are the set of the intersections between the horizontal and vertical channels. They are sometimes called the T-junctions. The channel intersections are denoted by the set $V \subset J_h \times J_v$.

The *channel assignment* is an assignment function $\chi : N \rightarrow V^*$ which assigns to each net a set of channel intersections. The set of channel intersections of a net must be connected: there must be a path of intersections, going from channel to channel, connecting any pair of intersections in the set.

The routing model is a graph $\mathcal{C}(V, E)$ of which the vertices V correspond to the channel intersections. Therefore this graph is called the *channel intersection graph*. Nodes that are neighbors in a channel are connected by an edge. Figure 2.3 shows an example of a channel intersection graph. The channel intersection graph is a planar graph. Note that the faces of the graph correspond to the circuits in the floor plan.

The length of an edge is the distance between the channel intersections. The distances in the routing model are not changed after

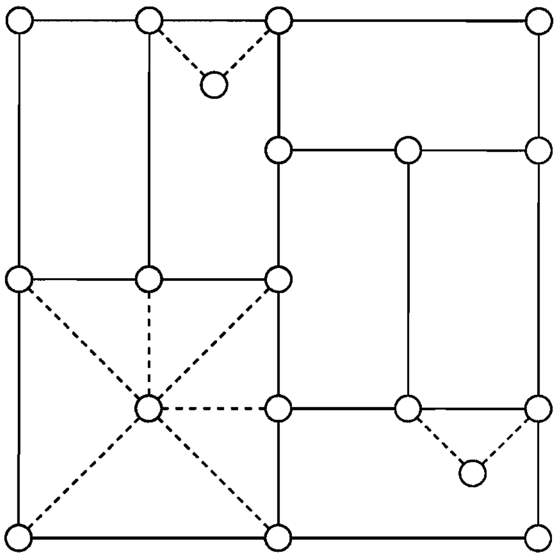


Figure 2.3. Example of a graph used as a global routing model.The drawn edges correspond to the edges of the channel intersection graph. The dashed edges are temporary edges that model the connections of a circuit.

a net has been routed. The sequence in which the nets are routed is therefore irrelevant.

The pins of the net are added to the graph as temporary nodes during the routing of the net. Temporary edges connect the temporary node to nodes that represent possible positions of the pin. A pin whose position is entirely unknown gets edges to all nodes surrounding the circuit. A pin whose position is exactly known only gets two edges. These two edges lead to nodes adjacent to the pin position. In figure 2.3 the temporary edges are indicated by dashed lines.

Some circuits allow a net to enter at different positions on its periphery. Making the length of the temporary edges small encourages routing through the circuit. Longer edges prohibit routing through the circuit.

Since the width of the channel is adapted, there is no limited channel capacity. We will not pay attention to the problem of controlling the channel density. The main criterion for the wiring will be the length of the wires: all wires are to be kept as short as possible.

The routing problem is now to find the shortest connected subset of edges in the extended channel intersection graph that connect to the temporary nodes. This is known as the shortest steiner tree problem in graphs. A heuristic for the steiner tree problem is presented in chapter 4.

The steiner tree is a set of nodes and edges of the channel intersection graph. Only some of the nodes in the steiner tree indicate a transition from one channel to another. These nodes have orthogonal edges in the steiner tree. Only those channel intersections are included in the channel assignment of the net.

2.4 The interfaces

From the floor plan, as determined by the slicing structure and the channel assignment, parameters must be derived for the design of the circuits. Figure 2.4 gives an overview of the flow of the design data. The design is specified as a net list of interconnected circuits. The circuits also each have a description, which depends on the type of circuit. An essential feature of top-down design is the information exchange between the circuit generators and the floor planner.

The circuit generators start by reading the descriptions of the circuits they must design. In bottom-up design the circuit generators would immediately design these circuits. In a top-down design method the circuit generators first predict the capabilities they have for the parameters. For some parameters this can be specified in the form of trade relations.

Using the net list and the capabilities of the circuits, the floor planner designs a floor plan. This floor plan is represented as a slicing structure and as a channel assignment for the nets. The slicing structure determines the ordering of the circuits on the chip. The design parameters that can be derived from the floor plan are passed to the circuit generators. The circuit generators can then design the circuits accordingly. Once the circuits have been designed, the building block layout can be assembled and the masks for the connecting nets can be generated.

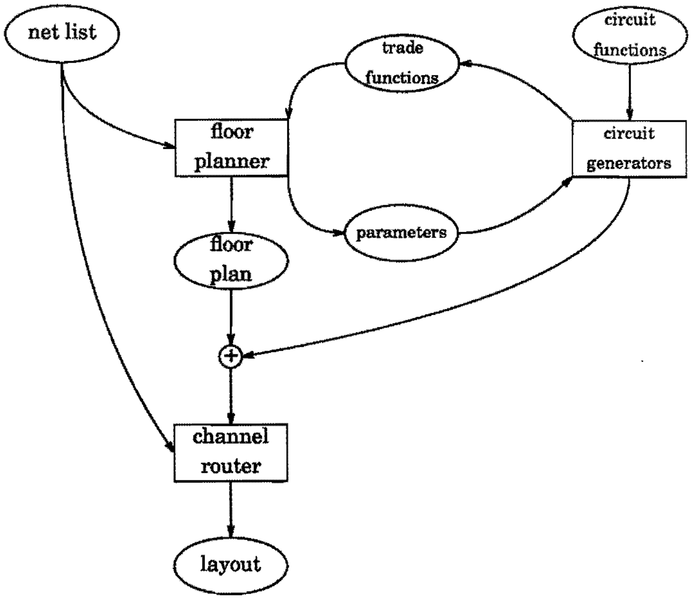


Figure 2.4. Interfaces between the programs.

We use the channel assignment to estimate the length of each wire and the widths of the channels. A statistical method for estimating the channel width has been proposed in [21]. The length of a wire can be used to estimate the capacitance of the wire. The output buffer driving the wires could be dimensioned to compensate this capacitance.

The channel width is not fixed; the width of the channel is determined by the number of wires that must be accommodated. The use of a slicing structure makes it easy to adapt the widths of the channels to match the requirements.

The width of a channel can be estimated from the routes found by the channel assignment. A lower bound for the width of the channel is the number of wires that must pass a certain point. The maximum number of wires that overlap at any point in the channel is called the *channel density*. Experience shows that this bound is rather sharp. After the channel assignment, shape optimization resizes the circuits according to widths of the channels.

A problem is that the topology of the channel intersection graph depends on the dimensions of the circuits and the widths of the channels. To determine the topology and distances in the channel intersection graph, the dimensions of the circuits must be known. It is therefore necessary to determine the optimal shapes of the circuits before the channel assignment can be done. It is advisable to redo the optimization after the channel assignment. Note, however, that the channel assignment, once determined is insensitive to topology changes in the channel intersections graph. The same nodes still exist in the graph, only the connecting edges have been changed. The channel assignment for a net is still valid.

The optimal shapes can be determined with the algorithm that is described in § 3.2. It may be a good idea to redo this optimization after the design of each circuit. When the predictions were inaccurate, it may be necessary to redesign some circuits.

Some blocks, like PLAs, may have limited flexibility and others may be designed beforehand. Of course, it is necessary to limit the number of fixed shape blocks. Another problem of fixed shape blocks is in the pin positions. A special heuristic is necessary to find a good orientation of the circuits. The pin positions can be determined once the orientation of the blocks is determined.

When all the circuit generators can produce the pins where they predicted that they would be, the channel assignment need not be redone. To avoid problems that may occur due to inconsistencies, the channel assignment is redone after all the circuits have be designed.

2.5 Mask generation

When the layout of the circuits has been designed, and the channel assignment of the nets is known, the layout of the routing can be generated. This is one of the most time consuming tasks in manual layout design. The mask generation of the interconnections is probably in any automatic layout design package the most time saving program.

A *layout* is a set of patterns that completely specifies the geometry of the circuit. The term *mask* will be used for each pattern. To keep the mask generating algorithms simple, the mask patterns will be restricted to sets of iso-oriented rectangles.

To increase the yield when the circuit goes into production, patterns are required to satisfy certain rules, the *design rules*. Two classes of rules can be distinguished: structural rules and numeric rules. Structural rules enforce or prohibit certain combinations while numeric rules quantify the width of, and spacings between patterns in a mask or in a combination of masks. The numeric rules are almost exclusively specifications of lower bounds, because it is assumed that the layout design techniques will try to keep the total chip small.

The process of packing the layout elements as close as the design rules allow is called *compaction*. A channel allows for one-dimensional compaction because the positions of the pins of a channel are fixed but the channel width is not. One-dimensional compaction is a simpler problem then two-dimensional compaction.

Contour compaction [15] introduces jogs anywhere where this is advantageous. It takes maximum advantage of the design rules by using different rules on different layers. It allows variable width wires and takes advantage of irregular channel boundaries.

A number of channel routers that use contour compaction have been described [59, 80, 28]. However, contour compaction alone suffices to get good layouts. It makes conventional track assignment unnecessary, which simplifies the channel routing considerably.

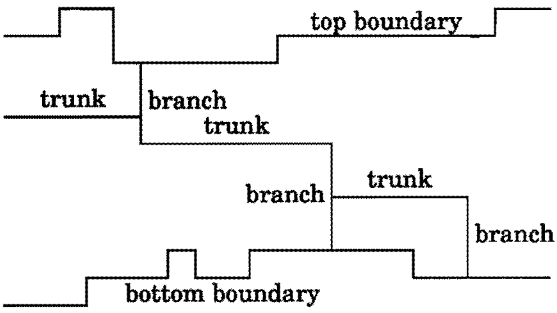


Figure 2.5. A routing channel with irregular boundaries. A net with 4 pins is decomposed into 2 pin sections. Each section has a trunk and two branches. The masks of a pair of branches to the same pin are superimposed.

The channel has two channel boundaries. We will assume that the channel is oriented horizontally, so there is a top and a bottom boundary. Wires can leave the channel at the open ends to the left and the right. Which nets have to leave and at which side is determined by the channel assignment. The pins on the boundary have known exact positions while the pins that leave the channel still have to be positioned.

Given the pins of a net, the net can be decomposed into one or more 2 pin segments. The global router ensures that there is never a net with only one pin. If there are several segments, the segments share a pin. Each pin connects at most to two segments.

Each two pin segment is realized by a horizontal piece called *trunk* and two vertical pieces called the *branches*. The vertical branches bring the wire into the channel. The horizontal trunk connects the branches together.

The compaction is done by maintaining two *contours*, one for each layer, that indicate the area occupied by previous trunks. A contour is a piecewise constant function, which gives the boundary between the area that is occupied by the wires and the free area. Only the trunks and their vias are considered during the compaction; the branches are added once the layout for the trunks is determined.

The contour is initialized with the shape of the bottom channel boundary. Trunks are processed one by one, and the channel is filled from the bottom to the top. The next trunk, with its associated vias, is placed against this contour, thereby using a minimum of space. The space used by this trunk and its vias, determined by the design rules, is added to the contour.

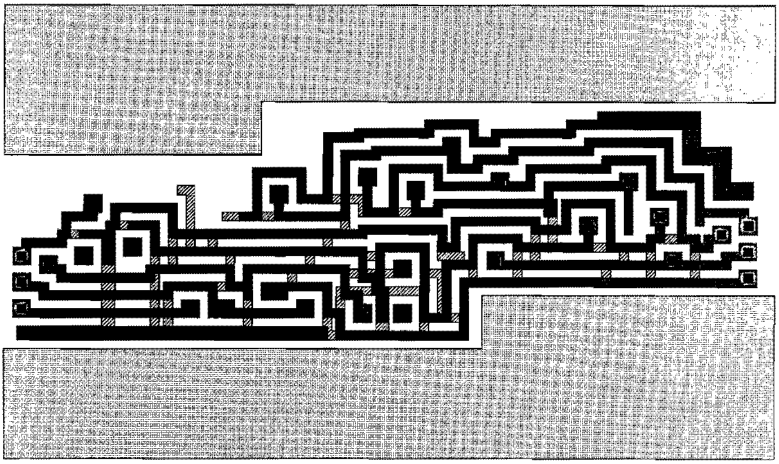


Figure 2.6. Routing after compaction phase.

As can be seen in figure 2.6 not all jogs that are generated this way are necessary to reduce the channel width. Each jog is believed to be a small reliability hazard due to electromigration. Also many jogs mean many rectangles in the layout. Wire straightening therefore increases the reliability of the chip and decreases the amount of disk space needed to store the layout.

Two passes are needed to get rid of unnecessary jogs. In the first compaction pass the width of the channel and the space available for each trunk is determined. All contours are saved, such that it will be known exactly what area can be used for each trunk in the second pass.

In the second pass the actual mask generation is done. The second pass starts at the top of the channel, and the trunks are compacted in the reverse order. Now there are two boundaries for each trunk, delimiting the area available on both sides. The straightening algorithm constructs a trunk with a minimum number of jogs that fits between the two contours.

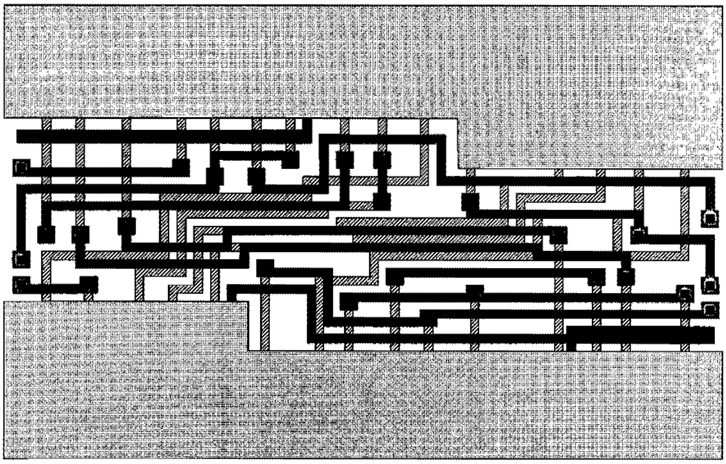


Figure 2.7. Routing after straightening phase.

Because the pins are not restricted to grid positions, a branch may have more than one opposite branch. Therefore gridless channel routing introduces more vertical constraints. A problem is that there may be cycles in the vertical constraint graph. In [44] a simple but less conventional approach was used, which guarantees that there will be no cycles. This will guarantee that the algorithm generates a valid layout.

In the approach of [44] the two pin segments are divided into five classes, indicated by the letters A through E. Branches and trunks are realized in both layers. Figure 2.8 illustrates the different classes. The bottom layer is realized in polysilicon, the top layer is realized in metal.

Class C segments, that have pins in exactly opposite positions of the channel, are handled as class E or D segments. Power and ground lines are routed planarly. Therefore the power and ground lines are routed at the top and bottom of the channel. In [44] the branches at the top of the channel use another layer than the branches at the bottom. However, to be able to connect to power and ground on the metal layer it is unavoidable that the branches connect to the sides of the cells on the polysilicon layer. Therefore the classes D and E use an extra via hole to change layer.

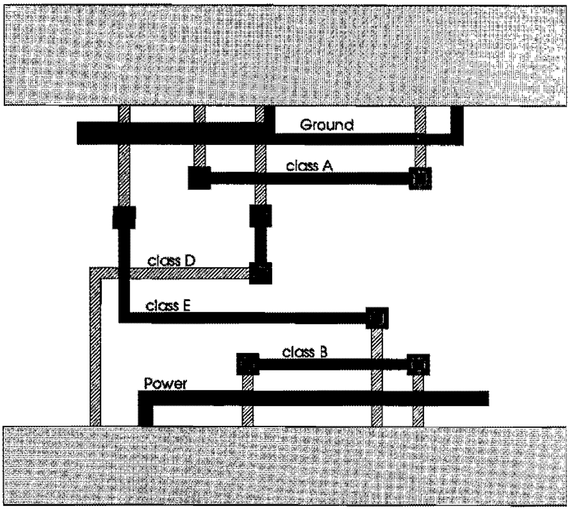


Figure 2.8. Different classes of two pin segments.

The segments will be ordered from the bottom of the channel to the top of the channel. The classes can be ordered in the sequence B–E–D–A. Within the classes A and B there are no vertical constraints, and any order is possible. Within class D the segments can always be ordered on the position of the bottom pin from right to left. The segments of class E can be ordered from left to right on the position of the top pin. This always constitutes a valid order, in which the vertical constraints are satisfied.

The vertical constraints in the classes E and D tend to form very long chains. This causes trunk ordering algorithms similar to [76,8] to become ineffective. It is necessary to use contour compaction to reduce the channel width.

Table 2.1 compares the channel widths for two randomly generated channels and Deutsch's Difficult Example [14]. The channels have a large number of pins. Realistic channels in building block layouts mostly have far fewer pins. The channels were routed using Mead & Conway [46] design rules. Since our router does not have the concept of tracks it is only possible to compare the width of the channel in lambda's. The pin positions were on a grid and the channel sides were straight. For comparison the last column is added which gives the density of the channel times 6. This could be the performance of a good track based router.

Table 2.1. Performance comparison for three Channels

example	gridless router *pin pitch* 5	gridless router *pin pitch* 6	track router *track pitch* 6
random1	128	120	132
random2	82	68	84
Deutsch's	144	116	114

The results for Deutsch's Difficult Example are almost as good as those of a good track based router, and for random examples even better. In most practical floor plans the pin density is much lower, and compaction is more rewarding than track assignment techniques.

3. Joint shape optimization

In this chapter we will consider the problem of joint optimization of the shapes of the circuits. The essential idea of the adaptor is the adaptation of the flexible circuits to their environment. For the shapes of the circuits this means that they have to be optimized simultaneously.

We will assume that all circuits occupy a rectangular area, and that only the length and width can be chosen. The objective is to find the optimal aspect ratio of each circuit such that the area occupied by the compound circuit has minimal dimensions. In the remainder of this chapter the shape of the circuits will be limited to a rectangle.

The possible shapes of the circuits are represented by a shape function. The shape function gives the smallest height of a circuit as a function of the width of the circuit. It is a model for the possible parameter options of the circuit. The circuits can be ordered in space in different ways. Whether the optimization algorithm is polynomial also depends on the kind of spatial ordering and the representation of the shape functions.

In the first section we will look at the different representations of the shape functions. Then we will look at the different possible spatial orders of the circuits, and the consequences for the optimization problem. In the following sections two specific optimization problems will be elaborated.

3.1 Shape functions

As multiple parameters of circuits are optimized together, a tradeoff relation between the parameters must be specified. In the optimization of the aspect ratio of the circuits, the parameters that can be traded against each other are the width w and the height h of the circuit. The height and width of a circuit are related to each other by a *shape function* $s(w)$. Allowable shapes for the circuit are those for which $h \geq s(w)$.

Width and height can be viewed as valuable resources. We may assume that a larger width will not lead to a larger height. A shape function is therefore a positive non-increasing function:

$$s(w_1) \geq s(w_2) > 0, \quad 0 < w_1 < w_2 \tag{3.1}$$

The *inverse* of the shape function, $s^{\leftarrow}(h)$ is defined as

$$s^{\leftarrow}(h) = \min\{w \mid s(w) \leq h\} \tag{3.2}$$

For a completely flexible circuit only the area α is known. Ideally we may assume that a certain area is necessary to perform the function of the circuit. In that case the shape function follows a hyperbola:

$$h(x) = \frac{\alpha}{x} \tag{3.3}$$

Obviously, flexible circuits have limits on the aspect ratio, and other circuits may not be flexible at all. Piecewise linear functions can be used to model a large variety of circuits. A *piecewise linear shape function* is defined as

$$s(w) = \begin{cases} \infty & \text{if} \quad w < w_1 \\ a_1 w + b_1 & \text{if} \quad w_1 \leq w < w_2 \\ a_2 w + b_2 & \text{if} \quad w_2 \leq w < w_3 \\ \ldots & \text{if} \quad \ldots \\ b_n & \text{if} \quad w_n \leq w \end{cases} \tag{3.4}$$

and can be represented in a computer by 3 vectors $\vec{w}, \vec{a}$ and $\vec{b}$. A piecewise linear shape function must be non-increasing:

$$a_i \leq 0 \ \wedge \ a_i w_{i+1} + b_i \geq a_{i+1} w_{i+1} + b_{i+1} \tag{3.5}$$

Notice that a piecewise linear function models the limitations on the

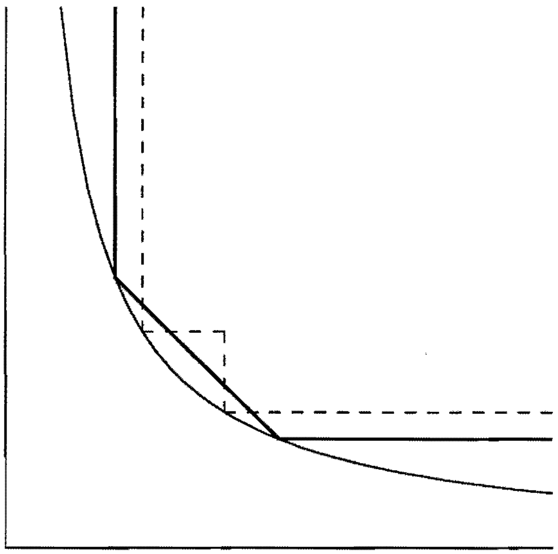


Figure 3.1. A hypothetical flexible circuit would be represented by a hyperbola. A convex piecewise linear shape function is used to approximate the hyperbola. The dashed line is a staircase function. It models a fixed shape rectangle with two orientations.

aspect ratio: $w \geq w_1$ and $h \geq b_n$.

A special case of the piecewise linear functions are the convex piecewise linear functions. These can be represented by a set of linear inequalities:

$$\mathbf{A}\begin{bmatrix} h \\ w \end{bmatrix} \geq \vec{c}, \quad \mathbf{A} \geq 0, \quad \vec{c} > 0 \tag{3.6}$$

Another special case of the piecewise linear functions are the *piecewise constant*, or *staircase shape functions*:

$$s(w) = \begin{cases} \infty & \text{if } w<w_1 \\ b_1 & \text{if } w_1 \le w < w_2 \\ b_2 & \text{if } w_2 \le w < w_3 \\ \ldots & \text{if } \ldots \\ b_n & \text{if } w_n \le w \end{cases} \tag{3.7}$$

Staircase functions can be seen as the enumeration of a number of options for the shape of the circuit. For circuits with a rigid shape there are two possible orientations giving two shape options. For reasons of complexity to be explained later, we will also introduce the *integer staircase functions* for which $w_i, b_i \in \mathbb{N}$.

The joint shape optimization problem is solved by computing the shape function of the whole using the shape function of the pieces. Once this shape function is known, the point with the lowest score $\varepsilon(w,h)$ can be selected. $\varepsilon(w,h)$ must be nondecreasing in both arguments. This guarantees an optimum solution on the points $(w,s(w))$ of the the shape function. It is a reasonable assumption, since a circuit that is purely larger in one dimension is always worse. Examples of such score functions are

$$\varepsilon(w,h) = wh \quad \text{and} \quad \varepsilon(w,h) = 2(w+h) \tag{3.8}$$

The first one measures the area of the floor plan, the second the perimeter. The second has a preference for small, close to square solutions.

When one of these two score functions is used in combination with a piecewise linear function, the solution is always on the convex corners of the piecewise linear function. The optimum can simply be found by evaluating this function for all w_i.

In the algorithms described in this chapter, the shape function of the floor plan is computed by combining the shape functions of the circuits and slices. The algorithms use two operations on shape functions: addition and sometimes taking the minimum of two shape functions. Addition arises when two slices with known shape functions are abutted to form one slice. The new slice has a shape function that is the sum of the two shape functions of its components. Depending on the orientation of the slicing line, the widths or the heights of the slices must be added. For two slices that are stacked vertically, the new

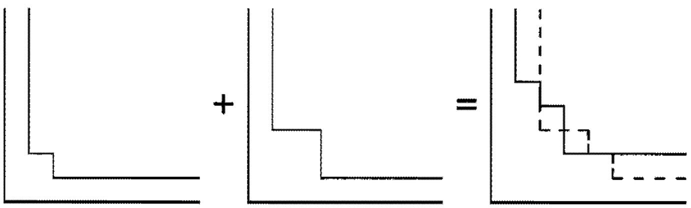


Figure 3.2. Addition of two shape functions. The shape function of a slice is computed by adding the shape functions of the subslices. The shape function of a vertically stacked slice is drawn, the shape function of a horizontally stacked slice is dashed.

shape function is computed by: $s_{new}(x) = s_1(x)+s_2(x)$. For slices that are abutting horizontally, the shape function is: $s_{new}^{\leftarrow}(y) = s_1^{\leftarrow}(y)+s_2^{\leftarrow}(y)$.

In some algorithms, there are multiple different implementations possible for a slice. For instance, the same subslices can be abutted in two ways: horizontally and vertically. Each of the different implementations has a shape function. The shape function of the slice, independent of implementation, is the minimum of the shape functions of the different implementations: $h_{new}(x) = \min(h_1(x), h_2(x))$. Note that minimization is not dependent on orientation.

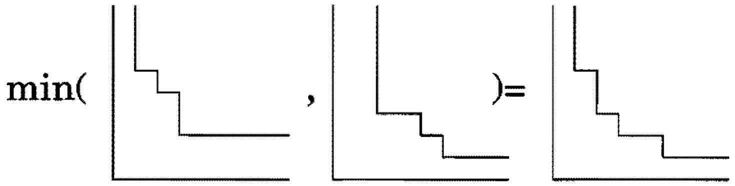


Figure 3.3. Computing the minimum of two shape functions. Taking the minimum of the shape functions of two implementation options summarizes the options possible for that slice.

Different spatial orders are used for representing the topology of floor plans. Let (x_i, y_i) be the coordinates of circuit i. The spatial ordering constraints are formulated in terms of inequalities:

$$x_i+h_i \le x_j \quad y_i+w_i \le y_k \tag{3.9}$$

In a general rectangle dissection the constraints for the x coordinates can be represented by the polar graph. The polar graph gives a partial order for the circuits. The constraints for the y coordinates are given by the dual polar graph.

Consider the joint shape optimization problem. The two dimensions of each circuit are related to each other by equation (3.4), while the dimensions of the circuits are mutually related by equation (3.9). This problem can be solved in exponential time with integer programming as described in [77]. A branch and bound algorithm was presented in [68]. The constraints of equation (3.9) combined with the equations for the convex shape functions of equation (3.6) form a linear program which can be solved in polynomial time [49]. If the shape functions are not convex, the problem becomes NP-hard [62]. An iterative numerical approach using the constraints of equation (3.3) was described in [67].

When the polar graph is a series parallel graph, the rectangle dissection can be described by a slicing tree. For a slicing floor plan, the joint shape optimization problem can be solved in polynomial time for the more general shape functions of equation (3.4). The shapes of the circuits in the slicing tree can then be optimized by the algorithm proposed in [52]. A simpler version of this algorithm, using shape functions of equation (3.3) was described in § 1.5 and first presented in [50].

In the next section we will review the algorithm of [52]. Although this is not a new algorithm we will treat it more extensively because it is important to the building block method, while it reflects the predictor-adaptor paradigm on a small sub-problem in an elegant way. Furthermore, understanding this algorithm may help to understand the combined slicing and shape optimization algorithm of the following section.

In [36] the algorithm was extended to determine the optimal orientation of the slicing lines. This can also be done in polynomial time, but only if the shape functions are limited to integer staircase functions.

Finally, the spatial relationships can be represented by a point configuration. In § 3.4 the two steps of slicing and shape optimization are combined into a single exact and polynomial algorithm. For a given point configuration, it is possible to determine the optimal slicing tree and circuit shapes. For integer staircase functions, this can be done in polynomial time. This new algorithm will be presented in § 3.3.

3.2 Optimization of a slicing floor plan

In this section we will look at the algorithm of [52, 62]. For a given slicing structure, and piecewise linear shape functions, this algorithm determines the optimal shapes of the circuits. It does this in polynomial time and it always finds the best solution.

The two algorithms of [62] and [52] differ only in the representation of the shape function. The first [62] uses stair case functions, while the other [52] uses more general piecewise linear functions.

The floor plan is represented by a binary tree: the slicing tree. The slicing tree represents the hierarchy of the slicing structure. The leaf nodes of the slicing tree are the circuits $c \in C$. The leaf nodes have only one edge $\forall_{c \in C}[°c=1]$. The other nodes are called slicing nodes J. There is a root node $r \in J$ which has two edges $°r=2$. All other nodes have degree 3: $\forall_{j \in J \setminus \{r\}}[°j=3]$. The 3 nodes adjacent to such a slicing node are represented by left(j), right(j) and parent(j).

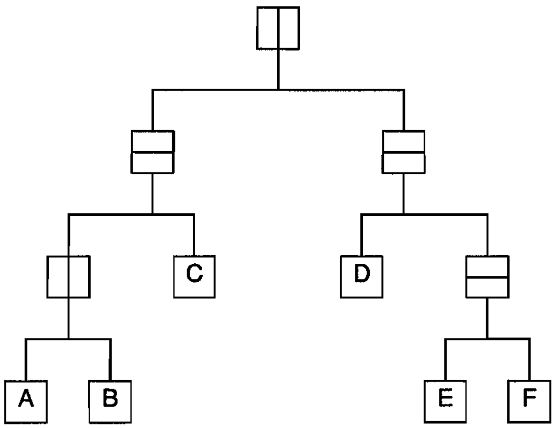


Figure 3.4. The slicing hierarchy of the slicing structure of figure 1.7

The nodes of the tree correspond to the slicing lines. Each slicing node has a label ω(j), which indicates the orientation of the corresponding slicing line: ω(j) can be either ⊟ or ◫. This binary tree representation of the slicing tree was chosen to simplify the algorithms.

The shape function of a slice can be computed by adding the shape functions of the sub-slices. The circuits have shape functions $s_c(w)$. From the shape function of the circuits, the shape functions of the slices, $s_j(w)$ are computed. For slices that are stacked vertically, (horizontal slicing line), the shape functions can easily be added. For horizontally abutted slices, the functions must be inverted before adding them, and the result must be inverted afterwards.

In this way the shape function of the floor plan $s_r(w)$ is computed. A pair (w,h) for which the score $\varepsilon(w,h)$ is minimal is determined. Once the shape of the floor plan has been determined, the dimensions of the slices can be derived.

The algorithm consists of two phases: a bottom-up phase in which information is collected, and a top-down phase, during which the decisions are taken. On a sub-problem, this reflects the general predictor-adaptor paradigm.

Algorithm 3.1. The predictor: computation of the shape functions. The function returns the shape function of slicing node v.

```
function predictor(v);
begin if not $v \in C$ then
        begin $s_{left(v)}$ :=predictor(left(v));
              $s_{right(v)}$ :=predictor(right(v));
              if $\omega(v)=\boxminus$
              then $s_v := s_{left(v)} + s_{right(v)}$
              else $\overleftarrow{s_v} := \overleftarrow{s_{left(v)}} + \overleftarrow{s_{right(v)}}$;
        end;
        return($s_v$);
end;
```

The predictor is a depth first search that collects information about the sub-slices. This is done by computing the shape function for each slicing node in the slicing tree in a bottom-up order. The shape functions are retained for the second pass of the algorithm.

From the shape function of the whole circuit, the most suitable height/width combination can be chosen. Then in a depth first search algorithm, the shapes of all sub-slices, and finally all the leaf nodes are determined. The second phase of the algorithm is a top-down phase, during which the decisions about the shapes of the slices and the circuits are taken.

Algorithm 3.2. Top down shape assignments. adaptor(v, w, h) assigns dimensions w, h to slice v.

```
procedure adaptor(v, w, h);
begin if $v \in C$ then
          begin width[v] = w;
                height[v] = h;
          end
          else if $\omega(v) = \boxminus$ then
          begin adaptor(left(v), w, $s_{\mathrm{left}(v)}(w)$);
                adaptor(right(v), w, $s_{\mathrm{right}(v)}(w)$);
          end
          else
          begin adaptor(left(v), $\overleftarrow{s}_{\mathrm{left}(v)}(h)$, h);
                adaptor(right(v), $\overleftarrow{s}_{\mathrm{right}(v)}(h)$, h);
          end;
end;
```

When adding two piecewise linear functions, the number of line sections in the functions can be at most $\#s_1+\#s_2-1$. At each junction node at least one line section is eliminated. So, the number of line sections for the entire floor plan is at most

$$1-\#C+\sum_{c\in C}\#s_c \tag{3.10}$$

When the line sections are kept sorted, the addition of two shape functions can be in linear time. Both passes are depth first search algorithms, that visit each node in the tree exactly once. Therefore, the complexity of the algorithm is $O(\#C^2)$ if the number of points per circuit is constant.

3.3 Optimal slicing

In this section we will consider the problem of simultaneously constructing the slicing tree and finding the optimal shapes of the circuits. The objective is to find a slicing tree that will lead to a minimum area floor plan. A point configuration with circuit coordinates x_c, y_c is given.

Each node in the slicing tree corresponds to a slicing line in the point configuration. Each slicing line divides the circuits in a slice into two groups. A horizontal slicing line partitions the circuits depending on their y-coordinate, a vertical slicing line partitions the circuits by their x-coordinate.

In [51] a heuristic for slicing the point configuration is proposed. This method solves the problem in two steps: first the slicing tree is designed, then the shapes of the circuits are optimized. The slicing lines are picked using heuristic criteria. Each slicing line corresponds to a node in the slicing tree. The first few slicing lines are chosen with respect to connectivity criteria. The slicing lines at a lower level of the hierarchy are chosen to optimize shape fitting. The slicing heuristic tries to keep the circuits close to square.

Once the slicing tree has been designed, the shapes of the circuits can be optimized using the algorithm of § 3.2. The slicing tree is constructed top-down, beginning with the nodes at the top of the hierarchy.

The new algorithm constructs a slicing tree for a given point configuration. It is possible to compute the shape function that represents the best solutions over all slicing trees that match the point configuration. Although an exponential number of slicing trees is possible, we will show that only a polynomial number of possible sub-slices needs to be considered. We will use dynamic programming over these slices to find the optimal slicing lines for the point configuration.

The circuits of a slice are always contained in a rectangular area of the point configuration that contains only points of the circuits of the slice. This is true for any sub-slice of any slicing tree that is consistent with the given point configuration. The set of circuits contained in a rectangular section of the point configuration will be called a *point slice*.

A rectangular area of the point configuration has four sides. There are #C positions for a side, separating the circuits in the point configuration. Of course the number of possible choices for the right side is limited by the choice of the left side. The same goes for the other two sides. Therefore there are at most $(\frac{1}{2}\#C(\#C-1))^2$ different rectangles. Many of these rectangles contain the same set of points. So, the actual number of point slices to be considered is smaller. The exact number of point slices depends on the point configuration, but obviously this number cannot be larger then $O(\#C^4)$.

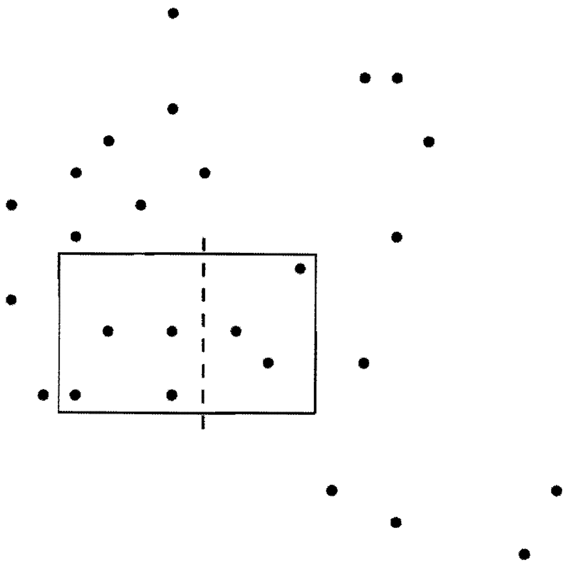


Figure 3.5. The different rectangular cuts of the point configuration are called point slices and are the partial solutions of the dynamic programming problem. The dashed line indicates a possible slicing of the set of circuits.

The shape function of a point slice is computed by combining the shape functions of smaller composing point slices. A point slice with the given set of circuits can be composed in different ways. There are different options for the choice of the slicing line that partitions the point slice. For a point slice with n circuits there are $2n-2$ options.

The shape function of each option is computed by addition of the shape functions of the sub-slices. The shape function of the slice becomes the minimum of the shape functions for the different options.

Initially only the shape functions of the slices with one circuit are known. The shape functions of slices with 2 circuits can be computed by combining two slices of 1 circuit. For a slice of two circuits two implementation options exist: one with a horizontal slicing line and one with a vertical slicing line. The shape functions of slices of 3 circuits are computed by combining slices of 1 and 2 circuits. The slices of 4 circuits are computed by combining slices of 2 and 2 circuits or of 1 and

3 circuits, etc..

This method of computing partial solutions from smaller partial solutions is called dynamic programming. Each partial solution can be used in the computation of several larger solutions. The partial solution needs only be computed once, which saves a large amount of time.

The partial solutions are subsets of the power set of circuits: $S \subset C^*$. Not all subsets need to be considered, only the point slices are enclosed by a rectangle in the point configuration. There are only a polynomial number of such point slices. This fact makes dynamic programming efficient.

Algorithm 3.3. Bottom up computation of shape functions. The function predictor(Q) returns the shape function for a slice with the circuits in set Q.

```
function predictor(Q);
begin if $s_Q := \emptyset$ then
          for $\omega = ⊟, ◫$ do
          begin $V := \emptyset$;
                $W := Q$;
                while #W>1 do
                begin if $\omega = ⊟$
                      then $q := q \in W \mid \forall_{t \in W}[y_q \le y_t]$
                      else $q := q \in W \mid \forall_{t \in W}[x_q \le x_t]$;
                      $V := V \cup \{q\}$;
                      $W := Q \setminus V$;
                      $s_V$ := predictor(V);
                      $s_W$ := predictor(W);
                      if $\omega = ⊟$
                      then $s_Q := \min(s_Q, s_V + s_W)$
                      else $s_{\overleftarrow{Q}} := \min(s_{\overleftarrow{Q}}, s_{\overleftarrow{V}} + s_{\overleftarrow{W}})$;
                end;
          end;
end;
```

The predictor computes the shape function for each point slice. The shape function of the point slice $Q \in S$ is computed by taking the minimum over all partitions of Q into a pair V, W that are allowed by the point configuration. The shape functions are saved in a global data structure for the second phase.

Algorithm 3.4. Top-down slicing decisions and shape assignment, the adaptor assigns dimensions w, h to set Q

```
procedure adaptor(Q, w, h);
begin if #Q = 1 then
          begin width[$q \in Q$] := w;
                height[$q \in Q$] := h;
          end
          else
          begin for $\omega$:=⊟,◫ do
                begin V:=$\emptyset$;
                      W := Q;
                      while #W>1 do
                      begin if $\omega$=⊟
                            then q := $q \in W \mid \forall_{t \in W}[y_q \le y_t]$
                            else q := $q \in W \mid \forall_{t \in W}[x_q \le x_t]$;
                            V:=$V \cup \{q\}$;
                            W:=Q\V;
                            if $\omega$=⊟
                            then $s' := s_V + s_W$
                            else $\overleftarrow{s'} := \overleftarrow{s_V} + \overleftarrow{s_W}$;
                            if $s'(w) \le h$ then goto found;
                      end;
                end;
                { this point should never be reached }
   found:       slice(V,W);
                if $\omega$=⊟ then
                begin adaptor(V, w, $s_V(w)$);
                      adaptor(W, w, $s_W(w)$);
                end
                else
                begin adaptor(V, $\overleftarrow{s_V}(h)$, h);
                      adaptor(W, $\overleftarrow{s_W}(h)$, h);
                end;
          end;
end;
```

From the shape function of the entire set C, the most optimal shape of the floor plan is chosen. The adaptor then traces back the computations of the first phase that led to this shape. While it does that, it slices the point configuration, and assigns dimensions to the circuits.

Any two circuits are separated by a slicing line during some stage of the slicing process. Exactly one of the two ordering relations in both dimensions is enforced by the slicing tree. So, for the purpose of slicing, the point configuration can be seen as two sequences of circuits. The distance information in the point configuration may be interpreted as additional connectivity information. However, it is not obvious how this information can be used in the slicing process.

The slicing algorithm takes the shape functions of the circuits into account. Connectivity information is disregarded with the exception of the information that is contained in the point configuration. The point configuration is an approximate placement. The design of the point configuration takes connectivity information into account, but may ignore the shapes of the circuits.

3.4 Efficiency of the slicing algorithm

Figure 3.6 shows an example of 30 arbitrary circuits was sliced using the optimal slicing algorithm. The circuits are rigid, but can be oriented both ways. An arbitrary point configuration was used.

The shape function for the floor plan of figure 3.6 is shown in figure 3.7. We see that the flexibility of this floor plan is very great. The shape function is an almost perfect hyperbola. After the slicing lines have been chosen, the flexibility of the floor plan is very small. Most floor plans have a limited number of circuits, typically 5-25, and for such numbers exact solutions can be found easily. Because the complexity of the algorithm is quite high, a heuristic may be more efficient for a larger number of circuits.

The slices separated at a high level have a large number of circuits. They can be given almost any shape. The first decisions of the first few slicing lines have little influence on the quality of the final result. The heuristic chooses the first few slicing lines on the the basis of shape fitting. If a slicing line is accepted, the shape of the slices is determined by the aspect ratio of the floor plan and the areas of the slices. This is similar to the algorithm of § 1.5. A slicing line is acceptable if all circuits in the slice can be fitted into the slice individually. If no acceptable slicing line can be found, algorithm 3.4 is called. The floor plan found by this heuristic is shown in figure 3.8.

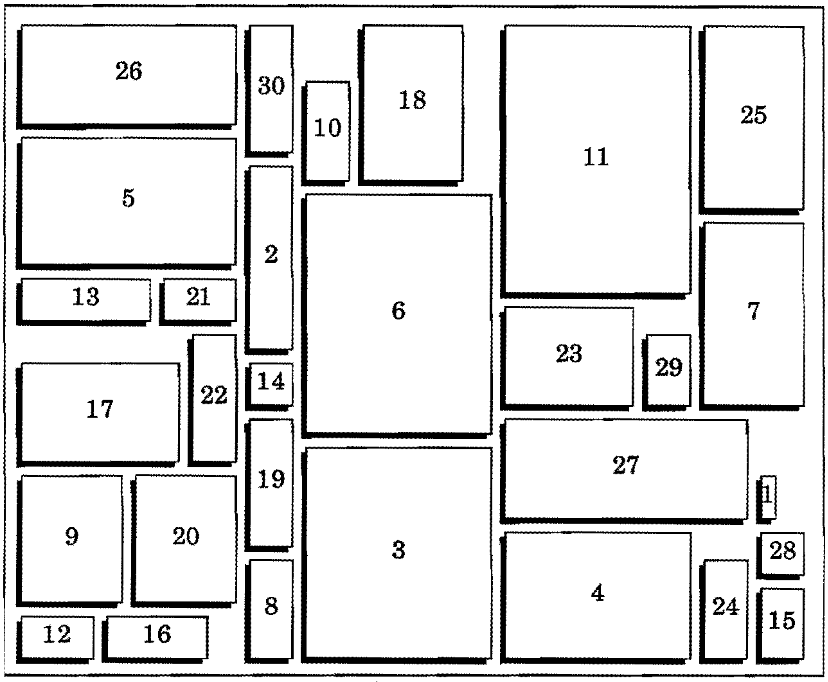


Figure 3.6. Example slicing result of the dynamic programming algorithm. This result has a 95.7% occupation and took about 5 minutes to compute on a single CPU of an Alliant/FX8.

For reasons of complexity, general piecewise linear functions cannot be used. If piecewise linear functions were used, the number of line sections could grow exponentially with the number of circuits in a slice. To keep the complexity of the algorithm down, the algorithm uses integer stair case functions. The length and width of each circuit are expressed as small integers. Because all the discontinuities are required to be at integer coordinates, the discontinuities in different functions will often occur at the same coordinate. Since such discontinuities will often coincide the number of discontinuities will not grow exponentially. Using integer staircase functions, the number of segments is limited by the maximum dimension of the slice.

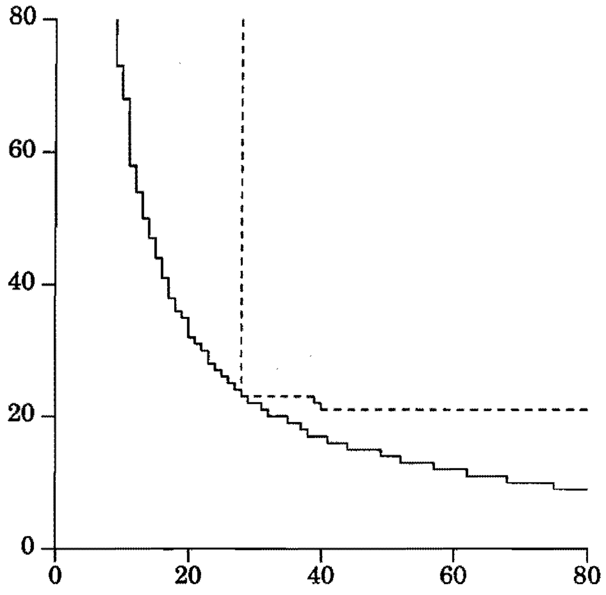


Figure 3.7. Shape functions for the floor plans of the 30 circuit example: The drawn shape function represents all possible shapes with a given point configuration. After the optimal slicing tree has been chosen, the shape of the floor plan is limited to the dashed shape function.

The shape functions have been implemented as arrays of integers. The index of the array is the argument of the shape function. For integer stair case functions, this is the most efficient implementation. Addition, minimization and inverting a shape function can all be implemented as simple 'for' loops. These loops can simply be vectorized.

The complexity of the algorithm is polynomial, although a rather high polynomial. The number of slices is limited by $O(\#C^4)$. Constructing a new slice from known sub-slices takes $O(\#C)$ additions and minimizations. Each minimization and each addition takes $O(w+h)$ operations, where w, h are the maximum integer dimensions of the floor plan. When the dimensions of the circuits are small constants, the maximum dimension of the floor plan grows linearly with #C. The complexity of the algorithm is $O(\#C^6)$.

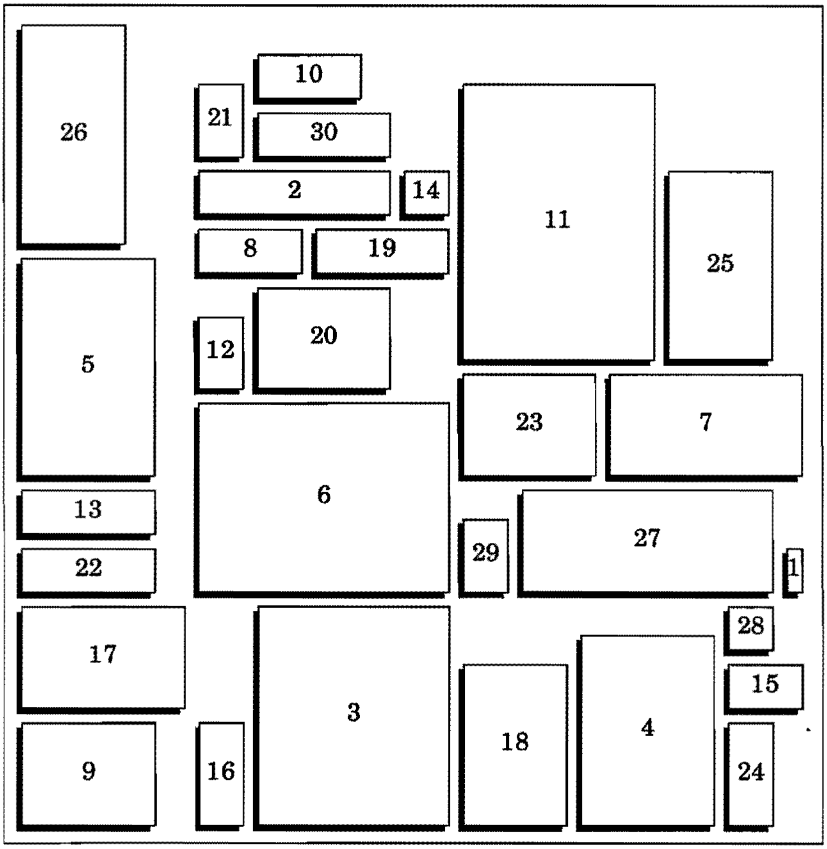


Figure 3.8. Example slicing result from the heuristic for the same problem. This result has a 81.5% occupation and took less than a second to compute on a single CPU of an Alliant/FX8.

The number of slices is $O(\#C^4)$, but the number of possible sets of circuits is exponential: $\#(C^*) = 2^{\#C}$. To avoid using an exponential amount of storage, the sets have been hashed. Using hashing, the access time to a set and its shape function is almost constant.

Connectivity information may also be important. Algorithm 3.4 chooses a slicing structure that realizes minimum dimensions. But there may be many different slicing structures that realize the same dimensions. A variation of the algorithm is conceivable which tries to enumerate the different solutions, and evaluates the solutions with respect to a measure of connectivity, for instance wire length.

4. A steiner tree heuristic

The steiner tree problem arises in the channel assignment of nets in floor plans. The channel assignment problem is reduced to finding the shortest steiner tree in the channel intersection graph. In this chapter we present a heuristic for the shortest steiner tree problem in graphs. The heuristic is based on a notion of hierarchy in the steiner tree. This hierarchy, the topology of the steiner tree, is determined using a shortest spanning tree algorithm.

A second stage maps the topology of the steiner tree onto the graph. It decides on the best positions for the steiner nodes in a top down order. This stage finds in polynomial time the best solution possible with the given topology. Because large numbers of steiner trees may have to be calculated special attention is given to an efficient implementation.

4.1 The shortest steiner tree problem in graphs

The shortest steiner tree problem in a graph can be formulated as follows: Given a graph $\mathcal{G}(V,E)$, with nodes $V=\{v_1,v_2...\}$ and edges $E=\{e_1,e_2...\}$ with positive integer edge weights $w\colon E\rightarrow\mathbb{N}$. Let $S\subset V$ be the subset of nodes to be connected, called the *leaf nodes*. The *steiner tree* of S is a connected acyclic subgraph $\mathcal{T}\subset\mathcal{G}$ such that $(S,\emptyset)\subset\mathcal{T}$. A steiner tree is a acyclic connection in the graph connecting the nodes of S. A *steiner node* is a node $t\in\mathcal{T}\setminus S \mid {}^\circ t>2$. The steiner nodes are the junction nodes of $\mathcal{T}$ in which the tree splits. The number of steiner nodes is $\leq\#S-2$. The *length* of a steiner tree is $\sum_{e\in E(S)} w(e)$. The problem is to find the minimum length steiner tree in a graph.

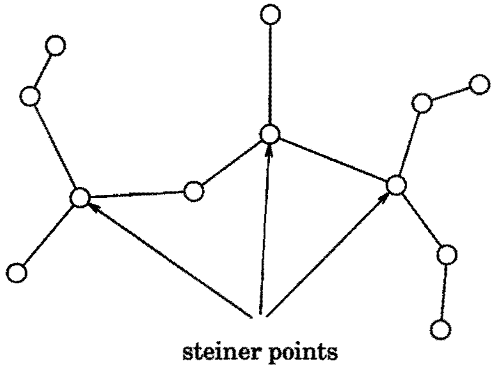


Figure 4.1. The shortest steiner tree connecting 5 leaf nodes could look like this. This steiner tree has 3 steiner nodes. The steiner tree is a subgraph of $\mathcal{G}$ (which is not shown here). Thinking about the steiner tree problem in the plane may help to understand the steiner tree problem in graphs.

The problem has been shown to be NP-hard for general graphs [34]. The global routing graph is planar, but even for planar graphs the problem remains NP-hard [24]. For series parallel graphs the decision problem and the optimization problem are in P [66]. For this class the problem can be solved by a dynamic programming method that takes advantage of the hierarchical structure of the series parallel graph.

Let v be an arbitrary node in the steiner tree. Let (v,w) be an edge of the steiner tree. Let $S_{v,w}$ be the subset of S that is connected to v when (v,w) is removed. Then there is a shortest steiner tree $\mathcal{T}_{v,w}$ with leaf nodes $S_{v,w} \cup \{v\}$ such that $\mathcal{T}_{v,w} \subset \mathcal{T}$. This is easy to understand. If a shorter tree $\mathcal{T}_{v,w}$ would exist this could be replaced in $\mathcal{T}$, making $\mathcal{T}$ shorter. Since $\mathcal{T}$ is the shortest, such a $\mathcal{T}_{v,w}$ cannot exist. So, the sub-trees of a shortest steiner tree are the shortest steiner trees of the leaf nodes in that sub-tree and a node v.

Exact solutions using branch and bound methods have been given by [18, 29]. Of course these methods require exponential running time and cannot be used for problems of over about 10 nodes. Since some wires may have a large number of connections we have to rely on a heuristic. For larger numbers of nodes heuristics have been presented by [40, 74, 65]. The worst case length of the steiner tree found by those heuristics is $2(1-1/\#S)$ times the length of the shortest steiner tree. A complete survey on steiner trees can be found in [71].

By far the fastest heuristic of the three mentioned is [74] which is a faster version of the heuristic of [40] with a complexity of $O(\#E \log(\#V))$. However, this algorithm does not guarantee the best solution for the found topology, as our algorithm does. In particular if the paths between the leaf nodes have to pass through many nodes, the heuristic of [40] performs poorly. This is a very common situation in large chips with many circuits. The heuristic presented here performs better at the cost of a slightly higher computing time. Its performs better because it guarantees that the solution is the best possible with the given topology.

Let us introduce the notion of distance in a graph. A *distance* $d(a,b)$ is defined between each pair of nodes $a,b \in V$. Like any distance measure it satisfies the following equations:

$$d(a,b) \geq 0, \quad \text{with } d(a,b)=0 \text{ if and only if } a=b. \tag{4.1}$$

$$d(a,b) = d(b,a) \tag{4.2}$$

$$\forall_{v \in V} \left[d(a,b) \leq d(a,v)+d(v,b) \right] \tag{4.3}$$

Equation (4.3) is known as the triangle inequality. The distance between two nodes can be computed from the edge weights:

$$d(a,b) = \begin{cases} w(a,b) & \text{if } (a,b) \in E \\ \min\limits_{v \in V} d(a,v)+d(v,b) & \text{otherwise} \end{cases} \tag{4.4}$$

We can now introduce the *length* of any graph with nodes in V. Let $H(W,F)$ be a graph with nodes $W \subset V$. Since the distance measure that was introduced above applies to the nodes of V, each branch of F can be assigned an edge weight d, and the length of the graph $H(W,F)$ is

$$L(H) = \sum_{(a,b) \in F} d(a,b) \tag{4.5}$$

Two interesting special cases exist. In the case that $\#S=\#V$ there are no steiner nodes and the problem becomes the *shortest spanning tree* *problem*. If #S=2 the problem becomes the *shortest path problem*. For both problems efficient polynomial algorithms exist [17]. We will use both algorithms in our heuristic.

Algorithm 4.1. Shortest Path

```
input: $C=\{a,b\}$; $\mathcal{G}(V,E)$;
output: $P$;
$P:=(C,\emptyset)$;
repeat $v : (a,v)\in E \wedge d(a,v)+d(v,b)=d(a,b)$;
        $P := P \cup (\{v\}, \{(a,v)\})$;
        $a := v$;
until $(v,b)\in E$;
$P:=P \cup \{(v,b)\}$;
```

The two node steiner tree problem can be solved by the shortest path algorithm. A *path* $P_{a,b}$ from node a to node b is a connected acyclic subgraph with $\forall_{p\in P\setminus\{a,b\}}\, {}^\circ p=2$ and ${}^\circ a=1$ and ${}^\circ b=1$. The distance from b to all other nodes in the graph can be computed using equation (4.4). When these distances are known, it is simple to find a path connecting the two leaf nodes (See algorithm 4.1). In our steiner tree heuristic this algorithm is used to find the paths between the leaf nodes and the steiner nodes. Notice that the shortest path is not completely determined by the begin and end node of the path.

4.2 The topology of the steiner tree

Our heuristic is based on a hierarchical representation of the topology of the steiner tree. Each hierarchy can be represented by a tree. Alternatively, each tree can be represented as a hierarchy. We will call this hierarchy the topology of the steiner tree. The topology is found by the shortest spanning tree algorithm. This algorithm always finds the best spanning tree. This does not guarantee that the topology is the best possible, but it is a good heuristic.

The *topology* of the steiner is a binary tree $\mathcal{T}(N,B)$, where N denotes the set of nodes and B the set of edges of $\mathcal{T}$ called the *branches* of the topology tree. The nodes N consist of the leaf nodes $S\subset N$ and the *junction nodes* $J=N\setminus S$. The tree is *binary* because $\forall_{j\in J}[{}^\circ j=3]$ and $\forall_{s\in S}[{}^\circ s=1]$. The number of nodes is $\#N = 2.\#S - 2$. Except for the case that #S=2 no branches connect two leaf nodes: $B \subset J{\times}N$

The *steiner assignment function* ξ:N→V determines the steiner nodes in the routing graph. The leaf nodes are always mapped to themselves: $\forall_{s \in S}\xi(s)=s$. The junction nodes can be viewed as potential steiner nodes. The branches in $\mathcal{Y}$ stand for paths in the routing graph. The steiner tree is the union of those paths. The steiner tree with assignment ξ is

$$\mathcal{T}_\xi = \bigcup_{(a,b)\in B} P_{\xi(a),\xi(b)} \tag{4.6}$$

The *topology class* C of a topology $\mathcal{Y}$ is the set of steiner trees that can be realized by choosing an assignment function ξ and paths between the leaf and steiner nodes:

$$C(\mathcal{Y}) = \bigcup_{\xi}\{\mathcal{T}_\xi\} \tag{4.7}$$

It is possible that two junction nodes are assigned to the same node in V or that they are assigned to a leaf node. In this case a path disappears, and the steiner tree has less then the maximum number of steiner nodes #J.

The topology tree $\mathcal{Y}$ has a *root* $r\in S$. By defining a root, the tree can represent a hierarchy. The neighbors of the nodes $j\in J$ will be represented by left(j), right(j) and parent(j). For the nodes $s\in S$ only parent(j) is defined. The edge (n,parent(n)) always points towards the root of the tree, except when n is the root itself. left(j) and right(j) are the other two edges of a junction node. It is not important which is which.

A variation of the spanning tree algorithm, the nearest neighbor algorithm, determines the topology of the steiner tree (See figure 4.2). The graph for which the shortest spanning tree is determined is the complete graph (S,S×S) with edge weights d:S×S→$\mathbb{N}$. Instead of producing the spanning tree the algorithm constructs a binary topology tree $\mathcal{Y}$.

The algorithm finds the shortest edge connecting a node $p\in S$ already in the tree to a node $q\in S$ not yet in the tree. However, instead of simply adding an edge, it removes the old edge connecting p to $\mathcal{Y}$, and adds a three edge star with junction node j_i (See figure 4.3).

There exists a steiner tree $\mathcal{T}$ with topology $\mathcal{Y}$ found by algorithm 4.2 that has a length that is not longer then 2(1−1/#S) times the length of a shortest steiner tree $\mathcal{T}_S$:

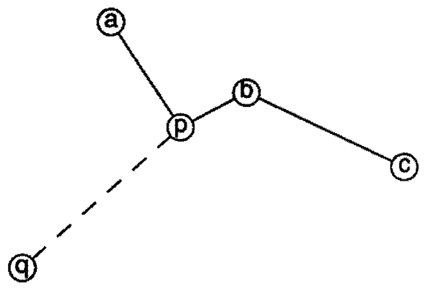


Figure 4.2. The spanning tree is extended with the shortest edge (p,q) that connects a node in the spanning tree (p) with one outside the spanning tree (q). The new edge is dashed.

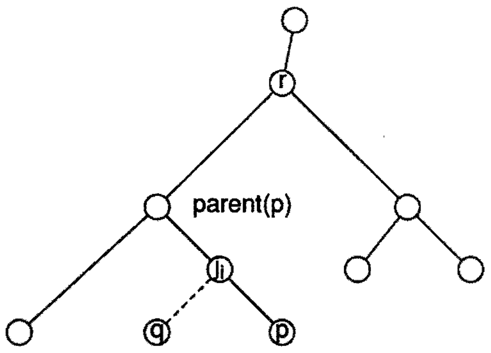


Figure 4.3. When a new edge (p,q) of the shortest spanning tree is found the topology $\mathcal{Y}$ is extended with an extra junction node j_i.

$$\exists_{\mathcal{T} \in C(\mathcal{Y})}[L(\mathcal{T}) \leq 2(1-1/\#S).L(\mathcal{T}_S)] \tag{4.8}$$

We will show this by showing that the length of $\mathcal{T}$ cannot be longer then the length of the shortest spanning tree. The assignment $\xi'(j_i)=p_i$ maps all junction nodes to the leaf nodes from which they were derived (The index refers to the index used in algorithm 4.2). Therefore ξ':N→S makes $\mathcal{Y}$ into a shortest spanning tree of (S,S×S). The length of this spanning tree is

Algorithm 4.2. Spanning tree

```
**input**: $d: S \times S \rightarrow \mathbb{N}$;
**output**: $\mathcal{Y}(N,B)$; r;
$i := 0$;
$r, q \in S : \forall_{(x,y) \in S \times S} [\, d(r,q) \leq d(x,y) \,]$;
$N := \{r, q\}$;
$B := \{ (r, q) \}$;
$R := S \setminus \{r, q\}$;
parent(r) := q;
parent(q) := r;
**while** $R \neq \emptyset$ **do**
**begin** $i := i + 1$;
          $(p,q) \in ((N \cap S) \times R) : \forall_{(x,y) \in ((N \cap S) \times R)} [\, d(p,q) \leq d(x,y) \,]$;
          $N := N \cup \{q, j_i\}$;
          $B := (B \setminus \{ (p, \text{parent}(p)) \}) \cup \{ (\text{parent}(p), j_i), (j_i, p), (j_i, q) \}$;
          $R := R \setminus \{q\}$;
          **if** p=r **then**
          **begin** parent(parent(r)) := $j_i$;
                    parent($j_i$) := r;
          **end**
          **else** parent($j_i$) := parent(p);
          parent(p) := parent(q) := $j_i$ ;
**end**;
```

$$L = \sum_{\{(a,b) \in B \,|\, \xi'(a) \neq \xi'(b)\}} d(\xi'(a), \xi'(b)) \tag{4.9}$$

A steiner tree with this assignment and using only shortest paths has the length of this spanning tree. So there exists a steiner tree with the topology $\mathcal{Y}$ as found by algorithm 4.2 that is not longer than L.

The shortest spanning tree of S×S cannot be longer than twice the length of the steiner tree $\mathcal{T}_S$. Suppose a steiner tree $\mathcal{T}_S$ is given. It is possible to visit every leaf node while crossing each edge in $\mathcal{T}$ twice. (See figure 4.4)

This corresponds to a cycle in S×S with twice the length of the steiner tree. By removing one branch of this cycle it becomes a spanning tree. The minimum length of the longest branch is 1/#S times the length of the spanning tree. Therefore the length of the spanning tree is 2(1−1/#S) times the length of the steiner tree.

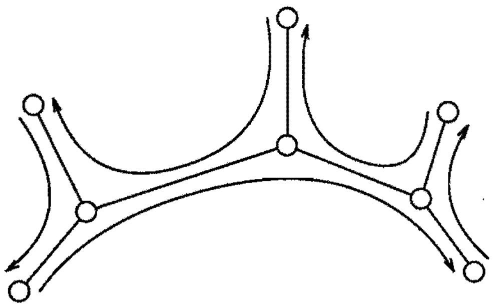


Figure 4.4. A cycle visiting all leaf nodes has a length of at most twice the length of the steiner tree. One edge may be left out to create a spanning tree.

In the next section we will show that it is possible to find the optimal assignment ξ for a given $\mathcal{Y}$in polynomial time.

4.3 Applying the paradigm

In the second stage the optimal assignment ξ is found, that is, an assignment is found such that

$$L'(\mathcal{T}) = \sum_{(a,b)\in B} d(\xi(a),\xi(b)) \tag{4.10}$$

is minimal. If a shortest steiner tree is in the topology class of $\mathcal{Y}$ then the result will be a shortest steiner tree. In any case the tree that is returned is not longer than the length of the shortest spanning tree. Notice that equation (4.10) does not necessarily represent the length of the corresponding steiner tree $\mathcal{T}$. Some of the paths may overlap, so that they counted twice in L': $L(\mathcal{T}) \leq L'(\mathcal{T})$.

The second stage is an optimization stage similar to the algorithm of [18]. However we do not enumerate all possible topologies. Only the topology of $\mathcal{Y}$ will be considered. Because of this restriction the algorithm is polynomial.

The *sub-tree* of a node $j \in J$ is the part of the topology tree that is hierarchically under that node, that is, the part that can be reached through left(j) and right(j). The function $\lambda : N \times V \to \mathbb{N}$ gives for the sub-tree of each node in J the length of that tree when connected to an arbitrary node $v \in V$. So $\lambda(\text{parent}(r), r) = L'(\mathcal{T})$. The function λ can be computed using the following recursive definition:

$$\lambda(n,v) = \begin{cases} d(n,v) & \text{if } n \in S \\ \min\limits_{i \in V} d(v,i) + \lambda(\text{left}(n),i) + \lambda(\text{right}(n),i) & \text{if } n \in J \end{cases} \tag{4.11}$$

Notice that to compute λ for each node $v \in V$ the best steiner node for each sub-tree is determined.

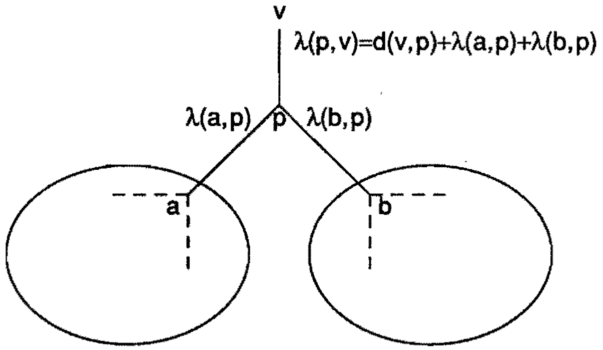


Figure 4.5. λ is computed recursively.

This algorithm consists of two depth first visits to the tree. The predictor computes λ. The adaptor traces back how this optimum was constructed, while choosing the steiner nodes.

Since this algorithm always finds an optimal assignment, it is immaterial what node $r \in S$ is used as the root node. This does not need to be the same node as in algorithm 4.2.

One way to look at this optimization is as a 'contraction' of the spanning tree. While the spanning tree contracts, the steiner nodes move away from the leaf nodes that they were assigned to by ξ'.

Algorithm 4.3. Recursive optimization of the assignment.

```
input: d: $S \times S \rightarrow \mathbb{N}$;$\mathcal{T}(N,B)$;r;
output: $\mathcal{T}$;

procedure predictor(n);
begin if $n \in S$
      then $\forall_{v \in V}[\lambda(n,v) := d(n,v)]$
      else begin predictor(left(n)); predictor(right(n)); end;

      $\forall_{v \in V}[\lambda(n,v) := \lambda(\mathrm{left}(n),v) + \lambda(\mathrm{right}(n),v)]$;
      $\forall_{v \in V}[\lambda(n,v) := \min_{w \in V} \lambda(n,w) + d(v,w)]$;
end;

procedure adaptor(n, k);
if $n \notin S$ then
begin v : $\lambda(n,k) = \lambda(n,v) + d(v,k)$;
      $\xi(s) := v$
      adaptor(left(n),v);
      adaptor(right(n),v);
end;

begin predictor(parent(r));
      adaptor(parent(r),r);
end;
```

4.4 An efficient implementation

Most of the time is spent in computing the distances d and λ. Once d and λ are known, the optimal assignment and the shortest paths are easily reconstructed. Note that both d and λ are computed by a minimization over the entire graph. d is computed using equation (4.4) and λ is computed using the minimization of equation (4.11). This minimization can be done by a single algorithm. This algorithm computes a function $\beta(v)$ which can either represent λ or d. Given an initial $\beta(v)$ the function computes

$$\beta'(v) = \min\{\beta(v)\} \cup \{w(v,u) + \beta'(v) \mid u \in V\} \qquad (4.12)$$

This a

Algorithm 4.4 is the wavefront expansion algorithm which computes β. It assumes that for some nodes the shortest distance is given. With the triangle inequality (4.3), β can be computed for the other nodes. This is

Algorithm 4.4. Wavefront expansion

```
    global: $\mathcal{C}$(V,E);$\beta$:V$\rightarrow$N ; heap;
    repeat getheap(v);
3           for all (v,u)$\in$E do
            if $\beta$(u)>$\beta$(v)+w((v,u)) then
            begin $\beta$(u):=$\beta$(v)+w((v,u));
6                 putheap(u);
            end;
    until heap=$\varnothing$
```

done starting at the node with the smallest β. The β of the neighbors of this node is computed, then the β of their neighbors, etc.. Both algorithm 4.2 and algorithm 4.3 can use this efficient wavefront expansion algorithm. It uses a heap for an efficient implementation of the selection of the smallest element of a set.

The algorithm considers each edge once and the heap operations have complexity O(#V). The complexity of the algorithm is O(#Elog(#V)). The routing graph $\mathcal{C}$ is planar so #E $\leq$ 3#V $-$ 6, and therefore the complexity is O(#V log(#V)).

Algorithm 4.4 can be used in the shortest spanning tree algorithm and for the computation of λ. In the shortest spanning tree algorithm, an unconnected node must be found, that is closest to the tree already formed. So, initially $\beta(v)=0$ for all nodes in the tree, and $\beta(v)=\infty$ for all other nodes. The algorithm will compute the distance to the set of nodes in the tree, beginning with the nodes closest to the tree. In the computation of λ, β is initialized as $\beta(v)=\lambda(\mathrm{left}(n),v)+\lambda(\mathrm{right}(n),v)$.

To improve the efficiency of the algorithm, the wavefront expansion can be limited to a part of the graph. In the spanning tree algorithm 4.2, the expansion can be stopped as soon as the distance to one of the remaining nodes in R is known. This is automatically the nearest neighbor, which is the node we are looking for.

In algorithm 4.3 λ needs only be computed for those nodes that are possible steiner nodes. We will now show that candidate steiner nodes must be located within a certain neighborhood of the sub-trees that they connect.

Consider the sub-tree below junction node p$\in$J of the topology $\mathcal{Y}$. Let $S_{p,\mathrm{parent}(p)}$ be the leaf nodes in this sub-tree. Let ℓ_p denote the length of the shortest spanning tree between these leaf nodes. Let v$\in$V be a candidate steiner point. Then v can only be a steiner point if

$$\lambda(\mathrm{left}(p),v)\leq \ell_p \ \wedge\ \lambda(\mathrm{right}(p),v)\leq \ell_p \tag{4.13}$$

Any node v for which $\lambda(\mathrm{left}(p),v)>\ell_p$ or $\lambda(\mathrm{right}(p),v)>\ell_p$ cannot become a steiner node for p. So the wavefront search of algorithm 4.4 can be stopped as soon as equation (4.13) is violated. This will limit the wavefront expansion to the immediate neighborhood of the nodes to be connected.

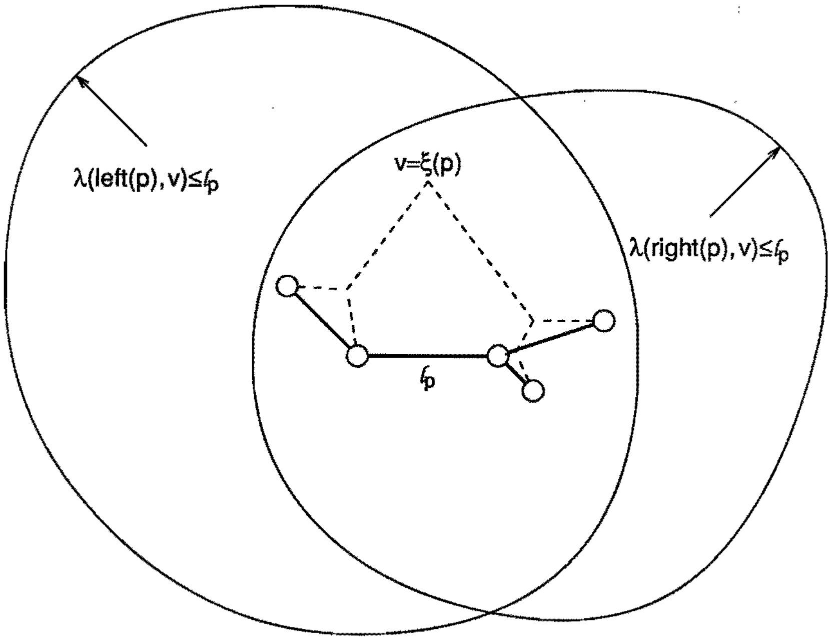

$\lambda(\mathrm{left}(p),v)\leq \ell_p$
$v=\xi(p)$
$\lambda(\mathrm{right}(p),v)\leq \ell_p$
ℓ_p

Figure 4.6. The length of the sub-trees *(dashed)* is limited by the length of the shortest spanning tree *(fat)*. The steiner node v must within both ranges.

We will derive equation (4.13) as follows: In a steiner node the two sub-trees are joined. So, for a steiner node candidate $v=\xi(p)$, λ can be computed by:

$$\lambda(p,v) = \lambda(\mathrm{left}(p),v) + \lambda(\mathrm{right}(p),v) \tag{4.14}$$

These two sub-trees that are connected through p could be built up differently. An alternative realization of the steiner sub-tree of p is the spanning tree of the nodes under p, which has length ℓ_p, plus an edge from a leaf node to p. To make this topology match a possible $\mathcal{T}$, the worst edge on either side is assumed. However, we may choose the best of the two sides. This number results in an upper bound for $\lambda(p,v)$:

$$\lambda(p,v) \le \ell_p + \min\left(\max_{q \in S_{left(p),p}} d(p,q), \max_{q \in S_{right(p),p}} d(p,q)\right) \tag{4.15}$$

The length of the worst edge to the nodes in the sub-tree $S_{left(p),p}$ is of course always smaller then $\lambda(left(p),v)$ of that subtree. Substituting equation (4.13) gives

$$\lambda(left(p),v) + \lambda(right(p),v) \le \ell_p + \min(\lambda(left(p),v),\lambda(right(p),v)) \tag{4.16}$$

from which we can derive a bound on λ for each of the sub-trees:

$$\lambda(left(p),v) \le \ell_p \wedge \lambda(right(p),v) \le \ell_p \tag{4.17}$$

5. Two-dimensional folding

Folding is a class of layout methods, like placement and routing are classes of layout methods. In folding, one-dimensional elements, called *strips*, are assigned to *tracks*.

Strips are small layout elements, often wire segments. Sometimes the strips can represent more complex compounds such as transistors and gates. These layout elements have approximately the same thickness, but varying lengths. The strips will therefore have a uniform width.

Two strips that are assigned to the same track cannot overlap. When more strips are assigned to the same track, we say that they have been *folded*. The objective is to use as few tracks as possible, thereby minimizing the area of the circuit.

In this chapter we will present a new algorithm for the two-dimensional folding problem. The algorithm uses an elegant divide and conquer heuristic. This heuristic gradually refines the folding by repeated partitioning.

Two-dimensional folding can be used for several layout problems. Usually layout styles that use some kind of folding introduce some additional constraints. In chapter 6 we will reformulate some common layout problems, which leads to some new layout styles.

5.1 The folding problem

In folding the objective is to assign strips to tracks such that the strips do not overlap and the number of tracks needed is minimal. In the simplest folding problem, all strips have a fixed left and right edge. In this case the folding problem is optimally solved for n strips in only O(n log(n)) time by the well known left edge algorithm. Often other objectives or constraints are present, complicating the problem.

The two-dimensional folding problem is the problem of assigning horizontal and vertical strips to *rows* and *columns*. The strips have an interconnection pattern that must be realized. This interconnection pattern can be thought of as a net list for the horizontal and vertical strips. Each horizontal strip has a number of vertical strips to which it is connected. A horizontal strip must reach all columns that contain strips to which it is connected. The span of a strip is therefore not fixed. Of course the strips in the same row must not overlap. But whether they overlap or not also depends on the assignment of the vertical strips to the columns.

Channel routing can be formulated as a one-dimensional folding problem. The unconstrained case can easily be solved by the left edge algorithm. For the constrained case heuristics have been given by [76] and others. Two-dimensional folding techniques have been developed for PLA's [12] where the objective was to minimize the area of the PLA. These techniques can handle certain constraints for the pins, but the possibilities of adapting the shape and pin positions are limited. Also there are several additional constraints on the order of the strips in PLA folding. Often only two strips can be folded into one column. Also the strips of the **or** and **and** planes must remain separated.

The circuit to be laid out is represented as a bipartite graph $\mathcal{F}(H,V,E)$. H is the set of horizontal strips, V is the set of vertical strips. The set of pins $E \subset H \times V$ contains the edges of the graph, that stand for the incidences between the vertical and horizontal strips.

The folding algorithm assigns the horizontal strips to columns and the vertical strips to rows, such that they do not overlap. Notice that the sequence of the incidences on a strip is completely free. The folding algorithm allows no constraints on this order, hence the name *unconstrained* folding.

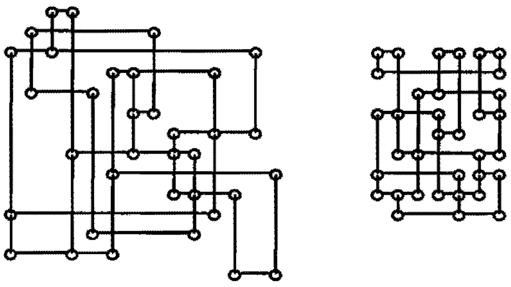


Figure 5.1. Two-dimensional folding: A circuit *(left)* with one strip per row or column is folded. The result *(right)* has strips that share rows or columns, and is therefore smaller.

We formally state the *unconstrained two dimensional folding problem* as follows: The circuit is to be realized on a grid of *rows* and *columns*. The set of grid points is represented by $\mathbb{Z} \times \mathbb{Z}$. The folding of a circuit is determined by the *column assignment function* $\phi: V \to \mathbb{Z}$ that assigns vertical strips to columns and the *row assignment function* $\psi: H \to \mathbb{Z}$ which assigns horizontal strips to rows. Let $v(s)$ denote the set of strips connected to strip s: $v(s) = \{t \in H \cup V \mid (s,t) \in E\}$. Notice that s and t are always in different sets because $\mathcal{F}$ is a bipartite graph. The *span* $\sigma \in \mathbb{Z}$ of a vertical strip $s \in V$ is an interval of rows defined as

$$\sigma(s) = [\min_{t \in v(s)} \psi(t), \max_{t \in v(s)} \psi(t)]. \tag{5.1}$$

The spans of strips that are assigned to the same column are not allowed to overlap:

$$\forall_{s_i, s_j \in V} [\phi(s_i) = \phi(s_j) \Rightarrow \sigma(s_i) \cap \sigma(s_j) = \varnothing] \tag{5.2}$$

Since the problem is symmetric the same goes for the horizontal strips. In the remainder of this chapter this duality will not be explicitly mentioned. The objective of the folding algorithm is to find a valid ϕ and ψ subject to some score function, for instance area.

The unconstrained two-dimensional folding problem was first posed in [16] and a simulated annealing solution was proposed. An improved simulated annealing algorithm was described in [73]. A new solution to this problem [81, 82] uses an elegant hierarchical divide and conquer approach similar to the approach of [7] to placement and routing of gate arrays. The algorithm allows for a large amount of freedom in choosing the aspect ratio and pin positions. Different amounts of folding in both dimensions are used to get the desired aspect ratio. For larger arrays two-dimensional folding also increases the functional density.

We may classify one-dimensional gate placement as a special case of two-dimensional folding. In the one-dimensional gate placement the column assignment function ϕ must be a permutation. The objective is to minimize the area, that is, to minimize the number of rows needed. This problem has been shown to be NP-hard [35]. Therefore, the assumption that two-dimensional folding is at least NP-hard seems valid, although proof is not provided here.

5.2 The hierarchical folding algorithm

The new algorithm is a divide and conquer heuristic. The design is repeatedly subdivided by straight orthogonal cutting lines. After each division the strips are partitioned into two *groups*. After the kth horizontal cut the horizontal strips H are partitioned into k+1 groups $H_0..H_k$.

$$H = \bigcup_{i=0}^{k} H_i \qquad \forall_{H_i,H_j} [H_i \cap H_j = \emptyset] \tag{5.3}$$

Similarly the vertical strips are partitioned into $\ell+1$ groups $V_0,..,V_\ell$ after ℓ vertical cuts. The sets are ordered in the grid space, that is, the sets imply a constraint on the functions ϕ and ψ.

$$\forall_{s \in V_i, t \in V_j} [i<j \Rightarrow \phi(s)<\phi(t)] \tag{5.4}$$

When a group V_i is partitioned into two groups V_i and V_{i+1} this implies a constraint on ϕ (although a solution remains always possible). The index of remaining groups V_j with $j>i$ increases by one. As the exact assignment has not yet been determined the span of a strip will be defined as

$$\bar{\sigma}(s) = [\ \min\{i \mid H_i \cap v(s) \neq \emptyset\},\ \max\{i \mid H_i \cap v(s) \neq \emptyset\}\ ] \tag{5.5}$$

To make a prediction of the resulting size of the array, and to evaluate the consequences of cutting line decisions, we use bounds on the number of rows or columns needed for a group of strips. The maximum number of columns needed is equal to the number of overlapping strips. In the worst case all strips that can overlap will actually overlap. An upper bound for the number of columns needed for a group of vertical strips is therefore given by

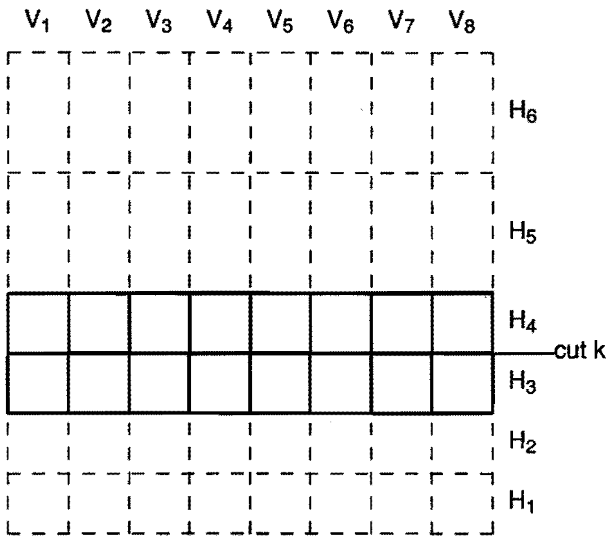


Figure 5.2. Each cut creates a 2xn subproblem, that is easier to conquer. Further cuts hierarchically divide the problem into smaller partitioning problems. Step by step the partitioning refines the assignment of the strips to rows and columns.

$$\mu(V_i) = \max_j \#\{s \in V_i \mid j \in \bar{\sigma}(s)\} \tag{5.6}$$

Notice that this is the exact number of columns if $\sigma = \bar{\sigma}$. A lower bound for the number of columns is determined by the number of strips that cross horizontal boundaries:

$$\delta(V_i) = \max_j \#\{s \in V_i \mid j \in \bar{\sigma}(s) \wedge j+1 \in \bar{\sigma}(s)\} \tag{5.7}$$

Since the incidences of the strips are not allowed to overlap there is another lower bound:

$$\gamma(V_i) = \max_{s \in H} \#(v(s) \cap V_i) \tag{5.8}$$

These upper and lower bounds are the predictor for the final size of the array. A dimension of the matrix is predicted as the mean of the upper and lower bounds. The area can be predicted as

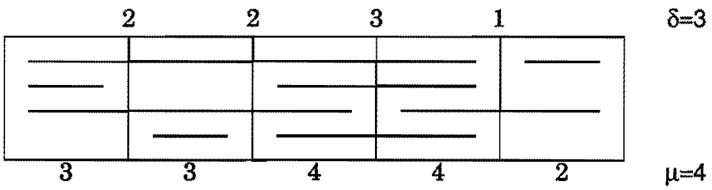


Figure 5.3. We can compute bounds for the width of a group. The upper bound μ is the maximum of the number of possibly overlapping strips in one box. A lower bound δ is the maximum of the number of strips that are cut.

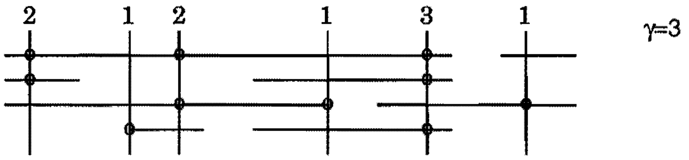


Figure 5.4. Another lower bound is γ, the maximum of number of connections to an orthogonal strip.

$$\alpha = \left(\sum_{i=0}^{\ell} \frac{\max(\delta(V_i),\gamma(V_i))+\mu(V_i)}{2}\right).\left(\sum_{i=0}^{k} \frac{\max(\delta(H_i),\gamma(H_i))+\mu(H_i)}{2}\right) \quad (5.9)$$

When the following criterion is satisfied, further cuts cannot improve the result.

$$\forall_{S\in\{V_1..V_\ell,H_1..H_k\}} [\mu(S)=1 \;\vee\; \mu(S)=\max(\gamma(S),\delta(S))] \quad (5.10)$$

When $\mu(S)\neq 1$ then the elements of S do not have a completely determined assignment function. An assignment with a minimum number of rows or columns can easily be constructed using a left edge algorithm. The left edge algorithm assigns two strips with an overlapping span to different columns.

$$\forall_{v_1,v_2\in V_i}[\bar{\sigma}(v_1)\cap\bar{\sigma}(v_2)\neq\emptyset \Rightarrow \phi(v_1)\neq\phi(v_2)] \quad (5.11)$$

This left edge assignment realizes the upper bound μ. This upper bound, and also the left edge algorithm, assume that all strips that can overlap, actually do overlap. Notice that it is now possible to exchange the columns within a group without creating any overlaps between horizontal strips. This freedom can be used to further optimize the assignment with respect to other criteria.

The shape is controlled by selecting the orientation of the next cutting line. A vertical cutting line tends to make the array wider and lower, a horizontal line makes the array higher and narrower. How this is used to control the shape will be explained in chapter 7.

5.3 Partitioning the groups

When a cut is made a group of strips is partioned into two groups. The partitioning problem can be formulated as a combinatorial problem: Given a group $S \subset H$, partition this set into two groups $P \subset S$ and $S \setminus P$ such that a *score function* $\varepsilon(P)$ is minimal. Since each element of S can be either in P or not, the number of possible partitions, or *states*, is $2^{\#S}$. The state is fully determined by the subset P, therefore the score function ε is a function of P. Since little is known about the score function ε, the general problem is NP-hard. Therefore we will resort to heuristics. For these heuristics, we will assume that there is some correlation between the scores of neighboring states. States are neighbors of they differ in at most one element: $\#(P_1 \cap P_2) = \#(P_1 \cup P_2) - 1$. We assume that in that case $\varepsilon(P_1) \approx \varepsilon(P_2)$. The score functions that were used are all simple arithmetic functions of μ, γ and δ. These three measures satisfy the correlation assumption. A score function that proved to be very effective was equation (5.9).

We will consider three strategies for partioning a subset of strips. All strategies start with an arbitrary partition, and transfer strips between the two subsets while trying to minimize the score function. The strategies differ however in the control of this process. The first strategy is based on a strategy as used in the mincut algorithm of [37]. It is a very effective method for escaping from local minima, but it is rather time consuming. The second strategy, based on iterative improvement, is faster but can easily get stuck in a local minimum. Finally, an attempt is made to combine the best features of both in a two stage algorithm.

The first strategy performs several *passes* while trying to improve the initial partition. Initially each strip is assigned to an arbitrary subset. It transfers strips one by one to the other subset in an attempt to find a better solution. An element which has been transferred becomes *locked* and cannot be moved any more. When all strips have become locked the algorithm has completed a pass. The algorithm performs several passes until no improvement can be found. Each time the initial state of a pass is the best state encountered so far. A pass is continued even when the current state becomes worse. This helps to escape from local minima.

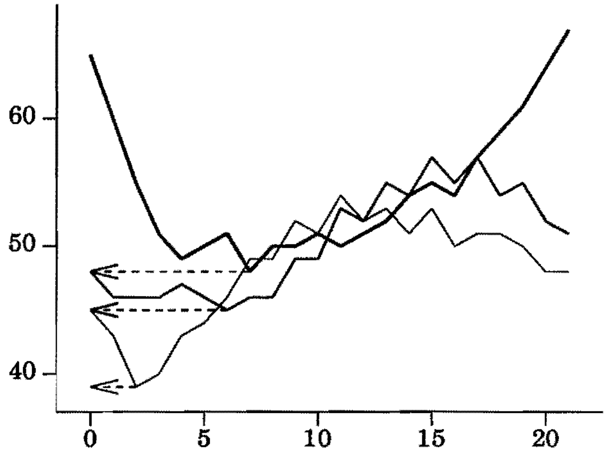


Figure 5.5. The first partitioning strategy performs several passes. The score is plotted during each of three passes. The last pass *(not shown)* did not give an improvement. The best state of the previous pass is the initial state of the next pass. Notice that the state is pushed out of the local minimum during the first and second pass.

To prevent all elements from moving to one set, elements are not allowed to be selected from a set which contains less than ⅓ of the total number of elements. The algorithm continues to transfer strips, until no valid move is possible (all elements are locked or all unlocked elements are in a set smaller than ⅓). In practice only a few passes are necessary to arrive at a minimum. This is reached when the initial state of the pass is the best state encountered.

The number of passes needed in practice is not very large. At least two passes are needed. The last of those two is needed to make sure that no further improvement can be found. For large numbers of strips (>100) 5 passes or so are needed. We will assume that the number of passes is

Algorithm 5.1. Partitioning

1. Begin with an arbitrary partition. Save this initial state.
2. Begin of a pass: unlock all strips.
3. Is one set smaller than ⅓? If yes then limit the following selection to members of the larger set.
4. Select the strip that gives the lowest increase or largest decrease of the score function.
5. Transfer that strip and lock it.
6. Is the present state better than the saved state? If yes then save the present state.
7. Are there still unlocked strips? And are they in a set larger than ⅓? If yes then go to 3.
8. The best state becomes the initial state of the next pass. Did we encounter a state that was better than the initial state? If yes then go to 2.

constant, as is assumed in [22], although O(log(#S)) may be more realistic.

Let O(ε) be the complexity of the computation of the score function. The algorithm transfers each strip exactly once, so #S transfers are done. Before a transfer is done, the best strip to be transferred is found, by trying out all #S strips. The complexity of one pass of algorithm 5.1is O(#S^2).O(ε). The computation of the score function involves the recomputation of the bounds μ,δ and γ. The computation of these bounds is rather costly. Using some redundant data structures, which are updated after each move, the effort can be reduced. The complexity of computing those bounds by updates is determined mainly by the number of subsets perpendicular to the subset being partitioned (k or ℓ depending on the orientation).

5.4 Performance improvements

One of the most time consuming features of algorithm 5.1 is determining the *best* strip to transfer in step 4. This is done by simply trying out *all* strips. Step 4 has a complexity of O(#S).O(ε).

In stead of finding the best strip, algorithm 5.2 accepts any strip that gives an improvement. Often, we do not have to search very long to find such a strip. Although the worst case complexity of algorithm 5.2 is very bad, it proves to be much faster then algorithm 5.1. The disadvantage of this algorithm is that it never accepts a transfer that increases the score. It cannot escape from a local minimum.

Algorithm 5.2. Iterative improvement

1. Begin with an arbitrary partition. Unlock all strips.
2. Is one set smaller than ⅓? If yes then limit the following selection to members of the larger set.
3. Select an unlocked strip at random.
4. Transfer that strip.
5. Is the present state better than the previous state? If no then transfer the strip back and lock it. If yes then unlock all strips.
6. Are there still unlocked strips? And are they in a set larger than ⅓? If yes then go to 2.

In [22] an implementation of algorithm 5.1 is described that uses the object function (Remember that we assumed that $S \subset H$ is a horizontal group):

$$\varepsilon = \#\{s \in V \mid k \in \bar{\sigma}(s) \wedge k+1 \in \bar{\sigma}(s)\} \tag{5.12}$$

With the use of special data structures it is possible to make the total effort needed for updating the date structures is linear with the number of incidences in the subset $O(\#S \times V \cap E)$ during one pass. Unfortunately this method is specific for the score function of equation (5.12). This score function tends to reduce $\mu(V_i)$ and $\delta(V_i)$ but ignores $\mu(H_i), \delta(H_i)$ and $\gamma(H_i)$. ($\gamma(V_i)$ cannot be influenced by the partitioning process). In particular, it is sometimes possible that the algorithm finds the same partition during the first horizontal and the first vertical cut. In that case, most incidences will be in diagonally opposing quadrants.

To avoid this effect, some extra strips are introduced temporarily during the partitioning. They must ensure an even distribution of the strips in the groups. An extra vertical strip was added between each non-overlapping pair of horizontal strips. This way there is a penalty for each pair of strips that could be assigned to the same track, but is assigned to different partitions. Notice that these strips are added only during the partitioning algorithm, and only influence the score function. Unlike the regular vertical strips, the temporary strips do not imply overlap.

So, first some extra strips were added between non-overlapping strips. Then algorithm 5.1 was used with the score function of equation (5.11). Then the temporary strips are removed, and algorithm 5.2 is used with score function equation (5.9) to 'fine tune' the partition to the true objective.

The quality of the solutions of the folding can be improved by remerging and repartitioning groups. When no improvement can be made by further partitioning groups, a second optimization stage is done. In this stage, groups are remerged if this is possible without increasing the score function. Some groups that were separated at a high level can now be merged. By repartitioning these groups, sometimes an improvement can be made. In this way, old partitions, that were taken at an early stage, can be partly reconsidered with more information available. The second stage optimization yields an improvement of a few rows and/or columns for both small and large circuits. Of course, remerging and repartitioning can be done at every level of the hierarchy.

5.5 Some practical experiences

First, we will look at the performance of the algorithm in terms of computational speed and area reduction. Table 5.1 shows the experimental results for the three partitioning strategies. Three different examples were used. For each strategy and each example circuit the final size and the CPU time on an Apollo DN3000 in seconds is given. The CPU times for algorithm 5.1 are much higher then for algorithm 5.2 but the results are much better. The combination of both (third column) seems to be a good compromise.

Table 5.1. Comparison of the three partitioning algorithms.

example	alg 5.1		alg. 5.2		5.1;5.2	
	time	area	time	area	time	area
cnt4	642.57	2385	26.50	3540	49.92	2236
four	213.33	891	14.42	1739	29.52	986
five	1895.05	3534	74.15	5727	128.83	4260

To test the stability of the algorithm, choices for which the algorithm has no preference were randomized. Several runs of the program, with identical input will generally result in different solutions. The folding algorithm has been applied to a 79 by 74 strips example. A sequence of ten runs with randomized choices resulted in an average area of 818 units. The standard deviation of the sequence was 61, or 7.45%. The best and worst solutions have areas of 729 and 930 units.

The algorithm has been applied to dense regular connection patterns of which the optimal solution is known. Notice that these examples are not typical. The algorithm finds solutions that are about a factor 2 worse than the optimum.

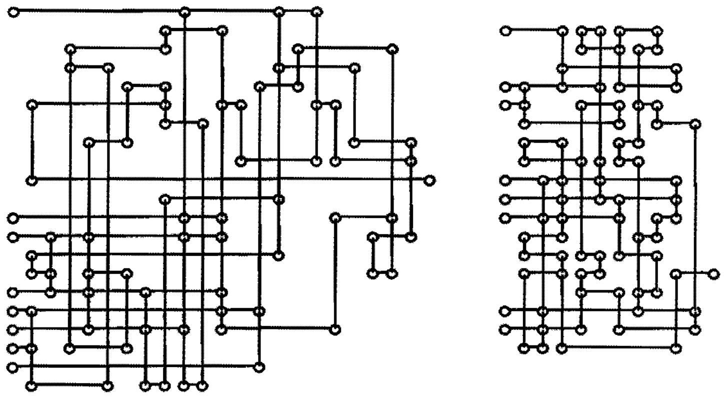


Figure 5.6. The result of the simulated annealing algorithm *(left)* compared to the result of the new algorithm *(right)*.

We compared the new algorithm to the simulated annealing algorithm of [16]. The following example was taken from that paper as a bench mark. The circuit contains 36 vertical strips and 28 horizontal strips. The pins were required to be at the same side as in the given example.

The result of [16] is given in figure 5.6 and uses 21 rows and 21 columns. The result of our algorithm uses 18 rows and only 10 columns, an improvement of 59%.

6. Layout of unidimensional modules

In this chapter we will present two new layout styles that are based on two dimensional folding. These styles are well suited for laying out net lists of small circuits that can be represented as strips in the folding algorithm. The folding creates a wire plan, to which the pin positions of the primitive circuits are adapted.

The first style is called *transistor matrix*. In this style the horizontal strips are transistors. The style is somewhat related to gate matrix which also lays out transistor net lists.

In the second style, called *nor matrix*, the horizontal strips are nor gates. The nor gates are implemented as static nMOS logic. This style is related to the PLA style, and even more to the weinberger array style. Like the weinberger array, this style can handle multi level logic. In both styles the vertical strips are the nets.

6.1 The wire plan

In both new layout styles, the *circuit* to be laid out is represented by a bipartite graph $\mathcal{N}(C \cup \{C\}, N, P)$ called the *net list*. In this graph C represents the *primitive circuits* and N represents the nets. The edges of the graph P represent the pins of the circuits, or the circuit net incidences. Nets that are connected to the *super circuit* C are I/O nets, so circuit C represents the periphery. In the first layout style, the primitive circuits are transistors. In the second, the primitive circuits are nor gates.

To use folding, a circuit should consist of interconnected horizontal and vertical strips. It is represented by a *folding graph* $\mathcal{F}$(H,V,E), in which H and V represent the horizontal and vertical strips. The edges E of the folding graph represent the connections between the horizontal and vertical strips. When strips are *folded*, they share a row or column. The folding algorithm (see chapter 5) folds the strips to save area.

The new layout styles are based on equating the folding graph $\mathcal{F}$(H,V,E) with the net list graph $\mathcal{N}$(C,N,P). The vertical strips are the nets, and the horizontal strips are the primitive circuits. The pins of the circuits are the incidences between the horizontal and vertical strips. The layout problem is now a two dimensional folding problem: the circuits and nets are folded, such that all connections are made while the strips do not overlap.

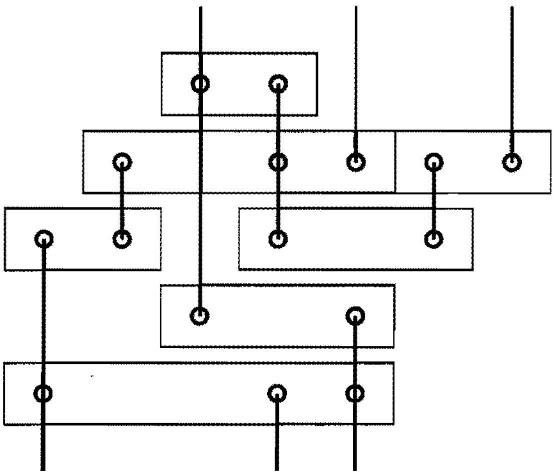


Figure 6.1. The wire plan is designed by the folding. The rectangles are the primitive circuits, are designed to match the pin positions. In the two styles presented here, the primitive circuits are either transistors or nor gates.

In previous chapters we looked at the design of circuits with an internal decomposition. First a *floor plan* was designed. Then the sub-circuits were designed according to this plan.

The circuits considered here also have an internal decomposition. The folding process could be regarded as a planning phase. The primitive circuits, transistors and nor gates, are then designed according to this plan. Because this plan only determines a wiring pattern, it is called the *wire plan*. Together with the folding graph, the assignment functions ϕ and ψ determine the wire plan of the circuit.

A circuit that uses folding is in an intermediate position in the hierarchy. Not only does it consist of sub-circuits, the circuit itself is a sub-circuit in a floor plan. So the wire plan of the circuit must be adapted to this floor plan. The pins of the circuit are positioned such that the wires in the floor plan remain short. The number of rows and columns is also determined by the floor plan (see chapter 7).

Like the wire plan is adapted to the floor plan, the primitive circuits are adapted to the wire plan. Specifically, the wire plan determines the positions of the pins of the primitive circuits.

To represent the periphery of the circuit C, four new strips are introduced $n,s \in H$ and $e,w \in V$ that represent the sides of the circuit. The folding algorithm assigns the side strips to a predetermined side:

$$\forall_{h \in H \setminus \{s,n\}} [\psi(s) < \psi(h) < \psi(n)] \tag{6.1}$$
$$\forall_{v \in V \setminus \{w,e\}} [\phi(w) < \phi(v) < \phi(e)] \tag{6.2}$$

It does that by introducing a special group for each of these strips at the beginning of the folding.

A pin for a net $v \in V$ at the top or bottom of the matrix is represented by an edge (v,s) or (v,n). For a pin that connects to the left or right side of the matrix an additional horizontal strip is introduced. Lets say that this new strip h carries the net from strip v to the left side w of the matrix. Two new edges are introduced: (v,h) and (h,w). The floor planner determines which net is assigned to which side.

Most cell generators found in the literature have no or very limited possibilities of adapting the pin positions. In [45, 10, 58] pin positions are adapted by stretching the circuit. Only the distances between the pins can be specified, but not the order. Terminals that can be assigned to any side in any order are allowed by [43]. So far most circuit generators that have this capability rely on placement and routing [48].

The layout algorithms commonly used for gate matrices and weinberger arrays reduce the layout problem to a restricted two dimensional folding problem. The vertical assignment function ψ is required to be a permutation. Since each vertical strip uses a whole column, the horizontal assignment function ϕ can be found using the left edge algorithm. Finding the optimal permutation ψ is NP-hard [35], but an efficient branch and bound algorithm is possible [2]. Most papers about weinberger arrays or gate matrix layout describe heuristics to find the permutation ψ.

6.2 Transistor matrix layout

The first layout problem that we will treat is the layout of a transistor circuit. The specification of the circuit is a net list of transistors.

A layout style called *gate matrix* [42] is commonly used for this problem. In this style, only one polysilicon strip is used per net for all transistor gates. All transistors that share the same gate net, are required to use that strip as their gate. The gate matrix style got its name from those gate strips. Other connections in the circuit are made using horizontal metal strips.

An automatic method for the gate matrix style was first proposed in [69]. Since then many algorithmic approaches to this problem have been published [70,13,19].

Gate matrix layout requires transistors with the same gate net to be on the same polysilicon strip. This requirement results in complex optimization problems and hampers efficient compaction. By dropping this requirement we arrive at a more elegant and symmetrical optimization problem: the two dimensional folding problem.

Like the gate matrix style, this new layout style can handle any circuit of transistors, while the transistor sizes may vary. Each transistor can have different electrical parameters. Logic with pass transistors or special buffers and latches can easily be handled. Even non-critical analog circuits can be laid out, but there is no control over the parasitics.

In addition, it is more area efficient than the gate matrix style and it can control the shape and pin positions accurately. Because the primitive circuits are transistors in this case we called the new style *transistor matrix layout* [82].

The horizontal strips represent the transistors and the vertical strips represent the nets. The nets are implemented in metal, while the transistors use diffusion and polysilicon. The folding algorithm assigns the nets to columns and the transistors to rows, such that they do not overlap.

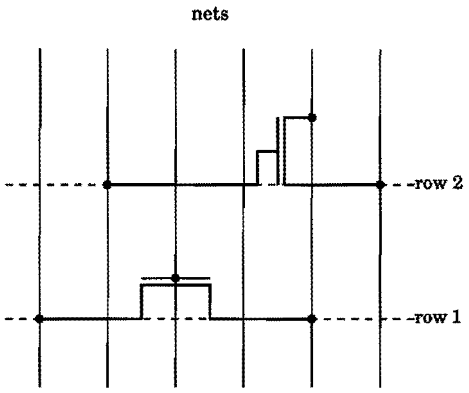


Figure 6.2. Transistor matrix layout: The transistors are assigned to rows, while the nets are assigned to columns. The net order determines the transistor configuration.

The order of the pins of a transistor is determined by the folding. There are no constraints on the order of the nets for the folding algorithm. Each transistor has three pins, the gate, the source and the drain. When the gate is the middle pin, a different transistor model is needed then when it is the left or right pin (see figure 6.2). Sometimes the gate is connected directly to the source or the drain. In that case, the transistor model has only two pins.

Since the folding algorithm assumes that all transistors are equally high, it is desirable that the transistors have a small and almost uniform height. The height of the transistors should preferably be close to the width of a horizontal polysilicon or diffusion wire.

6.3 Parameterized transistors

As a rule, in top-down design, design decisions are postponed as much as possible. In optimizing the design we always assumed that the sub circuits can be designed according to the specification determined by the global structure. In this way the global structure could be optimized easily, without taking all kinds of additional constraints into account. The burden of actually realizing the global specifications is moved down the hierarchy.

Finally, when the primitive circuits are designed, it is no longer possible to move the design problems down. The primitive circuits must meet their parameters without decomposition. It is however, much easier to design a single transistor according to a variety of parameters. Since there is no decomposition, there are no sub-circuits that need to be ordered. The transistor layouts are so simple that compaction is not necessary. The coordinates of the rectangles of the layout can be calculated as simple arithmetic expressions of the parameters.

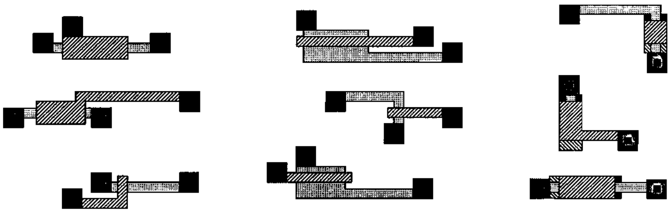


Figure 6.3. Some examples of automatically generated nMOS transistors. The transistors are designed according to geometrical, electrical and design rule parameters.

We can distinguish the following classes of parameters for the transistors: electrical, geometrical and design rule parameters. The electrical rules deal with the function and speed of the circuit. For an nMOS transistor they are the width and length of the transistor, which determine the drain current factor and the gate capacitance, and an optional diffusion implant, which controls the threshold voltage V_T.

The geometrical parameters are in the first place the positions of the pins of the transistor. It is essential for the folding that each pin can be assigned to any position. Furthermore, various shapes are possible with the same pin positions. The active area can be oriented horizontally or vertically, or the transistor can be mirrored in the X axis, to decrease the occupied area.

Finally, the design rule parameters are the minimum feature sizes of the transistor. As no compaction is involved in the generation of the layout of a single transistor, these parameters are handled easily.

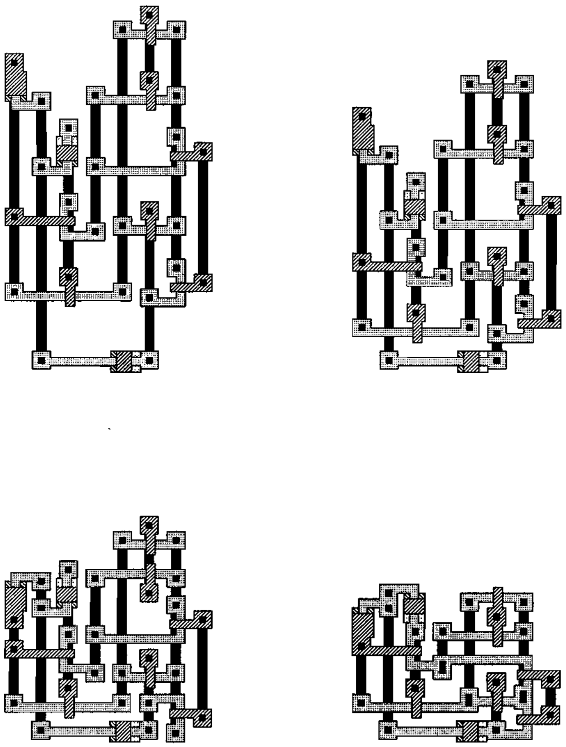


Figure 6.4. In this figure four methods of compaction are compared. Grid based compaction with a fixed grid *(top left)* and with a variable grid *(top right)* is not very efficient. Therefore the transistors are individually spaced *(bottom left)* and pins of the same layer are allowed to overlap *(bottom right)*.

For the compaction of the matrix, the transistors are modeled by the area they occupy in the polysilicon layer and the diffusion layer. This area is represented by a polygon. The minimum separation rule between the transistors is implemented by adding a margin of half this design rule on all sides.

Pins of different transistors are allowed to overlap, if they are connected to the same net and use the same layer. Therefore the diffusion pins only occupy area in the polysilicon contour, and polysilicon pins only occupy area in the diffusion contour.

The compaction works from the bottom to the top while it places the transistors one by one. It uses a grid for the nets, but the transistors are spaced individually (see figure 6.4).

During the compaction two contours, one for polysilicon and one for diffusion, are maintained to indicate the occupied area. The shape of the transistors can be designed according to the shape of the contours. Of course the selections of the transistors are mutually dependent, but no decisions are traced back.

6.4 Track assignment

In § 5.2 we mentioned that track assignment can be done optimally with the left edge algorithm. This algorithm ensures that the upper bound μ is realized. The left edge algorithm places two strips with an overlapping span $\bar{\sigma}$ on different tracks (equation 5.11).

Since the left edge algorithm uses the span $\bar{\sigma}$ to determine overlap between strips, the assignment in a group has no consequences for orthogonal groups. A *box* is a part of the grid that belongs to two orthogonal groups (see figure 5.2). Within a box no tracks are shared. Notice that it is now possible to exchange the tracks within a group without creating any overlaps between orthogonal strips. This freedom can be used to further optimize the net assignment with respect to another criterion.

Often the transistors in the matrix have either a long or a wide active area. These sizes can be much larger then the width of the wires. Since the height of the transistors must be approximately equal, a horizontal orientation of the active area is desired. When the pins of the transistor are sufficiently far apart, enough space is available to allow very wide or very long transistors. However, when the pins are close together, a vertical orientation is necessary. This may lead to large empty areas in the matrix.

The columns can be reordered to create sufficient space between the pins. When the columns within a group are reordered, this does not cause overlaps between transistors in the same row. The objective is to find a column sequence that will allow a horizontal orientation for many transistors.

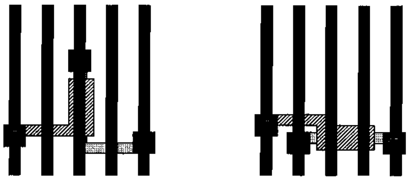


Figure 6.5. The influence of exchanging two columns: the transistor to the left does not have enough space to lie flat. To the right, enough space between the pins is created for a flat implementation by exchanging two nets.

Let for each transistor t the nets connected to gate, drain and source be $\{g_t, d_t, s_t\} = v(t)$ and let $w(t), l(t)$ denote the width and length of t. The number of columns needed by transistor t is

$$S(t) = \text{entier}\left(\frac{\max(w(t)+c_1, l(t)+c_2)}{p} + 1\right). \tag{6.3}$$

p is the metal pitch and c_1 and c_2 are constants. An assignment that allows sufficient room for all transistors satisfies:

$$\forall_t \ |\phi(s_t) - \phi(d_t)| \geq S(t) \tag{6.4}$$

Such a perfect assignment may not always be possible. In that case a solution is preferred that leads to a small number of violations. In a vertical orientation the size of the transistor determines the height of the row. A violation has a larger influence when the size of the transistor is larger. We therefore optimize the sum of sizes of the transistors that cannot be placed horizontally:

$$\sum_{\{t \mid S(t) > |\phi(s_t) - \phi(d_t)|\}} S(t) \tag{6.5}$$

Then columns are reordered to allow more horizontal transistors. This problem can be solved exactly using a dynamic programming technique. Its complexity grows exponentially with $\max_i \mu(N_i)$, so it is only efficient when the partitions are small. To avoid this exponential growth, a smaller subset of partial solutions could be retained as a heuristic.

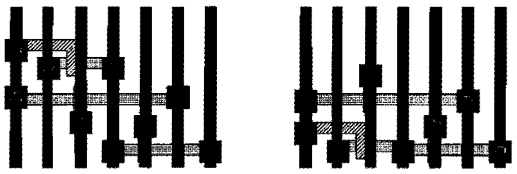


Figure 6.6. Reassignment of transistors to rows. Because the terminals that use the same layer are allowed to overlap, one track less is needed.

Another optimization is possible by reassigning the transistors of a group to rows. If the pins of two transistors connect to the same and if they use the same layer, they are allowed to overlap. This means that the lower bound γ as defined in the previous section is not always completely valid for estimating the width of a partition of transistors.

The left edge algorithm that assigns the transistors to rows uses σ, not $\bar{\sigma}$ to determine whether transistors overlap. It looks at what columns are spanned by the transistors, not just at what groups are spanned. The algorithm needs to know the exact order of the columns. Therefore the transistor to row reassignment must be done after the column ordering.

Since a more accurate model is used, which also allows the pins of the transistors to overlap, the left edge algorithm may need less tracks then μ. It may even need less tracks then γ. Of course it cannot need less then δ tracks.

6.5 Nor matrix layout

The second layout style has nor gates as its primitive circuits. The nor gates form the horizontal strips, and the nets are the vertical strips. The horizontal strip of a gate and the vertical strip of its output signal are connected pairwise forming a conducting cross. In figure 6.7 the schematic diagram of such a nor gate is given.

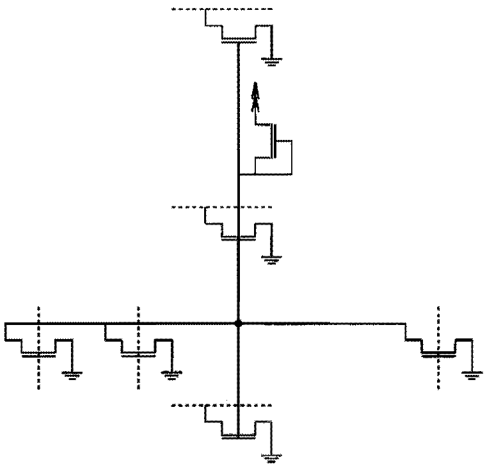


Figure 6.7. Cross of conducting strips representing the nor gate and its output net.

The nets are connected to the drains of the transistors; the sources are always connected to ground. This way the transistors function as the pull-down devices of the nor gates. The output is formed by a via, which connects the horizontal net to the vertical gate thus forming a conducting cross carrying the output net of the nor. A depletion transistor is connected to the gate as a pull-up. The pull-ups of several nors are placed in rows to allow them to be connected by a power net.

This layout style has a number of features, which make efficient folding possible. The most important one is that the sequence of the transistors and the output is irrelevant. This way the folding algorithm is completely free to determine the most optimal sequence. Secondly, the devices and vias that are located at the intersections of the gates and nets, have sizes that are comparable to the width of the wires. An important property of nors is that all pull-down transistors can have the same size, when the pull-ups have the same size. The size of the pull-downs does not need to depend on the number of inputs. To make two dimensional folding possible multiple rows of pull-up devices are introduced.

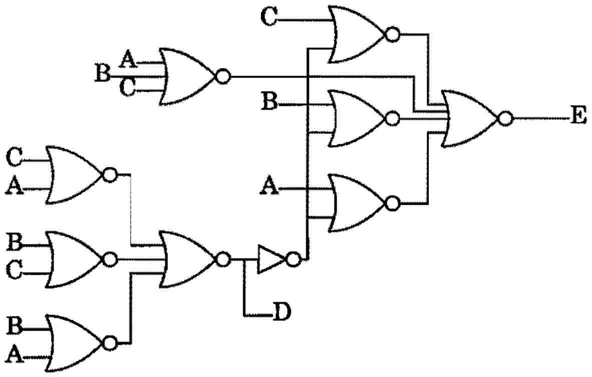


Figure 6.8. Full adder schematic circuit. The nor matrix layout style can handle any circuit that is described with multilevel nor's.

In figure 6.9 the schematic of a small example circuit is given. The circuit is a full adder which implements the following boolean equations:

$$D=AB+AC+BC \tag{6.6}$$
$$E=ABC+\overline{D}(A+B+C) \tag{6.7}$$

This 5 level implementation is not necessarily the best, but it illustrates the capabilities of the technique. Since a PLA realization needs only 3 levels, this is always a possible solution. The multilevel implementation however needs 7 transistors less then a 3 level implementation.

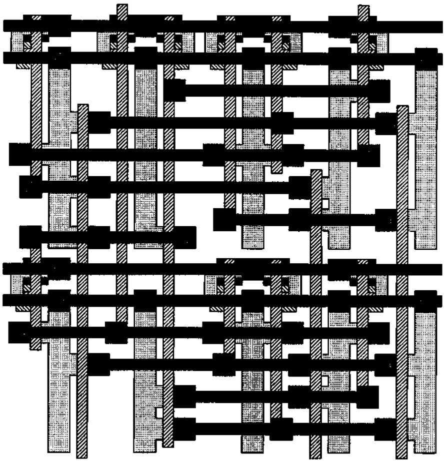


Figure 6.9. Mask layout of a full adder

The layout of this circuit can be seen in figure 6.9. The implementation presented here is suitable for a simple nMOS process with one layer of metal. The nets are realized in polysilicon and the nor gates in metal and diffusion. For the ground extra vertical diffusion strips are introduced everywhere. They do not take part in the folding process. The power and ground are distributed by horizontal metal rails. Notice that multiple rows of pull-ups have been placed, which allows several gates to share the same column.

A simple grid based compaction algorithm is applied before the masks are generated. When the arrays become larger, however, this compaction becomes less effective. The probability that somewhere on a grid line the maximum design rule must be applied increases with the size of the array.

6.6 The power and ground rails

If a single net were used for the ground or power, two-dimensional folding would become almost impossible. All gates would have to connect to the power and would therefore need a separate row.

Therefore we do not consider the power and ground during the folding. The power and ground nets are eliminated before the folding.

The power and ground rails are inserted in the array after the folding. In contrast to all other nets, the power and ground in a nor-matrix use rows, so they can be implemented in metal. The power and ground rail are placed in pairs, and always occupy a full row in the array.

The pull-up devices are placed on the intersection of its net and a power rail. The position of the power rails is therefore determined by the legal positions of the pull-ups. The ground rail is placed next to the power rail, so the pull-up can use the area underneath both rails. The first row of pull-ups must always be to the top of the array to connect the ground diffusions (see figure 6.9).

The problem of placing the power rails is equivalent to covering an interval graph with a minimum number of cliques. This problem can be solved efficiently [27].

$I(N,A)$ is a graph in which N is the set of nets that must be connected to a power rail. Two nodes are connected by an edge $a \in A$ when the nets overlap i.e. if $\sigma(n_1) \cap \sigma(n_2) \neq \emptyset$. I is an interval graph. The minimum number of cliques can be determined by a linear scan over the intervals. A pair of rails is placed whenever the placement of a pull-up of some gate can no longer be delayed.

Subsequently the pull-ups are assigned to valid positions on the power rails. Valid positions are the intersections of the power rails and the nets. For each net that needs a pull-up there is at least one valid position. This is guaranteed by the placement algorithm of the power rails.

The pull-up devices are wider than pull-downs and vias. Therefore two pull-ups are not desired to be adjacent between two ground diffusion strips. When two pull-ups are adjacent, more space is needed between the ground diffusion strips. Because grid compaction is used, this will affect the distance between the columns over the entire length.

The columns can be considered per pair, since the columns that share a ground diffusion strip are independent. Each pull-up has a number of the power rails on which it can be placed. This set of power rails is always consecutive. This problem can be formulated as a scheduling problem of tasks of unit length on two processors. This problem can be solved in polynomial time [25]. The algorithm can place the pull-ups in one scan over the power rails. It always places the pull-up with the closest 'deadline'.

6.7 A comparison of automatic layout styles

A number of experiments were done to compare the new layout styles to other automatic layout styles. All programs design for the same nMOS process, with the same design rules. We compared the results to the results of a conventional gate matrix generator and a standard cell place and route package. The standard cells were automatically generated linear transistor arrays. Some of the net lists were extracted from layouts produced by the standard cell package. The area of the minimum enclosing box is shown in the table. Compared with the standard cell package the area was 34%-63% smaller. The comparison with a gate matrix implementation showed an 27%-56% improvement.

Inspecting the layouts shows that the new transistor matrix layouts contain no empty areas. The standard cell program suffers from empty area in channels and of unbalanced columns. Also, the positions of the transistors are not matched as precisely as in the folding. The gate matrix suffers from its grid based compaction. Both old methods exhibit wire congestion in the center of the layout that pushes layout elements apart over the entire width or length of the circuit. Some of these problems may be fixed, but even then it seems unlikely that the results can be matched.

Table 6.1. comparison of layout areas

example		area of smallest enclosing box				
name	nr of xtors	xtor matrix	nor matrix	gate matrix	standard cell	PLA
hel84	12	0.043	-	0.072	-	-
data	24	0.088	-	0.166	-	-
adc	42	0.175	0.114	0.411	0.472	0.277
mp5	46	0.176	0.090	0.436	0.465	0.386
four	68	0.38	0.208	0.52	0.58	0.425
cnt4	96	0.57	-	1.35	-	-
five	177	1.33	0.595	2.99	2.29	0.81
six	332	3.85	-	-	3.95	1.58

Both new styles are nMOS styles. Folding seems to suit nMOS well because the transistors are not much wider then the wires. they can easily be intermixed with the wiring and each other. The circuits must have approximately the same width because they share a row. The more different the devices are from the wiring, the more efficient the separation of the device area and the routing area may be.

7. Shape control and prediction

As we stated before, the placement of arbitrary fixed shape rectangular circuits is very difficult. To alleviate this problem top-down floor planning does not use a fixed shape for a circuit. It assumes that a circuit can have any aspect ratio within certain constraints. The shapes of the circuits can jointly be optimized after the floor plan topology has been determined.

To make top-down floor planning feasible, it is necessary that the shapes of the circuits are controlled. To design a floor plan, the floor planner needs to know a prediction for the area of the circuit and the constraints on the shape before the circuit is designed.

Most circuit generators described in the literature have no or very limited possibilities of controlling the shape. The few circuit generators that have this capability mainly rely on a placement and routing [48]. Using placement and routing it is easy to change the shape of a circuit. In pluri cell circuits, automatically generated cells are placed in columns. The shape can be controlled by choosing the number of columns.

Accurate control of the shape is difficult if the circuit contains only a few primitive circuits. The control of the shape is finer when the circuit primitives are smaller. Folding allows the use of smaller circuit primitives, such as transistors. For smaller circuits shape control by folding is therefore more suitable.

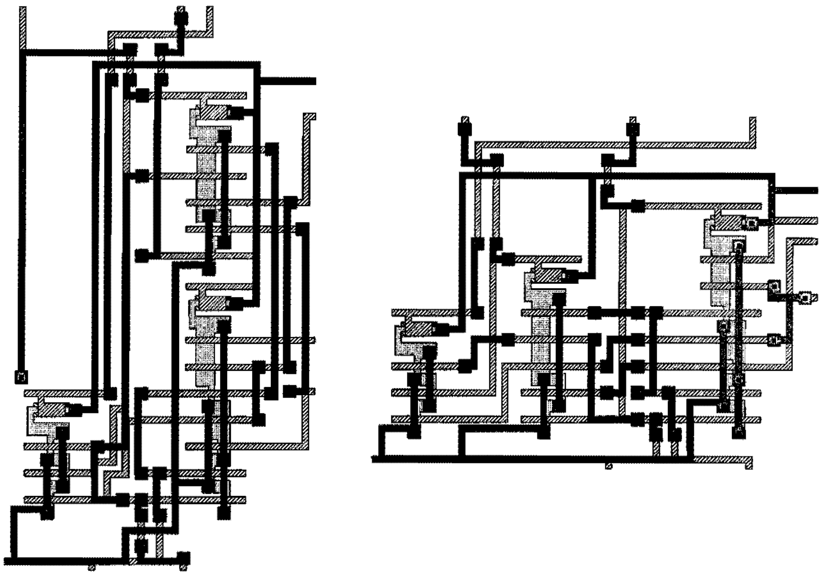


Figure 7.1. Two standard cell macros for the same function. By changing the number of columns, the shape of the macro can be controlled. Because the number columns is small, the control is rather crude.

In this chapter we will look at methods of shape control and prediction in two-dimensional folding. First, we will consider the control of the shape of the circuit. The orientation of the cutting line determines whether the rows or the columns are optimized. The direction of the cutting line can be used to steer the folding process to the desired aspect ratio.

The aspect ratio cannot be controlled equally well for all circuits. Long strips with many connections may limit the aspect ratio. In the second subsection we will consider partitioning these strips with a mincut algorithm.

Finally, to be able to design a floor plan, a prediction must be made for the shape function of the circuit. A prediction of the area can be made from the number of strips and their connections. Bounds on the aspect ratio are also derived.

7.1 Shape control

The folding algorithm assigns horizontal and vertical strips to rows and columns (see chapter 5). Two strips are folded when they are assigned to the same row or column. Using two-dimensional folding the number of rows and columns can be traded against each other. When more vertical strips are folded, fewer columns are needed. As a consequence, more horizontal strips overlap. They will need more rows. The number of rows can be traded against the number of columns accurately.

The folding algorithm repetatively partitions groups of strips. The partitioning algorithm tries to optimize a score function ε. A simple approach is to put the aspect ratio criterion into the score function of the partitioning algorithm. A suitable score function is

$$\varepsilon = \max(w, \rho h) . \max(h, \frac{w}{\rho}) \qquad (7.1)$$

In this formula w, h are estimates for the width and the height of the circuit and ρ represents the desired aspect ratio: the desired value of w/h. The formula represents the area of the smallest enclosing rectangle that has the desired aspect ratio ρ.

This seems to provide a good control mechanism for both area and shape optimization. It is not sufficient, however, because the cutting line orientation determines wether the width or the height can be optimized. When a horizontal group is partitioned, the height of the circuit cannot be reduced. Only the width of the circuit can be reduced.

To understand this we have to look at the consequences of cutting line decisions for the width and height of the array. Partitioning a group influences the width of the group being partitioned as well as the width of all orthogonal groups. The width of the group is the number of tracks it needs. It can be estimated with the upper bound μ and the lower bounds δ and γ.

Let H_k be the group that is partitioned into the groups H'_k and H'_{k+1}. Partitioning this group can only make it wider: a bound for the new sub-groups are together never smaller then the same bound for the original group.

$$\mu(H'_k)+\mu(H'_{k+1}) \geq \mu(H_k) \tag{7.2}$$
$$\delta(H'_k)+\delta(H'_{k+1}) \geq \delta(H_k) \tag{7.3}$$
$$\gamma(H'_k)+\gamma(H'_{k+1}) \geq \gamma(H_k) \tag{7.4}$$

The bounds become larger when the strips are not distributed evenly over the entire length of the new groups. (See figure 7.2). Using the left edge algorithm, and assigning the tracks to the new groups would lead to an even distribution. This way it is always possible to make a partition such that the upper bound does not increase and equality holds in equation (7.2). In short, partitioning a group cannot decrease its height, but the upper bound μ does not need to increase.

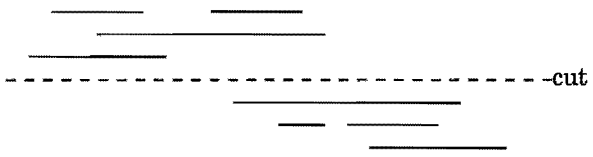


Figure 7.2. An uneven distribution of the strips over the two groups increases the height estimate of the group. The strips could be placed in 4 rows, but with this partition 6 rows are needed.

In the other direction, the bounds for the groups $V_1 \cdots V_\ell$ are also influenced by the partition. In this direction the gain is made because partitioning can decrease μ. To decrease μ and to keep δ down, few strips must cross the cutting line. Notice that γ cannot be influenced in this direction.

$$\mu'(V_i) \leq \mu(V_i) \tag{7.5}$$
$$\delta'(V_i) \geq \delta(V_i) \tag{7.6}$$
$$\gamma'(V_i) = \gamma(V_i) \tag{7.7}$$

We conclude that a horizontal cutting line tends to make the cell narrower and a vertical cutting line makes the cell lower. When the aspect ratio requires that the height be reduced, a vertical group must be partitioned. The control of the aspect ratio is therefore done by selecting the cutting line orientation. The orientation is chosen such that the estimated aspect ratio is corrected towards the desired aspect ratio. This is an effective control mechanism, which is also accurate because the repeated cuts allow for corrections that gradually become smaller.

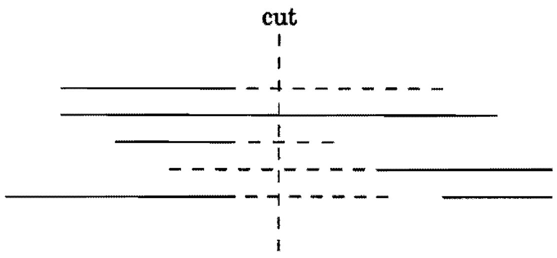


Figure 7.3. The height of a group can be reduced by a vertical cut. The density μ of the group is reduced by finding a partitioning that cuts few nets. This also optimizes the lower bound δ. The lower bound γ cannot be influenced.

The shape control algorithm estimates the aspect ratio, and determines the orientation of the next cutting line. It selects a group of this orientation and tries to partition it. If it is better, that is, if ε has become smaller, it is accepted. If it isn't, it is locked to prevent repeated selection. The algorithm ends if all groups are locked, that is, no group could be partitioned giving a better result.

When an improvement is found all groups are unlocked again because the partitioning of one group can influence the partitioning of all other groups. If the algorithm fails to improve the aspect ratio, it will try the other orientation. This will have a negative effect on the aspect ratio, but may further reduce the area.

Algorithm 7.1 Shape control

1. Determine a prediction for the aspect ratio W / H. Unlock all groups.
2. If $W/H < \rho$ then select a horizontal orientation else select a vertical orientation.
3. Select an unlocked group of this orientation. Partition this group.
4. Is the new state better than the previous state? If yes then go to 1.
5. Else merge the partition and lock this group.
6. Are there still unlocked groups of this orientation? If yes then go to 3.
7. Else select the other orientation. Are there still unlocked groups of this orientation? If yes then go to 3.

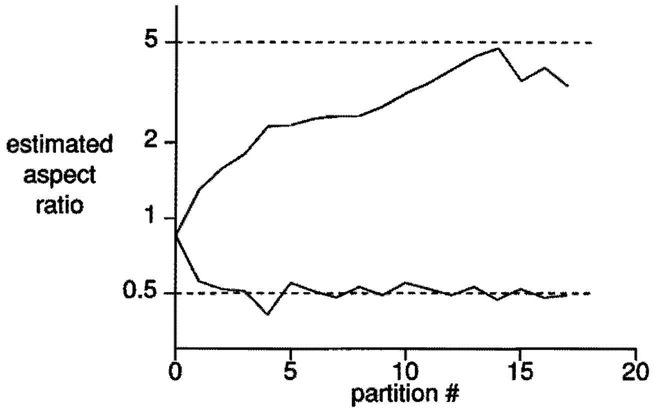


Figure 7.4. This graph shows the convergence of the aspect ratio to the desired value for two cases. The lower curve accurately approaches the desired ratio. The upper curve bends down because folding is not possible any more in the selected orientation. The aspect ratio gets worse, but the area improves.

Experiments have been done to verify that the shape can be controlled accurately. As an example, a nor matrix circuit was laid out with 10 different aspect ratios. In figure 7.5 five of the results are shown. The numbers indicate the desired aspect ratio. The arrays tend to be about 20% too high, which is due to different row and column pitches, and the assignment of power and ground after the folding. Since this factor is fairly constant it is easily compensated.

In figure 7.5 the result of this experiment is plotted in a graph. We can see that there is an almost linear relation between the desired aspect ratio and the actually obtained aspect ratio. Within a wide range, the aspect ratio can be controlled accurately. At the same time the area is almost independent of the shape. This is what can be expected of a completely flexible circuit.

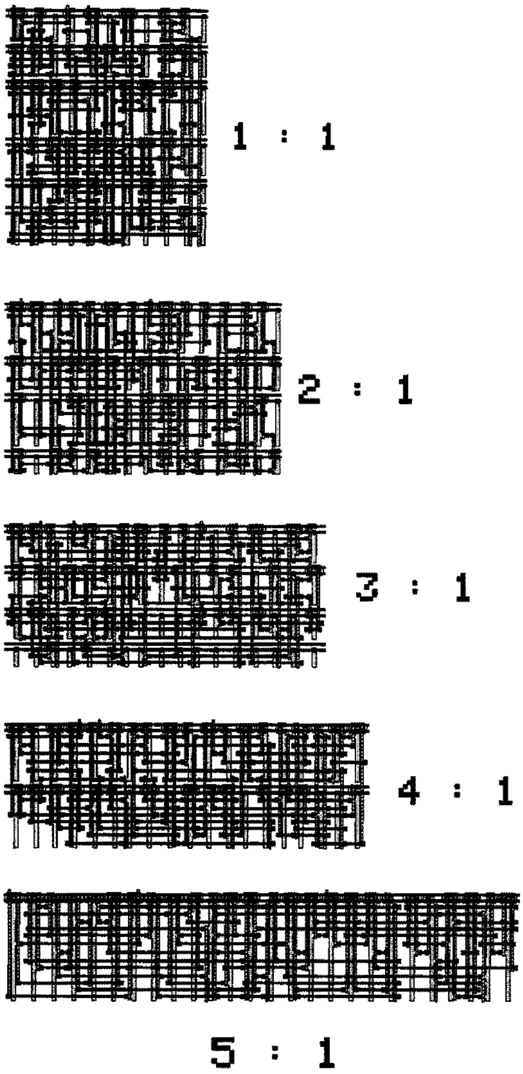


Figure 7.5. Five times the same nor-matrix circuit, but with different aspect ratios.

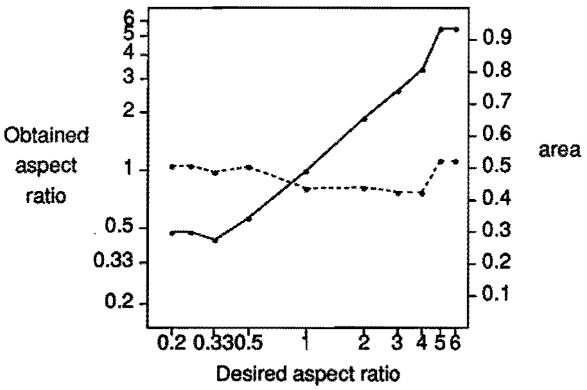


Figure 7.6. Control over the aspect ratio. The relation between the desired and the resulting aspect ratio *(drawn)* is almost linear. Meanwhile, the area *(dashed)* remains almost constant.

7.2 Shape function prediction

In top-down design, the floor plan is made before the circuits are generated. The floor planner needs however some information about the shape of the circuits. As is described in chapter 3 information about the shape of the circuit is represented as a positive non-decreasing function. A prediction for this shape function is needed. Methods for the prediction of shape functions have been described in [78, 9].

A completely flexible circuit has a constant area and an unconstrained aspect ratio. As figure 7.6 shows, circuits that use folding behave that way within a large range of the aspect ratio. In this range their shape function can be approximated by a hyperbola.

Since the aspect ratio is not completely free, bounds on the aspect ratio are needed. To predict the possible shapes of the circuits, we have to predict the area of the circuit, and the limits of the aspect ratio.

The area of the circuit can be predicted from the size of the input. In particular we used #H, #V and #E as characteristic values. The area of the final array can be predicted as:

$$\alpha \approx (\#V.\#H)^{0.75} \tag{7.8}$$

This relation is quite accurate over a number of examples as is shown

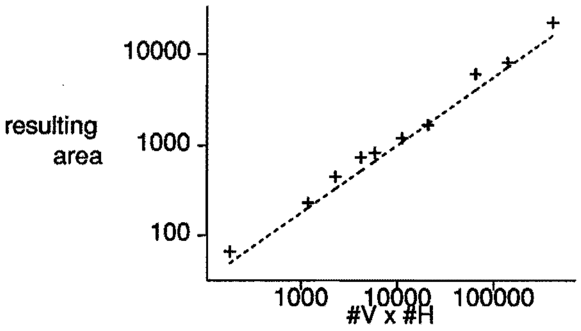


Figure 7.7. The area of the final array as a predicted from the number of vertical and horizontal strips #V.#H.

in figure 7.7.

In figure 7.8 the relation $\alpha \approx a\#E^2 + b\#E$ is fitted to measured data. Clearly, this is also a very good estimation mechanism. Perhaps it is possible to make a more accurate prediction using both numbers, but to do that it is necessary to do more experiments with a wider variety of circuits, because the deviations from the predicted values now seem hardly significant.

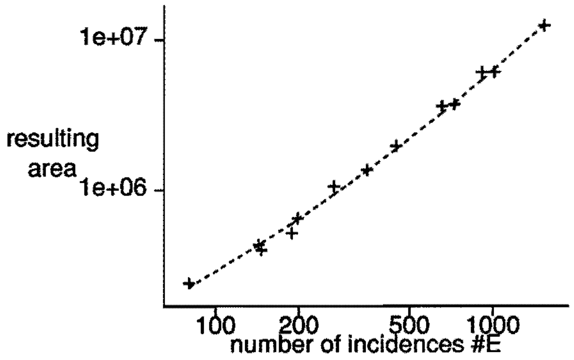


Figure 7.8. The area of the layout as predicted from the number of incidences #E.

It is clear that the circuit cannot be deformed to every aspect ratio, so there are bounds on the aspect ratio. To inform the floor planner, we have to make a prediction of these bounds. These predictions can be made using the bounds μ, δ and γ. They can be computed very simply for the unpartitioned groups.

The upper bound μ is equal to the number of strips in a group. At the extreme, all horizontal strips are assigned to a separate column. Clearly, the array cannot become higher than this.

The lower bound γ is the number of connections on a strip. Each of these orthogonal strips must be placed on a separate row or column. This constitutes a lower bound on this dimension of the array. The lower bound δ translates to the number of pins on the sides of the array.

If we assume that the area is known, and independent of the aspect ratio, we can compute from each lower bound an upper bound on the other dimension. For instance, let $w \geq \gamma(V)$ then $h \leq alpha/\gamma(V)$. This way we can compute 3 upper bounds and 3 lower bounds for each dimension, from which we can compute bounds for the aspect ratio. The bounds computed this way are not sharp, but can be used as an indication.

7.3 Partitioning before folding

Unfortunately, the shape cannot always be controlled as accurately as figure 7.6 suggests. Nor-matrix layouts are usually quite flexible, but transistor matrix layouts are often hard to control. The reason for this is that transistor net lists often contain nets that are connected to many transistors. Power and ground nets and nets like the clock and reset signal, must often be present everywhere. This means that initially there is a very high lower bound $\gamma(H)$ for the height of the matrix. For larger circuits it is often not even possible to make it square.

To alleviate this problem, it is necessary to partition the nets into several vertical strips. The vertical strips are connected by introducing a horizontal connection strip.

To decide which connections should go to which strip is not trivial: a good decision needs to know the global structure of the net list. Preferably, each vertical strip connects to horizontal strips that will be

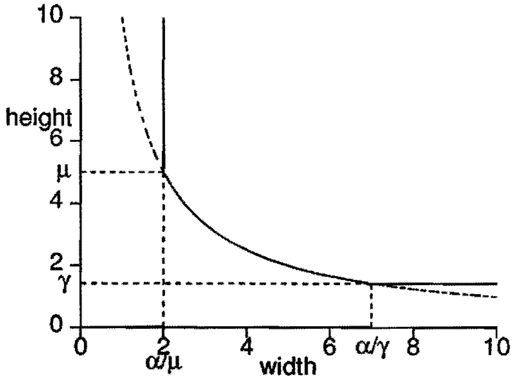


Figure 7.9. The shape function is a hyperbola within the flexible range. The limits of this range are in this case determined by the μ and γ of the horizontal strips. The bounds are 'mirrored' in the hyperbola.

close together in the final layout. Therefore, the choice is made globally by partitioning the horizontal strips into groups. Within each group a net is represented by a vertical strip. Nets that occur in several groups, and that have several vertical strips, are connected by new horizontal connection strips. These connection strips are the only elements between the groups.

The nets are partitioned with the fast mincut heuristic of [22]. Given is the bipartite folding graph $\mathcal{F}(V,H,E)$. The nodes V are partitioned into two sets V_1,V_2. The set must be about the same size: $|\#V_1-\#V_2|<\frac{1}{3}(\#V_1+\#V_2)$. Under this constraint the heuristic minimizes the number of connections between the two sets:

$$\#\{h\in H \mid v(h)\cap V_1\neq\emptyset \wedge v(h)\cap V_2\neq\emptyset\} \tag{7.9}$$

The algorithm is repeated several times, until the groups are small enough. When a wide array is desired, more cuts must be made to be able to get the desired aspect ratio. With each cut we introduce a number of extra horizontal strips. Partitioning the matrix into many groups may therefore lead to a larger area. It also may impair the possibility of creating very high circuits. It is therefore important to use a suitable number of cuts.

The optimal number of groups depends on the size of the matrix and the desired aspect ratio. As a heuristic we adopted the rule that the number of columns of a group after the folding must be constant. Empirically we determined that a group must be about c=15 columns wide after the folding. The number of groups is entier(w/c). Using $w=\sqrt{\rho\alpha}$ and equation (7.8), the optimal number of groups is computed with

$$\frac{1}{c}\sqrt{\rho}\,(\#V.\#H)^{0.375} \tag{7.10}$$

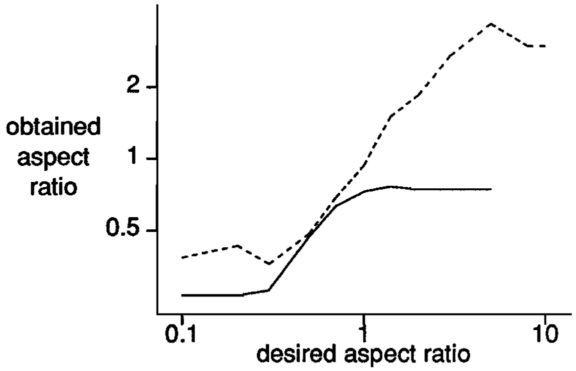


Figure 7.10. The obtained aspect ratio versus the desired aspect ratio for two versions of the same circuit. The unpartitioned circuit *(drawn)* is not very flexible. The other circuit *(dashed)* is partitioned into 8 groups before the folding. It has a much wider aspect ratio range.

Figure 7.10 shows the influence of partitioning on the flexibility of a circuit. For the unpartitioned circuit the aspect ratio could be varied from 0.27 to 0.74. When the same circuit was partitioned into 8 groups before the folding, the aspect ratio could be varied from 0.39 to 2.96.

8. Discussion

This chapter gives a more global view of the design problem and the proposed methods. First we will look back to the beginning of the book. At the beginning of the book we outlined a design philosophy. We will reconsider the principles outlined there, and evaluate them in the concrete terms of the algorithms that have been presented. Then we will look at how these algorithms can be put together to form a design package. Finally we will indicate a number of problems that have not been solved, and that require further research.

8.1 The predictor-adaptor paradigm

In this book the predictor-adaptor paradigm has been applied to a number of different problems. It has been applied to the floor planning problem, to the shape optimization problem, to the steiner tree problem, the folding problem and others.

In some cases it led to efficient algorithms. The best example of this is the shape optimization algorithms, which is both polynomial and optimal. In other cases it led to a decomposition of the problem into stages that gradually refine the design by taking more decisions. In the case of floor planning this is obvious: the layout problem is decomposed into the steps of floor planning - circuit generation - routing. The floor planning process itself goes through the steps of designing a point configuration and slicing, then channel assignment and shape optimization.

Sometimes it leads to a decomposition of the problem into similar, but smaller problems. In the slicing the solution space could be covered by combining the shape functions of smaller problems to shape functions of larger problems. Algorithms that do this are often referred to as 'dynamic programming' [3]. In the case of folding it lead to a decomposition of the partitioning problem into a smaller progressively smaller partitioning problems. The track assignment was gradually refined until the exact assignment was known. Tackling a problem by decomposing it into similar, but smaller sub-problems is known as 'divide and conquer' [4].

Since the design of the circuits is done according to parameters, this leads to some design criteria for the algorithms. For instance, the folding has to be able to take information about the optimal shape and pin positions into account.

The advantage of custom layout over gate-array is that there is more freedom to design the circuits. A circuit generator for custom layout should be able to exploit this freedom by taking many layout parameters into account. The transistor matrix layout style provides the best basis for such a flexible circuit generator that can take many important parameters into account.

The transistor matrix layout style shares with gate matrix layout the property that it can design a layout for any network of transistors. The transistors are adaptable in width and length. In many designs this is important because a lot of circuitry depends on electrical properties. Examples of such circuits are flip-flops, dynamic circuits, pass transistor logic, comparators, buffers, schmitt-triggers, clock drivers, precharged circuits, tri-state buffers and mueller C-elements. They can only be described in general terms as electronic networks. It is often important that the electric parameters of these circuits can be controlled. For instance the delay and fan-out of a buffer are important.

The advantages of transistor matrix layout over gate-matrix layout are clear: the layout is considerably smaller and the aspect ratio and pin positions of the layout can be controlled accurately. Esthetically it is pleasing to see that the transistor matrix layout style is entirely consistent with the top-down design philosophy.

We showed already that transistor matrix layout can handle parameters like shape and pin positions. Also the transistors have parameters for the width and the height and possibly the threshold voltage V_T.

Adaptation to the design rule parameters is realized by the compaction. We proposed different compaction algorithms for different styles. The one-dimensional compaction of the channels and of the transistor matrix layout is done by contour compaction. The weinberger array was compacted using grid based compaction.

Design rules can be very diverse. This makes it difficult to design compaction algorithms which can take every type of design rule into account. Writing a compaction routine is not easy. The principles are often simple, but if there is no simple and general framework to formulate the design rules, it will be sensitive to bugs.

A single general grid based compaction has been proposed for all cell generators [58]. Grid based compaction is usually the easiest to implement. However in both the channel routing and the transistor matrix, the layout method depends heavily on the compaction scheme. The way they are implemented now would make grid based compaction inefficient.

In floor planning decisions are made on the basis of predictions of the shape functions. The quality of the decisions depends entirely on the quality of the predictions. It is essential that these predictions are accurate, without a good predictor floor planning is not feasible.

The shape functions can be predicted by making a number of trial designs. Trial designs have the drawback that it is almost impossible to make a design for all different aspect ratios. When several parameters will influence each other a lot, the problem of making a prediction becomes an even larger problem. Predictions can also be made from some key parameters such as the number of transistors or the number of nets in the design. Luckily, the evidence so far suggests that good predictions can be made, at least for the layout methods that we studied. It also seems that the pin positions are of little influence to the predicted sizes of the circuits.

A circuit in the hierarchy provides simplification by replacing the complex internal structure of the circuit by a simpler model. In the building block layout, *all* circuit parameters are captured by a single model. The building block layout method assumes a complete isolation of the blocks for all parameters. This was necessary to be able to handle any circuit in any layout style in a single general method.

Such a complete isolation of the circuits may not always be a good solution. It is not always possible to find a good model for every parameter of the circuit. Also a different hierarchy may be more suitable for a different parameter. If complete isolation is unnecessary it is usually undesirable.

Slicing structures are an example of a hierarchy in which the slices have a model for some properties but are transparent in other properties. The hierarchy is present, but it is only used where it is advantageous. We used the slicing hierarchy only for the parameters for which it is advantageous: the dimensions of the slices. The dimensions of a slice can be modeled by only two numbers. They can easily be derived from the dimensions of the sub-slices and there relative positions. For the pin positions it would be more difficult to find a model that can easily be derived from the component slices and their relations. Therefore we did not use the slicing hierarchy in the channel assignment.

Another example is the hierarchy of groups in the folding. The hierarchy is gradually refined by partitioning groups into smaller groups. During the partitioning the groups are modeled by the (partial) evaluation of the parameters μ, δ and γ. Since this hierarchy serves no other purpose and since the parameters can be derived directly from the groups the hierarchy is not retained.

The floor planning problem was originally introduced in the layout of large row based designs. The problem was that a few large blocks, like memories or PLA's had to be embedded among a large number of small standard cells. There are no placement algorithms that can handle this problem gracefully. Therefore the placement can be done in two steps. First a floor planning phase and then a detailed placement phase. In the floor planning blocks that represent the memories and PLAs are placed together with blocks which represent groups of functionally related standard cells. These last blocks could be flexible since the number of rows of standard cells was not fixed.

The advantage of building block layout is that the inside of the block can be entirely hidden. These cells, mostly memories and register files, are often much more area efficient because they use a special array structure. Smaller circuits, like registers or PLAs can be implemented more efficiently in a general layout method. Their internal efficiency due to a special structure does not measure up to the inefficiency of the external wiring needed to connect it with the other circuits. The reason for using building block layout essentially depends on the need to include special structures on the chip.

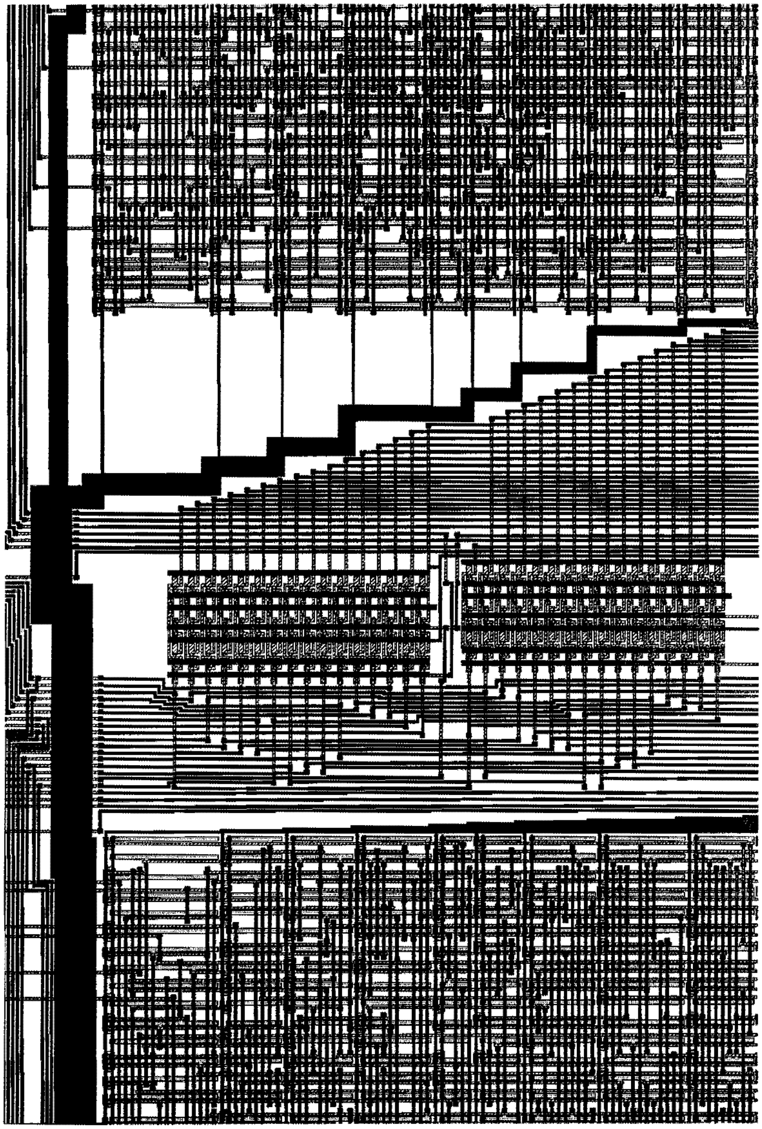
The design of the interconnection structure is the most important problem in layout. As the scale of the integration grows, the wiring becomes the dominating factor. Therefore the wiring should be designed as early as possible. It is however not clear how the wiring can be designed before the placement is known. The objective of the placement is however to create an efficient wiring pattern. Several approaches have tried to combine placement and wiring for this reason [7,11].

Designing a wiring pattern first, and then placing the circuits accordingly, is called *wire planning*. In [5] a wire planning approach is described for stackable designs such as the data path of a micro processor. If there are a sufficient number of routing layers the circuits can be placed under the wiring pattern. Since most of the chip area is wiring space we may assume that there is sufficient space underneath the wires to accommodate the circuits.

Wires should preferably be as straight as possible. Efficient regular patterns used in VLSI have straight wires only. The rationale is that a straight wire can connect things that are far apart more efficiently. It has, with the same wire length, a longer reach then a wire which has several branches.

The layout styles based on folding have these properties. The wires are required to be straight. The track assignment can be seen as a wiring plan. The circuits are placed underneath the wires and connected to the right wires after the folding. The circuits are designed with pin positions that match the environment in which they are used.

Figure 8.1. *(right)* A section of a draft layout for the bit-blitter chip that has been designed using the described algorithms. The design is done in nMOS. The two large regular arrays are barrel shifters that have been implemented using the nor-matrix style. The small modules in the center are registers.



8.2 Architecture of a design package

To make the algorithms available to a designer, they must form a *package*. Such a package consists of various programs that can be used to design an integrated circuit. It could consist of various programs for design synthesis such as logic synthesis and layout design, and various programs for analysis, such as design rule checking and simulation. The programs in the package can work together by exchanging design data. It is important that the package is consistent in the data exchange and in the tasks of the programs [6]. First we will look at the requirements for consistency in tasks, then we will look at the exchange of design data.

The task of a layout design package is to design a layout. The user must be able to have confidence in the correctness of the design. Confidence can be obtained in two ways: by analysis and by synthesis. Compliance with the design rules can be checked by a design rule checker. That the design does what it was intended to do can be checked by extraction of the net list and simulation of the net list. Programs to do this were the basis of the ICD package.

Another way of guaranteeing correctness is to use synthesis programs that produce correct designs. Such a synthesis program produces a layout from a higher level specification. When the whole design is done by such synthesis algorithms, the end result will be *correct by construction*. It is still necessary however to check the correctness of the higher level specification. To do this the designer needs a simulation program for this higher level specification. The tasks of a design package depend on each other: different synthesis programs require different analysis programs.

When custom layout is preferred over gate-array layout techniques it is usually for reasons of efficiency or for the need for analog circuitry on the chip. Often it is only possible to achieve the required function, circuit density or speed by using specialized design methods for several circuits. It may for instance be necessary to incorporate on the chip circuits like AD and DA converters, dynamic RAMs, phase locked loops, CCD-memories etc.

Such specialized circuits are designed by specialized methods, which are often not general enough to be automated. We cannot expect an automatic layout package to have an answer to every layout problem of these specialized circuits. Since the internal structure of these circuits is very different, it is necessary to use a layout technique for the entire chip that hides the internal structure of the circuits, and which can work with the circuits in terms of a model: the shape and pin positions. This is what the building block method was invented for.

A custom layout package should be open to differently designed circuits. The package should allow specially designed circuits or special circuit generators to be incorporated in the package. It should also allow easy interfacing to other synthesis and analysis programs. A package which allows this is called an *open* package.

A drawback of such an open package is that there is a large variety of data that is needed by the layout design generators. The package may accept data in the form of a net list, a piece of layout, random logic, trade relation predictions, design rules, timing information etc. A functional description of the package in terms input and output is then very difficult. Also it is more difficult to guarantee correctness by construction or to check that the input of the package is correct.

A complete chip design package requires too much expertise to be architectured by a single person. Yet the programs in such a package, although designed by different people at different times, must be able to work together. It is therefore necessary that the assumptions that must be made about the environment in which such a program operates are minimized. A program should preferably use only the most general methods of communicating with its environment. The only way of communicating with the environment which is sufficiently standardized is communication by means of files.

Like any complex system, the software package should be structured hierarchically. The practical levels in the hierarchy are often the procedure, the source file and the program. A couple of programs make for instance a circuit generator. The circuit generators for instance which generate the folded circuits consist of 3 or more different programs.

This separation into different programs has many advantages. Each of the programs has input and output files. The structure of these files can be defined exactly, thus facilitating the functional description of the programs. This allows a more accurate description then strong typing of the parameters of routines. The programs can check the input files for correctness, a much more rigorous checking than strong type checking. The separation into programs dissallows the use of global data structures [75]. Such an approach to programming is also known as *functional programming* or *object oriented programming* [55]. In this approach the programs are regarded as operations that are performed on the files which are the objects. Although most programming languages are unsuitable for object oriented programming, any program written in any language can be a primitive operation when the objects are files. Because the communication is through files only, the package does not need to be programmed in a single language.

On computers that have a limited amount of memory, an additional advantage is that the active piece of code is much smaller. The small programs function like overlays. Also the mess of dynamic data structures that most programs create is cleaned up automatically.

The partitioning of the package into small programs has several advantages for the programmers. The programmers that are working on the package, can work independently. Each programmer can keep a set of examples that he/she can use as test material. Text files can easily be inspected and edited by the programmer. Bugs created by one programmer in one program will not affect the others. When debugging, only the program with the bug has to be located and debugged. Since all the intermediate results are stored in files, finding the incorrect program is easy. The execution time of this single program may be much shorter than the execution time of the entire package. The repeated tests will therefore take less time.

Since an open package is extensible by design and since different people make these extensions, format conversions are unavoidable. A couple of measures can be taken to facilitate extension and conversion of formats. File formats can be made extensible by using labels for different kinds of data. The programs should look only at the data that is of interest to them and ignore unknown labels [20]. To facilitate conversions the file formats could be limited to an organization of columns and records. Such a representation of data is commonly used in relational data bases. The UNIX operation system provides many utilities, for instance awk, that perform operations on such files [38, 1].

8.3 Further research

One of the more important issues that has not been addressed in this book is the design of the timing of the integrated circuit. Much of what has been said about parameter optimization and trade-off in the first chapter applies directly to timing. Timing becomes increasingly important. For application specific integrated circuits speed is often the main reason for the design of a special chip.

Tradeoffs between area and delay must be made at an earlier stage than the layout stage. During the layout stage it is however necessary that the timing specifications are realized despite the fact that not all capacitances can be controlled. It is therefore necessary to change the sizes of the transistors to improve the timing.

The transistor matrix layout method allows any size of transistor to be used. For an accurate control of the timing resizing of the transistors after folding is necessary. Methods for resizing transistors have been published in [31, 23]. It would be useful if this could be done in such a way that timing of a circuit can be guaranteed.

Presently, the design rule parameters are not implemented in a satisfactory way. The design rules should be taken into account in the compaction as well as the transistor generation. The transistor generator should be able to take design rule parameters into account. By taking into account more environment information of the shape of neighboring transistors, a more efficient compaction may still be possible.

The transistor matrix generator can presently not handle delay and power consumption as parameters. To handle such parameters requires some understanding of the circuit. A separate program that translates the functional description of the circuit to a network should handle these parameters. It could take into account the delays between the in- and outputs and adjust the transistor sizes accordingly. This algorithm would be limited to a smaller class of circuits then transistor matrix layout can design.

Our present implementation of the matrix layout generator is not able to generate CMOS layouts. To make CMOS layouts only a small modification to the mask generation routines is necessary. The n transistors must be placed in a p-well and the p transistors must be placed in an n-well. The groups that are created by the folding can be partitioned into an n and a p part. Each part can have its own well.

In the comparison of layout styles the nor matrix style has the smallest area for the same function. The problem with nor-matrix is that it depends on the nMOS technology. There is no simple CMOS equivalent of the nor-matrix.

Folding can also be done by the simulated annealing algorithm of [73]. It would be interesting to compare the two approaches. Also it would be interesting to investigate whether folding and Bursteins placement and routing [7] could be combined. Bursteins placement and routing allows for bent strips, which is more suitable for very large arrays. Of course, in stead of simple transistors or nor gates more complex gates could be used. It is however questionable whether this would lead to an efficient result.

Another interesting subject would be the design of folded circuits that can be abutted to form arrays. Such circuits are called *tiles*. This requires a accurate control of the pin positions during the folding. The sides of the tile that are abutted could be connected during the folding. When a tile is to be abutted horizontally only, left and right side of the circuit are connected thus forming a cylinder. This cylinder can be cut and repeated to get the required repetition. For two-dimensional abutment also the top and bottom should be connected. The matrix becomes a torus and can be repeated in two dimensions.

The channel routing described in chapter 2 allows only for two layers of routing. As technology has progressed, more layers have become available. Nowadays, three or more layers of metal are common. The router as described could easily be changed to allow for three layers. However as the number of layers increases, the assumption that we made earlier that the wiring area and the area of the function blocks must be separate becomes less valid. An area router will be needed [33]. Area routers have the disadvantage that they cannot guarantee completion as channel routers can. Channel routers can guarantee completion because they can increase the channel width when needed. An area router that could do the same would be useful.

In this book we addressed the automation of custom layout design. For chips that must be developed fast, a gate-array is probably more suitable. The methods proposed in this book for building block layout are applicable to sea-of-gates too. It would be interesting to devise a sea-of-gates pattern that can be used in combination with folding.

8.4 Conclusions

The predictor-adaptor paradigm is not only applicable to floor planning but also to various design sub-problems. It leads to algorithms that are targeted towards global optimality by stepwise refinement. We showed how the paradigm can be used in routing, slicing and folding algorithms. A new polynomial and optimal slicing algorithm was presented.

We introduced design with unidimensional circuits rather then rectangular circuits as a more suitable method for designing compounds of small circuits. Compared to conventional methods the new layout styles give much smaller circuits.

Top-down design also determines the tasks of the various design algorithms and the interfaces between them. Custom layout allows indefinitely many layout styles. A design package that does not constrain the number of styles is necessarily open.

Top-down design is essential to achieve the maximum benefit of custom layout. Future research could address parameters that have been ignored so far. The most important of these parameters is delay. Many other refinements to existing software are possible. Particularly to the transistor matrix generator many interesting enhancements are still possible.

References

[1] AHO, A.V. AND B.W. KERNIGHAN, *The AWK programming language,* Amsterdam : Addison-Wesley, 1988.

[2] ASANO, T., "An optimum gate placement algorithm for MOS one-dimensional arrays," *Journal of Digital Systems*, vol. 6, pp. 1-27, 1982.

[3] BELLMAN, R., *Dynamic programming,* Princeton : Princeton University Press, 1957.

[4] BRASSARD, G. AND P. BRATLEY, *Algorithmics - theory and practice,* pp. 105-141, Englewood Cliffs : Prentice-Hall, 1988.

[5] BRAYTON, R.K., C.L. CHEN, J.A.G. JESS, R.H.J.M. OTTEN, AND L.P.P.P. VAN GINNEKEN, "Wire planning for 'stackable' designs," *Proc. Int. Symp. on VLSI Technology, Systems and Applications*, pp. 269-273, Taipeh : ERSO/ITRI, Taipeh, May 13-15, 1987.

[6] BROOKS, F.P., *The mythical man-month,* Reading : Addison-Wesley, 1975.

[7] BURSTEIN, M., S.J. HONG, AND R. PELAVIN, "Hierarchical VLSI layout: simultaneous placement and wiring of gate arrays," *Proc. IFIP Int. Conf. on Very Large Scale Integration*, ed. F. Anceau and E.J. Aas, pp. 45-60, Trondheim, 16-19 Aug. 1983.

[8] CHEN, H.H. AND E.S. KUH, "A variable-width gridless channel router," *Digest Int. Conf. on Computer Aided Design*, pp. 304-306, New York : IEEE, Santa Clara, Nov. 18-21, 1985.

[9] CHEN, X. AND M.L. BUSHNELL, "A module area estimator for VLSI layout," *Proc. 25th Design Automation Conf.*, pp. 54-59, Piscataway : IEEE, Anaheim, June 12-15, 1988.

[10] CHU, KUNG-CHAO AND RAMAUTAR SHARMA, "A technology independent MOS multiplier generator," *Proc. 21st Design Automation Conf.*, pp. 90-97, Piscataway : IEEE, Albuquerque, June 25-27, 1984.

[11] DAI, W.M. AND E.S. KUH, "Simultaneous floor planning and global routing for hierarchical building block layout," *IEEE Trans. on Computer Aided Design*, vol. CAD-6, pp. 828-837, 1987.

[12] DEMICHELI, G., "Multiple constrained folding of programmable logic arrays: theory and applications," *IEEE Trans. on Computer Aided Design*, vol. CAD-2, pp. 151-167, July 1983. errata in Vol. CAD-3, July 1984, p.256.

[13] DEO, N., M.S. KRISHNAMOORTY, AND M.A. LANGSTON, "Exact and approximate solutions for the gate matrix layout problem," *IEEE Trans. on Computer Aided Design*, vol. CAD-6, pp. 79-84, 1987.

[14] DEUTSCH, D.N., "A dogleg channel router," *Proc. 13th Design Automation Conf.*, pp. 425-433, Piscataway : IEEE, San Francisco, 1976.

[15] DEUTSCH, D.N., "Compacted channel routing," *Digest Int. Conf. on Computer Aided Design*, pp. 223-225, New York : IEEE, Santa Clara, Nov. 18-21, 1985.

[16] DEVADAS, S. AND A.R. NEWTON, "Genie: a generalized array optimizer for VLSI synthesis," *Proc. 23rd Design Automation Conf.*, pp. 631-637, Piscataway : IEEE, Las Vegas, June 29 - July 2, 1986.

[17] DIJKSTRA, E., "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269-271, 1959.

[18] DREYFUS, S.E. AND R.A. WAGNER, "The steiner problem in graphs," *Networks*, vol. 1, pp. 195-207, 1972.

[19] DUNLOP, MOD., A.E., "Session 8A: Gate matrix layout," *Digest Int. Conf. on Computer Aided Design*, pp. 312-327, New York : IEEE, Santa Clara, Nov. 11-13, 1986.

[20] EDIF STEERING COMMITTEE,, *EDIF - electronic design interchange format - version 2 0 0,* Washington : Electronic Industries Association (EIA), May 1987.

[21] EL GAMAL, A. AND Z.A. SYED, "A stochastic model for interconnections in custom integrated circuits," *IEEE Trans. on Circuits and Systems*, pp. 888-894, Sept. 1981.

[22] FIDUCCIA, C.M. AND R.M. MATTHEYSES, "A linear time heuristic for improving network partitions," *Proc. 19th Design Automation Conf.*, pp. 175-181, Piscataway : IEEE, Las Vegas, June 14-16, 1982.

[23] FISHBURN, J.P. AND A.E. DUNLOP, "TILOS: a posynomial programming approach to transistor sizing," *Digest Int. Conf. on Computer Aided Design*, pp. 326-328, New York : IEEE, Santa Clara, Nov. 18-21, 1985.

[24] GAREY, M.R., D.S. JOHNSON, AND L. STOCKMEYER, "Some simplified NP-complete graph problems," *Theoretical Computer Science*, vol. 1, pp. 237-267, 1976.

[25] GAREY, M.R. AND D.S. JOHNSON, "Two-processor scheduling with start times and deadlines," *SIAM Journal on Computing*, vol. 6, pp. 416-426, 1977.

[26] GAREY, M.R. AND D.S. JOHNSON, *Computers and intractability - A guide to the theory of NP-completeness,* New York : Freeman, 1979.

[27] GOLUMBIC, M.C., *Algorithmic graph theory and perfect graphs,* Computer Science and Applied Mathematics, New York : Academic Press, 1980.

[28] GROENEVELD, P., H. CAI, AND P. DEWILDE, "A Contour-based variable-width gridless channel router," *Digest of Technical Papers of Int. Conf. on Computer Aided Design*, pp. 374-377, New York : IEEE, Santa Clara, Nov. 9-11, 1987.

[29] HAKIMI, S.L., "Steiner's problem in graphs and its implications," *Networks*, vol. 1, pp. 113-133, 1971.

[30] HALL, K.M., "An r dimensional quadratic placement algorithm," *Management science*, vol. 17, no. 3, pp. 219-229, Nov. 1970.

[31] HEDLUND, K.S., "Models and algorithms for transistor sizing in MOS circuits," *Digest Int. Conf. on Computer Aided Design*, pp. 12-14, New York : IEEE, Santa Clara, Nov. 12-15, 1984.

[32] HELLER, W.R., G. SORKIN, AND K. MALING, "The planar package planner for system designers," *Proc. 19th Design Automation Conf.*, pp. 253-260, Piscataway : IEEE, Las Vegas, June 14-16, 1982.

[33] HEYNS, W., W. SANSEN, AND H. BEKE, "A line expansion algorithm for the general routing problem with a guaranteed solution," *Proc. 17th Design Automation Conf.*, pp. 243-249, Piscataway : IEEE, Minneapolis, June 23-25, 1980.

[34] KARP, R.M., "Reducibility among combinatorial problems," *Complexity of Computer Computations*, ed. R.E. Miller and J.W. Tatcher, pp. 85-104, New York : Plenum, 1972.

[35] KASHIWABARA, T. AND T. FUJISAWA, "NP-Completeness of the problem of finding a minimum clique number interval graph containing a given graph as a subgraph," *Proc. Int. Symp. on Circuits and Systems*, pp. 657-660, 1979.

[36] KELLER, W., F. SCHREINER, B. SCHURMAN, E. SIEPMANN, AND G. ZIMMERMANN, "Hierarchisches top down chip planning," manuscript, University of Kaiserslautern, 1987.

[37] KERNIGHAN, B.W. AND S. LIN, "An efficient heuristic procedure for partitioning graphs," *The Bell System Technical Journal*, pp. 291-307, Feb. 1970.

[38] KERNIGHAN, B.W. AND R. PIKE, *The UNIX programming environment*, Prentice-Hall software series, Englewood Cliffs : Prentice-Hall, 1984.

[39] KODRES, U.R., "Geometrical positioning of circuit elements in a computer," *Proc. AIEE fall general meeting*, October 1959. Paper nr. 1172

[40] KOU, L., G. MARKOWSKY, AND L. BERMAN, "A fast algorithm for steiner trees," *Acta Informatica*, vol. 15, pp. 141-145, 1981.

[41] LAPOTIN, D.P., "Mason: a global floor-planning tool," *Digest Int. Conf. on Computer Aided Design*, pp. 143-145, New York : IEEE, Santa Clara, Nov. 18-21, 1985.

[42] LOPEZ, A.D. AND H.F.S. LAW, "A dense gate matrix layout method for MOS VLSI," *IEEE Trans. Electron Devices*, vol. ED-27, pp. 1671-1675, 1980.

[43] LURSINSAP, C. AND D. GAJSKI, "Cell compilation with constraints," *Proc. 21st Design Automation Conf.*, pp. 103-108, Piscataway : IEEE, Albuquerque, June 25-27, 1984.

[44] MAREK-SADOWSKA, M. AND E.S. KUH, "A new approach to channel routing," *Proc. Int. Symp. on Circuits and Systems*, pp. 764-767, New York : IEEE, Rome, May 10-12, 1982.

[45] MATHESON, T.G., M.R. BURIC, AND C. CHRISTENSEN, "Embedding Electrical and Geometric Constraints in Hierarchical Circuit-Layout Generators," *Digest Int. Conf. on Computer Aided Design*, pp. 3-5, New York : IEEE, Santa Clara, Sept. 12-15, 1983.

[46] MEAD, C.A. AND L.A. CONWAY, *Introduction to VLSI systems,* Addison Wesley, 1980.

[47] NADLER, G., "Systems methodology and design," *IEEE Trans. on Systems, Man and Cybernetics*, vol. SMC-15, no. 6, pp. 685-697, Nov/Dec. 1985.

[48] NODA, T., Y. FUJINO, AND M. TERAI, "Fully automatic layout of sea of gates by soft macro approach," Preprint, Mitsubishi Electric, 1988.

[49] OTTEN, R.H.J.M. AND M.C. VAN LIER, "Automatic IC-layout: The Geometry of the Islands," *Proc. Int. Symp. Circuits and Systems*, pp. 231-234, New York : IEEE, Newton, April, 1975.

[50] OTTEN, R.H.J.M., "Layout structures," *Proc. Large Scale Systems Symp.*, pp. 349-353, Virginia Beach, 1982.

[51] OTTEN, R.H.J.M., "Automatic floorplan design," *Proc. 19th Design Automation Conf.*, pp. 261-267, Piscataway : IEEE, Las Vegas, June 14-16, 1982.

[52] OTTEN, R.H.J.M., "Efficient floorplan optimization," *Proc. Int. Conf. on Computer Design*, pp. 499-503, New York : IEEE, Port Chester, Oct. 31 - Nov. 3, 1983.

[53] OTTEN, R.H.J.M. AND L.P.P.P. VAN GINNEKEN, *The annealing algorithm,* Boston : Kluwer, 1989.

[54] PATTEE, H.H., *Hierarchy theory - The challenge of complex systems,* New York : Braziller, 1973.

[55] PETERSON, G.E., *Object oriented computing,* Washington : IEEE Computer Society Press, 1987.

[56] PREAS, B.T. AND W.M. VAN CLEEMPUT, "Placement algorithms for arbitrarily shaped blocks," *Proc. 16th Design Automation Conf.*, pp. 474-480, Piscataway : IEEE, San Diego, 25-27 June, 1979.

[57] RAO, P., R. RAMNARAYAN, AND G. ZIMMERMANN, "SPIDER, A chip planner for ISL technology," *Proc. 21st Design Automation Conf.*, pp. 665-666, Piscataway : IEEE, Albuquerque, June 25-27, 1984.

[58] REVETT, M.C., "Custom CMOS design using hierarchical floor planning and symbolic cell layout," *Digest Int. Conf. on Computer Aided Design*, pp. 146-148, New York : IEEE, Santa Clara, Nov. 18-21, 1985.

[59] ROYLE, J., M. PALCZEWSKI, H. VERHEYEN, N. NACCACHE, AND J. SOUKUP, "Geometrical compaction in one dimension for channel routing," *Proc. 24th Design Automation Conf.*, pp. 140-145, Piscataway : IEEE, Miami Beach, June, 1987.

[60] SCHOENBERG, I.J., "Remarks to Maurice Frechet's article "Sur la definition axiomatique d'une classe d'espace distancies vectoriellement applicable sur l'espace de Hilbert"," *Annals of Mathematics*, vol. 36, pp. 724-732, 1935.

[61] SIMON, H.A., "The architecture of complexity," *Proc. of the American Philosophical Soc.*, vol. 106, pp. 467-482, Dec. 1962.

[62] STOCKMEYER, L., "Optimal orientations of cells in slicing floorplan designs," *Information and Control*, vol. 57, pp. 91-101, 1983.

[63] STOK, L. AND R. VAN DEN BORN, "EASY: multiprocessor architecture optimistation," *Proc. of the Int. Workshop on Logic and Architecure Synthesis for silicon compilers*, Grenoble, May 1988.

[64] SZEPIENIEC, A.A. AND R.H.J.M. OTTEN, "The genealogical approach to the layout problem," *Proc. 17th Design Automation*

Conf., pp. 535-542, Piscataway : IEEE, Minneapolis, June 23-25, 1980.

[65] TAKAHASHI, H. AND A. MATSUYAMA, "An approximate solution for the Steiner problem in graphs," *Math. Japonica*, no. 24, pp. 573-577, 1980.

[66] WALD, J.A. AND C.J. COLBOURN, "Steiner trees, partial 2-trees, and minimum IFI networks," *Networks*, vol. 13, pp. 159-167, 1983.

[67] WATANABE, H. AND B. ACKLAND, "Flute - A floorplanning agent for full custom VLSI design," *Proc. 23rd Design Automation Conf.*, pp. 601-607, Piscataway : IEEE, Las Vegas, June 29 - July 2, 1986.

[68] WIMER, S., I. KOREN, AND I. CEDERBAUM, "Optimal aspect ratios of building blocks in VLSI," *Proc. 25th Design Automation Conf.*, pp. 66-72, Piscataway : IEEE, Anaheim, June 12-15, 1988.

[69] WING, O., "Automated gate matrix layout," *Proc. 15th Int. Symp. on Circuits and Systems*, pp. 681-685, New York : IEEE, Rome, May 10-12, 1982.

[70] WING, O., S. HUANG, AND R. WANG, "Gate matrix layout," *IEEE Trans. on Computer Aided Design*, vol. CAD-4, no. 3, pp. 220-231, July 1985.

[71] WINTER, P., "Steiner problem in networks: a survey," *Networks*, vol. 17, pp. 129-167, 1987.

[72] WIRTH, N., "Program development by stepwise refinement," *Communications of the ACM*, vol. 14, pp. 221-227, 1971.

[73] WONG, D.F. AND C.L. LIU, "Array optimization for VLSI synthesis," *Proc. 24th Design Automation Conf.*, pp. 537-543, Piscataway : IEEE, Miami Beach, June, 1987.

[74] WU, Y.F., P. WIDMAYER, AND C.K. WONG, "A faster approximation algorithm for the steiner problem in graphs," *Acta Informatica*, vol. 23, pp. 223-229, 1986.

[75] WULF, W. AND M. SHAW, "Global variables considered harmful," *Sigplan Notices*, pp. 28-33, Feb. 1973.

[76] YOSHIMURA, T. AND E.S. KUH, "Efficient algorithms for channel routing," *IEEE Trans. on Computer Aided Design*, vol. CAD-1, pp. 25-35, Jan. 1982.

[77] ZIBERT, K. AND R. SAAL, "On computer aided hybrid circuit layout," *Proc. Int. Symp. on Circuits and Systems*, pp. 314-318, San Francisco, 1974.

[78] ZIMMERMAN, G., "A new area and shape function estimation technique for VLSI layouts," *Proc. 25th Design Automation Conf.*, pp. 60-65, Piscataway : IEEE, Anaheim, June 12-15, 1988.

[79] VAN GINNEKEN, L.P.P.P. AND R.H.J.M. OTTEN, "Stepwise layout refinement," *Proc. Int. Conf. on Computer Design*, pp. 30-36, New York : IEEE, Port Chester, Oct. 8-11 1984.

[80] VAN GINNEKEN, L.P.P.P. AND J.A.G. JESS, "Gridless routing for general floor plans," *Digest Int. Conf. on Computer Aided Design*, pp. 30-33, New York : IEEE, Santa Clara, Nov. 9-12, 1987.

[81] VAN GINNEKEN, L.P.P.P., J.T.J. VAN EIJNDHOVEN, P.R.M. VAN TEEFFELEN, AND T.J. DECKERS, "Soft macro cell generation by two dimensional folding," *Proc. Int. Symp. on Circuits And Systems*, pp. 727-730, Espoo, June 1988.

[82] VAN GINNEKEN, L.P.P.P., J.T.J. VAN EIJNDHOVEN, AND A.H.C.M. BROUWERS, "Doubly folded transistor matrix layout," *Digest Int. Conf. on Computer Aided Design*, New York : IEEE, Santa Clara, Nov. 7-10, 1988.

[83] VON BERTALANFFY, L., *General system theory - foundations, development, applications*, Braziller, 1973.

Index

Notations

Scalars

w_c, h_c	width and height of a circuit
x_c, y_c	x and y coordinates of a circuit
α_c	area of a circuit
$s_c(w)$	shape function
$s_c^{\leftarrow}(h)$	inverse shape function
ε	score function of an algorithm
$\mu(S)$	upper bound for the number of tracks in group S
$\delta(S), \gamma(S)$	lower bounds for the number of tracks in group S
ξ	steiner assignment function
χ	channel assignment function
ϕ	column assignment function
ψ	row assignment function

Sets

$V=\{v_1, v_2, \cdots\}$	set of elements v_1, v_2 ...
$v \in V$	set membership
$V=\{v \in W \mid P(v)\}$	set of members of W for which P(v) is true
$V \cup W$	set union
$V \cap W$	set intersection
$V \backslash W$	elements of V that are not in W
$V \times W$	set product
$V \subset W$	V is a subset of W

$\#V$	number of elements of V
$\emptyset$	empty set $\#\emptyset=0$
$\mathbb{Z}$	whole numbers $\cdots -2,-1,0,1,2 \cdots$
$\mathbb{N}$	natural numbers $1,2,3 \cdots$
$\mathbb{R}$	real numbers
C	set of circuits
P	set of pins
N	set of nets
$\sigma(s)$	span of a strip expressed in tracks
$\bar{\sigma}(s)$	span of a strip expressed in groups

Graphs

$\mathcal{G}(V,E)$	graph named $\mathcal{G}$ with nodes V and edges $E\subset V\times V$
(V,E)	unnamed graph
(v,w)	edge $(v,w)\in E$
$\nu(v)$	neighbors of a node $\{w\in V \mid (v,w)\in E\}$
$^{\circ}v$	degree of a node $\#\nu(v)$
$(V,E)\cup(W,F)$	graph union $(V\cup W, E\cup F)$
$(V,E)\cap(W,F)$	graph intersection $(V\cap W, E\cap F)$
parent(v)	parent node of a node
left(v)	child node of a node in a binary tree
right(v)	other child node of a node in a binary tree
$\omega=$⊟ or ◫	orientation of a slicing line
$\mathcal{C}(V,E)$	channel intersection graph
$\mathcal{N}(C\cup\{C\},N,P)$	bipartite netlist graph
$\mathcal{F}(H,V,E)$	bipartite folding graph
$\mathcal{Y}(N,B)$	topology of steiner tree tree
$\mathcal{T}$	steiner tree
$\mathcal{P}_v(J_h,C)$	vertical polar graph
$\mathcal{P}_h(J_v,C)$	horizontal polar graph

Mask layers

■	contact hole
■	metal
▨	polysilicon
░	diffusion
▧	burried contact
■	depletion implant

Biography

Lukas P.P.P. van Ginneken was born on June 5, 1960. He studied electrical engineering at Eindhoven University, the Netherlands, from which he graduated with honors on December 20, 1984. Since then he has been working on a Ph.D. degree in the Design Automation Section. He hopes to graduate on the work in this thesis on April 11, 1989.

Mr. van Ginneken has worked in Mathematical Sciences Department of IBM's Thomas J. Watson Research Center, Yorktown Heights, New York, during most of 1984 and at various times during 1985, 1986 and 1987. From December 1984 to November 1988 he was an employee of the Foundation for Fundamental Research on Matter in the Netherlands. Since March 1989 he holds a postdoctoral position in the Computer Science department of the T.J. Watson Reseach Center.

Mr. van Ginneken has published several papers on simulated annealing and automated custom layout design. He is a co-author of a book called 'The annealing algorithm'.

STELLINGEN

bij het proefschrift van
Lukas P.P.P. van Ginneken

THE PREDICTOR-ADAPTOR PARADIGM

1. De transistor matrix layout methode maakt het mogelijk om circuits in vele parameters aan te passen aan de ontwerpspecificaties.

   Dit proefschrift

2. Gegeven een puntconfiguratie en de uitwisselingsrelatie tussen de dimensies van de modules, is het mogelijk om een met de puntconfiguratie consistente slicingstructuur te vinden in polynoom begrensde tijd, zodat de oppervlakte van de slicingstructuur minimaal is, indien de dimensies van de modules integers zijn.

   Dit proefschrift

3. Hoewel berekening van de krimpfactor van een eerste slicinglijn, maximaal $O(n^2)$ operaties kost, waar n het aantal punten in de puntconfiguratie voorsteld, kunnen de krimpfactoren van alle eerste slicinglijnen eveneens in $O(n^2)$ operaties worden berekend.

   L.P.P.P. van Ginneken: 'Gridless Routing for Generalized Cell Assemblies: Report and User Manual', TU Eindhoven research report 87-E-180, September 1987.

4. In simulated annealing van een probleem met een kwaliteitsfunctie die samengesteld is als de som van vele termen vormt de uitdrukking $E(t)-\sigma^2(t)/t$ een nauwkeurige schatting van het optimum.

   R.H.J.M. Otten and L.P.P.P. van Ginneken: 'Stop Criteria in Simulated Annealing', *Proc. Int. Conf. on Computer Design*, p.549, IEEE, 1988.

5. In simulated annealing vormt de uitdrukking

$$a \ln(1/\beta) + \frac{\overline{\max(0,\Delta\varepsilon)}}{t} + (a-1)\ln(1-a)$$

waarin β de uniforme selectiekans van een toestandsverandering is, Δε de scoreverandering veroorzaakt door die toestandsverandering, t de controleparameter en a de gemiddelde acceptatiekans, een goede indicatie voor het aantal toestandsveranderingen dat nodig is om de evenwichtsverdeling te herstellen.

L.P.P.P. van Ginneken and R.H.J.M. Otten: 'An Inner Loop Criterion for Simulated Annealing', *Physics Letters A*, 130(1988)429.

6. Om de timing van een geintegreerd circuit te kunnen garanderen is het noodzakelijk om de snelheid van de circuits op een laag nivo te kunnen beinvloeden.

7. De introductie op grote schaal van de VLSI chip technologieën met meerdere metaallagen zal een drastische vereenvoudiging van de automatische routing tools tot gevolg hebben.

8. De *pointer* kan worden gezien als de *goto* van de datastructuren. Pointers moeten daarom evenals goto's als schadelijk worden beschouwd.

9. De gebruikersvriendelijkheid, complexiteit en functionaliteit die verwacht worden van software voor hardware ontwikkeling is veel groter dan wat gangbaar is voor software die gebruikt wordt bij software ontwikkeling.

10. In het huidige economische bestel bestaan kosten grotendeels uit distributie kosten. Dit verklaart de grote hoeveelheden afval: de vuilnisman is de goedkoopste vorm van distributie.

The Predictor-Adaptor Paradigm

Lukas P.P.P. van Ginneken

This book shows how the predictor-adaptor paradigm can be applied to many aspects of automatic layout design. It features a new polynomial algorithm for optimal slicing of point configurations and a new heuristic for the Steiner tree problem in graphs. In these algorithms, the predictor predicts the design freedom, and the adaptor adapts the sub-designs to the specifications.

Design by adaptation implies design of circuits according to parameters. An elegant hierarchical folding algorithm makes accurate control of the aspect ratio of circuits possible.

A new layout style based on this folding algorithm, called transistor matrix layout, applies parametrized design to the smallest possible circuits: the transistors. The new layout styles give much smaller layouts than conventional layout styles.

ISBN 90-9002703-3