1190

Philosophical Dissertation

Technology Independent Layout Design

submitted to the Faculty of science of the University of Neuchâtel to obtain
the degree of Doctor of Science

by

Ruud Riem-Vis

Institute of Microtechnology
University of Neuchâtel
Switzerland

September 1993

Technology Independent Layout Design

Ruud Riem-Vis

IMPRIMATUR POUR LA THESE

Technology Independent Layout Design

de Monsieur Ruud Riem-Vis

UNIVERSITE DE NEUCHATEL

FACULTE DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel
sur le rapport des membres du jury,

Messieurs F. Pellandini, N. de Rooij, M. Declercq (EPF-Lausanne) et
Madame G. Saucier (INPG-Grenoble)

autorise l'impression de la présente thèse.

Neuchâtel, le 13 octobre 1993

Le doyen:

A. Robert

Abstract

This thesis presents an approach to technology independent layout design, which takes the compaction problem as a central issue. Whereas conventional approaches create the constraint graph automatically out of an initial layout, the tool TILT (Technology Independent Layout Tool) uses an hierarchical object oriented layout language (al2) to define all the required geometries and constraints.

The compacted results become predictable through the use of constraint weights and costs. The compaction time is minimized with the application of several graph reduction techniques, and different algorithms have been tested on practical cases.

The language al2 integrates recent developments in the domain of object oriented programming into a layout design environment. It allows to capture the design intent and eases the construction of portable, process independent layout generators.

Concrete examples in both the digital and analog field, for CMOS and bipolar type of technologies illustrate the large application domain of the approach. Still, the absence of explicit place and route functions makes the tool best suited for the generation of basic elements, standard cells and regular modules.

Résumé

Cette thèse présente une approche à la conception de layout indépendant de la technologie qui est conçu autour du problème de la compaction. Contrairement à des approches conventionnelles qui génèrent le graphe de contraintes automatiquement à partir d'un plan de masque initial, l'outil TILT (Technology Independent Layout Tool) utilise un langage hiérarchique et orienté objet (al2) pour définir les géométries et contraintes.

Le résultat après compaction devient prévisible à cause de l'application des contraintes pondérées et sélectives. Les temps de compactage sont minimisés par l'utilisation de plusieurs techniques de réduction de graphes. Différents algorithmes ont été testés sur des cas pratiques.

Le langage al2 intègre des développements récents dans le domaine de la programmation objet dans l'environnement de la conception de layout. Il permet de saisir l'intention du concept et il facilite la création de générateurs portable et indépendants du procédé de la fabrication.

Des exemples concrets dans le domaine digital et analogique, pour des technologies de type CMOS et bipolaire, illustrent le grand champ d'application de l'approche. Cependant, l'absence des fonctionnalités de placement et routage rend l'outil mieux adapté pour la génération des éléments de base, cellules standards, et modules réguliers.

Acknowledgements

It has been a long way from the primary school to the doctor's degree. Suprisingly, a long and intensive study was predicted by a psychologist who examined my future possibilities when I was 11 years old.

So, I would first like to thank my parents for their support and encouragements, as well as the Dutch government which attributed a bourse during my Master's study.

An obligatory part of this study concerned the execution of a three months training period. After canceling an opportunity in Brazil, destiny brought me to the Institute of Microtechnology in Neuchâtel. Initially planified for three months in the Sensors and Actuators group of Prof. Nico de Rooij, I found myself starting my diploma work 8 months later at the chair of electronics and signal processing of Prof. Fausto Pellandini. Special thanks to them, who allowed me to pass a splendid time at the IMT.

Destiny also brought me to Paris, where I met my wife. Thanks for her patience and love during the busy time when I was preparing my dissertation. Another journey in Paris, allowed me to contact Prof. Gabriele Saucier who I would like to thank for her participation in the committee.

I also appreciate the participation of Prof. Michel Declercq, actually director of the Laboratory of General Electronics at the Swiss Federal Institute of Technology of Lausanne (LEG-EPFL).

Many hours have been spent with discussions, some of the ones I had with Ganiyou Maliki, my colleague, were quite animated. Fortunately, walls in Switzerland are thick, and both of us wanted the best out of it. Thank you, Ganiyou! Furthermore, I express my gratitude to the two other members of my group, Pascal Vaucher and Sandro Ponta, whose contributions have been of great importance for the validation of the tool.

Then, I received a lot of help from the students which participated to the project. In particular, I wish to thank Marco Berkhout, Wiebe van de Veen, Richard Logtmeijer, Jacques Margairaz, Laurent Dardel, and Antoine Allaman.

The year I passed at the CSEM, was of great benefit to me, especially due to the contributions of Michel Joss, Jean-Marc Masgonty and Christian Piguet.

Of course, I do not forget to thank my other colleagues including the secretary board, who gave me valuable help and support in different domains. In particular, the proofreading done by Jean-Pierre Amann and Philippe Moeschler allowed me to improve the text.

Finally, I express my acknowledgements to all the instances who financially supported this work, in particular: the Charles-Edouard Guillaume foundation, Fontainemelon; the Autophon Foundation, Solothurn; ASULAB SA, Neuchâtel; Stiftung Hasler-Werke, Berne; ROLEX SA, Geneva; ROLEX SA, Biel; Complication SA, La Côte-aux-Fées; ASCOM Microelectronics SA, Bevaix; Federal Commission for the Encouragement of Scientific Research (CERS 1907.1 and CERS 2291.1); and the National Fund for Scientific Research (FN 34292.92).

Contents

1 Introduction 1

1.1 Overview 1
1.2 History 2
1.3 VLSI design 3
1.4 Economical context 4
1.5 Outline of the text 5
1.6 References 5

2 Layout design 7

2.1 Introduction 7
2.2 Hand-crafted layout 8
2.3 Gate arrays 9
2.4 Standard cells 10
2.5 Symbolic design approaches 11
Fixed grid layout 11
Gate-matrix layout 12
Sticks 13
Virtual grid 13
Graph-based compaction 14
2.6 Automated layout synthesis 14
Procedural module definition 15
Silicon compilers 15
2.7 TILT 16
History 16
Design and data flow 17
Compaction 18
Implementation 19
2.8 Conclusions 21
2.9 References 21

3 Compaction **23**

3.1 Introduction 23
3.2 The compaction problem 24
3.3 Constraint management 26
Longest path determination 26
Redundancy 28
The critical path 31
Slack distribution 32
Centering 35
Zero cycle reduction 36
Costs 39
3.4 Hierarchy 40
An example 40
Principle 41
Extraction 42
Compaction sequence 43
Protection frames 44
Instance reduction 45
3.5 Incremental compaction 45
Constraint building 46
One-level constraint resolution 46
Hierarchical constraint resolution 46
3.6 Overconstraints 47
3.7 Interfacing 48
Layout properties 49
Geometrical primitives 49
The box 50
3.9 Performance 51
Computational efficiency 52
3.10 Active and exclusive constraints 54
3.11 Conclusions 56
Algorithms 56
Interface 57
Incrementality 57
Limitations 57
3.12 References 58

4 The language al2 **59**

4.1 Introduction 59
4.2 Language design 60
Variable name propagation 60
Dynamic constants 61
Declarative or imperative? 62
Memory management 62

4.3 Entities, types, classes and objects 63
Constructors and destructors 63
Creations and references 64
NULL-pointer propagation 64
Deductive typing 65
Class hierarchy 66
4.4 Expressions and statements 67
A pyramid of contacts 67
Dynamic arrays 69
Generic records 69
Assignments 70
Operator summary 72
Built-in function summary 73
Statement summary 73
4.5 Exemplars 74
Technology encapsulation 75
4.6 Reuse 75
Overloading 76
Defaulting 77
Instance matching 77
4.7 Compilation 78
The preprocessor 79
Syntax and semantics 80
Resource allocation 81
Code generation 81
System functions and operators 86
Adding on external C functions 87
4.8 Interpretation 88
Incremental execution 89
4.9 Debugging 90
4.10 Conclusions 90
Semantics 91
Implementation 91
Improvements 92
4.11 References 92

5 Applications 93

5.1 Introduction 93
Outline 93
Figures with layout 94
5.2 The CMOS and the bipolar library 95
Technology 96
Environment 98
Geometry 98
Topology 100

Summary 103
5.3 Standard cells 103
Floor planning 103
Internal interconnections 104
External interconnections 106
Using the second metal 106
Variable fanouts 108
Further parametrization 109
Performances 110
Summary 112
5.4 A static RAM generator 112
The memory matrix 112
The line decoder 115
The complete generator 117
Summary 121
5.5 Analog building blocks 121
CMOS transistor generators 122
An operational transconductance amplifier 123
Bipolar transistor array 125
Summary 125
5.6 A switched capacitor silicon compiler 126
Laker biquads 126
Capacitor ordering and switch orientation 127
Capacitor field 128
Routing 128
Examples 128
Summary 130
5.7 Conclusions 130
5.8 References 130

6 Conclusions 133

6.1 Technology independent layout design 133
Technical viewpoint 133
Economical interest 134
6.2 TILT 135
The approach 135
Applications 135
Implementation 136
Limitations 136
6.3 Suggestions 137
Technology independent design 137
Data flow 138
A mixed approach 138
Netlist 138
6.4 References 139

Annex A: Technical data **141**

A.1 Utilities and libraries 141
A.2 Al2 compiler options 141
A.3 Compactor options 142

Annex B: Formats **143**

B.1 Al2 syntax 143
B.2 Layout image 145

Annex C: Application code **147**

C.1 The UNION function 147
C code 147
C.2 ADDCONT 149
Al2 header 149
High level al2 body 149
Low level al2 body 150
C version 150
C.3 MOS transistor 153
Al2 header 153
Al2 body 153

1 Introduction

It's a mystery to me - the game commences
for the usual fee - plus expenses
confidential information - it's in a diary
this is my investigation - it's not a public inquiry

Dire Straits

The layout design process corresponds to the creation of a set of masks to be used during the fabrication of integrated circuits. After a short overview of the approach followed in this dissertation, a historical, technical and economical context is given. Finally, the outline of the text is presented.

1.1 Overview

At first sight, technology independent layout design is a *must* for every dynamic VLSI design system. However, the commercial support for this kind of approaches has been very weak, so far [PIG90]. This thesis may provide some explanations of this phenomenon.

To start with, the most common layout design methodologies are analyzed. It appears that only symbolic and procedural design systems allow for a sufficient independence of the target technology. The former generally uses some kind of graphical entry, and includes the early fixed grid [LAR71, GIB76], the virtual grid [WES81, BOY83], and the relative grid approaches [CHO77, WIL78, HSU79]. The latter is language based [BAT81, LIP82], and offers an expressive environment suited for advanced parametrization.

Much of the text is concerned with the tool TILT [RIE92], which has been developed by the author and his team over the last four years. This procedural design system inherits much of the ideas found in the ALI system [LIP82] and in LISA [SWA88]. It combines the power of imperative and declarative formulations in an attempt to manage the design complexity of technology independent layout generators. Based on a powerful implementation of a relative grid compactor, an in-house developed layout language, called *al2* [MAL93], is used to parameterize the design and to generate a set of technology specific constraints. Contrary to the symbolic design approaches,

TILT allows for hand-crafted equivalent densities, but restricted to Manhattan[1] type of layout.

The increased computing power of modern workstations explains the regained interest for the expensive relative grid compaction. Chapter 3 presents several graph complexity reduction techniques, as well as different algorithms to achieve an optimally spaced result with a uniform slack distribution.

The design and implementation of the original object oriented language al2 are presented in chapter 4. Some particular properties are highlighted, and the advantages of a bootstrap compiler implementation are illustrated. Although al2 was designed for the description of layout, its particular semantics related to incremental behavior, makes it interesting for other application domains.

The validation of the approach is the prior occupation of chapter 5 [BER91, LOG92, MAR93]. Here, different case studies are presented to illustrate the power and limitations of the method. Each of the examples focuses on some typical aspect of the design, and shows how this can be implemented in the TILT system. Various tables and figures allow to judge the quality of the generated layout.

With the experiences obtained on the different applications, we must conclude that real technology independent layout design can only be justified for some special cases. Although it is possible to create high quality layout with technology independent generators, the development time of these generators is significant. Therefore, only in some specific situations, the investment may be economically worthwhile.

1.2 History

In 1986, when I was a student at the Twente University in the Netherlands, I participated at an optional VLSI design course. After a few theoretical introductory lessons, several teams of 8 to 10 persons were formed, each of them had to design a real VLSI chip in some weeks of time. The layout generation was an essential part of the job, and I remember the terrible input format of the CAD tool. Every coordinate of each polygon corner had to be specified in a text file.

Our team had chosen to realize a 32-bit microprocessor with a particular RISC-style architecture. Together with two colleagues, I was in charge of the design and implementation of the control unit of this processor. The floor planning phase had attributed a thin but tall area for this part of the chip. Being in zoom mode on the eight floor of the building, it took a quarter of an

[1] in a Manhattan type layout, all geometries have their sides parallel to the x or y-axis

hour to scroll one step. The software simply redrew the complete layout, starting at the basement. We drunk a lot of coffee...

When I arrived in the group of Prof. Pellandini in 1987, better tools were available, but still, very much effort was invested in the design of hand-crafted layout [DEF89, SJÖ90]. Months were spent to create a near to optimum solution for a new distributed arithmetic processor unit. Unfortunately, time has made the layout obsolete, but still the know-how remains valuable. This observation has motivated myself to push the research and development of a tool which would be able to capture the design *intent* instead of the physical result. Four years of intensive labor have passed since, and the research group grew from one to six persons.

1.3 VLSI design

The design of a large integrated circuit has grown far beyond the capabilities of a purely manual approach. Whereas early microprocessors (e.g. Intel 4004), were merely assembled by hand, today, the use of CAD tools is a necessity. The VLSI design process has become a sequence of transformations, starting at the functional or behavioral description of the system, and ending with a set of mask descriptions which are used for the manufacturing of the circuit.

Different tools are required at different moments in the design process. At the highest level, the designer is concerned with the structural and behavioral description of the system. Together with the standardization efforts spent on the VHDL language [VHS87], a rich research activity is noticed over the past few years. The reason for the growing popularity of this approach is the fact that it has been recognized that it is not only important to produce correctly working chips, but also to specify what they do. Especially in an industry with a continuously evolving production technology, a system with the same functionality will be realized differently tomorrow than it is today! In the beginning, VHDL was primary used for the specification of digital systems, nowadays, modeling, testing and synthesis are domains that are engaged. Even analog design becomes more and more affected to this field and language extensions have been proposed in order to make the language accessible for non-discrete systems [COT92].

A more concrete vision of the system is obtained with the *netlist*. Such a net describes the elements of the systems and the connectivity in between them. Note that a netlist is coupled to a specific technology, such as ECL, I2L, TTL, nMOS, CMOS, BicMOS, or GaAs, because of the technology class specific device characteristics. The choice of a specific technology will primary depend on the specifications and know-how of the company. At the netlist level,

primary tools are circuit simulators. SPICE has been a reference here for years, but recently, new simulators gain interest. The goal of these simulations is to verify whether the proposed circuitry will be able to meet the specifications or not. Speed, power dissipation, yield, stability, etc., are among the factors to be considered.

Once the netlist has been simulated and approved, the time has come to map the circuit into silicon. Much effort has been invested to automate this phase. Indeed, in some cases it is possible to perform this translation in a straightforward way. Still, a large number of designs need the intervention and creativity of a layout designer. As the manual translation of a netlist into a physical layout may be very time consuming, different methodologies have been proposed to reduce this effort. The next chapter gives an overview of the most current approaches. Emphasis is put on the degree of technology independence that can be obtained with each of them.

1.4 Economical context

The state of the art of a technology depends largely on the economical profit it may be engender. Since the discovery of the field-effect transistor by Shockley in 1952, things have gone very fast. All over the past four decades, new technologies have been proposed which allowed for faster, smaller, and less power consuming devices. This development has been so fast that today's technologies are only poorly exploited.

CAD tools have followed this evolution. Progress in the field of algorithms, data structures, data bases and hardware has improved the performance of this technology a lot. Today, most leading CAD software has emerged towards a framework that integrates the different tools into a consistent and compatible environment. Due to the enormous effort which is required to achieve this, only a few (large) companies can afford such a complete and expensive system. Small and medium sized enterprises are often obliged to assemble their design on a printed circuit board, or to make use of external design facilities.

From the CAD vendor's point of view, the rapid changes and the variety of technologies imply that they only supply tools which can be sold to many of their clients. They need to make money! As a consequence, problems which ask for non-standard solutions are often difficult to solve with these tools, and they are badly supported by the vendors. Fortunately, the pressure from the client's side has resulted in the establishment of *open* systems. Now, the user may integrate his own tools into the framework.

Technology independent layout tools are good examples of non-standard solutions. A large part of the market relies on several types of gate-arrays and

standard cells. Both approaches make extensive use of libraries, which can be purchased from the same CAD vendor! Thus, providing a tool which makes a layout description completely technology independent could reduce the turnover.

Technology independent layout tools may complicate the design task because additional effort is needed to assure the portability. Therefore, they are of more interest as a back-end facility for the creation and maintenance of libraries. Now, the investment may be worthwhile because the time-to-market of new technologies may be reduced.

Another chance for the application of technology independent tools arises when no libraries are available from the CAD vendors. This may for instance be the case for very recent technologies, or if special specifications are required with respect to speed or power consumption. In particular, the Swiss watch industry has a need for low-power circuitry, and many research programs have been sponsored by this group.

1.5 Outline of the text

After this introduction, a survey of the most common layout design styles, including the tool TILT, is given in chapter 2. The following chapter is concerned with compaction techniques, whereas chapter 4 concentrates on the language al2. Several applications are given in chapter 5, followed by the global conclusions of this thesis.

Annex A gives some technical figures concerning the tool, whereas annex B provides the complete syntax of the language al2 and the layout image format. Some illustrative code samples are presented in annex C.

Notice that all chapters contain their own specific list of references, and chapter 2 to 5 are concluded individually. This is done because the chapters treat fairly different items. Someone which is only interested in a single topic will have a specific conclusion on the subject with local references, directly at the end of the chapter. Furthermore, it may be easier to search for a reference in a smaller list.

1.6 References

[BAT81] J. Batali, N. Mayle, H. Shrobe, G. Sussman, and D. Weise, *"The DPL/Daedalus Design Environment"*, VLSI 81: Very Large Scale Integration, J.P. Gary ed., pp.183-192, Academic Press, 1981.

[BER91] M. Berkhout, *"SCSC a Switched Capacitor Silicon Compiler, Volume I: Design Manual"*, internal report, IMT University of Neuchâtel, Switzerland, March 1991.

[BOY83] D.G. Boyer, N. Weste, *"Virtual Grid Compaction using the Most Recent Layers Algorithm"*, Proc. IEEE Conf. on Computer Aided Design, pp.92-93, September 1983.

[CHO77] Y.E. Cho, A.J. Korenjak, and D.E. Stockton, "*FLOSS: An Approach to Automated Layout for High-Volume Designs*", Proc. 14th DAC, pp.138-141, June 1977.

[COT92] R. Cottrell, K. Nolan, M. Brown, "*VHDL Analog extensions: Process, Issues and Status*", Proc. Eurodac '92, pp.713-717, Hamburg, Sept. 1992.

[DEF89] I. Defilipis, U. Sjöström. M. Ansorge, and F. Pellandini, "*A 2-Dimensional 16 Point Discrete Cosine Transform Chip for Real Time Video Applications*", Proc. GRETSI'89, Vol.2, pp.813-816, Juan-Les-Pins, France, June 1989.

[GIB76] D. Gibson & S. Nance, "*SLIC - Symbolic Layout of Integrated Circuits*", Proc. 13th DAC, pp.434-440, June 1976.

[HSU79] M.Y. Hsueh & D.O. Pederson, "*Computer-Aided Layout of LSI Circuit Building-Blocks*", Proc. IEEE Intern. Symp. on Circuits and Systems, pp.474-477, 1979.

[LAR71] R.P. Larsen, "*Computer-Aided Preliminary Layout Design of Customized MOS Arrays*", IEEE Transactions on Computers, Vol. C-20, No.5, pp.512-523, May 1971.

[LIP82] R.J. Lipton, S.C. North, R. Sedgewick, J. Valdes, G. Vijayan, "*ALI: A Procedural Language to describe VLSI Layouts*", Proc. 19th DAC, pp.467-474, June 1982.

[LOG93] R. Logtmeijer, "*Analysis of a Layout Generator for Switched Capacitor Filters*", report 328 PE EC 09/92 IMT University of Neuchâtel, Switzerland, Sept. 1992.

[MAL93] G. Maliki, S. Ponta, R. Riem-Vis & P. Vaucher, "*Al2, Un langage de description de layout*", IMT report version 3.0, Switzerland, March 1993.

[MAR93] J. Margairaz, "*Réalisation d'un générateur de mémoire RAM à l'aide de TILT*", diploma work at the IMT University of Neuchâtel, Switzerland, March 1993.

[PIG90] C. Piguet, "*Technology Independent Layout Tools*", Intensive Summer Course on CMOS VLSI Design '90, EPFL Lausanne, Aug. 27 - Sept. 14 1990.

[RIE92] R. Riem-Vis, G. Maliki, & F. Pellandini, "*TILT, a Technology Independent Layout Tool*", WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, March 12-13 1992.

[SJÖ90] U. Sjöström, I. Defilippis, M. Ansorge, and F. Pellandini, "*A Discrete Cosine Transform Chip for Real Time Video Applications*", Proc. IEEE Int. Symp. on Circuit and Systems ISCAS'90, Vol.2, pp.1620-1623, New Orleans, USA, May 1990.

[SWA88] M. van Swaaij, "*LISA. A Declarative Language For Interactive Layout Generation*", CSEM technical report no. 193, Neuchâtel, Switzerland, 1988.

[VHS87] "*VHSIC Hardware Description Language*", Language Reference Manual, IEEE-1076, New York, 1987.

[WES81] N.H.E. Weste, "*Virtual Grid Symbolic Layout*", Proc. 18th DAC, pp.225-233, June 1981.

[WIL78] J.D. Wiliams, "*STICKS - A graphical compiler for high level LSI design*", AFIPS Conference Proceedings, Vol.47, pp.289-295, June 1978.

2 Layout design

This chapter presents an overview of common items in the field of layout design. After an introduction into some general techniques to manage the design complexity, an overview of common design styles is given. Emphasis is put on the degree of technology independence which can be obtained with these different approaches.

2.1 Introduction

If we want to map high performance systems into silicon, it is imperative to reduce the complexity of the design. There are a number of techniques which are commonly used [WES85]:

- hierarchy
- domain structuring
- modularity
- regularity
- locality
- abstraction

Hierarchy is based on the "divide and conquer" philosophy: divide a module into sub-modules and repeat this operation until the complexity of the sub-modules is at a comprehensible level of detail.

A system has different faces: a *behavioral*, a *structural*, and a *physical* [GAJ83]. In the behavioral domain, the design is seen as a black box; the relations between inputs and outputs are given without reference to the final implementation. A structural view sees each part of the design as the composition of interconnected subparts. In the physical (or layout) domain, information is provided how the subparts in the structural domain are located in a two dimensional plane. It is important to separate these data, while still keeping a relation between them.

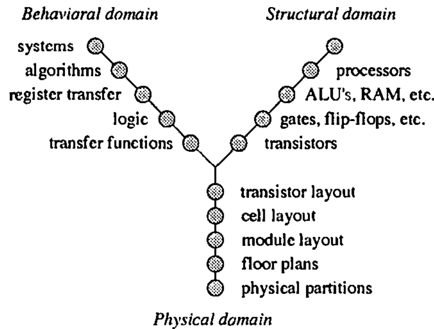


Fig.2.1 The visualization of the three design domains in Gajski's Y-chart.

Modularity is a result of hierarchy. In general, modules are functional decompositions of a circuit. They allow for the definition of an unambiguous interface between the module's inside and outside containing behavioral, structural and physical data. Modules are often well suited to be parameterized (RAM, ROM, n-bit adder etc.), which enhances their reusability.

The repeat of identical cells in a design is an example of regularity. It helps to improve the performance of a design system because instances, instead of copies, may be used. Regularity my be explored at all levels of the design hierarchy: identical transistors at the circuit level, identical gate structures at the logical level, bit slices at the architectural level etc.

Locality helps to hide internal details from the outside world. This means for instance that modules are interconnected first, only the required connections are exported to higher hierarchical levels.

Lower level details of IC design may be simplified using abstraction. This technique helps to hide structural and physical domain information as well as process design rules. This technique is extensively used in symbolic design strategies.

Abstraction not only simplifies the design, it may also increase the flexibility with respect to the evolution of technologies. The same design might be used for different technologies, and technology updates may be propagated more easily.

2.2 Hand-crafted layout

Hand-crafted layout is the term applied to less constrained design techniques that involve, at some stage, the layout of functional subsystems at the mask level. This is the oldest form of chip design and still the most widely used. Essentially, it requires that a design is divided among designers with expertise in logic, circuit, and process details.

Note that as total freedom is allowed at the physical level, the structural specification and hence the behavioral description may differ from the one required. This imposes the need for adequate circuit extraction techniques, which take as input a mask description and then present the designer the corresponding circuit description (netlist).

The technology dependence of the hand-crafted layout is complete. As all information is entered at the geometrical level for a specific technology, there is no way to reuse the layout directly. In [CON92] *shrink and size* migration techniques have been analyzed, but stay somewhat primitive. Only a linear scaling may be achieved easily.

2.3 Gate arrays

The gate-array (together with the sea-of-gates) approach has gained a widespread popularity as an LSI-VLSI implementation medium, and it has even been used in the domain of analog circuits [DEC92] (see also chapter 5.5). Together with the standard cell approach, it benefits from the compatibility with TTL[1] design that makes it easy to transfer a system from breadboard to silicon. Moreover, the cost of a gate-array is potentially the lowest of the methods described in this chapter, but the application domain of gate-arrays is restricted to specific classes of integrated circuits.

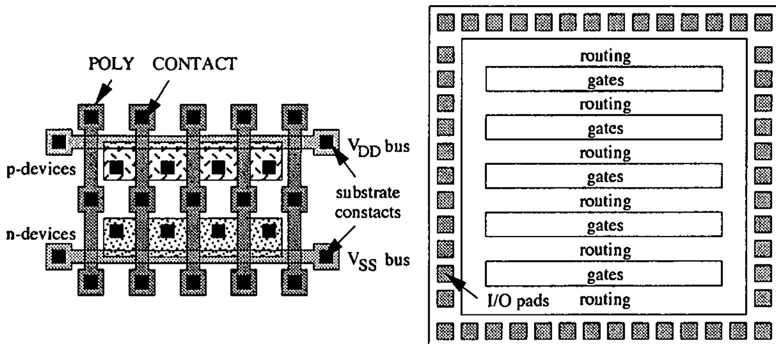


Fig.2.2 Typical gate-array gate organization and floor plan

Gate arrays exist in many different forms, but they can be characterized by a design that uses a large number of identical blocks, each of which consists of a number of circuit elements; n- and p-transistors in a CMOS process. The area outside the blocks is reserved for the interconnection between these elements.

[1] TTL design is used here as an example of a data book oriented design style

The more recently developed sea-of-gates method differs from the previous in the sense that no specific interconnect areas are reserved. Instead, the routing is done on top of the transistors.

So, the basic topology of a gate-array is fixed. This allows a vendor to process the wafers until a given fabrication step (usually the metallization). The implementation specific steps may concern a routing in a single metal layer, and eventually contacts, a double metal routing with contacts, and vias etc. As a result, the mask cost is reduced to 15-25% compared to a full custom design. Further savings concern the fast design time (due to specific gate-array netlist-to-mask mapping software), and the reduced time-to-market.

A drawback of the gate-array approach may be the wasted chip area. All blocks are in place, whether they are used or not. Furthermore, the fixed circuit configuration may lead to inefficient occupation of the area (cf. memory). Several vendors have tried to circumvent these problems by providing gate-arrays with predefined memory blocks.

2.4 Standard cells

The standard cell approach is grounded on a set of predefined logic and circuit blocks, principally in the digital domain. Predefined logic is based on a fixed floor plan: the cells have a fixed height, and all connections are located on the bottom or the top of the cells, so that these cells can then be abutted side by side. Cells are placed in rows, and routed through channels. Circuit blocks consist of regular circuitry such as RAM, ROM or PLA. Again the method is attractive for TTL designers; the cells can be organized according to a TTL data book.

The main advantage of standard cells compared to gate-arrays is the higher flexibility, especially on the floor plan level. Contrary to gate-arrays, specific blocks can be included in the design. Still, gate-arrays may compete with standard cells because a gate-array technology allows usually smaller gates. The time-to-market time of standard cell circuitry is longer as all masks need to be fabricated.

Moreover, standard cells can be synthesized automatically [HWA93, SHI88], and they may be optimized for speed or power consumption [MAS91]. This property extends the application domain of this approach.

Still, in many cases, the standard cells themselves are designed by hand and depend upon a single technology. Their design effort is recovered from the fact that they are reused multiple times in different designs. A complete set of standard cells (100-400 cells) is usually available in a library, and can, as such, be purchased from different CAD vendors.

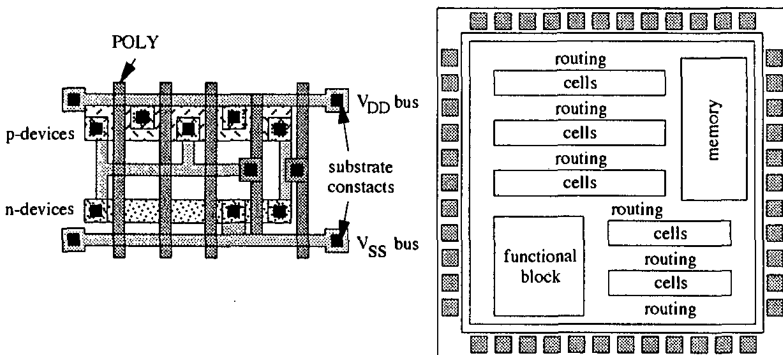


Fig.2.3 Typical standard cell predefined logic and floor plan

The standard cell layout style must be considered as technology dependent, because the introduction of a new technology requires an additional design effort other than the characterization of the technology itself.

2.5 Symbolic design approaches

The idea of the symbolic layout representation is to simplify the lower level details of IC design by hiding process rules and capturing structural and physical domain information. This approach may be compared to an assembler code in the software domain. It is sufficiently low in the design hierarchy to allow for most operations, but the system designers is not confronted with over-detailed process specific characteristics.

Symbolic design approaches are real technology independent layout styles: they allow to add a new technology (within some limitations) without modifying the original cell descriptions.

Notice that symbolic design approaches are generally applied in digital design environments. Apart from historical reasons (stronger development of tools for digital circuitry), this can be explained by the fact that analog layout requires is much more sensitive to its physical implementation (symmetry, parasitic effects etc.). A symbolic abstraction does not allow for a sufficiently precise description.

Fixed grid layout

The idea behind a fixed grid symbolic layout is to impose a uniformly spaced grid in both X- and Y-directions (a matrix). The spacing between adjacent grid lines is chosen sufficiently large to avoid (most) design rule errors. Then, symbols are defined for each valid combination of mask layers that can be put on a grid location.

The design is made by placing symbols on the grid; each symbol is replaced by its layer(s) later on. The major drawback of this method is the low density which can be obtained, due to the large grid spacing. Therefore, a posterior compaction phase is often required to reduce the cell's size. Nevertheless, the result is too far from the optimal geometrical layout to be of interest for industrial applications.

Gate-matrix layout

The gate-matrix design style uses a grid to represent the gate positions in a circuit. Here, rows of diffusions and columns of polysilicon are used to create transistors at their intersections. Both silicon and polysilicon is used to interconnect vertically, and metal is used for both horizontal and vertical tracks. Again, symbols are used to distinguish the different cases.

The gate-matrix technique provides a higher density than the fixed grid layout style because of its regular organization of the circuit elements. However, the strong "in the cell" routing still provides relatively poor occupation of the available space.

Note that this method explores regularity in the layout style, whereas locality is provided by the in-the-cell routing technique. Especially bit-sliced structures benefit from these properties. Circuit extraction may be done at the symbolic level. However, local device optimizations are not possible, electrical characterization is difficult, and the description of more advanced layout (floor planning) may become tedious.

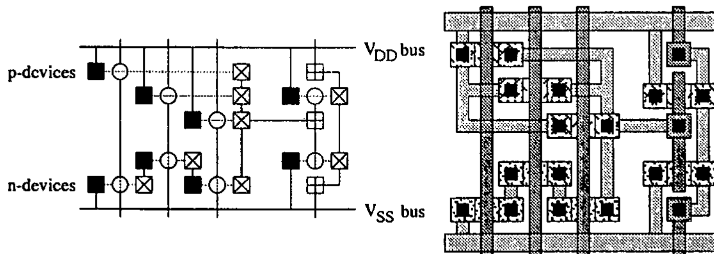


Fig.2.4 Typical gate-matrix symbolic input format and resulting layout.

A special gate-matrix layout style is the branch-based gate-matrix approach. Here, all circuitry is described as a set of branches only. This restriction makes the branch-based method very suitable for automatic synthesis.

Sticks

Contrary to the grid based approaches, which uses a grid spacing to satisfy the design rules, the sticks design styles is based on a free form topological description of a layout. This description is entered via an interactive graphics system, and consists of colored masks representing mask level interconnections (the *sticks*). Only the relative placement of the symbols is taken into account, and the final result is computed by passing a compaction phase. As the constraints for the compactor are created based on an initial solution, the compaction sequence (*XY*, *YX*, or even *XYX*, *YXY*) plays an important role. Indeed, dependent on the choosen sequence, the compactor may be blocked by non-stretchable elements.

The symbolic editor of COMPASS [COM91] is an example of a sticks based editor. It allows to enter small topologies rapidly and to create the layout for different technologies. However, these technologies need to have the same characteristics, and parametrization of cells is very limited. Furthermore, hierarchy is not supported. As a consequence, the method is not suitable for the creation of generators.

Virtual grid

The virtual grid design style can be considered as a result of the experiences of the fixed grid, gate-matrix and sticks type systems. Now, circuit elements (transistors, wires, contacts etc.) are manipulated directly. These elements are placed on a grid to facilitate the design capture. The spacing between adjacent grid lines is determined by the interference of circuit elements on their neighbors. Once a circuit is defined, a virtual grid compactor determines the final location of the grid lines.

The designer places the elements regardless of the design rules. Instead, it is the system which determines the spacing to be used to assure a design rule error free result. It is also the system which may resolve context dependent cases. Finally, the grid is also used to define the circuit connectivity.

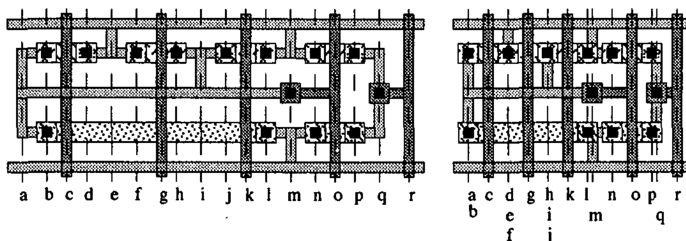


Fig.2.5 The virtual grid method illustrated with a horizontal compaction. Elements which are initially on the same grid line have the same final coordinate.

Virtual grid systems have been quite successful. This is probably due to the fact that virtual grid compactors are fast and that they do not distort the layouts more than perhaps the designer wants. As the result is predictable, users "trust" the system.

Still, one of the drawbacks of pure virtual grid compactors is the constraint that elements on the same grid line (either horizontal or vertical) remain on the same axis. Recent approaches [DUF92] allow for local *off grid* placement of elements to increase the density.

Graph-based compaction

The graph-based (or relative grid) compaction is a gridless approach which allows for the best results in terms of density: circuit elements are only related to each other with spacing rules. Now, a directed constraint graph is built where the nodes represent the circuit elements (or groups of them), and the edges constraints between these elements.

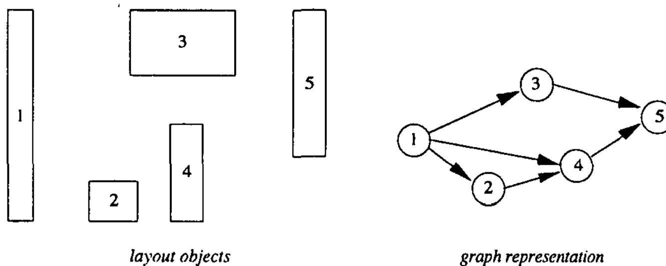


Fig.2.6 Graph based compaction

Once the graph has been constructed, the path with the greatest spacing requirement is determined (the *critical path*). All nodes on this path are on their final location, the remaining ones need to be placed in some convenient way (for instance while applying a yield enhancement strategy).

So far, the graph based compaction method has been considered to be time consuming. Chapter 3 will demonstrate how graph reduction techniques, and the improved computation performance, may eliminate this drawback.

2.6 Automated layout synthesis

Many attempts have been made to map the behavioral descriptions via intermediate forms, such as boolean logic, to layout. The interest for such an automated synthesis is clear, is does not only free the designer from the tedious task of layout generation, but it also allows him to verify the circuit down to the layout level. Among the main trends are [SHI83]:

- register transfer level to logic level
- global design space "explorers"
- local design space synthesizers
- logic array synthesizers
- silicon compilers
- human creativity based systems

Most approaches are based on a boolean logic lower level form, which limits the types of synthesizable circuitry. Here, two methods are presented which allow to synthesize arbitrary circuit types, even in the analog domain.

Procedural module definition

A particular design style or architecture may be captured by a procedural modeling approach. Here, the design variations that might by required by the designer are coded into a *generator* program. Such a program is written in some high level language that specifies some lower level format; either at the symbolic or at the mask level. The former is more flexible in response to technology scaling and variations.

An important feature of procedural layout (contrary to most graphics based systems), is the ability to parameterize designs in terms of repetitions, conditions or other procedural constructs. This flexibility is essential for the creation of module generators.

An interesting approach is given with the tool ACAP [EES91], which allows for the graphical capture of module generators. By using incremental compaction to position objects in a layout, and capturing these operations, textual generators can be constructed without typing a line of code.

Silicon compilers

A silicon compiler may be defined as an automatic translation tool that converts a behavioral description into a mask level description. Usually, a silicon compiler has the notion of a fixed floor plan, with particular architectural constructs. They can be extremely useful to synthesize well defined types of circuits rapidly (such as a sequence of second order filters), and to explore space/time trade-offs.

The best known silicon compilers are probably the so-called data-path compilers [GRA92, JER86, MAR86, ROW87, AZA90]. Here, some kind of high-level behavioral description is used to generate first a gate-level netlist, which can then be used for the mapping to gate-array, standard cell or bit-slice type layout architectures. Note that in general, the layout of all the leaf-cells is to be provided for the specific technology. An exception is formed by real automatic layout synthesis programs, such as Gencell [ROB92], but the

modeling of the circuitry may become troublesome. Other examples of silicon compilers may be found in Cathedral II [MAN88].

2.7 TILT

The idea behind *TILT* [RIE92] (Technology Independent Layout Tool) was to offer a layout design environment that allows to capture design knowledge in generators, independent on the fabrication technology. This technology independence should be interpreted in two ways, namely:

- a generator can be executed for different technologies that are of the same family (e.g. CMOS, or bipolar)
- the tool is applicable for different types of technologies, e.g. CMOS, bipolar, BiCMOS, or "exotic" technologies

While analyzing layout practice, it appeared that leaf cells were very often made by hand, and that the composition and interconnection of cells was usually done with commonly available place and route tools. The main reason for this approach is that it is the only way (so far), which can obtain the optimum density of the cells. The time-effort trade-off is the only limiting factor.

The aim of TILT was to be competitive with respect to hand-crafted layout. Therefore, the decision was taken to structure the tool such that it would allow for the description of arbitrary layout, restricted to Manhattan type geometries. The advantages of the tool, compared with hand-crafted layout, would concern the technology transparency, the possibilities for parametrization, and reusability. From the beginning, no particular attention was paid to the creation of a netlist; at this point the tool is similar to a hand-crafted approach. The motivation for this choice was that a netlist is technology independent anyway (at least in a very high degree), and reliable methods to create a netlist from a symbolic description are well understood.

This section presents the global structure of the tool. After a short historical time setting, an overview of the main elements of the tool is given. Not all parts will be treated in detail in this document. Only the principal items, the *compaction* and the *layout description language*, are presented in two separate chapters. The in-house experiences (see chapter: *Applications*), illustrating the different features, provide some realistic case studies.

History

One of the preliminary decisions during the definition of a CAD-tool project concerns its desired relations with other software. In many cases, a tool is integrated from the beginning into an existing framework. The advantages are multiple, especially at the level of user interfaces, data compatibility, and

data-bases. From the technical point of view, such an approach may be preferred, as a research project may concentrate on the core problem, not on the overhead.

On the other hand, the dependency on a commercial framework implies an increased cost for those which are only interested in the tool being developed. Given the cost of frameworks provided by Mentor, Cadence, Compass and others, this may be a decisive factor, in particular for small and medium sized companies. Also, the portability may be an important issue during the design, for instance for reasons of maintenance. The price to pay is the large increase of overhead during the design phase, for example to establish a convenient user interface. Often, many interfaces and data structures need to be implemented from scratch, which represents a large design effort.

Based on the experiences obtained with LISA [SWA88], a decision was made to implement TILT under UNIX on multiple workstations in ANSI C [KER88], and to base the user interfaces on X-Windows [QUE90]. Given the distribution of this software, a large portability could be obtained without too much additional effort. This choice was made in 1988. Today, C^{++} [STR87] together with Motif [OSF90] or XView [HEL89], would be more convenient.

Also, emphasis was put on the open character of the system. It is impossible to foresee all the requirements of potential users. They must be able to add the missing features themselves.

Design and data flow

As the primary goal of the tool was to be able to create technology independent parameterized generators, the need for a high degree of flexibility was apparent. This desire is reflected in the tool by two items:

- the use of a language to describe the generators
- the use of a compactor to create the layout

The choice of a language is motivated by the required degree of parametrization and precision. Graphical input is well suited to enter topological information, but a textual description is better adapted to express algorithms and data flow. As such, a language is a good medium for capturing design knowledge. Also, the experiences in the domain of symbolic layout have shown their limitations. A simple conditional placement is usually cumbersome or impossible to achieve.

A compactor, as a core part for the effective layout generation, can be motivated by the fact that it is an excellent constraint solver. Structure can be well captured with a set of constraint declarations, which in turn may be implemented using a graph. Graph theory provides powerful and well understood algorithms to solve these relations efficiently. Remark that the

compactor is used as a main process, not as a kind of post-processing utility in order to improve initially too bad layout.

A technology independent design should not imply a worst case design. Technology specific topological options and properties may ask for different solutions. As the treatment of these alternatives has a more sequential character, they are handled by the language. Thus, the compactor "sees" a technology *specific* set of relations.

The coupling between the language and the compactor is straightforward: a generator program is created out of the source code, and the execution of this generator provides a set of constraints, which are solved by the compactor. The layout is obtained by the transcription of the compactor data into a mask description format.

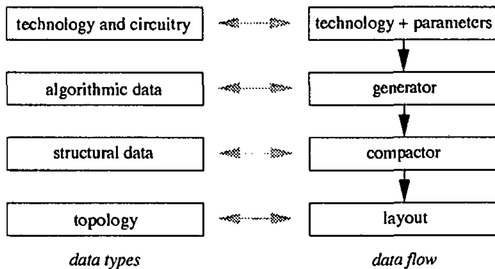


Fig.2.7 Abstract view of the tool TILT

The abstract view of the tool shows two aspects: one which focuses on the data itself, the other on the data flow. These items correspond in fact to the two stages in the layout design process: the *creation* of the generator (= capture of data), and the *execution* of the generator (= creation of data).

Compaction

In conventional compaction based approaches, a compactor is used to reduce the size of a too widely spaced layout. The required constraints are extracted out of this initial layout, using additional information on the technology dependent design rules. As such, this operation can realy be seen as a *compaction*; the results tends to be smaller than the original.

The method applied in TILT is essentially different from the technique described above. Here, the constraints are generated directly, not out of an initially badly spaced layout. As a result, one is able to describe arbitrary layout[1], the only condition is to create the correct constraints. Thus, the role of the compactor is not to compact, but to solve the constraint graph.

[1] restricted to Manhattan type layout

As a side effect, it is not possible to use the benchmarks that are available to compare the performance of the compaction. These benchmarks allow to measure the size reduction of some typical layouts. As the TILT approach directly generates the constraints, such a comparison is not possible.

In the conventional approach, non-stretchable elements (e.g. wires) may block the compaction process, and thus limit possible size reduction. Some approaches try to solve this obstruction by inserting dog-legs automatically, but the results become difficult to foresee. Moreover, the order of compaction (x- or y-direction first) plays a role, because the output of the compaction in the first direction is used as the input for the compaction in the second direction.

The ambiguity of the compaction order does not exist in TILT. This results from the fact that the constraints in both the x- and y-direction are created at the same time without *any* interdependence. A similar behavior would be obtained if all the constraints in the conventional approach are created before starting the compaction. If an element *a* is located on top of and and the right of element *b*, it will remain at this relative position all over the compaction phase.

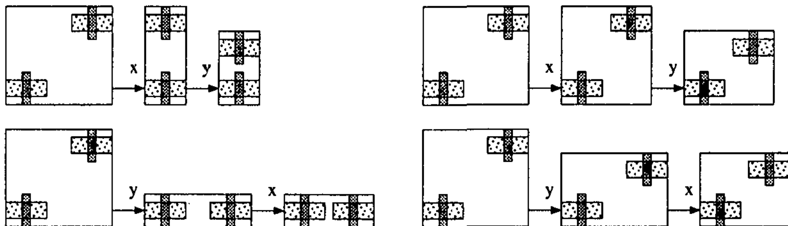


Fig.2.8 *Conventional compactors (left) are sensitve to the order of the x- and y-direction, because the second compaction step creates its constraints out of the results of the first step. In TILT (right), all constraints in both directions are known before starting the compaction (spacing), which makes it independent on the direction.*

Whereas such an approach would lead to poor results in a conventional compaction approach, this property is desired in TILT. Here, the presence of a constraint corresponds to a desired positioning.

Implementation

The way from an abstract data flow to a running tool is long. We will not discuss all the intermediate versions here (there were so many), nor the solutions which would work best in theory. Instead, a brief overview of the different TILT utilities and their internal relations are shortly discussed. It provides a realistic overview of a realized and tested layout design environment.

The first part of the system concerns the *creation* of the technology independent layout generators and the technology data bases. Here, the design and the process data are captured. A detailed knowledge of the language al2, and eventually C is required. Furthermore, it is an advantage to be familiar with common UNIX facilities such as *make* and library archiving.

The second part of the system only *executes* the generators. At this stage, only the technology and the parameters are to be given, and no detailed knowledge of the al2 language is required. At run time, the generator solves the description it contains, and it creates the primitives for the compactor. Overconstraints will occur during the resolution of the constraint graph, but should have been eliminated during the test of the generators.

Finally, a few data conversion programs allow to transform the internal layout format into CIF [MEA80], PostScript [REI90], or to display the layout on the screen. The latter is of great importance during the creation of the generator, because it allows to check the result visually. The interaction between the textual layout description and the image on the screen is largely simplified because the variable names are automatically propagated to the geometries on the screen.

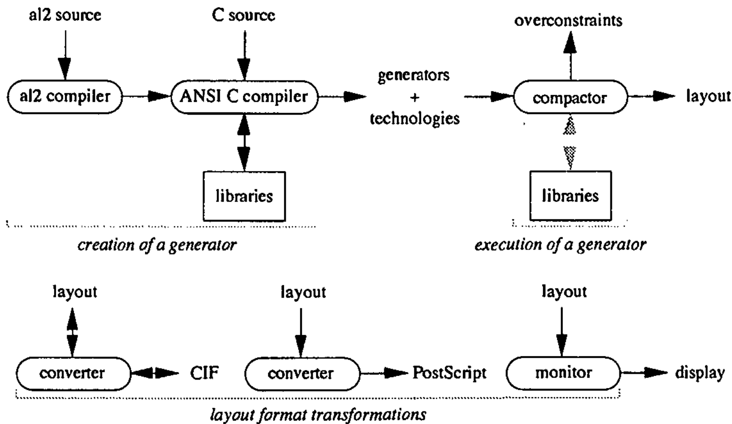


Fig.2.9 Different items of the TILT implementation

The separation of the generator creation and execution phase largely simplifies the establishment of links with commercial systems. As a matter of fact, the generators are UNIX-like pipeline style programs that produce their result on the standard output stream with an integrated interface. This means that they can easily be integrated into a commercial mainframe, without disturbing the consistency of the software.

2.8 Conclusions

The technology independence of layout design on most commercial VLSI systems is a *virtual* technology independence. The user is able to select the technology mapping onto a specific set of library elements. Libraries are to be purchased from the CAD vendor, and layout of eventual non-supported technologies need to be provided explicitly.

Symbolic design approaches offer a better alternative in the sense that they try to separate the topological data from the technology specific data. These methods are generally constructed on top of a grid based graphical entry. As such, they allow to capture the connectivity implicitly, but they have only limited possibilities with regard to parametrization and hierarchy.

Silicon compilation is restricted to a specific type of layout architectures, such as a data-path or bit-serial and bit-parallel signal processing circuitry. Again, the technology independence is virtual, it is based on the definition of technology specific leaf-cells.

The TILT-approach is different from the previous ones in the sense that it is designed for parametrization. The combination of an object oriented description language and powerful compaction techniques allows to create reusable and technology independent layout generators. As such, the method is complementary to the previous approaches and can be used, for instance, for the creation of libraries.

2.9 References

[AZA90] Azato, Ditzen & Dholakia, "*A data-path multiplier with automatic insertion of pipeline stages*", IEEE journal of solid-state circuits, vol.25, No.2, pp.383-397, April 1990.

[COM92] COMPASS Design Automation V8R3, "*LAYOUT TOOLS, Symbolic Editor, VIP Language*", 1991.

[CON92] B. Conq, R. Etienne, T. Perez-Segovia, "*Design Library Portability: A Case Study*", WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, pp.174-182, March 12-13 1992.

[DEC92] M. J. Declercq, P. Duchêne, O. Buset, "*Design & Performance of Analog Arrays for Semicustom IC Design*", Proc. Eurochip workshop on VLSI training, Grenoble, 1992.

[DUF92] J.C. Dufourd, J.F. Naviner & F. Jutand, "*PREFORM: a process independent symbolic layout system*", Proc. ICCAD 1990.

[EES91] R.A. Eesley & M.A. Tarsi, "*ACAP - A System for the Interactive Graphical Capture of Module Generators*", Proc. CICC 1991.

[GRA92] V. Gravoulet & B. Marie, "*A netlist and layout datapath compiler*", WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, March 12-13 1992.

[HEL89] D. Heller, "*XView Programming Manual, for version 11 of the X Window System*", IO'Reilly & Associates, Inc., Third edition, 1989.

[HWA93] C.Y. Hwang, Y.-C. Hsieh, Y.-L. Lin, and Y.-C. Hsu, *"An Efficient Layout Style for Two-Metal CMOS Leaf Cells and Its Automatic Synthesis"*, IEEE Trans. on CAD, Vol.12, No. 3, March 1993.

[JER86] Jerrya, Varinot, Jamier & Courtois, *"Principles of the SICO compiler"*, Proc. DAC, 1986.

[KER88] B.W. Kernighan & D.M. Ritchie, *"The C Programming Language, 2nd edition"*, Prentice Hall, 1988.

[MAN88] H. de Man, J. Rabaey, J. Vanhoof, G. Goossens, P. Six, and L. Claessen, *"CATHEDRAL II - a Computer Aided Synthesis System for Digital Signal Processing VLSI Systems"*, Computer Aided Engineering Journal, pp. 55-66, April 1988.

[MAR86] Marshburn, Lui, Brown, Cheung, Lum & Cheng, *"DATAPATH: A CMOS Datapath Silicon Assembler"*, Proc. DAC, 1986.

[MAS91] J.M. Masgonty, C. Arm, C. Piguet, *"Technology- and Power-Supply-Independent Cell Library"*, Centre Suisse d'Electronique et de Microtechnique CSEM SA, IEEE 1991, Custom Integrated Circuits Conference CICC'91, San Diego CA, USA, May 12-15 1991.

[MEA80] C. Mead & L. Conway, *"Introduction to VLSI Systems"*, pp.115-127, second printing, October 1980.

[MIC87] G. De Micheli, A. Sangiovanni-Vincentelli & P. Antognetti, *"Design Systems for VLSI Circuits, Logic Synthesis and Silicon Compilation"*, Martinus Nijhof Publishers, 1987.

[OSF90] *"OSF/MOTIF, Programmer's Guide, Revision 1.0"*, Prentice Hall, 1990.

[PIG90] C. Piguet, *"Technology Independent Layout Tools"*, Intensive Summer Course on CMOS VLSI Design '90, EPFL Lausanne, August 27 - September 14.

[QUE90] V. Quercia & T. O'Reilly, *"X Window System, User's Guide"*, O'Reilly & Associates, Inc., Third edition, May 1990.

[REI90] G.C. Reid, *"PostScript, Language Program Design"*, Adobe Systems Inc., seventh printing, December 1990.

[RIE92] R. Riem-Vis, G. Maliki, & F. Pellandini, *"TILT, a Technology Independent Layout Tool"*, WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, pp.88-93, March 12-13 1992.

[ROB92] R. Jamier, *"Gencell: a CMOS random logic generator"*, WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, pp.148-157, March 12-13 1992.

[ROW87] Rowson, Walker & Dholakia, *"A datapath compiler for standard cells and gate arrays"*, IEEE Proc. CICC, May 1987, pp. 149-152, 1987.

[SHI83] S.G. Shiva, *"Automatic Hardware Synthesis"*, Proc. IEEE, Vol. 71, No. 1, pp.76-87, January 1983.

[SHI88] Y. Shiraishi, J. Sakemi, M. Kutsuwada, A. Tsukizoe, and T. Satoh, *"A high packing density module generator for CMOS logic cells"*, Proc. 25th IEEE DAC, pp.439-444, June 1988.

[STE84] G.L. Steele Jr., *"Common LISP: The Language"*, Digital Press, 1984.

[STR87] B. Stroustrup, *"The C^{++} Programming Language"*, Addison Wesley, 1987.

[SWA88] M. van Swaaij, *"LISA, A Declarative Language For Interactive Layout Generation"*, CSEM technical report no. 193, Neuchâtel, Switzerland, 1988.

[WES85] N. Weste K. Eshraghian, *"Principles of CMOS VLSI Design, A Systems Perspective"*, Addison-Wesley, 1985.

3 Compaction

3.1 Introduction

Compaction and spacing programs have been in use since the early 1970s [LAR71]. These programs attempt to minimize the distances in between layout components and the overall surface occupied by these components, while satisfying a set of constraints.

Still, these programs have not been as widely used as might be expected. The principal reasons which can be mentioned are:

- early compaction systems did not support both lower-bound and upper-bound constraints
- electrical considerations are rarely taken into account
- it is difficult to edit and to express constraints

Especially those approaches which apply compaction as a post-process on an initially widely spaced layout suffer from this lack of control over the final result. The result is often somewhat arbitrary and difficult to foresee.

According to [NEW87], a useful layout spacing program in a modern CAD framework should have the following capabilities:

- full support of upper-bound and user-defined constraints
- *detection* and *identification* of overconstrained elements
- an efficient constraint-graph solution algorithm
- adjustable positioning of non-critical-path circuit elements
- constraints both *within* and *among* circuit elements to provide a uniform approach to technology changes
- dependencies *among* constraints to enforce symmetry during spacing, for analog and high-performance digital layouts

It may be noted that most of the requirements concern the control of the

generated result. So far, no layout compactors are known which handle all of these requirements. The approach which is presented in this chapter is not an exception to this rule, but it meets most of them.

After this introduction to spacing and compaction programs, the following section defines the actual compaction problem, and some conventions for the terminology used. The next section presents the two major modules of a conventional compaction system: the *constraint graph builder*, and the *constraint graph solver*. It is interesting to notice that local optimizations and overconstraint detection are performed during the graph construction. Due to these optimizations, run times of the critical path analysis and slack distribution are reduced.

A full section is dedicated to a hierarchical approach of compaction. It is shown that the use of an extracted graph model leads to acceptable resolution times. The following section considers different items that may be of interest to an incremental compaction strategy.

Then, the interface between the layout and the constraint graph is presented. It is shown that a well chosen high level interface may help to describe the compaction problem in an efficient way. Furthermore, it provides a clear identification mechanism, essential during the report of overconstraints.

Overconstraints are often treated as a side effect of the compaction problem. However, a clear and efficient report of unfeasible sets of constraints is essential because only feasible graphs yield a valid result.

After a paragraph treating some special constraint types, statistical results on the performance of the implemented algorithms are presented. A number of measurable parameters are defined, and different scenarios are compared. The conclusions terminate this chapter.

3.2 The compaction problem

A general formulation of the compaction problem is usually expressed in terms of two constraint graphs $\Gamma_x(V,E)$ and $\Gamma_y(V,E)$. These graphs represent constraints among objects in the horizontal and vertical direction. In the following description, the subscripts x and y have been dropped for convenience. A formal definition of the compaction problem P is given below [GON79]. Both the total size, and the total cost have to be minimized constrained to a set of inequality relations (fig.3.1).

Each vertex $v_i \in V$ represents an *object*, and each directed edge $e_{ij} \in E$ represents a *constraint* between vertices v_i and v_j. An edge e_{ij} has a value s_{ij} and a cost c_{ij} associated to it. Values are used to express a distance between

vertices, whereas costs on edges indicate the desire to privilege the edge value to be maintained. A *fanin* edge at v_i is defined as a directed edge whose head is incident at i. Similarly, a *fanout* edge at v_i is defined as a directed edge whose tail is incident at i. The total number of fanin and fanout edges at vertex i are denoted $N_{fi}(i)$ and $N_{fo}(i)$, respectively.

$$(P)\quad \begin{cases} \min\ v_{max}-v_{min} \\ \min \sum c_{ij}(v_j-v_i),\ i,j=(1,\ldots,N) \\ v_j-v_i \geq s_{ij} \\ v_i \geq 0,\ c_{ij},s_{ij},v_i \in \mathbb{Z} \end{cases}$$

Fig.3.1 The compaction problem P as a set of linear inequalities on integer variables.

The inequality relations that imply vertex pairs are explicitly generated by the user or the system. The relations which state that each vertex should be larger or equal to zero, are implicit relations; no constraints are created to represent them. Note that all constraint and vertex values are integers. This restriction is primary included for reasons of data efficiency. As the photo masks used to create a layout have limited resolution, it does not limit the possibilities of the compactor: all coordinates are multiples of the resolution.

$$(a)\ v_j-v_i \geq s_{ij}$$

$$(b)\ v_j-v_i \leq s_{ij} \Leftrightarrow v_i-v_j \geq -s_{ij}$$

$$(c)\ v_j-v_i = s_{ij} \Leftrightarrow \begin{cases} v_j-v_i \geq s_{ij} \\ v_j-v_i \leq s_{ij} \end{cases} \Leftrightarrow \begin{cases} v_j-v_i \geq s_{ij} \\ v_i-v_j \geq -s_{ij} \end{cases}$$

Fig.3.2 The explicit inequality relations for (a) a lower-bound constraint, (b) an upper-bound constraint, and (c) a fixed-bound constraint.

Three basic constraint types can be expressed in terms of explicit inequality relations: the *lower-bound* constraint (minimum spacing), the *upper-bound* constraint (maximum spacing), and the *fixed-bound* constraint (fixed spacing). They permit to describe most of the practical cases. Exceptions are formed by the so called *active* constraints and *exclusive* constraints. They will be treated in more detail further in this chapter.

Without loss of generality, the constraint graph can be converted into a directed polar graph with the addition of a source vertex v_{so}, and a sink vertex v_{si}. The source vertex is connected to all vertices v_k where $N_{fi}(i) = 0$ with a zero weighted, zero cost edge $e_{so,k}$, and all vertices v_k with $N_{fo}(i) = 0$ are connected to the sink vertex v_{si} with a zero weighted, zero cost edge $e_{k,si}$. Dependent on the implementation, it may be possible to use only *virtual* zero weight, zero cost edges inside the algorithms.

3.3 Constraint management

Usually, the constraint graph is constructed from the relative placement of the components and a set of spacing rules. Constraints may be either *user constraints* or *system constraints.* User constraints are defined by the user and are under his direct control. System constraints are automatically inserted by the compaction system, and no direct control is left to the user. In principle, user constraints are random. Beside the problem of overconstraints, it is possible that a constraint is already satisfied by other constraints in the graph. In this case, the constraint is *redundant,* and its insertion may result in a loss of performance.

This paragraph presents how part of the redundant constraints can be removed during the graph construction. Then, the different algorithms used to solve the constraint graph are presented.

These algorithms are explained in a chronological order: from the longest path determination, via the critical path reduction, to the slack distribution, and centering. Then, a new zero cycle reduction algorithm is presented which may eliminate the explicit critical path reduction phase.

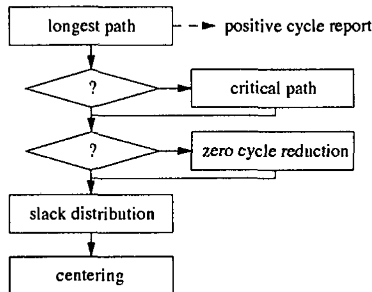


Fig.3.3 The compactor allows for different scenarios such as critical path reduction, elimination of zero cycles, both, or none of them.

For reasons of clarity, costs are ignored until the presentation of the slack distribution. Furthermore, the compaction is assumed to be non-hierarchical, as the introduction of hierarchy in the compactor is shown to be compatible with the mechanisms presented here.

Longest path determination

The first step of the graph resolution concerns the determination of the longest path. At the mask level, this corresponds to the minimum overall size of the layout. A necessary condition for the longest path to exist is that the

graph must be free of strictly positive cycles[1], because they represent contradictory constraints[2]. A cycle is a path which starts and ends in the same vertex. The length of a path is the sum of the edge values which are on the path.

It is known that the *existence* of positive cycles can be detected in the time required to solve a legal graph [BAL82]. As an example, the Bellman and Moore algorithms use the iteration count to verify whether positive cycles exist or not. However, they cannot report where these cycles are located in the graph.

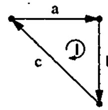


Fig.3.4 *Example of a cycle with length l=a+b+c. Dependent on the value of l, we speak about a negative cycle, a zero cycle, or a positive cycle. Only positive cycles lead to unfeasible graphs.*

The spacer described in [KIN84] handles overconstraints by consecutively ignoring constraints until the graph becomes feasible, resulting in an illegal layout. SPARCS [BUR86] uses a two pass process: a first pass attributes a level to each vertex, and a second pass detects the paths that are involved in an overconstraint.

```
Min(Γ)
{
    count = 0, change = True;

    foreach(vertex v_i in Γ)
        v_i.min = 0, schedule(v_i);

    while(count < Γ.vertexCount and change) {
        increment count, change = False;
        while(v_j = getNextVertex())
            foreach(fanout edge e_jk)
                if(v_k.min < v_j.min+L_jk)
                    change = True, v_k.min = v_j.min+L_jk, schedule(v_k);
    }
    return(count);
}
```

Alg.3.1 *An adaptation of the Bellman algorithm to set the minimum locations in a directed graph. If count equals the vertex count on return, one or more strictly positive cycles exist.*

Only graphs *without* positive cycles are of interest, and these are to be solved as fast as possible. For that reason, it should be avoided to slow down

[1] Strictly positive cycles will be called positive cycles, but they exclude zero cycles

[2] The simplest example of a positive cycle is created with the set of constraints: A is on the left of B, *and* B is on the left of A

an algorithm with cycle detection and report. Instead, a dedicated algorithm should be applied when positive cycles exist. This cycle report algorithm should be designed for clarity because the user is meant to intervene. Run time characteristics are not of prior interest here.

An algorithm based on Bellman, which sets the minimum legal positions in a feasible graph, is given in alg.3.I. The iteration count is used to detect whether positive cycles exist or not. First, all vertices are initialized to zero and scheduled for operation. Then, all scheduled vertices are treated until no scheduled vertices remain, or when the iteration count reaches the number of vertices in the graph (overconstraints).

The longest path algorithm returns the number of iterations. If it equals the number of vertices in the graph, at least one strictly positive cycle exists in the graph, but no indication is available where they occur. Otherwise, the sink vertex will contain the smallest upper border which satisfies the constraints. It yields the solution to the first part of the compaction problem.

Redundancy

In general, not all constraints in a constraint graph are strictly necessary to obtain the desired result. Especially in those cases where the user may add on constraints himself, we must assume that he cannot know beforehand whether this constraint will already be satisfied by some other constraints or not. However, the presence of these additional constraints in a graph $\Gamma(V,E)$ will reduce the performance of the algorithms; there are more constraints in the graph than effectively required. Part of this redundant information can be removed during the graph construction, namely:

- *dominant constraint:* if multiple constraints between vertices v_i and v_j occur, only the largest constraint need to be maintained[1]
- *fixed-bound constraint:* if two vertices are related with each other with a fixed distance, only one of these vertices needs to be considered

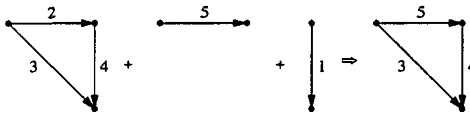


Fig.3.5 Only the dominant constraints need to be maintained during the graph resolution.

The detection of the dominant constraint among vertices v_i and v_j is straightforward: verify whether an edge e_{ij} exists, and if so, replace it by the new edge if the new value is larger than the existing one. Otherwise, the

[1] Still all constraint weights need to be considered, because they influence the slack distribution

constraint value is simply ignored. Note that this method introduces a search procedure to verify the existence of a constraint among the vertices v_i and v_j. Generally, the number of fanins and fanouts per vertex is smaller than 10, which limits the search time considerably.

As a side effect, the conservation of the dominant constraints only may reduce the number of possible cycles as well. In the case that two edges in parallel are both on a cycle, only the largest cycle will be preserved. As a consequence, zero cycles will be masked by strictly positive cycles, and negative cycles will be masked by zero cycles.

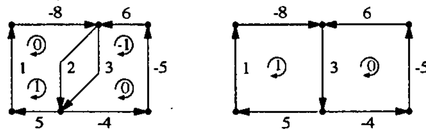


Fig.3.6 While considering only dominant constraints, only the longest cycles are preserved. On the left, the graph is shown when all constraints are inserted, on the right only the dominant constraints remain.

Fixed-bound constraints can be implemented with a lower-bound constraint and an upper-bound constraint in parallel, thus creating a zero cycle in the graph. Instead of inserting this zero cycle into the graph and finding it later to reduce it, one of the two vertices only needs to be considered. This vertex fusion process may lead to a significant reduction of the graph complexity for mainly two reasons:

First, it is applicable in all those situations where a vertex position is fixed relative to another. This is the case for two vertices which share a fixed-bound constraint, but also for those vertices which are on the critical path.

Second, the number of edges may be reduced because the vertex fusion includes the transfer of fanin and fanout edges[1]. Given two vertices v_i and v_j, all fanin edges e_{kj} and fanout edges e_{jl} are transferred to fanin edges e_{ki} and fanout edges e_{il}. The edge values are updated by adding the value $\delta_{ij} = v_j - v_i$. Possible edges e_{ij} or e_{ji} are removed, and there is a probability that an edge e_{ki} or e_{il} already exists, in which case only the largest edge is maintained.

Overconstraints may be detected during the vertex fusion. Two distinct cases may occur and need to be verified:

- if an edge e_{ij} or e_{ji} exists between the two vertices v_i and v_j, its value has to be smaller or equal to the position of v_j relative to v_i
- if a fixed-bound constraint has to be inserted between two members of the same vertex, the constraint value has to be equal to the member's difference in offset

[1] Note that the accumulated constraint weights need to be transfered as well

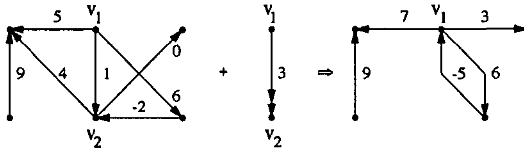


Fig.3.7 The vertex fusion process: all dominant fanin and fanout edges of v_2 are transferred to v_1 while adding the difference between the vertex positions to the edge values. Then, v_2 is removed from the graph (the fixed constraint is represented by a line with a double arrow).

In both cases, the following strategy can be followed: ignore the edge e_{ij} or e_{ji}, report the conflict, and continue the graph construction. In this way, other cycles may be detected in the same pass. The overconstraint message only needs to report the fixed-bound constraint being inserted and the constraint which corresponds to the edge e_{ij} or e_{ji}.

```
Fusion(Γ,va,vb,δ)
{
    foreach(fanout edge ebi)
        if(target equals va) {
            if(ebi+δ > 0)
                report positive cycle;
            remove ebi from Γ;
        }
        else if(no fanout edge eai exists)
            ebi = ebi+δ, move ebi to eai;
        else if(eai < ebi+δ)
            eai = ebi+δ, remove ebi from Γ;

    foreach(fanin edge eib)
        if(source equals va) {
            if(eib+δ > 0)
                report positive cycle;
            remove eib from Γ;
        }
        else if(no fanin edge eia exists)
            eib = eib+δ, move eib to eia;
        else if(eia < eib+δ)
            eia = eib+δ, remove eib from Γ;

    remove vb from Γ;
}
```

Alg.3.2 The vertex fusion procedure transfers all edges which concern v_b to v_a except for edges among these vertices. It reports if positive cycles are reduced.

The fusion of vertex pairs needs to be implemented as a reversible process because in some cases vertex positions are only fixed temporarily (e.g. critical path).

The critical path

Once the graph is known to be free of strictly positive cycles, the next step is usually the determination of the critical vertices in the graph. These vertices have the special property that their minimum possible location equals their maximum possible location. The computation of the maximum location can be done with an algorithm similar to the algorithm used to determine the minimum location.

All vertices which are on the critical path are already placed on their final location. In the context of vertex fusion, it is efficient to merge these vertices with the source vertex v_{so}, which is always on the critical path itself. In practice, this results in a considerable reduction of the remaining number of vertices and edges in the graph.

```
Max(Γ)
{
    foreach(vertex vi in Γ)
        vi.max = vsi.min, schedule(vi);

    change = True, vso.max = 0;
    while(change) {
        change = False;
        while(vj = getNextScheduledVertex())
            foreach(fanin edge ekj)
                if(vk.max > vj.max-Ljk)
                    change = True, vk.max = vj.max-Ljk, schedule(vk);
    }
}

Critical(Γ)
{
    if(Min(Γ) == Γ.vertexCount)
        Cycle(Γ);
    else {
        Max(Γ);
        foreach(vertex vi in V different from vso)
            if(vi.min == vi.max)
                Fusion(Γ,vso,vi,Li,so);
    }
}
```

Alg.3.3 The critical path is formed by those vertices which have equal minimum and maximum value. All these vertices can be merged with the source vertex, which is always on the critical path itself.

Unfortunately, the approach proposed in [CAT90] did not follow this strategy. Here, the zero cycles and the positive cycles were searched for during the longest path construction, resulting in quite large CPU times. The main advantage of reducing the critical path first is that all zero cycles on the critical path are reduced at the same time. Furthermore, it can be shown that the remaining critical vertex cannot be member of any zero cycle in the graph.

As the critical path may contain many vertices which can be connected to many non-critical vertices, a zero cycle reduction algorithm will run faster after the reduction of the critical path. The time invested to find this path is small, in the order of time to solve the legal graph.

Slack distribution

So far, only the elements on the critical path are placed on their final position. Still, the optimal position of the *non-critical-path* elements has to be computed. The original definition of the compaction problem defines that these elements need to be placed in such a way that the total weighted sum of the difference between the largest and the smallest vertex value is minimized. This cost function Λ will be the kernel of our slack distribution algorithm.

$$\Lambda = \sum_{i,j} c_{ij}(v_j - v_i) \Leftrightarrow \begin{cases} \Lambda = \sum_i a_i v_i \\ a_i = \sum_j c_{ij} - \sum_j c_{ji} \end{cases}$$

Fig.3.8 The cost function of the compaction problem is a linear equation of N variables.

Unfortunately, the minimization of such a function is known to have an exponential complexity. For that reason, different strategies were adopted in the past to limit the computational effort (see also fig.3.10):

- *low-left corner compaction:* place the vertices on their minimum legal position [WIL78,MOS81]
- *any corner compaction:* the user tries the minimum or maximum legal position in x and y direction and chooses the best
- *average slack:* such that the empty space is uniformly distributed, used in CABAGE [HSU79]
- *connectivity driven slack:* strongly connected elements are placed closer together than poorly connected ones [WOL85]
- *force driven slack* [BUR86]: use a positive integer force F_{ij} on each edge e_{ij} to indicate the relative attraction between the vertices incident to the edge and use a heuristic positioning algorithm such that:

  $$d_i \sum_{fanin} F_{ki} == d_o \sum_{fanout} F_{ij}$$

  where d_i represents the free space before the vertex, and d_o the free space after the vertex

With the increasing computational power of common platforms at a continuously decreasing cost, it becomes more and more interesting to review the possibility to calculate the optimum locations. The rest of this section will focus on an algorithm which finds such a solution.

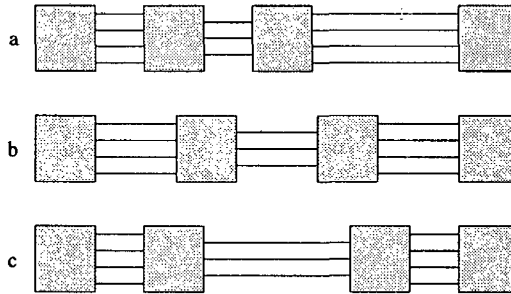


Fig.3.9 *Illustration of different slack distribution techniques: (a) left-bottom corner compaction, (b) average slack, (c) connectivity driven.*

First of all, it should be noted that the cost function Λ is linear. As a consequence, all solutions to this problem must be located on intersections between this function and the inequalities in the solution space. This solution space is known as the *simplex*, and different applications have been reported [MAR88, YOS85]. In other words, each vertex v_i must have seen its value set through a saturated edge e_{ij} or e_{ji}, i.e. an edge whose value is *equal to* the difference between the position of the target and source vertex.

$$v_j - v_i == L_{ji} \text{ or } v_i - v_j == L_{ij}$$

Fig.3.10 *A minimum solution of the cost function Λ implies that each vertex has at least one saturated edge with another vertex.*

Together with the sign of the vertex cost a_i, the possibilities for decreasing the cost function Λ are:

- a_i is *negative* and the edge has its *head* incident at i $\Rightarrow$ increase v_i until a fanout edge e_{ij} becomes saturated
- a_i is *positive* and the edge has its *tail* incident at i $\Rightarrow$ decrease v_i until a fanin edge e_{ji} becomes saturated

Although it is possible to apply a direct algorithm which searches in the list for *individual* vertices to be moved, a more efficient approach is known which moves vertex *groups* with the aid of a tree structure to mark the saturated edges [CAT90].

Such a tree can be constructed as follows: take the critical vertex v_{so} as the root[1] of the tree T, scan the saturated fanout edges and insert the vertices which are not yet in the tree T. Given N vertices in the graph Γ, this tree is a representation of N-1 equations which are all on the simplex.

Two basic strategies can be applied to construct the tree T: *breadth-first* or *depth-first*. The breadth-first construction completes a tree level before treating an underlying tree level. It provides wide trees with minimum depth.

[1] The index of the critical vertex refers to the source vertex, which is always on the critical path

The depth-first construction creates the underlying tree levels before completing the current level. It yields deep but thin tree structures.

Maier used breadth-first tree construction to detect and locate positive cycles in a graph during the tree construction [MAI79]. Experiments show that for layout, breadth-first initial trees converge faster to an optimum solution than depth-first trees. However, a later section will present that depth-first trees are able to detect and reduce zero cycles in a convenient and efficient way. Therefore, it may be advantageous to apply a depth-first approach anyway.

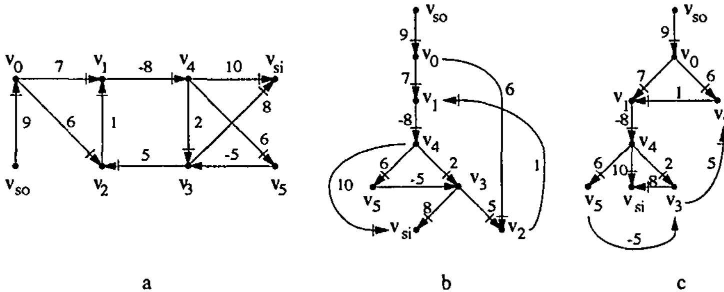


Fig.3.11 Given a graph Γ (a), the initial tree T of saturated edges (= marked arrows) can be constructed using a depth-first strategy (b), or a breadth-first strategy (c).

The tree structure indicates which vertex groups are related to each other through saturated edges. This means that if a value is added to all members of a subtree, still all vertices are related through saturated edges.

```
Slack(Γ)
{
    T = CreateTree(Γ);
    while(vj = getNextPivot(T)) {
        markSubtree(vj);
        ejk = findEdgeToSaturate(vj);
        δ = Ljk - ejk;
        updateSubtreePosition(vj,δ);
        cost = vj.cost;
        updateAncestorCost(vj,-cost);
        moveSubtree(vj,vk);
        updateAncestorCost(vk,cost);
    }
    deleteTree(T);
}
```

Alg.3.4 Global structure of the slack distribution algorithm.

This property leads to the introduction of the *reduced vertex cost*, which is defined as the sum of all vertex costs a_i in a subtree. Now, candidates for a subtree, called a *pivot*, are those vertices for which the following holds:

- the reduced cost f_i is *negative* and the edge has its *head* incident at i
  ⇒ increase all vertices in the subtree v_i until a fanout edge e_{ij} becomes saturated
- the reduced cost f_i is *positive* and the edge has its *tail* incident at i
  ⇒ decrease all vertices in the subtree v_i until a fanin edge e_{ji} becomes saturated

Once a subtree and the new edge are found, all vertex values in the subtree are modified until the new edge is saturated. The edge which is related to the subtree has become unsaturated and is removed from the tree. Then, the tree is reconstructed using the new saturated edge.

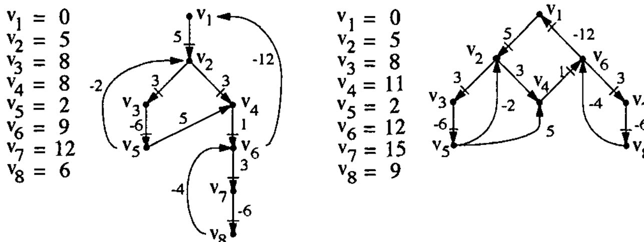


Fig.3.12 Illustration of tree pivoting: the edge e_{24} is unsaturated and the edge e_{16} is saturated (source [CAT90]).

The procedure of finding a pivot, followed by a desaturation of the edge is repeated until no pivots are left. At this moment, the minimum cost solution is found.

Centering

Once the slack distribution algorithm has finished its work, the total cost of the graph is minimum. However, inside the final tree there may exist subtrees which have a cost of zero. This means that such a subtree may be moved without increasing the graph cost.

The margin inside which the subtree can be moved without violating the constraints will be called the *free space*. The centering algorithm places each subtree in the middle of its available free space. At the geometrical level, this feature is translated into an optimum occupation of the local area. This solution leads to "nice-looking" layouts, because the free space is uniformly distributed.

The implementation of the centering algorithm is straightforward using the vertex fusion approach. Starting at the root of the tree used during the slack distribution, all child vertices with a cost different from zero are merged with the parent vertex. The resulting graph is used to place the vertices between their minimum and maximum position.

```
Center(Γ)
{
    foreach(vertex vi in Γ)
        if(vi.reduced_cost == 0)
            schedule(vi);

    while(vj = getNextScheduledVertex()) {
        foreach(vk = sonOfVertex(vj))
            if(vk.reduced_cost != 0)
                Fusion(Γ,vj,vk,Ljk);
    }

    Min(Γ);
    Max(Γ);
    foreach(vertex vi in Γ) {
        vi.pos = (vi.min+vi.max)/2;
        Split(Γ,vi);
    }
}
```

Alg.3.5 The centering of zero cost subtrees is obtained by merging all the non-zero cost vertices with their parent, followed by an iterative average positioning algorithm.

Note that this algorithm collapses the total graph into a few vertices. The cost of this operation, in terms of CPU time, is quite large: up to 50% of the total compaction effort!

Zero cycle reduction

The reduction of the zero cycles before the application of the expensive Simplex algorithm is a good idea; the graph to be solved may become much smaller. However, the detection of the zero cycles as presented in [CAT90] is complex. Based on [DAN66], it uses the fact that a feasible graph $\Gamma_k(V,E)$ consisting of k vertices and their edges can be constructed out of the vertex v_k and the feasible graph $\Gamma_{k-1}(V,E)$ consisting of $k-1$ vertices and their edges. Now, all strictly positive and zero cycles will pass through v_k if Γ_{k-1} contains no strictly positive or zero cycles.

Although an improvement of a factor 4 is reported[1], the method can be criticized on the following points:

- the critical vertices are not reduced before the application of the algorithm, so even zero cycles on the critical path are detected and reduced
- the cycle check is performed for each new vertex on a graph which becomes larger at every iteration
- there is no selection criterion proposed for the vertices to be added at each new iteration

---

[1] Run times are compared under equal conditions, i.e. before optimization of the memory management

- the choice of the initial graph is arbitrary and not guided by the property to be discovered (the cycles)
- the algorithm has worst case run time if no zero cycles exist

An alternative and faster method to detect and remove the zero cycles can be obtained if a Maier-like algorithm [MAI79] is integrated into the construction of the initial Simplex tree. The rest of this paragraph will be dedicated to the presentation of this algorithm.

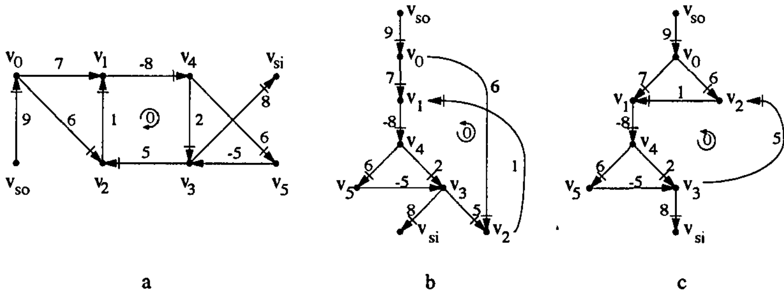


Fig.3.13 A zero cycle will consist of vertices on the same branch when a saturated tree is constructed out of a graph (a) with a depth-first method (b). The breadth-first approach does not allow this cycle location (c).

The principle is based on the fact that all vertices on a zero cycle are connected with each other through saturated edges. As the tree T is constructed following the saturated edges and using a depth-first strategy, a zero cycle is detected if the head of a saturated fanout edge e_{ik} in v_i is incident to a vertex v_k which is an ancestor of v_i in the tree T.

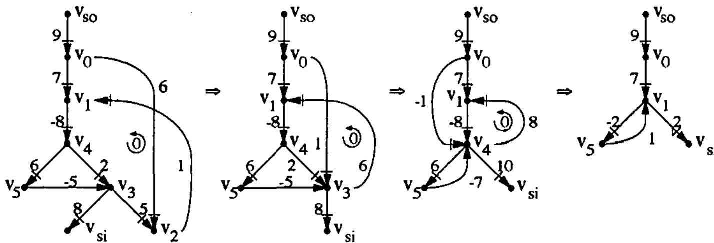


Fig.3.14 If a zero cycle is detected, the vertex is merged with its ancestor. The final tree can be used as the initial tree for the slack distribution.

If this case occurs, v_i is a member of a zero cycle, like its ancestor. Now, v_i is merged with its ancestor, and the tree construction is continued. When the tree is totally constructed, it is guaranteed to be free of zero cycles, and at the same time, an initial tree is constructed for the slack distribution algorithm.

The same principle can be applied to avoid critical path determination. As

both the source and the sink are already known to be critical, they are merged directly, which transforms the critical path into a zero cycle. Applying the zero cycle reduction algorithm now, means that the critical path is automatically reduced.

```
CreateTree(Γ,T,vi)
{
    change = True;
    vi.busy = True;
    while(change) {
        change = False;
        continue = True;
        while(continue and foreach(saturated fanout edge eik))
            if(not member(T,vk)) {
                putInTree(T,vk);
                if(CreateTree(Γ,T,vk) == False) {
                    change = True;
                    continue = False;
                    removeFromTree(T,vk);
                    Fusion(Γ,vk,vi);
                }
            }
            else if(vk.busy == True)
                continue = False;
    }
    vi.busy = False;
    return(continue);
}
```

Alg.3.6 Zero cycles are detected and removed during the construction of the initial tree of saturated edges.

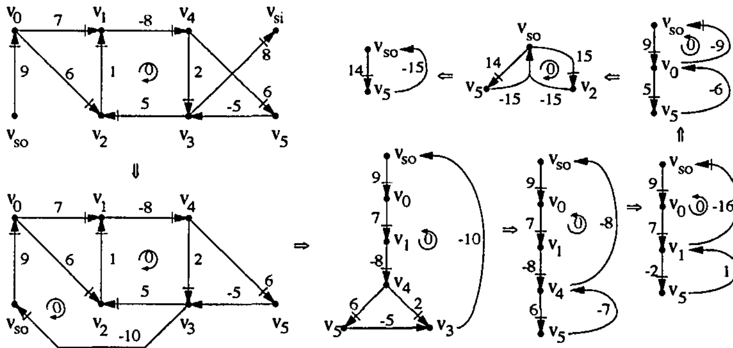


Fig.3.15 If the source and sink vertex are merged before application of the zero cycle reduction algorithm, all critical vertices are reduced automatically.

Although the algorithm avoids the determination of the critical path, it should be noted that it will be applied on an initially larger graph. On the other hand, zero cycles are reduced during the application of the algorithm,

which in turn, decreases the graph size. A comparison between this algorithm and the classical method, which first reduces the critical path explicitly, is presented later in this chapter.

Once the tree T has been created, the graph Γ is free of zero cycles and the minimization of the cost function Λ may begin. Its performance will be optimum as no zero cycles remain.

Costs

Apart from the constraint *values* (often called *weights*), an important role may be given to the constraint *costs*. They are of interest for the slack distribution and the centering. Costs may reflect part of the design intent. They allow to obtain a detailed control over the final result, which is the global optimum solution.

The primary application of costs is to define a cost on the constraint which defines the minimum width or length of an element. A polysilicon layer has a higher resistance than a metal layer, which may be translated into a layer dependent cost. As the compactor will minimize the global cost, the constraints with a high cost will be more contracted than the constraints with a low cost.

In fact, this helps to solve ambiguous cases. In the example below, a metal and a polysilicon wire are aligned at the left side of the box, and another set at the right side. Then, two wires of the opposite type are interconnected with a contact. Due to the higher cost on the length of the polysilicon wire, its length is minimized at the cost of an increased metal wire length. Coupling costs to resistivity helps to assure that layers with poor resistive properties will by minimized prior to layers with better conduction.

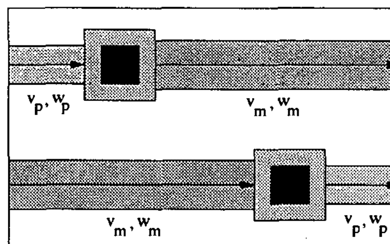


Fig.3.16 Costs on constraints are used to reflect the desire to contract (or expand) a constraint more than another. Here, $w_p > w_m$.

Costs may also be chosen negative. In this case, the effect is opposite; the vertices are placed as far as possible away from each other, however, without increasing the longest path. This situation may be useful to place sensitive circuitry, without loosing surface. Or, someone may indicate that the width of a power line should be as large as possible within the available space.

Contrary to constraint values which may be redundant, all constraint costs need to be taken into account. As an example, consider the vertices v_1 and v_2 in a graph, among which two constraints exist, c_1 with value 3 and cost 4, and c_2 with value 5 and cost 0. The dominant constraint reduction will insert only one edge e_{12} with value 5. However, the cost of constraint c_1 should be considered: it expresses the desire to contract v_1 and v_2.

$$v_1 \xrightarrow{3,4} v_2 \;+\; v_1 \xrightarrow{5,0} v_2 \;=\; v_1 \xrightarrow{5,4} v_2$$

Fig.3.17 Even though only dominant constraint values need to be considered for graph resolution, all constraint costs have to be taken into account.

An analog motivation may be held for vertex fusion; again all constraint costs need to be taken into account even if they are masked by dominant ones. This effect can be obtained if constraint costs are accumulated in the vertices, thus providing the *vertex cost*. Now, the cost of the vertex which is removed, is added to the master vertex.

The cost of zero valued constraints can be used to control the slack distribution. In fact, in this case, only the cost will have an effect. Inserting such a constraint between the source and the vertex allows to push the vertex from the source or to pull the vertex to the source. Note that these constraints can never modify the longest path.

3.4 Hierarchy

So far, the effort was concentrated on the optimizations of the spacing algorithms, but other possibilities exist to reduces the time required to solve a problem described by a set of constraints.

A spectacular improvement is obtained by the introduction of hierarchy. The key issue of hierarchy is to exploit regularity; references instead of copies are used in order to avoid to recompute results which are in fact already known. In VLSI tools, it has become an essential part in the attempt to control the amount of data needed for a design.

Hierarchy corresponds well to the natural approach of *divide and conquer*; large problems are managed by dividing them into smaller problems until they become easy to solve. Then, they are put together to form the final solution.

An example

It is important to notice that the data structure, onto which the compaction algorithm will be applied, will correspond to a layout. The number of geometries in such a layout may be extremely large. If we take for example a static random access memory, each cell may be composed of 6 transistors, which in turn may consist of about ten rectangular geometrical primitives.

Even when discarding the interconnections and surrounding layout, a memory matrix with 1000 bits would correspond to approximately 1'000x60≈60'000 rectangular primitives, four times as much vertices, plus the constraints in between these vertices!

Then, we know that many compaction algorithms have a complexity of NM, where N equals the number of vertices, and M the number of edges. According to the examples given in [CAT90], practical cases contain 5 to 10 times as much edges as vertices, whereas the spacing time on a 0.9 MIPS machine corresponds to 10'000 vertex edges per second. Even on an ideal 50 MIPS machine with sufficient memory and without page faults, the memory matrix example would run for months!

Fortunately, we may profit from hierarchy to handle this complexity. This paragraph presents how this is implemented in the compactor. The key point is formed by the *substitute graph*, which is automatically extracted from the cell.

Principle

In a hierarchical layout description environment, the data structure can be defined in terms of *cells*, and primitives such as *rectangles*, *instances* and *constraints*. An instance represents an occurrence of a cell, where only a subset of the cell data is duplicated. Those rectangles, which are copied into a cell instance, are called *ports*. Ports serve to communicate between a cell and an underlying cell. Each cell is composed of primitives only, and cells which do not contain any instances are called *leaf cells*.

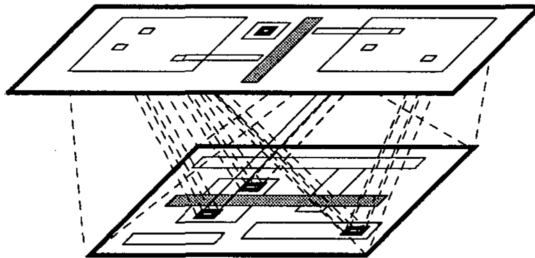


Fig.3.18 Cells are composed of rectangles and instances. Ports are rectangles which are visible at the caller level; they allow to communicate between the different levels.

The *root cell* is defined as the cell which is not called by any other cell; it contains the whole layout. The *depth* of a cell corresponds to the largest number of cells which are to be passed to arrive at the root cell. As a consequence, the root cell has depth zero.

Instances are not inevitably rigid hard-copies of the original cell ports. The graph representation of a layout allows for four types of instances[1]:

- the *fixed* instance, which preserves the exact form and port positions in the cell
- the *semi-fixed* instance, which preserves the exact size of the cell, but ports and related geometries may be moved within their free margin
- the *flexible* instance, which maintains the topology of the cell, but all ports and internal primitives may be moved as a result of external constraints
- the *flattened* instance copies all cell elements and constraints into the current level, so that no additional hierarchical level is introduced

It is the last type which makes the compaction system presented here a very powerful one. Constraints which are only known at higher description levels, may modify the positioning of lower-level geometries.

Extraction

The role of a cell extraction is to provide a *model* for the cell. This model should preserve all of the internal cell properties, transposed to the ports. The type of the cell model depends on the instantiation type.

The *fixed* instance requires that the form and position of all the cell ports is exactly preserved. This means that the cell can be fully compacted, including the minimum cost and center algorithm. Then, all the port vertices are merged with the source vertex, using the port positions as the offset value. Note that the reduction of the cell complexity is maximal, only two vertices (one for each direction) are used to model the total cell.

A slightly more flexible solution is offered by the *partial* extraction. Again the cell is fully compacted, but the free margin of the port edges is preserved. This free margin corresponds to the distance with which the port edge may be moved without increasing the total cost in the original cell graph. Practically, the graph that has been obtained after the fusion of the vertex clusters in the centering algorithm is used to extract this type of model. Cell ports that are in the same vertex cluster are related to each other through fixed-bound constraints, whereas the positioning relative to other clusters is obtained through lower-bound constraints. So, the partial extraction preserves the minimum cost of each individual cell, but it exploits the free margin of the ports.

---

[1] Note that an instance is composed of edges and constraints. In this sense it is not treated as a special case by the compaction algortithms

Finally, the *flexible* extraction allows for the modification of the cell size and port positions. Now, even internal cell primitives may move, but only through the bias of external constraints which have been placed on the ports.

The flexible extraction corresponds to a so-called *substitute graph* [STA84]. Each node in this graph represents a port vertex, and the constraints in this graph are the longest paths among these vertices. Only direct longest paths are considered, i.e. paths which do not contain any port vertices. As a result, the substitute graph models the constraints imposed by the internal components of a cell on its ports.

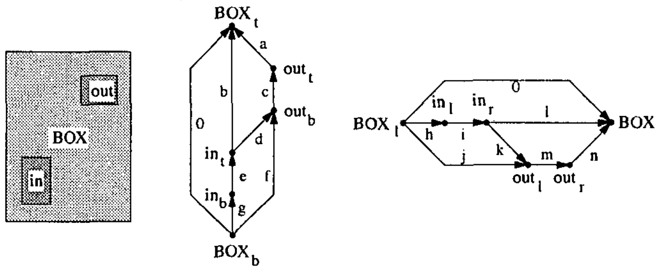


Fig.3.19 Example of a cell with three ports (BOX, in and out) and its corresponding substitute graph in x- and y-direction.

Unfortunately, the propagation of the vertex cost through multiple levels of hierarchy is troublesome. In fact, the only compatible solution here would be to represent all elements in the cell abstraction, which would cancel the effect of hierarchy. A correct cost minimization, while not representing all cell elements in the extraction, would mean that the vertex cost in the extraction graph would be a function of the distance of the vertex to the bounding box (source and sink). In this case the cost function would become non-linear, so that the linear programming theory, which served as a basis for the compaction techniques here, would not apply.

Therefore, only the vertex cost of the individual ports is included in the cell extraction. This may sometimes lead to some unexpected behavior, because the context of underlying cells is lost. This can either be solved by exporting more ports (thus providing a more detailed context) or by choosing the flattened instantiation type. In the latter case, the hierarchy is lost, but the distribution is optimal.

Compaction sequence

Up to now, we have seen that a substitute graph is used as a cell instantiation. Repeating this mechanism up through the hierarchy, the minimum locations of the elements at the top level are found. However, the final positions of the underlying cells still need to be computed!

As we know that the cell instances at the top level are on their final positions, this information is used to *freeze* the port positions of the underlying cell. In fact, this means that the cell ports have become critical, so they are merged into a single node. Then, the cell is compacted with the previously presented algorithms.

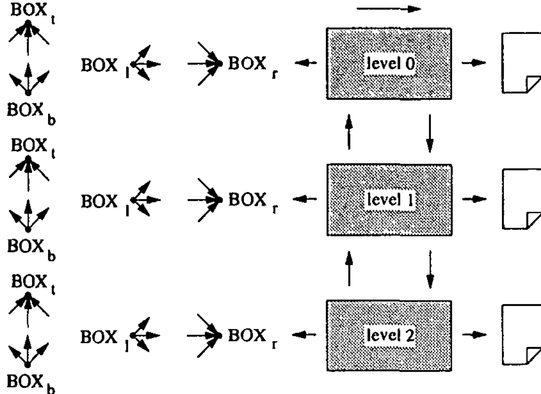


Fig.3.20 Substitute graphs are used during the bottom-up compaction to obtain the minimum locations of the top-level description. The final positioning of the underlying cells is obtained in the top-down compaction phase.

Note that port costs do not play a role in the top-down phase of the hierarchical compaction. All ports are merged into a single (critical) node, which cancels their effect.

Protection frames

In order to be able to treat a cell level without explicit interaction with underlying levels, the use of protection frames is common [BUR86]. In *PYTHON* [BAL82] and *SPARCS* [BUR86], per-layer sets of bounding rectangular polygons form the basis of these frames. They influence the spacing efficiency against the computational efficiency: the closer a frame approximates the bounding box (= smallest rectangle surrounding all geometries in a cell), the smaller the amount of information necessary to process to correctly space that frame. This means less computer time for spacing the frame, but it will also mean the greatest area loss. On the other hand, merging the geometry internal to the cell to form the protection frames will provide the greatest area efficiency, since all of the unused area within the cell may be used at the next level of hierarchy.

The compactor presented here does not use per-layer protection frames, but a single one. Furthermore, this frame is not automatically placed by the system, instead, it is the user who explicitly places the frame. Beside the

computational gain, the user defined positioning of the frame allows for an optimum area efficiency. Now, it becomes possible to share resources, such as power lines or bulk contacts, among cells.

This way of using a user defined protection frame is similar to the abutment box used in the Compass system. It has the side effect that each cell can be designed design rule error-free.

Instance reduction

In general, an hierarchical compactor implementation will copy the whole extraction for every instantiation of a cell. However, another possibility is to create the instance members only when they are needed. This so called *instance reduction* allows for a further reduction of the graph complexity because in practice, not all instance members will be referred to.

Unfortunately, experiences show that the effect is not as spectacular as expected. Every instantiation requires a new extraction to be computed, and reducing the number of instance ports increases the number of non-critical vertices in lower level graphs.

Another side effect is noticed when a single hierarchical level is drawn on the screen. When applying instance reduction, only those cell ports which are referred to are present in the caller level, which gives rise to a mutilated representation of the underlying cells.

3.5 Incremental compaction

Based on reports on the performance of compaction algorithms we have focused on an incremental implementation of the whole TILT system for a long time. The idea here was to re-compute only the modifications, and not to re-run the generator for every try.

As a rule of thumb, incremental implementations will run slower than "batch" implementations. This is mainly due to the more complicated data structure which is needed to be able to undo all what has been done. However, benefit of this behavior may be expected during the development phase, which is particularly interactive. A non-incremental version could be used in a batch type environment. The incrementality may be applied at different stages of the compaction process:

- *constraint building*
- *one level constraint resolution*
- *hierarchical constraint resolution*

All of these items may be explored independently. The pros and cons are discussed in the following paragraphs.

Constraint building

The nature of a stream with vertices and constraints is quite well suited for incremental implementations. The reason is that there is no particular sequence within the group of vertices and constraints; both the vertices and the constraints may be constructed in any order.

The cost to pay for an incremental constraint building resides in the fact that all constraints need to be preserved somewhere in the data structure. Each constraint or vertex may be deleted from the graph at any moment. For example, a redundant constraint may become important if the dominating constraint is deleted. If there is sufficient memory available, or if the compaction algorithm needs to preserve the constraints anyway (for instance to be able to split vertices after a temporary fusion), this should not be considered to be a drawback.

One-level constraint resolution

From the algorithmic point of view, the data interdependency in a constraint graph is very large. Removing a constraint will generally disturb the equilibrium which has been obtained after the slack distribution and centering, or it may even modify the critical path. The same arguments hold for the addition of new constraints.

Only the insertion or deletion of redundant zero cost constraints between already existing vertices will not modify the result. These (rare) cases can be detected during the graph construction, they have no impact on the algorithmic level.

The conclusion must be that incremental behavior on the algorithmic level may not provide any performance gain, unless new special algorithms are applied. The data interdependency is too large to expect any benefit here in the near future.

Hierarchical constraint resolution

The largest impact of incremental spacing may be obtained from the hierarchical graph resolution. A cell modification will only influence the cell itself, and the cells which depend on it. All other cells remain unchanged, and do not need to be re-computed.

Our experience has shown that a pure incremental implementation of the hierarchy leads to a too complicated data structure. If each operation in each object of a description can be undone, this implies a full set of influence and dependency relations to be maintained. This overhead loads the system considerably, so that the benefit of incremental updates is lost.

Instead, it is suggested to allow the incremental behavior only at the top level. Now, all the underlying objects remain the same, and no influence relations are required to process the modifications.

Still, an incremental approach requires the top-down slack distribution to be re-computed after every modification. In the case of large hierarchical structures (such as memories), this may take a considerable time. Here, the use of a data-base becomes a necessity.

3.6 Overconstraints

A constraint driven approach works fine as long as the constraints are compatible among each other. The difficulties arise when the graph becomes unfeasible because of contradictory constraints. For instance, an element *a* is placed on the left of element *b*, but elsewhere in the description, a construction declares that *b* is on the left side of *a*.

A solution is to avoid that these cases may occur, for example by extracting them from some initial topology (e.g. sticks). If an initially widely spaced layout is used as a starting point of the compaction algorithm, the constraints will in principle correspond to a feasible solution.

However, in a language based system, overconstraints are more difficult to avoid, and it is important to report them in a comprehensive way. The problem is that the constraints which cause a loop, and the vertices on this loop, are far away from the high-level description the user has given.

If the compactor has knowledge of the high-level instruction that created the constraint, it becomes possible to localize the source of trouble rapidly. For instance, an instruction identifier could be passed to the compactor, so that it knows who created what. In any case, the loop itself needs to be reported as it provides more detailed information on the cause of the overconstraints.

Another issue is to realize that the vertices should be named if we want the loop report to be clear. This is why the compactor used in TILT obliges all geometrical primitives to be named.

It is recommended to report all loops in the graph if overconstraints are detected. The basic idea behind this algorithm is to use the fact that the longest path algorithm does not converge if a positive loop exists. The vertices on the loop will remain scheduled when the iteration count exceeds the number of vertices in the graph. The loop itself is found by following the vertex ancestors.

All positive loops can be reported if a loop is transformed into a zero cycle after it has been reported. Applying the same algorithm again will report the next positive loop (if any) in the graph.

```
ReportLoops(Γ,T,vi)
{
    check = True;
    while(check) {
        check = False, change = True, count = 0;

        foreach(vertex vi in Γ)
            vi.min = 0, schedule(vi);

        while(count < Γ.vertexCount and change) {
            increment count, change = False;
            while(vj = getNextVertex())
                foreach(fanout edge ejk)
                    if(vk.min < vj.min+Ljk) {
                        change = True;          schedule(vk);
                        vk.min = vj.min+Ljk;    vk.anc = vj;
                    }
    }

    if(count == Γ.vertexCount) {
        vj = getNextVertex();
        clearSchedule();
        length = 0;
        while(notYetScheduled(vj)) {
            schedule(vj);              vk = vj.anc;
            length = length + Lkj;     vj = vk;
        }
        printLoop();   ejk = ejk - length;   check = True;
    }
}
```

Alg.3.7 Once a positive loop is detected, it is reported, and transformed into a zero cycle to be able to report the remaining positive loops in the graph.

We may argue about the way to proceed when a loop is detected. In general, the execution is aborted, and the loops reported. However, it is also possible to remove constraints one by one, until the graph becomes feasible, but resulting in an illegal layout [KIN84]. Although the compactor may continue now, the result may be somewhat arbitrary.

3.7 Interfacing

The compaction problem has been defined as a set of constraints and vertices which cost is to be minimized. In the case of layout, some particular properties may be explored to provide a higher abstraction level, and an efficient data representation.

It is important to realize that vertices in a constraint graph do not logically correspond to a single polygon or rectangle side in the layout. Instead, a vertex models an object, which may correspond to a much more complicated data structure.

As an example, the SPARCS [BUR86] program caries out a layout data abstraction in terms of two object classes: *immutable* and *mutable* objects. An object may be as simple as a point or an edge, or may be as complicated as a set of Manhattan polygons.

Immutable objects are used to represent contacts, transistors, standard cells etc., whereas mutable objects are used to describe elements such as wires, buses and parameterized cells. Mutable objects may both change form and translate; immutable objects may only translate.

The main benefit of this abstraction resides in the huge data reduction which can be obtained in the constraint graph. In fact, any immutable object of arbitrary complexity can be represented by a single node in the graph because all its components stay at fixed positions relative to the object's origin. However, two important drawbacks should be noticed: (1) the object type must be chosen beforehand, and (2) a set of mutable objects which become immutable due to the imposed constraints is not detected.

In the context of vertex fusion, the advantages of a large data reduction are obtained using only one object class: the *cluster*. Each cluster object contains a set of zero or more geometry sides. In certain cases, a cluster is transformed from one object into another resulting in a larger cluster, and an empty cluster. Only non-empty clusters are considered by the compaction algorithms.

Layout properties

Usually, a silicon foundry specifies a set of rules to be satisfied in order to obtain the desired properties (the *design rules*). These rules can be subdivided into two classes: *topological* rules, and *electrical* rules.

Topological rules concern the form of the geometrical primitives, and the relations among them. Many technologies allow only polygons to be used; circles or curves are excluded. Relations between the geometries concern the distances in between them, their width etc.

Electrical rules specify the conditions which must be fulfilled once a circuit is connected to a power supply. Such a rule may concern the maximum current per layer width, its resistance, capacitance, or the break through voltage of a junction.

Geometrical primitives

In the context of compaction, it is crucial that all geometries are identified, especially for the definition of the constraints amongst them. Only if each vertex in the constraint graph has a unique name, overconstraint reports may become clear. For this reason, polygons as a basic structure have been

rejected, in favor of the *box*, because each side can be addressed with the attributes *left*, *right*, *bottom* and *top*.

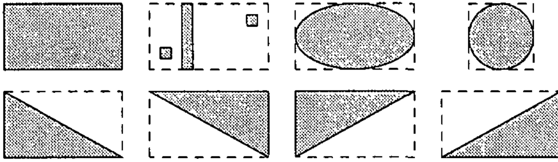


Fig.3.21 Using a virtual box as a primitive, rectangles, cells, ellipsoids and triangles can all be identified with the attributes left, right, bottom and top.

Having a closer look at the box structure, it becomes clear that it is not only suited for modeling rectangles. Using the box as an outline, complete cells, ellipsoids and triangles may be described in the same manner[1].

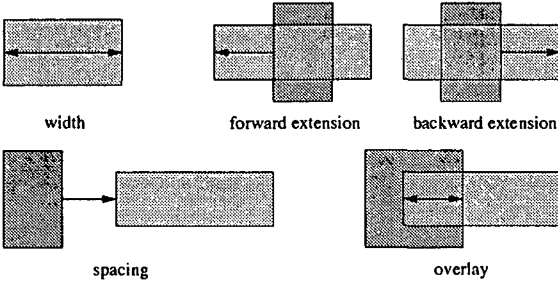


Fig.3.22 Width, spacing, extension and overlay are the 4 basic one dimensional constraints to compose structures using boxes.

The use of a rectangular primitive as the basic structure reduces the necessary name space for the vertices. Instead of naming each individual vertex, 4 vertices will share the same root. Another advantage appears at the constraint side: the use of 4 distinct constraint classes avoids the specification of the rectangle side. These classes are:

- *width*, a left-to-right or a bottom-to-top relation on the same box
- *spacing*, a right-to-left, or a top-to-bottom relation among two boxes
- *extension*, a left-to-left, a right-to-right, a bottom-to-bottom or a top-to-top relation among two boxes
- *overlay*, a left-to-right or a bottom-to-top relation among two boxes

The box

In a previous section, it has been shown that the compactor uses two

[1] Note that the compactor "sees" no difference between a box of a rectangle, ellipsoid or triangle

particular vertices: the *source* and the *sink*. These two vertices are particular in the sense that all other vertices are in between them. In this sense, they play the role of a *box* that contains all the cell primitives.

Another set of particular vertices is formed by the rectangular protection frame. This frame is positioned by the user and may play the role of a border to be respected by surrounding cells.

Both of these boxes are created automatically by the compactor when a new cell is created. They may be addressed by the user through the names *BOX[0]* and *BOX[110]*, where the latter corresponds to the protection frame.

In some cases, it may be useful to insert a constraint between a geometry and the source or the sink, for instance to align an element on the top of the cell. However, the graphs should remain consistent, i.e.: the source must be a vertex without fanin edges, and the sink a vertex without fanout edges. This property must be verified by the compactor during the insertion of the constraints.

As an example, consider the case where a spacing is specified between the right side of the box, and a rectangle. If such a constraint would be allowed, the sink would become a fanout edge, and the box would not be a box anymore: one vertex would be outside the box.

3.9 Performance

The comparison of layout compaction algorithms is usually made with benchmarks. These comparisons are based on the compaction of an initially created layout, and they often focus on surface gain and run time.

A procedural approach is not very well suited for such a kind of comparison. First of all, the input is a layout description, and not a topology. This means that there is no initial layout which may be taken as a reference. And second, the surface criterion is not very useful because in principle, a hand crafted layout may be described.

Instead, it is more interesting to present the efficiency of the different algorithms presented in this chapter when applied to layout[1]. In fact, very few results are published on the mean complexity of a constraint graph when applied to layout compaction.

A warning should be made concerning the interpretation of the results. Each layout is a special case, so it is difficult to make general conclusions. Nevertheless, the figures give a good idea of the possibilities of the algorithms.

[1] see Annex A for a technical data sheet of the compactor implementation

Computational efficiency

This chapter focuses on different graph complexity reduction techniques. The goal of these reductions was to prepare for smaller graphs to be processed by the time expensive slack distribution algorithm. This paragraph presents some of the figures that can be measured on these algorithms. They have been applied to a variety of circuits, trying to cover the most common cases. The following effects have been examined:

- polarization constraints
- redundant constraints
- fusion of fixed constrained vertices
- critical path reduction
- zero cycle reduction

The polarization constraints are needed to make the graph *polar*, i.e. to assure that all vertices are between the *source* and the *sink* vertex. Some algorithms become "cleaner" when the graph is polar. Especially the construction of the simplex tree or the zero cycle reduction tree would become less regular to implement.

As a measure, we define the polarization factor of a hierarchical constraint graph with N levels, as the total number of polarization constraints compared to the total number of original constraints:

$$\text{polarization factor} = \frac{\sum_i^N \#\text{polarization constr}_i}{\sum_i^N \#\text{original constr}_i}$$

It gives an indication how large the additional complexity is to obtain regular algorithms.

The effect of the redundant constraint filtering should be measured in another way. Now, the graph reduction of each cell is weighted with respect to the vertex-constraint product. This proportional contribution reduces the influence of small cells in the final result.

$$\text{redundancy factor} = \frac{\sum_k^N \#\text{reduced_vertices}_k \#\text{reduced_constr}_k}{\sum_i^N \#\text{total_vertices}_i \#\text{total_constr}_i}$$

A similar formula is applied to measure the impact of the critical path reduction and the zero cycle elimination algorithm. It is interesting to see that the fusion of critical vertices compared to the fusion of zero cycles provide almost identical results if fixed constraints imply vertex fusion. This means

that a description defines very few (undesired) zero cycles. However, only the zero cycle reduction technique yields the same (minimum) vertex-constraint product reduction, independent on the fixed constraint implementation. Note furthermore that the zero cycle reduction step removes *all* zero cycles, a property which possibly allows for optimizations in the slack distribution algorithm.

average vertex-constraint products for different steps in the compaction process[1]								
cell type	**polarization**		**redundancy**		**critical path**		**zero cycles**	
3 input and gate	0.05	0.04	0.16	0.51	0.09	0.21	0.07	0.05
2-to-4 line decoder	0.02	0.02	0.15	0.47	0.08	0.17	0.06	0.05
ota	0.12	0.08	0.25	0.94	0.19	0.49	0.18	0.13
120 element bipolar-array	0.09	0.06	0.07	0.95	0.04	0.02	0.04	0.02
32x8bit static RAM	0.09	0.01	0.06	0.96	0.04	0.05	0.04	0.02

The table above shows that the reduction factors of the critical path reduction and the zero cycles elimination are very close. This means that, in an average layout description, very few zero cycles are created as a combination of user constraints. However, the zero cycle reduction algorithm is more efficient if fixed constraints are implemented with two parallel lower-bound constraints. As it eliminates all zero cycles, the graph becomes minimum in both cases.

graph complexity of the different examples				
cell type	**#rectangles**	**#constr fix**	**#constr par**	**fix / par**
3 input and gate	195	4310	5451	1.26
2-to-4 line decoder	485	22299	25902	1.16
ota	509	5188	7547	1.46
120 element bipolar-array	995	6934	10518	1.52
32x8bit static RAM	4213	34781	271432	7.80

average total compaction CPU times (in s)[2] *with different graph reduction techniques*[3]								
cell type	**none**		**critical path**		**zero cycles**		**both**	
3 input and gate	1.2	2.0	1.2	1.9	1.2	1.8	1.0	1.8
2-to-4 line decoder	6.5	12.4	7.4	12.1	8.0	13.7	7.4	14.2
ota	3.3	6.5	3.3	6.8	3.6	5.3	3.4	5.4
120 element bipolar-array	13.8	22.4	15.2	44.3	15.0	33.3	15.1	45.5
32x8bit static RAM[4]	42.0	684.8	50.9	468.8	53.7	542.4	55.7	490.9

However, the table of the computation times does not really reflect the effects of the critical path reduction or the zero cycle filtering. Taking the inaccuracy of the CPU measurements into account, the different techniques provide similar over-all performance.

[1] the first values in a column correspond to those obtained with fixed constraint fusion, the second without
[2] including graph construction, resolution and output generation on a SUN4/50 IPX with 32Mb of central memory
[3] the first values in a column correspond to those obtained with fixed constraint fusion, the second without
[4] the large run times for the implementation without fixed constraint fusion are due to excessive page fualts

Still, the advantage of a merged fixed bound constraint implementation over a parallel one is evident. The compaction algorithms run about twice as fast, and are more economic with memory.

Another viewpoint to the compaction is provided if a partitioning is made as a function of the activity. It appears that about 50% of the total compaction run time is effectively spent on the resolution of the graph, in particular with the centering. Near to 25% of the time is passed in the vertex fusion procedure, and the time to extract the substitute graphs is particularly low. Even if the vertex fusion could be made ideal (= zero time), the performance would only be slightly improved.

partitioning of the total CPU time in the different parts of the compaction process[1]								
cell type	construction		resolution		centering		fusion	
3 input and gate	38%	43%	43%	45%	31%	38%	16%	19%
2-to-4 line decoder	32%	33%	57%	58%	45%	48%	25%	29%
ota	30%	34%	43%	47%	36%	36%	26%	27%
120 element bipolar-array	4%	4%	58%	61%	49%	52%	29%	30%
32x8bit static RAM	16%	19%	51%	54%	41%	42%	25%	32%

We may conclude that an additional performance gain may be expected from the optimization of the centering algorithm (a factor of two?). More improvements may be obtained through the use of an improved data structure that allows for a more efficient implementation of cell instances and vertex fusion.

3.10 Active and exclusive constraints

The constraint types that have been presented so far, define all constraints between two vertices in the same plane (x or y). They allow to describe most of the practical cases; all of the applications that are presented in a later chapter have been implemented using these constraints types only.

Still, other types of constraints exist that may be useful for layout descriptions, such as:

- *active constraints*, to describe symmetry which is of special interest for analog circuitry to optimize electrical performance
- *exclusive constraints*, constraints which allow to define non-contiguous ranges of legal values
- *inter-dimensional constraints*, introducing a dependency between the x- and y-graph.

A distinction has to be made between symmetry of fixed objects, and symmetry of stretchable objects. The first case is easily obtained through fixed

[1] the first values are minimum values, the second are maximum values, measured over different scenarios

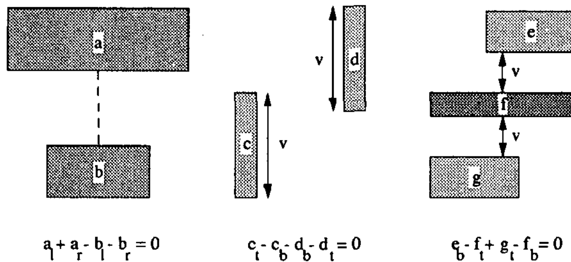
instantiation of cells. As the instances are rigid, the symmetry will be preserved automatically. The second case [OKU89] is somewhat more complicated, and can be generalized to a class of constraints between 4 vertices at a time.

$$v_j - v_i + v_l - v_k \leq s_{ijkl}$$

$$v_j - v_i + v_l - v_k = s_{ijkl}$$

Fig.3.23 The class of symmetry constraints defines a relation between 4 vertices.

In particular, the equality relation with s_{ijkl} equal to zero is useful for layout descriptions. They allow for different types of symmetries such as centering, equal widths or heights, and symmetrical spacing around an element.



$a_l + a_r - b_l - b_r = 0$ $\quad$ $c_t - c_b - d_b - d_t = 0$ $\quad$ $e_b - f_t + g_t - f_b = 0$

Fig.3.24 The class of symmetry constraints defines a relation between 4 vertices.

The implementation of the symmetry constraints is not straightforward because it requires the use of so-called *active* constraints. Their resolution is not possible with the algorithms presented in this chapter. These constraints are called active, because their values change during the compaction if either the source or the target vertex is moved. Note that convergence problems may occur while combining these kind of constraints in an environment with "normal" lower-, upper- and fixed-bound constraints.

The category of exclusive constraints would be useful for the description of something like [NEW87]: *"G1 may come as close to G2 as RR12 spacing units, or it may lie on directly on top of and be enclosed by G2, but neither of its edges may lie within RR12 spacing units of G2"*. A similar rule is involved with the merging of contacts or vias; they may be placed as far as the via-via spacing rules, *or* they may be collapsed. Unfortunately, at this time, there is no general solution for this problem. In particular, the difficulty is to proof the convergence of such a solution.

Furthermore, this feature could be expanded to the second dimension, which would allow to describe: *"G1 is either on the left of G2, or G1 is on top of G2"*. However, the placement of surrounding elements in an environment with exclusive constraints will become tedious, because the relative position of

neighbor elements cannot be known beforehand anymore. Probably, different descriptions for different cases yield a solution which is easier to manage.

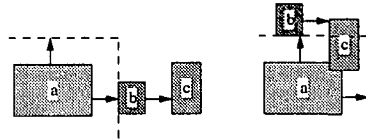


Fig.3.25 The placement of surrounding geometries becomes tedious if exclusive constraints would be allowed. In this example b is either on the right or on top of a. Element c will touch a if no constraint was foreseen.

Constraint types which allow to relate the x- and y-dimension would be useful to describe area constraints. But, again, problems may arise with respect to the transparency of the placement. Also, the interdependency between the x- and y-graphs has a serious impact on the computation time.

3.11 Conclusions

Algorithms

Hierarchical layout compaction based on rectangular primitives only is able to provide optimally spaced solutions in acceptable CPU time. To achieve this, it is essential to apply two graph reduction techniques: redundancy filtering and vertex fusion. The latter only, as a response to fixed-bound constraints, yields approximately twice as fast compaction times as the conventional parallel implementation with two lower bound constraints. Moreover, this method is less memory expensive.

On the contrary, the combination of these graph reduction techniques with an explicit reduction of the critical path, either with a conventional detection of critical vertices, or with a less common zero cycle reduction algorithm, provides no significant performance gain. Apparently, the cost to reduce the critical path, is similar to the extra time spent with the slack distribution.

The addition of costs on the constraint values is essential for the control of the slack distribution. Although a simple, non-optimal, solution is used for the propagation of costs through the hierarchy, the overall results are near to the ones which can be obtained with a fully flattened version.

The centering of costless vertex clusters is an essential extension to the slack distribution. Not only, it removes the ambiguity of the solution after cost minimization, but it also fully explores the available space. The uniform distribution may even have positive effects on the yield of the circuit. However, the current implementation is expensive: it consumes almost half of the CPU time. A solution which does not merge the vertex clusters explicitly may solve this problem.

Interface

Contrary to most other compactors, the basic primitives of the program are rectangles, and constraint between the rectangle's sides. This increased abstraction level does not only reduce the memory overhead, but it also allows for a clear overconstraint report. A further improvement could be obtained if complete objects (such as contacts or transistors) and high level parameterized topology functions (such as *place* and *align*) would be used as the lowest interface to the compaction algorithm. However, this would require a complete review of the compactor data structures.

Incrementality

The implementation of a pure incremental compactor is probably not advantageous, because the algorithms are merely sequential. Instead, this technique may be applied at the graph construction level, because the only sequence to be respected is that vertices should exist before constraints.

Still, the trade-off between the increased complexity of the data structure, and the performance gain has to be made. In any case, a non-incremental version should be preferred for non-interactive applications.

Limitations

The actual limits of the presented compactor are more in the domain of memory requirements than in the domain of computational effort. All geometries are named, but there is no specific name management integrated into the program. An optimization of the internal data structure, and the integration of the polarization constraints into the algorithms could be worthwhile here.

Especially, a performance gain is expected if the vertex fusion and splitting mechanism is optimized. The latter causes actually that all the vertex fusions have to be redone, which is not very efficient. I believe that the most efficient implementation can be obtained if the vertex fusion is only used as a response to fixed-bound constraints.

A further improvement may be expected if intermediate graphs may be preserved (e.g. in a data-base). Especially during the iterative development phase, this would allow for a significant performance gain.

Active constraints may be useful for the description of symmetry in the topology. Their implementation may be a valuable extension for the definition of sensitive parts of analog circuitry. On the contrary, exclusive and inter-dimensional constraints are less practical. They complicate the topological description, and require much additional computational effort.

3.12 References

[BAL82] M. W. Bales, "*Layout Rule Spacing of Symbolic Integrated Circuit Artwork*", ERL Memo No. UCB/ERL M82/72, University of California, Berkeley, May 1982.

[BUR86] J. Burns, A. R. Newton, "*SPARCS, A New Constraint-Based IC Symbolic Layout Spacer*", Proc. IEEE Custom Integrated Circuits Conf., pp.534-539, May 1986.

[CAT90] G. Cathebras, "*Contribution à la compilation structurelle des circuits intégrés CMOS*", Ph.D Thesis, University of Montpellier, France, February 1990.

[DAN66a] G. Dantzig, W. Blattner, M. Rao, "*Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem*", Proc. of the International Symposium on Graph Theory, pp.77-83, Rome, July 1966.

[DAN66b] G. Dantzig, W. Blattner, M. Rao, "*All shortest routes from a fixed origin in a graph*", Proc. of the International Symposium on Graph Theory, pp.85-90, Rome, July 1966.

[GON79] M. Gondrand, M. Minoux, "*Graphes et algorithmes*", Collection de la Direction des Etudes et Recherches d'Electricité de France, 1979.

[KIN84] C. Kingsley, "*A Hierarchical, Error-Tolerant Compactor*", Proc. 21st. Design Automation Conference, pp.126-132, June 1984.

[LAR71] R.P. Larsen, "*Computer-Aided Preliminary Layout Design of Customized MOS Arrays*", IEEE Transactions on Computers, Vol. C-20, No. 5, pp.65-112, May 1971.

[NEW87] A.R. Newton, "*Symbolic layout and procedural design*", Design Systems for VLSI Circuits, Logic Synthesis and Silicon Compilation, pp.65-112, Martinus Nijhoff Publishers, 1987.

[MAR88] D. Marple, M. Smulders, H. Hegen, "*An efficient compactor for 45° layout*", Proc. 25th DAC, pp.396-402, June 1988.

[OKU89] R. Okuda, T. Sato, H. Onodera, and K. Tamuru, "*An efficient algorithm for layout compaction problem with symmetry constraints*", Proc. ICCAD '89, pp.148-151, 1989.

[STA84] J.A. Starzyk, "*Decomposition approach to a VLSI symbolic layout with mixed constraints*", Proc. ISCAS '84, pp.457-460.

[YOS85] T. Yoshimura, "*A graph theoretical compaction algorithm*", Proc. ISCAS '85, pp.1455-1458.

4 The language al2

Je vis de bonne soupe, et non de beau langage.
Vaugelas n'apprend point à bien faire un potage.

Chrysale, acte II, "Les Femmes savantes", Molière

4.1 Introduction

A language is a basic expression form in many domains, such as every day speaking, communication with deaf people, photography, love, music and computer science. Usually a language is well adapted to the purpose it is used for: it offers basic building blocks for the corresponding application. Trying to describe something in a language which is not well suited for the given application can be extremely difficult, or even impossible. For example, someone could try to describe a piece of music for a piano by designing all the positions of the piano player's hands. Although this seems possible, a partition would become quite cumbersome and difficult to read.

In the context of layout generation, an analog problem is encountered. Layout may become very large, and is subject to many constraints. If someone tries to describe a layout generator in a conventional programming language (FORTRAN, C, PASCAL etc.), he will certainly manage, but very much effort will be required to handle the geometries which are to be created in an efficient way.

Certainly, there have been some attempts to describe layout in a more natural way. Mentor Graphics uses L [MEN92], Compass the VIP language [COM91] etc. However, none of these languages has been designed for technology independence. Instead, some rules on the utilization of the language have been provided to meat this constraint. The design of al2 has been defined with technology independence as a design objective from the beginning.

Another argument for the definition of a new language may be its independence on the design system. Nowadays, the cost and the dependency on commercial design systems are very high, and not everyone can afford to pur-

chase such systems. In order to avoid these kind of financial or even political problems, the choice for the development of a new and independent language may be worthwhile.

Finally, it should be noted that layout design and programming are two different disciplines: a good layout designer does not automatically yield a good programmer. Nevertheless, layout design and programming meet in the domain of generator design. The danger exists that a layout designer gets "lost" when programming in a so called *anarchical* language. Of course, he is able to describe almost everything, but this liberty increases the learning time, and reduces the possible reuse of code and design. A better choice may be a *tiranical* language, which constrains the user to a specific programming model. Al2 should be classified as a tiranical language. It is the compiler which takes care of many of the tedious programming tasks at the cost of a reduced flexibility.

4.2 Language design

The design of a language is subject to multiple iterations, essentially due to evolving requirements and experience. This paragraph should be seen as the result of this evolution, not as the working package which was used to design the language.

The decision the develop TILT was taken after the analysis of the interactive layout design system LISA [SWA88]. Many of the language features find their origin there and have been adopted and improved.

Al2 has a number of properties which are not very common in other programming languages. The most particular ones will be presented and motivated in this paragraph, illustrated with examples.

Variable name propagation

One of the reasons why procedural design systems have not become as popular as might have been expected was the limited relationship between the graphics and the language. Usually, graphics and procedures were treated as two disjoint descriptions of the design. The graphical mask layout was the *result* of the procedure, rather than an active part of it. Consequently, the debugging of a generator is a cumbersome task.

In al2, a strong language-graphics relationship was one of the design objectives. It had to be possible to select an instruction in the language, and to highlight the corresponding parts on the graphical part, and vice-versa. In a final stage, it should even be possible to hide the textual description behind a graphical interface.

One of the essential conditions for a consistent language-graphics relationship is that it is guaranteed that each geometry will have a *unique* identification. Instead of leaving this responsibility to the user, the system may provide a unique variable name to object mapping under four conditions, namely:

- it is not possible to define local variables within a block
- local variable names are superseded by the caller level
- variables may be assigned to an expression only once
- record and index attributes are added to the variable name

These conditions avoid that two geometries may be created with the same name. This strong binding between the source description and the generated result eases debugging considerably and is an essential condition for the establishment of an interactive system. As each object is identified in a unique way, erroneous descriptions are localized more rapidly.

```
c.cont  := RECT(...);
c.poly  := RECT(...);
c.metal := RECT(...);
```

Fig.4.1 Part of the description of the body of ADDCONT

```
cont[1] := ADDCONT(...);
cont[2] := ADDCONT(...);
input := ADDCONT(...);
```

Fig.4.2 Variable names and attributes are automatically propagated to the geometrical objects. In the example above, 9 rectangles will be generated with the names: "cont[1].cont", "cont[1].poly", "cont[1].metal", "cont[2].cont", "cont[2].poly", "cont[2].metal", "input.cont", "input.poly", and "input.metal".

The example above illustrates this automatic identification feature. Note that the caller determines the name of the object. The attributes on every level are automatically added on to the variable names.

Dynamic constants

As a variable can only be assigned once to an instruction, it is not possible to use an instruction such as $i=i+1$. In fact, each variable is a constant! As they are automatically allocated by the system from the heap (like objects), they are called *dynamic constants.*

However, loops are very useful to describe repetitive structures. Al2 solves this problem with the *set* and *cursor* abstraction. First, a set variable is created to hold all values which are normally affected to the loop variable. Then, an iteration over all these set-values within the same instruction is obtained with the cursor operator <>. It forces the instruction inside which it occurs to be evaluated for every value of the cursor.

The constraint that variables may be affected only once means that the order of execution of instructions can be determined from the context. This property has been introduced into the language as a feature: instructions may be

given in any order! As a consequence, instructions are *context-free*, i.e. the result of their evaluation depends only on the contents of their input variables.

```
r := RANGE(1,10);
cont[<r>] := ADDCONT(...);
```

Fig.4.3 Loop behavior is implemented with the use of sets and cursor abstractions. Here, a set r with 10 values is created, and a cursor is used to define 10 contacts.

If instruction ordering is random, the language may be well suited for incremental applications: only the modifications need to be communicated to the program core. The relations between the different variables are sufficient to be able to recompute only those parts which are influenced by the modification. Fast response in the iterative process of layout design becomes possible.

Declarative or imperative?

If we consider layout design as a kind of drawing, it is easy to see that it is not possible to define a specific order to create the drawing. In terms of language properties we may say that drawing is *declarative*, contrary to most conventional programming languages which are called *imperative*, or *procedural.*

Declarative languages are more suited to express a set of *constraints*, or *structural relationships*, rather than indicating how to solve them. For example, the equation $x+y=1$ puts a constraint between x and y, but it does not specify how to compute them.

On the contrary, a procedural language supports only unilateral computation, i.e.: perform predetermined operations on their inputs to produce desired outputs (e.g. $y=1-x$). Procedural languages are well suited to express algorithms. Sequencing, conditionals, loops and functional decomposition are typical ingredients of such a language.

Both approaches have their advantages and drawbacks. Describing the constraints of a regular layout in a purely declarative environment may be cumbersome, the same holds for the imperative approach. Al2 tries to take the best out of both approaches: the declarative form is used to create the structural relations, whereas the procedural form is used to control the data flow.

Memory management

The generation of parameterized layout has a very dynamic character, and asks for a run time allocation of memory to hold the required structure. In order to free the user from the process of memory management, all objects are automatically allocated and freed by the system. It is not even possible to allocate objects explicitly.

This integrated memory management avoids the risk of *dangling pointers*; the system assures that objects which are not referenced to are freed from the heap. This garbage collection feature is essential to manage the data complexity in large structures.

4.3 Entities, types, classes and objects

Most (though not all) languages are said to have the notion of "type", which attaches an intuitive set of properties to variables in the language. For example, we know that integers can be added, subtracted, multiplied and divided.

Initially, types were created as a formalism based on set-theoretic properties. Thus the *type* integer meant the set of all integers. Nowadays, types have become more notational conveniences that lost most of their theoretical inheritance. Types can be seen as an abstraction of *entities* (phones, accounts, transistors) in the application domain.

Classes are software abstractions which express *solutions*. They are related to objects as types are related to entities.

Like most symbolic environments, instances of *each* data type in al2 are objects, as contrasted with the built-in types in C++ (int, char, long, short etc.). For example, all data in al2 benefits from garbage collection. However, al2 has not the same flexibility as a high-level object-oriented programming language like Smalltalk. Like in C++, variable names are tightly tied to the *address* of the storage of the object it denotes, and not as a *label* which can be peeled off one object and applied to another at will. This strong binding lets the compiler ensure that a given variable always goes with an object of a given type.

Constructors and destructors

Two important ideas behind the design of the C++ language were that variables should always be automatically initialized, and that classes should control their own memory allocation. In C++ these tasks are fulfilled by two special member functions: the *constructor*, and the *destructor*.

These ideas have been adopted in al2, however, in a slightly different form. First of all, a particular property in al2 is that all variables are pointers to objects. The compiler initializes these pointers to NULL, and it marks that they have never been defined. Once an assignment is affected, a variable is marked to be assigned. As a consequence, it can be detected if a variable is attempted to be reassigned (even if the new value equals the old one). Thus, a variable which is never assigned to an expression will equal the NULL pointer.

Functions and operators in al2 act as constructors. They define how the object is to be created, and what value is to be put inside. In fact, the creation of an al2 program consists of the definition of constructors only.

Once a variable is out of scope, the destructor is automatically called by the compiler. This destructor is predefined by the compiler and is capable to dispose the memory allocated by any object class.

Creations and references

Now we have seen that all variables are pointers, the results of functions and operators in al2 may be classified into two categories: *creations* and *references.*

Functions and operators which create a new object form the largest class. Here, the compiler calls the constructor to allocate new memory from the heap, and to initialize the object with the result of the operation. A special case may occur when an operand equals the NULL pointer: dependent on the semantics of the function or operator, the constructor is *not* called; the result is a NULL pointer as well. All creations which get out of scope, are automatically deleted by its destructor.

Each function or operator which returns an existing part of an object makes a reference. A reference does not create anything new, and as a consequence, no destructor is called when it gets out of scope. References may either be made to a part of an object which is passed as an argument (e.g. a table reference *a[x]*), or to some kind of predefined object (e.g. *NULL(a)* refers either to the object *TRUE* or *FALSE*).

NULL-pointer propagation

As soon as pointers are introduced in a language, there exists the danger for dangling pointers and access to NULL-pointers. In al2, dangling pointers are avoided with the object destructors. The access to NULL-pointers is handled locally inside each operator and function. For example, all arithmetic operators test whether one of their arguments equals the NULL-pointer. If so, no computation is performed, and the result becomes the NULL-pointer. The same mechanism is applied for reference type functions and operators.

al2 code

```
s := RANGE(0,4);
i = <s>;
r[i] := ADDRECT(POLY);
PLACE(LEFT,r[i],r[i+1]);
```

C code

```
recordType  r[5];
int         i;

for(i=0; i<=4; i++)
   r[i] = ADDRECT(POLY);
for(i=0; i<4; i++)
   PLACE(LEFT,r[i],r[i+1]);
```

Fig.4.4 NULL-pointer propagation can be used to avoid upper- and lower-bound checks

The benefit of this so-called NULL-pointer propagation is illustrated with the elimination of upper- and lower-bound checks in table references. Fig. 4.4 shows a table with five entries, and the function *PLACE()* will insert constraints between two rectangles if both rectangles exist. As soon as a non-existing entry is referred to, a NULL-pointer will be returned, and the *PLACE()* function will perform no action.

Deductive typing

Contrary to most other typed languages, which require explicit declaration of a variable's type or class, al2 uses an implicit *typing by deduction* approach on its variables. Deductive typing means that the programmer does not declare all variable types, but that this work is done by the compiler, taking the context into account. This idea is based on three principles:

- Terminal types are defined by the syntax
- Type directives are terminals with a predefined type.
- Expression types are defined by the operator

The first item states that the compiler uses the form of the terminal to determine its type. For example, a terminal *234* will be detected as an *integer*, and a terminal *23.4* as a *float*.

Type directives are used whenever the compiler cannot determine the type from the context. This is for instance the case for input variables, whose type is not known as such. The second remark states that type directives themselves are terminals, so they can be considered as ordinary expressions. This means that a type directive does not impose the type of the variable, only its own type.

Contrary to the practice in C and C^{++}, the third statement means that the type of the result of an operation is only determined by its operator, and not by the variable into which the results is to be stored.

```
al2 code                          C code

a := 3;                           int    *a=NULL, *b=NULL,
b := a+5;                                *c=NULL, *d=NULL, *e=NULL;
c := a+0.5;                       a=(int)malloc(sizeof(int));
d := 3;                           *a = 3;
d := 3.5;    # type conflict      b=(int)malloc(sizeof(int));
e : integer;                      *b = (*a)+5;
e := 4.7;    # type conflict      c=(int)malloc(sizeof(int));
                                  *c = (*a)+0.5;
                                  d=(int)malloc(sizeof(int));
                                  *d = 3.5;
                                  e=(int)malloc(sizeof(int));
                                  *e = 4.7;
```

Fig.4.5 In al2, the compiler deduces the variable types from the context. Even type directives are only considered as hints.

Deductive typing eases the work of a programmer, and it allows for random instruction ordering. In al2 this practice is feasible because all classes are based on predefined types.

Note finally that all typing in al2 is static. Identifiers may have only one single type, and all types are determined at compile time.

Class hierarchy

The actual version of al2 provides the user with 7 built-in data types, which are the basis for all applications. The user cannot add on its own data types, but it is possible that new ones will be added in the future to ease the resolution of particular problems.

The class hierarchy shows that all data types are derived from the class *root*. All properties defined in a super class are inherited by a subclass, but a subclass may redefine the semantics locally.

The base class root defines that all its subclasses contain a pointer, which is initialized to *NULL*. Furthermore, the base class requires that each subclass has to provide its own assignment, constructor, and destructor routines.

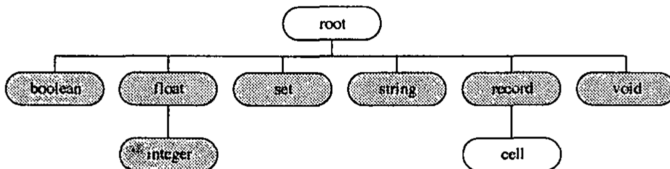


Fig.4.6 Class hierarchy in al2. The classes marked in a gray background are implemented as concrete data types.

Another property defined by the root class is the ability to distinguish between a first assignment, and reassignment. As has been stated before, a variable may only be assigned once to a value. This also holds for the NULL pointer value.

```
a := 2;              # first assignment of a to 2
a := 2,              # run time error, reassignment attempted
IF(FALSE) b := 3;    # b equals NULL
c : integer;         # c equals NULL
d := b+2;            # first assignment of d to NULL
d := 3+c;            # run time error, reassignment attempted
```

Fig.4.7 Examples of properties which are valid for all objects in al2.

The interdiction to assign a variable more than once, is essential in order to guarantee that instruction ordering may be random. The paragraph that deals with the compiler implementation explains how this feature is implemented.

4.4 Expressions and statements

Like in most other conventional languages, an al2 program is composed of a list of statements. Together with a rich set of operators, they allow to control the data flow in a piece of code.

An example is used to illustrate the most common facilities, followed by a summary of the operators, built-in functions and statements. A full description of the syntax can be found in annex A.

A pyramid of contacts

Consider the (hypothetical) case where someone would like to create a "pyramid of contacts", each of which at the minimum distance with regard to its neighbors. If 5 contacts are taken at the base of the pyramid, such a situation should look something like this:

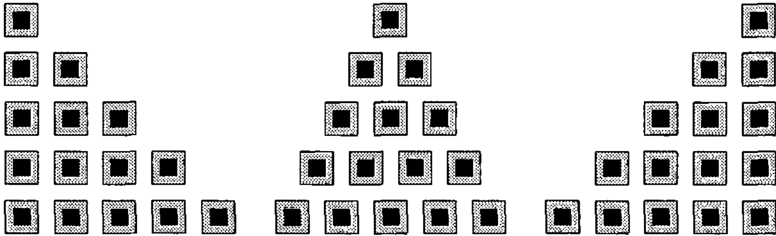


Fig.4.8 Three pyramids of contacts, with parameterized placement and base count.

The header of the description declares the type of the object, and the environment it will use. The role of the environments is important; it allows to import a set of predefined variables (such as *POLY, LEFT* etc.), well adapted to the application.

```
TYPE("cell");
ENVIRONMENT("LAYERS","DEF_RESS");
INPUT(base,option);
base : integer;
option : string;
```

Fig.4.9 Header of a pyramid cell.

Then, there is an *INPUT* declaration, which defines the set of variables that can be given as a parameter at invocation of the program. The types of these variables are specified because the compiler has no context to deduce them. Variables which are not provided in the argument list, are automatically initialized to NULL.

The body of the description starts with a typical construction to retrieve default values with the *GETFIRST* function. This function will return the first non-NULL argument. In fig. 4.10, this means that the base of the pyramid will consist of 5 contacts if no base value is given.

The iteration operator <> is used to define a *cursor* which will scan all the items in the argument. Every iteration operator defines a new independent cursor; in the example, x will be scanned for every value of y. The reference operator = is particular in the sense that it is evaluated only once per instruction. This property is used to refer to the same iteration in different expressions of the same instruction. Note that it is not possible to store an iteration in a variable; the instruction *x:=<set>;* would cause a redefinition error at run time.

```
n = GETFIRST(base,5);
set := RANGE(1,n);
x = <set>;
y = <set>;
IF(x>=y) {
   c[x,y] := ADDCONT(POLY,METAL);
   PLACE(LEFT,c[x,y],c[NEXT(x),y]);
}
PLACE(BOTTOM,c[x,y],c[x,NEXT(y)]);
PLACE(BOTTOM,c[NEXT(x),y],c[x,NEXT(y)]);
IF(option == "LEFT")
   ALIGN(LEFT,c[x,y],c[NEXT(x),NEXT(y)]);
IF(option == "RIGHT")
   ALIGN(RIGHT,c[x,y],c[x,NEXT(y)]);
```

Fig.4.10 Body of an al2 description for the generation of a pyramid of contacts.

The function *NEXT* (cf. *PREV*) is closely related to the iteration operator. It returns the next value of its argument, which should be a cursor. The use of *NEXT(x)* is recommended over $x+1$; not only it is more expressive, but also it is independent on the chosen index range. *NEXT* and *PREV* are the only alternatives to recover preceding or following cursor values if a table cursor is given.

The block construction *{...}* allows to define a common condition for multiple instructions. As instruction ordering may not be considered, the preprocessor expands the block instruction as if the condition would be placed before each single instruction. This means that all contacts will be generated before any *PLACE* function is executed.

```
IF(x>=y)
   c[x,y] := ADDCONT(POLY,METAL);
IF(x>=y)
   PLACE(LEFT,c[x,y],c[NEXT(x),y]);
```

Fig.4.11 A block construction is expanded by the precompiler.

Another feature which is frequently used is the transparency to the NULL-pointer. In our example, the vertical placement of the contacts is defined without testing whether the contact exists for the particular indices x and y. This construction is possible because all operators and functions (except for the function *NULL*), perform no action when one of their operands equals the NULL pointer. As a consequence, al2 code may be written independently on the boundary conditions, which results in very readable and expressive code.

The price to pay is a slight reduction in run time efficiency because the functions are also called with NULL-valued arguments.

As explained in the chapter which treats the compactor, geometries are automatically centered within their free margin. For the pyramid example, this yields the desired result. An optional parameter is added to allow to align the contacts on the left or right side. Again, the NULL-transparent semantics is used to obtain an expressive construction.

Dynamic arrays

Tables, vectors, arrays or whatever they are called, are powerful constructs which allow to hold multiple objects of the same type in a single structure with direct access. Normal vector and array implementations are based on a continuous block of memory which is addressed with a subscript. The address of an item is linearly related to the value of the subscript (direct access).

Al2 does not implement arrays as a linear continuous mapping of a block of memory. Instead, some kind of tree or list structure is used to hold a discontinuous set of subscripts. This means that the size of memory occupied by an al2 table only depends on the number of entries in the table; a table with the entries a[2] and a[3] is as big as a table with the entries a[2] and a[1000]. The reason for this organization is imposed by the fact that variables are not explicitly prototyped: the size of an array is not known until run time. We call this concept *dynamic arrays.*

The dimension of a dynamic array may be arbitrary large[1], and have varying depth. Furthermore, each entry may contain an item, and a sub-table. The distinction is made with the sub-table reference *.

```
a[1] := 3;
a[1,2] := 4;
a[1,3] := 5;
a[2,45,1,1] := 6;
WRITE(a[1]);          # will write 3
WRITE(*a[1]);         # will write [2]4,[3]5
```

Fig.4.12 Dynamic arrays allow for an arbitrary tree like organization of data. Entries may both contain objects and sub-tables.

Generic records

The use of structured data types (*record* in Pascal, *structure* in C) allows for logical organization of the data in separate records. Usually, a type is coupled to a specific structure, and functions are defined to access the fields of the variables.

But suppose that we are interested to write a generic function which is able to write the contents of a field of a certain type, independent on the record

[1] Limited only by the number of entries, not by the depth.

type. In C or Pascal, this implies that the type of the variable needs to be known by the function, to be able to convert the variable to the correct type. Furthermore, the function needs to be updated every time a new record with the corresponding field is added.

Al2 proposes a solution called *generic records*, which has properties similar to lists in LISP [STE84]. All record constructions are of the same type, and records preserve their field names, even during run time. Together with the function *SEARCH*, it becomes possible to define generic operations on fields with a certain name and type.

```
r.rectangle.name : string;
r.rectangle.layer : integer;
w.well : record;
c := ADDCONT(POLY,METAL);
d := ADDRECT(POLY);
t := ADDCMOS(NTR);
cs = SEARCH(c,r);
ds = SEARCH(d,r);
ts = SEARCH(c,r,w);             # ignore the well
PLACE(LEFT,cs,ts);              # iterate over all entries
PLACE(TOP,ds,ts);
```

Fig.4.13 The generic record structure, together with pattern searching, allows for a generic approach to placement, transparent to the data organization. In the example, the search operation is placed before calling the PLACE function, in practice the SEARCH routines are placed inside the PLACE function.

This feature is extremely important in the scope of information hiding and encapsulation. Now, it becomes possible to define a placement of rectangles, independent on the way they are organized in a variable. The search function will scan the record looking for the specific pattern, and it will iterate over all the entries found. Patterns may also be filtered out. This allows us, for instance, to ignore transistor wells of the same type when placing two transistors.

Assignments

In al2, 4 types of assignments exist: *assignment by value*, *assignment by reference, redirection* and a *link*. Each of them has a specific semantics and application domain; they will be shortly discussed here.

```
a := ADDRECT(POLY);
b := a;
```

Fig.4.14 An assignment by value only assigns or copies the object, it does not execute the object constructor. The object b will be a physical copy of a, but only one rectangle (called a) is known by the compactor.

The assignment by value *:=* is by far the most common assignment operator. If the right-hand expression has not yet been affected to any variable, and is different from NULL, the object label is peeled off, and the identifier on the left-hand side becomes the owner of the object. Otherwise, a new object is constructed using the constructor of the class. Note that the

assignment operator only copies the object on the right hand side, it does not re-execute the functions which created the object.

The assignment by reference = assigns the address of the right-hand expression to the left hand identifier. This identifier can then be used in instructions to improve the readability of code. If multiple occurrences occur in the same instruction, the reference is only evaluated once. Assignments by reference are only executed if another assignment type asks for its resolution.

```
set := RANGE(1,4);
x = <set>;
IF(x >= 2)
   c[x] := ADDCONT(POLY,METAL);      # ok
IF(<set> >= 2)
   c[<set>] := ADDCONT(POLY,METAL); # redefinition
```

Fig.4.15 Example of constructions with iterations. The reference assignment is needed to assure that the iteration of set is computed only once for each value in set. A redefinition error occurs at run time if the iteration is literally substituted.

The redirection assignment :: is used to control the name propagation in circular dependencies. This behavior becomes clearer with the example of a function which generates a contact cut between a polysilicon layer, and a metal. The redirection assures that the record fields *cont*, *poly*, and *metal* are included in the name of these rectangles.

```
cont :: ADDRECT(MIN_FIX,CPOL);
poly :: ADDRECT(MIN_STR,POLY);
metal :: ADDRECT(MIN_STR,METAL);
c.cont := cont;
c.poly := poly;
c.metal := metal;
c -> EXTEND(MIN_COVER,cont,poly);
c -> EXTEND(MIN_COVER,cont,metal);
# error1: EXTEND(MIN_COVER,cont,poly);
# error2: c -> EXTEND(MIN_COVER,c.cont,c.poly);
# error3: cont := ADDRECT(MIN_FIX,CPOL);
```

Fig.4.16 The redirection assignment is used in functions to assure correct name propagation.

Notice that functions only compute those parts which are required to compute the output. This means that in any case, the *EXTEND* call needs to be related to the output variable *c* (error1). The fact that instruction ordering is random, together with the deductive typing mechanism, means that the function *EXTEND* cannot take the variable *c* as an operand (error2). If so, the compiler cannot determine the type. A third error may occur if the indirection is not used at all (error3). Now, the name of the caller (cf. *d* in *d[i]:=ADDCONT();*) will not be propagated; each contact will have the name cont!

The last assignment type is the link ->, which creates a dynamic relation between the variable on the left hand side, and variables on the right hand side. Links are used to express assignments of expressions which do not return a

value that will be referred to later on. Most often, these are expressions which insert a constraint, rather than returning a value. As links do not return any value, they are ignored in type checking.

Operator summary

An overview of the al2 operators is given in the table below. The unary operators are right associative, the relational operators non-associative, and the remaining left associative.

Each box holds operators with the same precedence. An operator has higher precedence than operators in lower boxes. Parentheses may be used to resolve precedence conflicts.

Operator summary		
.	member selection	expr . field
[ ]	subscripting	expr [expr_list]
*	subtable reference	* expr
( )	function call	expr (expr_list)
<>	iteration (cursor)	< expr >
-	unary minus	- expr
NOT	complement	NOT expr
!	cast (type conversion)	directive ! expr
**	power	expr ** expr
*	multiply	expr * expr
/	divide	expr / expr
MOD	module (remainder)	expr MOD expr
DIV	division (integer)	expr DIV expr
+	add (plus)	expr + expr
-	subtract (minus)	expr - expr
<	less than	expr < expr
<=	less than or equal	expr <= expr
>	greater than	expr > expr
>=	greater than or equal	expr >= expr
==	equal	expr == expr
<>	not equal	expr <> expr
AND	logical AND	expr AND expr
OR	logical inclusive OR	expr OR expr

All operators propagate NULL-valued operands. So, all operators return NULL, if at least one of their operands equals NULL. Furthermore, the type of the result is only determined by the type of the operands and the operator, and not by the expression into which the result is to be stored. In the example below, a type conflict error message is given, because a table should be of a single type.

```
b[1] := 2**5;     # should be type-casted
b[2] := 2.5**5;
```

Fig.4.17 All members of a table definition should be of the same type. An operator determines its type only as a function of the operands and the operator. The example will yield an error message that the type of b could not be defined.

This bottom-up typing style is an essential issue in the deductive typing mechanism, because it allows to define an expression's type in an implicit way. Type casting can be used to avoid ambiguous cases.

Built-in function summary

The language al2 provides a number of standard library functions. Functions have the advantage of being verbose, and almost self-explanatory. Future releases may add on new functions, but their names will only consist of capitals. The example given earlier in this section uses some of these functions. A detailed description of their properties may be found in [MAL93].

Built-in function summary		
NULL	NULL pointer test	NULL (expr)
GETFIRST	get first non-NULL argument	GETFIRST (expr_list)
ADDRESS	address of variable	ADDRESS (expr)
SEARCH	search pattern	SEARCH (expr_list)
SCAN	scan expressions	SCAN (expr_list)
NEXT	next member in iteration	NEXT (expr)
PREV	previous member in iteration	PREV (expr)
ORD	order of member in iteration	ORD (expr)
EXP	natural antilogarithm	EXP (expr)
LOG	natural logarithm	LOG (expr)
SET	create set	SET (expr_list)
UNION	add sets	UNION (expr_list)
RANGE	create range	RANGE (expr_list)
INTERSECT	multiply sets	INTERSECT (expr_list)
CUT	subtract sets	CUT (expr_list)
MIN	minimum value	MIN (expr_list)
MAX	maximum value	MAX (expr_list)
RECT	create rectangle	RECT (expr_list)
CELL	create cell	CELL (expr_list)
WIDTH	width constraint	WIDTH (expr_list)
SPACING	spacing constraint	SPACING (expr_list)
EXTENSION	extension constraint	EXTENSION (expr_list)
OVERLAP	overlap constraint	OVERLAP (expr_list)
WRITE	write object value	WRITE(expr_list)
NORM	absolute value or length	NORM (expr)
SIGN	sign of value	SIGN (expr)

The function *NULL()* allows to consult the state of this pointer: it accepts one object as a parameter, and it returns a boolean object *FALSE* if non-NULL, and *TRUE* if NULL. It is widely used to control the data flow.

Statement summary

Al2 is a language with very few statement types. The main reason for this is the fact that the instruction ordering is undefined, so each instruction must be context-free. The statement summary shows that the syntax is very similar to "common" languages as Pascal, Algol or C.

As al2 is more restrictive compared to C, it takes a very short time to be able to write an error free program. Experiences with students have shown that most syntactical hooks are understood during the first day.

Statement syntax
statement: { *statement_list* } IF (*expression*) *statement* *structure* := *expression* ; *structure* [*expr_list*] := *expression* ; *identifier* = *expression* ; *identifier* :: *expression* ; *identifier* [*expr_list*] :: *expression* ; *identifier* -> *expression* ; *statement_list:* *statement* *statement statement_list* *structure:* *identifier* *structure . field*

Still, the semantic control of a program takes somewhat more time. Here, a new user needs to understand more about the philosophy behind the language, for example concerning the parametrization and reuse of code and design.

4.5 Exemplars

Writing a program in al2 consists of the definition of class constructors. This constructor must be based on one of the predefined prototypes or exemplars, which can be seen as a super class, whose properties are inevitably inherited by the class being defined.

Actually, five exemplars are defined in the language: *cell*, *class*, *function*, *environment*, and *technology*. Each one has its own semantics and properties. The efficiency of programming in al2 depends largely on this imposed structure.

All exemplars require that the subclasses are specified in two files, the *definition*, and the *body* (= implementation).

The definition file plays the role of the class's interface. This file can be compared with a C header file, but it may contain the interface of only one.

class definition. For example, it defines the base exemplar, and the environment which is seen by the class. Definition files allow for top-down design because they are compiled independently on their corresponding implementation.

The implementation of the class constructor is specified by an al2 *body* file. Here the actual operations are specified. An al2 body automatically loads the corresponding header during its compilation.

An alternative solution is to specify the implementation directly in C. Now, the full C semantics is available, and time-critical parts may be implemented efficiently. The communication between the al2-variables and the C-variables is obtained through a simple interface.

The choice between an al2- or a C-implementation depends on the performance constraints and semantical requirements. In general, layout specific routines are easier to write in al2 because they benefit from the typical al2 semantics. However, time-intensive applications, such as a router, will largely outperform an equivalent al2 solution. Also, functionalities with a recursive character, for example a function to compute the factorial, may be impossible to implement in al2.

Technology encapsulation

Process specific data is stored in separate executable *technology* type objects, and cannot directly be accessed within a description. Instead, the technology data is accessed through generic variables which are instantiated at run time by the technology object. In this manner, technology files may be exchanged or added without recompiling the generator, and the designer is forced to describe the topology based on generic terms.

4.6 Reuse

One of the key items to manage the increasing complexity of layout design is reuse of topology. It may be somewhat surprising that today, still a large effort is spent drawing circuitry with a layout editor. The situation with respect to technology independent design is even worse; whole libraries are redesigned by hand when new processes are released.

One of the main causes here is that historically, layout was made as a function of a specific design. Given the fast evolution of technologies, once this design was implemented, its layout became rapidly obsolete.

With the increasing popularity of object oriented languages, the foundations of reusable software were created. One of the characteristics of this type of software is the high degree of possible customization. A good

example are the widgets in the X-toolkit [NYE90]. Here, a small set of largely configurable basic window building blocks have been reused in an uncountable number of applications all over the world.

One of the conditions to enhance the creation of reusable code is to provide a sufficient parametrization of the cell being created. For instance, if a certain transistor type is designed for reuse, one should foresee the possibility to configure the elements gate size, orientation, fanout etc. Al2 provides some mechanisms which may improve the parametrization and its application:

- NULL-pointer propagation
- pattern matching
- overloading
- defaulting
- instance matching

Null-pointer propagation and pattern matching (see *records*) have already been discussed, the other ones are shortly presented here. More complete examples are given in the chapter *Applications.*

Overloading

Adapted from the object oriented world, the overloading technique fits well to the design intent. In fact, it corresponds to the possibility to assign the same function name to different sets of arguments.

Consider a function which returns the largest of two arguments. The types of the arguments may be either *boolean*, *integer*, *real* or *string*. Without overloading, different functions with different names would be needed to implement this because type casting will fail on the string types. The overloading mechanism will recognize the expression to be executed by matching the types of the input expression.

```
TYPE("function");

FUNCTION(fbb,b1,b2);        fbb : boolean;
FUNCTION(fii,i1,i2);        fii : integer;
FUNCTION(fir,i1,r2);        fir : float;
FUNCTION(fri,r1,i2);        fri : float;
FUNCTION(frr,r1,r2);        frr : float;
FUNCTION(fss,s1,s2);        fss : string;

b1 : boolean;  b2 : boolean;
i1 : integer;  i2 : integer;
r1 : float;    r2 : float;
s1 : string;   s2 : string;
```

Fig.4.18 Definition of an overloaded function.

Observe that the resulting code becomes much cleaner; the same name is used for, semantically speaking, the same action. Also, the same type of operation is concentrated in a single body, and the maintenance of the documentation is largely simplified. More in the field of layout, this feature

really helps to increase the readability. Each time two elements are to be placed, the function *PLACE* is used, independent on the number and type of parameters to be passed.

```
IF(b1 > b2) fbb := b1;               IF(b1 <= b2) fbb := b2;
IF(i1 > i2) fii := i1;               IF(i1 <= i2) fii := i2;
IF(i1 > r2) fir := float ! i1;       IF(i1 <= r2) fir := r2;
IF(r1 > i2) fri := r1;               IF(r1 <= i2) fri := float ! i2;
IF(r1 > r2) frr := r1;               IF(r1 <= r2) frr := r2;
IF(s1 > s2) fss := s1;               IF(s1 <= s2) fss := s2;
```

Fig.4.19 Body of the overloaded function.

New function calls with other argument types and number of arguments may be added without any modification of the existing code. This means that an extension does not require the already existing calls to be re-verified.

Overloading is only implemented for function type objects. The built-in operator objects also apply this mechanism but the user cannot add his own operator calls. The overloading mechanism is not implemented for the class type objects, due to the instance matching technique. Still, a solution is to define an interface function which is overloaded and calls the corresponding class object.

Defaulting

Another technique that increases the comfort of parametrization is defaulting. The al2 semantics defines that all non-assigned variables are initialized to the NULL-pointer. This feature can be used to define the effective value of a crucial parameter.

```
s := GETFIRST(nb_seg,3);    # return first non-NULL argument
tr := ADDSNAKEMOS(NTR,s);
```

Fig.4.20 Illustration of defaulting with the function GETFIRST.

For example, consider a snake transistor generator which takes the number of segments as an input parameter. It may be more elegant to generate a three segment transistor if the parameter is omitted. The function *GETFIRST* is well suited for this task. As it returns the first non-NULL argument in its argument list, it is a simple but efficient way of using default values.

Instance matching

The larger the design, the larger the probability that the same cell is used more than once. In order to manage the data flow, it is important to detect multiple occurrences of the same cell. In an environment where cell calls are parameterized, this can only be achieved if the system may recognize different calls with identical parameters.

In al2, such an instance matching mechanism is defined. For this purpose, cells keep an account of all cell activations. The object checks whether a previous call has been made with the same set of parameters. If so, the result of this call is directly returned, and the cell is not re-executed. Otherwise, the cell is solved, and the result is preserved and recorded by the object.

In fact, each cell maintains a very small data-base. Although very primitive, the effect is surprising and efficient, because the search times remain short, due to the local storage.

4.7 Compilation

It takes quite some effort to define a new language; it takes another bit to provide a good compiler for it. If we only think about machine specific optimizations, portability and so on, the question soon arises whether it is really necessary to use a new language.

Fortunately, there exists an approach we call *bootstrap compilation*. Here, the source code is translated into another language for which an on site compiler is available (for example C). The rest of the work is done by this compiler.

A few remarks are useful on this method:

- all syntax errors should be detected during the translation of the original source code into the intermediate code, and the second compilation should be error free.
- the development time is reduced drastically, and the portability is almost for free. The method reuses the huge effort which is usually invested in commercial language compilers.
- the resulting object code will not be as efficient as directly compiled source code. Its amount is difficult to estimate, but can be an order of magnitude lower if the source language fits badly to the intermediate language.
- if a very common language (such as C) is chosen, fast prototyping is possible on a variety of system architectures, allowing for early stage comparisons.
- the compiler design effort can be concentrated on the "front-end": high level syntax and semantics checking.
- a bootstrap compilation does not only inherit compiler features, but also valuable development tools such as debuggers.

This paragraph will only present the essential parts of the compiler design. It will show some of the "tricks" which were used to implement typical al2 semantics. The presentation follows the items which are essential in most

compiler designs: the preprocessor, the syntax and semantics, the resource allocation, and the generation of the target code.

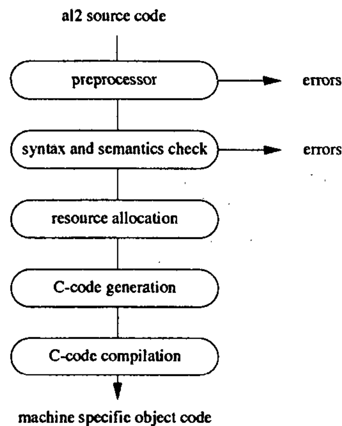


Fig.4.21 Data flow of the al2 compiler.

The preprocessor

The role of a preprocessor is to prepare the data for the compiler, so that it can achieve its task in an efficient way. A preprocessor is usually a filter type program which has no notion of type, scope or structure. The C preprocessor removes comments, and substitutes include files and #define directives.

The advantage of a preprocessor is that the compiler "sees" cleaner input code, and that important errors (such as missing include files) are detected in an early stage. However, the fact that a preprocessor has no notion of type, is a source for trouble. As a result of this, in C^{++}, the use of the language construct *const* is recommended over the use of *#define*.

Beside comment-filtering, the main task of the al2 preprocessor is to allow for incremental compilation. It translates the free format al2 code into a *one instruction per line* format. Now, the differences between two preprocessed al2 codes may be extracted with the standard UNIX program *diff*. Only these differences need to be re-executed in an incremental compiler or interpreter environment.

A special feature of the al2 precompiler is the fact that it expands conditional instruction blocks into separate instructions. This is necessary because of the constraint that instruction ordering is random, even inside instruction blocks. Only if the instructions are effectively expanded, a meaningful difference may be computed.

Syntax and semantics

The terms *syntax* and *semantics* are widely used in informatics, but they are often used in the wrong way. Here, we will speak of syntax if it concerns the grammatical rules in the language. The semantics has more to do with the *meaning* of the words (commands).

It may be stated that it is much more difficult to verify the semantic of a language, than its syntax. The reason for this is that a syntax may be specified much more easily with rules than a *meaning*. The semantic of a language has much more to do with its behavior. The precise definition of both the syntax and the semantics is essential to allow for system independent implementations.

```
%% /* Beginning of rules section */

list    :   /* empty */
        |   list stat '\n'
        |   list error '\n'         { yyerrok; }
        ;
stat    :   expr                    { printf("%d\n",$1); }
        |   LETTER '=' expr         { reg[$1] = $3; }
        ;
expr    :   '(' expr ')'            { $$=$2; }
        |   expr '+' expr           { $$=$1+$3; }
        |   expr '-' expr           { $$=$1-$3; }
        |   expr '*' expr           { $$=$1*$3; }
        |   expr '/' expr           { $$=$1/$3; }
        |   expr '%' expr           { $$=$1%$3; }
        |   expr '&' expr           { $$=$1&$3; }
        |   expr '|' expr           { $$=$1|$3; }
        |   '-' expr %prec UMINUS   { $$= -$2; }
        |   LETTER                  { $$=regs[$1]; }
        |   number
        ;
number  :   DIGIT                   { $$=$1; base=($1==0)?8:10; }
        |   number DIGIT            { $$=base*$1+$2; }
        ;
%%
```

Fig.4.22 Typical yacc format of the rules section. Note the close resemblance with the BNF syntax specification format.

The lexical and syntax analysis is usually performed by a parser. Its role is to detect errors in the input format, and to check type consistency of the description. Then, an intermediate code is generated which is compiled into the final object code.

The implementation of a parser is largely simplified with the use of the automatic parser generator *yacc* [JOH90] and the lexical analyzer *lex* [LES90]. Yacc allows the user to specify the structure of the input language in terms of rules, close to the BNF format. Each rule consists of a sequence of lexical items (provided by lex), such as an integer, an operator or a keyword. The parser tries to match a rule by shifting a new lexical item on a stack until it is able to reduce a rule. Once a rule is reduced, the action related to it is executed. Error patterns can also be captured by rules. This allows for very clear and precise error reports, which eases debugging of source code.

As instruction ordering in al2 is random, type checking may only be started after the parsing of the source file is completed. In al2, typing is done *by deduction*. This means that all routines need to provide a unique input to output mapping, and that terminal types need to be determined from the lexicals. The dependencies between the variables are used to scan the parsing tree in the right order.

Resource allocation

Once a description's syntax and semantics are correct, the effort can be concentrated on the generation of the target code. Scanning the parsing tree once more, it is determined which routines are effectively required, and which variables (including intermediate ones) are effectively needed. This stage may be qualified as resource allocation; we know exactly which functions and variables need to be declared for our description.

Another function of the resource allocation phase is to determine the sequence of the instructions. As al2 instructions may occur in any order, this sequence is to be derived from the dependencies between the variables.

Code generation

The final stage of the al2 compiler concerns the target code generation. This target code should be error-free, as syntax errors are already detected in previous stages. For reasons of portability between different platforms, the al2 compiler generates an ANSI-C format. It is this code which will be compiled into final object code by an on-site ANSI-C compiler.

```
TYPE("function");
FUNCTION(out,n);
n             : integer;
base          : integer;
out.table[]   : integer;
out.dim       : integer;
```

Fig.4.23 Al2 definition file of a base converter function.

```
b := 2;
i = <r>;
IF(n<0)
   out->WRITE("no negative numbers allowed");
IF(n>=0) {
   dim := integer|(LOG(n)/LOG(b)+1);
   r := RANGE(0,dim);
   out.table[i] := (n DIV integer|(b**i)) MOD b;
   out.dim := dim;
}
```

Fig.4.24 Al2 body file of a base converter function.

The different parts of the code, generated by the compiler are best illustrated with a simple example. It converts an integer into an arbitrary base representation.

First, the body passes through the al2 precompiler. Here, comments are stripped, conditions are expanded, and each statement is put on a single line. Note how the statement separators are replaced by newlines, whereas newlines are replaced by semicolons.

```
b:=2
;i=<r>
;;IF(n<0);  out->WRITE("no negative numbers allowed")
;IF(n>=0) {;   `IF(n>=0) '
 `IF(n>=0) 'size := integer|(LOG(NORM(n))/LOG(b)+1)
`IF(n>=0) ';   r := RANGE(0,size)
`IF(n>=0) ' ;  out.table[i] := (n DIV integer|(b**i)) MOD b
`IF(n>=0) ';      out.size := size
`IF(n>=0) ';}
.;;
```

Fig.4.25 Al2 body after passing the preprocessor

When compiled, the al2 code above will produce the following C-code:

```
#include "Tclc.h"                                              /*1*/
extern void free(void* pointer);                               /*2*/
extern TclcProto(biggerEqual_ii);
extern TclcProto(cast_ir);
extern TclcProto(plus_ri);
extern TclcProto(quotient_rr);
extern TclcProto(LOG_i);
extern TclcProto(NORM_i);
extern TclcProto(assign_g);
extern TclcProto(record_g);
extern TclcProto(RANGE_ii);
extern TclcProto(iteration_e);
extern TclcProto(mod_ii);
extern TclcProto(div_ii);
extern TclcProto(exponentation_ii);
extern TclcProto(index_g);
extern TclcProto(point_g);
extern TclcProto(recordTable_g);
extern TclcProto(smaller_ii);
extern TclcProto(WRITE_s);
static TclcProto(CONVERT)                                      /*3*/
{
char *_list=(void*)&_aList,_buffer[BUFSIZ];                    /*4*/
jmp_buf _exception;
TclcRstack _pile[100],*_base=_pile,*_head=_pile;
static int  _ti[3]={0,2,1,};                                   /*5*/
TiltTdata   i,b,dim,_id0,r,n,out;                              /*6*/
SEED(7);                                                       /*7*/
TiltTdata _ex[7];

switch(argCount) {                                             /*8*/
case 0 : {
n=TclcGetData(0);                                              /*9*/
START;                                                         /*10*/
out=UNDEF;                                                     /*11*/
{
static int _ty0[2] = {104,4,};
static char *_la0[2] = {"table","dim",};
TclcCreateRecord((TiltTrecord*)&out,2,_la0,_ty0,0);            /*12*/
}
n=_OK(n);                                                      /*13*/
b=UNDEF;
b=assign_g(1,(TiltTdata)&_ti[1],b);
_UPDATE;                                                       /*14*/
```

```
b=_OK(b);
dim=UNDEF;
_EQU(___id0);                                              /*15*/
if(_TEST(_id0)) {                                          /*16*/
_ex[0]=NORM_i(1,n);
_FREE(_ex[0],4);                                           /*17*/
_ex[1]=LOG_i(1,_ex[0]);
_FREE(_ex[1],5);
_ex[2]=LOG_i(1,b);
_FREE(_ex[2],5);
_ex[3]=quotient_rr(2,_ex[1],_ex[2]);
_FREE(_ex[3],5);
_ex[4]=plus_ri(2,_ex[3],(TiltTdata)&_ti[2]);
_FREE(_ex[4],5);
_ex[5]=cast_ir(2,"integer",_ex[4]);
dim=assign_g(1,_ex[5],dim);
}
_UPDATE;

dim=_OK(dim);
_EQU(___id0);
if(_TEST(_id0)) {
_FIELD("dim");                                             /*18*/
out=record_g(2,"dim",(dim)?TclcCopyInteger((int*)dim):NULL,out);
}
_UPDATE;

r=UNDEF;
_EQU(___id0);
if(_TEST(_id0)) {
_ex[0]=RANGE_ii(2,(TiltTdata)&_ti[0],dim);
r=assign_g(1,_ex[0],r);
}
_UPDATE;

r=_OK(r);
_EQU(___id0);
if(_TEST(_id0)) {
_FIELD("table");
{ TiltTdata _tmp=point_g(2,"table",out);
_EQU(__i);
_INDEX(i);                                                 /*18*/
if(i){
_ex[0]=exponentation_ii(2,b,i);
_FREE(_ex[0],5);
_ex[1]=cast_ir(2,"integer",_ex[0]);
_FREE(_ex[1],4);
_ex[2]=div_ii(2,n,_ex[1]);
_FREE(_ex[2],4);
_ex[3]=mod_ii(2,_ex[2],b);
_tmp=index_g(2,i,_ex[3],_tmp);
}
out=recordTable_g(2,"table",_tmp,out); }
}
_UPDATE;

_ex[0]=smaller_ii(2,n,(TiltTdata)&_ti[0]);
_FREE(_ex[0],3);
if(_TEST(_ex[0])) {
WRITE_s(1,"no negative numbers allowed");
WRITE_nl(0,NULL);
}
_UPDATE;

out=_OK(out);
_FREE_i(b);_FREE_i(dim);_FREE_e(r);                        /*19*/
return(out);
} break;
```

```
}

{ static jmp_buf _jmp;
__i : bcopy((char*)_exception,(char*)_jmp,sizeof(_jmp)); /*20*/
{ static TiltTdata _ex[2];
_ITERATION;                                              /*21*/
_ex[0]=iteration_e(1,r,&_seed,_seed.code);
_ITER(_ex[0]);
i=_ex[0];
 }
_JMP; }

{ static jmp_buf _jmp;
___id0 : bcopy((char*)_exception,(char*)_jmp,sizeof(_jmp));
{ static TiltTdata _ex[2];
_ex[0]=biggerEqual_ii(2,n,(TiltTdata)&_ti[0]);
_id0=_ex[0];
 }
_JMP; }
}

TclcProto(CONVERT_i)                                     /*22*/
{
char *_list=(void*)&_aList;
return(CONVERT(0,TclcGetData(0)));
}
```

Fig.4.26 Intermediate C-code as produced by the al2 compiler.

Other object types (environments, classes, etc.) are translated into a similar structure, in particular with respect to the principal function body, and the individual instruction translation. The computation mechanism uses a stack to keep track of the iterations and cleanup operations. This stack should not be confused with the one used in the *stack machine* approach.

A detailed step-by-step explanation is given below:

1 include the standard TILT environment, which defines the standard types, and a set of interfaces to basic TILT functions.
2 prototyping of the routines which are defined in other object modules. This prototyping is necessary to meet the requirements of the *ANSI-C* standard.
3 macro which declares the body of the routine, and start of the private implementation section. All routines have the same type of parameter list, i.e. an integer which indicates the number of arguments in the list, and a list of arguments.
4 list of common variables used for command flow management.
5 all integer and real terminals are declared in a static array. This allows for an efficient implementation as we may directly take the address of the variables.
6 declaration of all the variables used in the object.
7 declaration of the seed used for iterations, and an array *_ex* used to store temporary expressions.
8 switch to select the instruction set which is to be executed for the corresponding function call (see also 22).
9 macros to fetch the arguments from the parameter list.
10 beginning of the resolution. This macro initializes the local stack.
11 all variables are initialized exactly once to the predefined value *UNDEF*. It is this

initialization which allows us to detect whether an identifier is defined exactly once or not. This, because the identifier value is passed to every assignment routine, which verifies whether the entry to be assign equals *UNDEF*.

12 the record types and field names are defined as static variables in the object. Here, the record prototype is created, which will be filled in by the different assignment functions.

13 the macro *OK()* is applied exactly once for each variable. Its role is to set variables which are never assigned to a value, to the value *NULL*.

14 the macro *UPDATE* executes the stack cleanup function. This function tries to pop the calculation stack and treats the items which are found. These items include instruction to dispose memory of intermediate expressions, and jumps to iteration points.

15 the macro *EQU()* causes a jump to an assignment by reference. The current context is first save in the variable *_exception*, then the macro jumps forward to the implementation of the assignment (see 20), using the label given as an argument. Notice that such an implementation has the advantages of a function call (sharing of resources), but it does not require any parameter passing. Furthermore, the implementation is shared between all instruction sets.

16 the macro *TEST()* is used to control conditional statements. It returns *0*, if its argument equals *FALSE* or *NULL*. Otherwise it returns *1*, so that the statement block following the condition will be executed.

17 the macro *_FREE()* pushes the current expression and its type on the stack so that it will be disposed during the stack cleanup procedure.

18 the macros *FIELD()* and *INDEX()* are used for the automatic identifier name propagation. In this way, a geometry may inherit the name of the record field or the index of the table entry.

19 all variables which are not returned or exported by the object are disposed here. The compiler keeps track of the assignments made by the user, and is able to determine which variables will never be referred to anymore.

20 implementation of the *assignments by reference*. The assignments are placed after the return of the routine and are entered following an *EQU()* macro call. As an identifier of an assignment by reference may be used in multiple instructions, a *longjmp* is made to branch back to the position where the context was saved.

21 the macro *ITERATION()* initializes the cursor mechanism. It passes the seed to the iteration procedure, which uses it to store an array of iteration values. Furthermore, the macro saves the context in the seed; this context will be used later on to jump back to the correct position.

22 each possible call of an overloaded function is translated into a unique function name. Like in C^{++}, a call to a function *f*, maps to an internal name *f_something*, where *something* is derived from the types of the function's arguments. Each function-argument combination is mapped to a unique switch label in the private implementation section, which allows to share code.

A remark is in place here. It can be seen that the code generated from a simple al2 description becomes quite large; 20-40 as much lines! The main reason for this is the fact that al2 implements quite a few object oriented properties (such as overloading and automatic object management), which are not or badly supported by C.

As an estimation, the number of code lines could be only 2-4 times as much if C^{++} would have been chosen as the intermediate language. In that case, concrete data types could be defined for the TILT object types, operators and function could be overloaded, and the memory management could be integrated into the objects. The resulting code would be more efficient and easier to maintain. Unfortunately, C^{++} was not considered by the design team when the tool was developed.

System functions and operators

The implementation of every operator and system function follows a clear and modular schedule. They contain a principle routine which is called for all specific compiler operations, such as type checking, and the two pass code generation phase.

```
void routine_ii(…) {…}
void routine_ir(…) {…}
void routine_ri(…) {…}
void routine_rr(…) {…}

void routine(int mode,…) {
  switch(mode) {
    case SEMANT :
      …
      break;
    case PASS1 :
      …
      break;
    case PASS2 :
      …
      break;
  }
}
```

Fig.4.27 General structure of al2 operators and functions showing the principle routine called by the compiler, and the type specific implementations.

Beside this principle routine, one routine for each possible combination of input parameter types has to be provided. It is the implementation of the overloading mechanism (see also Annex C).

Another possibility is to use a single implementation function, but to pass an additional integer value to select the appropriate functionality. This solution results in a smaller number of global functions, and allows for resource sharing, thus being more efficient during the link phase. However, an additional test is to be performed at run time, slowing down the execution speed slightly. The built in operators and functions use the first technique,

whereas the transcription of al2 code is based on the second technique. Both implementations come very close to the structure which would be used in a C++ environment.

```
void routine_(int id,...) {
  switch(id) {
    case ...
      ...
  }
}

void routine(int mode,...) {
  switch(mode) {
    case SEMANT :
      ...
      break;
    case PASS1 :
      ...
      break;
    case PASS2 :
      ...
      break;
  }
}
```

Fig.4.28 Alternative structure to implement al2 operators and functions which uses a lower number of global functions and allows for resource sharing.

Adding on external C functions

Time critical or not external functionalities implemented in C can be added on easily. The process consists of three operations:

- the definition of an al2 header file
- the parameter transfer between the C function and the al2 function
- the establishment of a link between the C object and the al2 object

The definition of an al2 header file is the same as if the function would be implemented in al2. It defines that the object is a function, and which parameter combinations are valid. The fact that the header is the same for both an al2 and a C implementation is important because this allows to interchange them in a transparent way.

The parameters transfer between an external C function and the al2 call consists of the conversion of the different data formats. Here the values are recovered from the input parameters, and the result is to be encoded into a data format which is available in al2. An empty al2 body can be used to generate a prototype of the module of this interface. Note that a somewhat detailed knowledge is necessary about the way the internal al2 data structures are implemented.

The link between the objects created with the C function, and the objects created with the al2 code is needed to allow the loader to solve all the external references. Any kind of object code may be coupled to the system in the way described above, which makes it a very flexible mechanism.

4.8 Interpretation

There are many situations were a particular source format is to be translated into another, say target, format. In principle, there exist two basic methods to achieve this transformation: *compilation* and *interpretation*. The different characteristics of these approaches may be illustrated with a small example, where a the source format is Japanese, and the target format English.

If the Japanese text is interpreted, each occurrence of a particular sign is directly substituted by a sequence of English words (interpretation). It may be clear that the resulting English text may be somewhat irregular and inefficient, because there is no knowledge about the context.

A compilation of the Japanese text would first read the whole Japanese text, and then produce a text which takes particular characteristics of the English language into account. Now, the result will become more regular and may be adapted to the English grammar.

Interpretation is usually applied in interactive situations, such as a discussion between the Japanese emperor and the English queen. Compilation is efficient in those cases where the translated result is likely to be used multiple times, for instance for a patent description. Compilation may benefit optimization methods to improve the performance.

Concretely, the basic function of interpretation is to execute successive source-language statements by translating them into actions. The implementation of this function in an interpreter is particularly straightforward. For each sequence, the same sequence of steps is performed:

1 *obtain statement*
2 *determine action*
3 *perform action*

The execution of a machine-language program by a micro processor is a good example of an interpreter built in hardware. The interpretation sequence is the following:

1 *fetch an instruction from the location specified by the instruction counter (IC)*
2 *increment the IC in preparation for the next fetch*
3 *decode the instruction*
4 *execute the instruction*
5 *repeat step 1 to 4*

The normal sequential execution can be altered in step 4 by modifying the content of the IC. It is in this way that conditionals, loops, and jumps are implemented.

One of the first popular programming languages was interpreted (BASIC). As a matter of fact, the cyclic character of an interpretation is much the same in high level languages. However, in al2 the instruction ordering is random. This means that the al2 instructions cannot be executed "as such", the execution order needs to be determined before. Furthermore, the language al2 is very modular and hierarchical. Other specific constraints on the automatic naming and memory management complicate the implementation of a, usually straightforward, interpreter.

The advantages of interpretation may be obvious: there is no time lost to compile and link the code. However, a price is paid in efficiency, because instructions in loops are reinterpreted for every loop variable, and no optimizations are possible (such as the elimination of invariant parts in a loop). However, the recent developments show that interpretation regains interest; partially because of the increasing computing power, and partially because of its programming comfort (spreadsheets). Still, al2 is particularly well suited to be interpreted because loops are concentrated in the instruction itself: no price is paid to re-interprete instructions in a loop.

Several experiences have been obtained during the TILT project with our attempts to realize an interpreter for the language al2. In particular an incremental version of an interpreter has been subject to experiments.

Incremental execution

Whereas interpretation eliminates the phase of code compilation, a further gain may be obtained with the introduction of incrementality. Such a method is characterized by the fact that only the differences between two updates are executed, not the whole new version. Incremental software approaches, will allow us to gain an order of magnitude for instance during simulation, because it avoids to compute what is already known, so it fits much better to the stepwise refinement engineering approach[1].

The syntax of al2 is well suited for an incremental solution because of the self-contained instructions, and the parallel character of a layout description. As the language is primarily used to describe geometries and constraints, it is probable that the difference between two descriptions will have only a local effect. This contrary to a typical sequential language, where a modification in an early stage would cause all following instructions to be reinterpreted.

The price to pay is the increase of instruction run time and interpreter complexity. Consequently, incremental execution should only be applied during the development phase of an application. A compiled version should be used for repeated utilization of the generator.

[1] statement of John Darendinger, IBM, in a forum at EDAC'93, Paris

During the development of the TILT software, two incremental prototypes were developed. The first was realized at an early stage of the project, when no hierarchy had been introduced. At that moment, the interpreter was the only way to execute the source code.

A second version was realized later on in the project. Here, the incrementality was complete: someone could modify any source file, and the system was able to update the description completely. However, the performance loss was enormous. The possibility to modify a description at any level in the object hierarchy, imposed a complete influence and dependency relation scheme. Furthermore, the object management was reversible (create and delete), and many objects were to be loaded into memory, resulting in a large system complexity. Therefore, the decision was made to concentrate the effort on a compiler. No new incremental version has been realized since.

Still, a *close to compiler implementation* of an incremental interpreter may be worthwhile if it is realized in C^{++}, and if the incremental character is limited to the top level of the description. On the other hand, a non-incremental, but hierarchical, interpreter will have a lower complexity, and may sufficiently reduce the iterative design cycle.

4.9 Debugging

A language without a debugger, is like a car without a tool box. Everything works fine as long as the periodic services are made and nothing crashes. Indeed, a debugging tool is at least as important as the compiler. The implementation of a language specific debugger may take quite some time. Again, benefit may be taken from the intermediate language. Keeping the translation from the source language to the intermediate language as clean as possible, the on-site debugger may be used for this purpose.

Still, a "medium" al2 user should not need to use any specific debugging facility. As the language al2 is much more restrictive than C, experiences show that the simple use of the function *WRITE* is sufficient to discover the origin of the problems.

4.10 Conclusions

The design of a new language or format is a tedious and time-intensive task. The format should be well defined, a compiler or interpreter is to be implemented, and afterwards a new language is to be learned. Still, languages pop-up everywhere, especially in the academic world. A language that is developed for a specific application environment may help a lot to manage the problems to be solved and can improve the reusability of the code.

It is a good practice to base the language's syntax on some known language. Nothing worse than some exotic format which has to be learned from the beginning.

Semantics

Al2 has been designed for the implementation of technology independent layout generators. Its object oriented character allows to hide the access to technology parameters, and allows for a consistent separation of generic and technology specific data. Several original features have contributed to the establishment of a powerful semantics, such as:

- deductive typing
- insensitivity to instruction ordering
- automatic variable name propagation
- NULL-pointer transparency
- dynamic arrays
- generic records
- pattern search
- automatic object allocation and destruction
- function overloading
- defaulting
- instance matching

The insensitivity to the order of the given instructions implies that variables may be affected to an expression only once. This concept of *dynamic constants* implies that recursive structures may not be implemented.

Implementation

The development of a new language had a large impact on the project. We may state that during 2-3 years our activity was concentrated either on the language definition, the interpreter implementation, or the development of the compiler. The use of the tools *yacc* and *lex* is essential here. They allow to define a complete parser of the input format in a reliable and flexible way.

The implementation of a compiler is largely simplified if it produces an intermediate format in the same language as the source code of the compiler itself (instead of machine dependent assembler or object code). The advantages of such an approach are multiple: apart from the reduced design time, the compiler is easily ported on a platform for which a compiler of the intermediate language exists, and the linker and machine specific optimizations are inherited for free. Also, someone benefits from the tools which are already available for that specific language, such as a debugger and library facilities. Finally, it eases the possibility to add external functions, resulting in an *open system.*

The key-point is that the development of a new language should be based as much as possible on existing features in the system. This reduces the design time considerably, and it increases the chances to be adopted in a larger system.

Improvements

The time needed to implement a compiler by translating a language into another language is largely influenced by the semantics of the intermediate target language. The choice to implement the compiler in C was historical: at the time the project started, it was the most popular language for any serious tool development. Today, the language C++ would be preferred, because its semantics already supports much of the required object oriented features.

As an estimation, I believe that the size of the standard library, in terms of source lines, could be reduced with more than 50% if it would be rewritten in C++. An even more important reduction can be expected from the translation of the al2 objects into the intermediate code. The run time performance would be improved as well.

4.11 References

[AHO86] A. Aho, R. Sethi, J. Ullman, *"Compilers"*, Addison-Wesley, 1986.

[CAL79] P. Calingaert, *"Assemblers, Compilers, and Program Translation"*, Computer Science Press, 1979.

[COM91] COMPASS Design Automation V8R3, *"LAYOUT TOOLS, Symbolic Editor, VIP Language"*, 1991.

[COP92] J. O. Coplien, *"Advanced C++, programming styles and idioms"*, AT&T Bell Laboratories, *Addison Wesley*, 1992.

[JOH90] S. C. Johnson, *"Yacc: Yet Another Compiler-Compiler"*, AT&T Bell Labaratories, Muray Hill, New Jersey 07974, HP BSD UNIX Programmer's Manual section 6, No. 017272-A00, 1990.

[KER88] B.W. Kernighan & D.M. Ritchie, *"The C Programming Language, 2nd edition"*, Prentice Hall, 1988.

[LES90] M. E. Lesk & E. Schmidt, *"Lex - A Lexical Analyzer Generator"*, AT&T Bell Labaratories, Muray Hill, New Jersey 07974, HP BSD UNIX Programmer's Manual section 7, No. 017272-A00, 1990.

[MAL93] G. Maliki, S. Ponta, R. Riem-Vis & P. Vaucher, *"Al2, Un langage de description de layout"*, IMT report version 3.0, Neuchâtel, Switzerland, March 1993.

[MEN92] Mentor Graphics, *"L Database and Language Users Guide"*, Software version 5.2_1, 1992.

[NYE90] A. Nye and T. O'Reilly, *"X Toolkit Intrinsics Programming Manual"*, O'Reilly & Associates, Inc., Second edition for X11, Release 4, September 1990.

[STE84] G.L. Stelee Jr., *"Common LISP: The Language"*, Digital Press, 1984.

[STR87] B. Stroustrup, *"The C++ Programming Language"*, Addison Wesley, 1987.

[SWA88] M. van Swaaij, *"LISA, A Declarative Language For Interactive Layout Generation"*, CSEM technical report no. 193, Neuchâtel, Switzerland, 1988.

5 Applications

5.1 Introduction

It is relatively easy to imagine a new concept. However, its implementation may be somewhat more complicated, and, more important, may discover some unforeseen features. Furthermore, a concept can be technically well founded and yield good results, but it may fail because it does not fit with user habitudes.

TILT may be considered as a new concept, using an object oriented hierarchical language while prohibiting the direct access to technology parameters, and which directly creates the geometries and constraints for a spacing program. The method focuses on parametrization, in order to capture the design knowledge, and to increase the reusability of the layout descriptions.

How does such an approach look like in practice? Which results can be obtained, and at which cost? This chapter tries to provide an answer to these questions. It presents the various projects which have been running at our institute. These projects were defined such that they allow to validate some specific properties of the system.

Outline

The first section presents the basic design environments for CMOS and bipolar technologies. It treats a technology file definition and the set of topology oriented functions which allow to encapsulate low-level technology parameters.

Then, a section is dedicated to the design of parameterized standard cell generators. It appears that a specific set of convenience functions may help to

reducc the development time, while still preserving hand-crafted design's equivalent densities.

The validation of hierarehy was performed with a static RAM generator. It gives an indication on typical run times and acceleration techniques for large regular circuitry, as well as limitations on memory requirements. It also illustrates the use of flexi-cells: cells which adapt to context.

Then, some generators for analog circuitry are presented. Different types of transistor generators are treated, both for a CMOS and a bipolar technology. Also, a bipolar "gate-array" structure is given, with the construction of a regular floor plan and repetition of elements.

Finally, the open character of the system is illustrated with a switehed capacitor silicon compiler. Based on a rigid floor plan, it is able to generate Laker biquad filters, using the capacitor values as parameters. Here, placement and routing procedures have been added in the tool's environment.

Figures with layout

All of the layout examples that are given in this chapter were integrated into the document with the help of the tools PostScript® driver. Instead of repeating a legend with each layout, it is given once below, both for CMOS and bipolar layout styles.

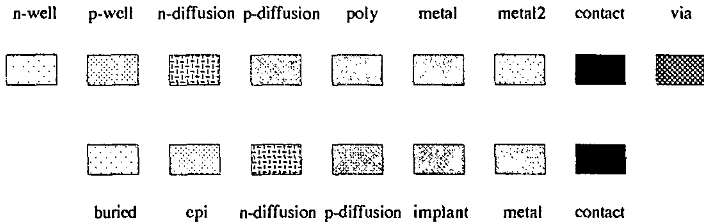


Fig.5.1 Pattern to layer mappings of CMOS technologies (top range), and bipolar technologies (bottom range).

Another important item is the scale of the layout. In this document, 5 different scale factors have been used in order to provide a sufficient level of detail in the examples. Figure 5.2 defines a ruler for each of them, and the scale number is mentioned with each layout example.

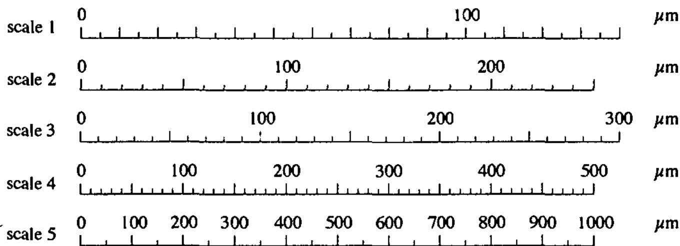


Fig.5.2 Definition of the different scale factors that were used for the examples in this chapter.

5.2 The CMOS and the bipolar library

From the geometrical point of view, the al2 language only defines six basic functions: *RECT()*, *CELL()*, *WIDTH()*, *SPACING()*, *EXTENSION()* and *OVERLAY()*. These functions correspond to the interface of the compactor, and they allow to define the spatial constraints while addressing the geometries at the rectangle level, not at the edge level. At this point, the set of geometries and constraints is technology specific. The constraint values are already computed using the current technology.

Whereas this interface is convenient to control the compactor, it is somewhat uncomfortable for a layout designer. Imagine the amount of code to describe a 10 transistor cell, containing some hundreds of rectangles! And what about the technology independence?

Thus, some higher abstraction level was needed which is closer to the type of layout we want to describe. This feature has been implemented in the form of a CMOS and a bipolar library. They provide a technology independent tool kit containing a set of general configurable operations out of which a cell can be composed: a set of technology descriptions, a symbolic environment, declaration of elements, and constraints between these elements.

Members of the CMOS technology independent library			
Technology	**Environment**	**Declaration**	**Topology**
VTI12U	DEF_RESS	ADDBULK	ABUT
VTI2U	LAYERS	ADDCONT	ALIGN
SACMOS2U	RULES	ADDMOS	ATTACH
MIETEC2U		ADDRECT	CONNECT
			EXTEND
			PLACE

Each of these tool kit function groups will be discussed in the following paragraphs. They form the kernel of the TILT design system and are used in all of the subsequent applications.

Technology

The characteristics of a fabrication process are described in an al2 technology object. A particular feature of such an object is the fact that it provides an executable program. On invocation, it accepts the name of the variable whose result is printed on the standard output stream. When executing an al2 generator, the technology output stream is redirected and parsed internally, thus establishing a dynamic link.

The first data type which is described in the technology object concerns the *design rules*. They are commonly formulated with the help of *design layers*, a kind of symbolic layers which are only used during the design. The role of these layers is to simplify the definition of the design rules, because they can be related with specific situations. Finally, the set of design layers is mapped to a (smaller) set of physical layers.

As an example, consider the n-type diffusion layer. Different design layers are introduced to specify the design rules for a diffusion that is part of the active area in a transistor, a diffusion used to implement a resistor, a diffusion used to create bulk contacts, or a diffusion used in the remaining cases (interconnect). Now, an extension rule of a diffusion on a gate, will only be defined for the transistor diffusion, and different clearances may be defined between diffusions which are part of an active area or not.

Not only the specific design rules can be described in the technology object. In fact, any kind of technology *dependent* data, such as the square resistance and unit capacitances, or the correspondence between the fanout and the transistor width, should be defined here.

```
 # Diffusion Rules
#########################################################

TR_DIFF      := SET(NTR,PTR);          # set of transistor diffusion
ND_DIFF      := SET(NDIFF,NTR);        # set of n-doped diffusion
PD_DIFF      := SET(PDIFF,PTR);        # set of p-doped diffusion
GUARD        := SET(NGUARD,PGUARD);      # set of guard diffusion
DIFF         := SET(ND_DIFF,PD_DIFF,GUARD);    # set of diffusion

SPACING[<DIFF>,<DIFF>]          := 3.5;
# inside n-doped tub
EXTENSION[<ND_TUB>,<PD_DIFF>] := 5.0;
# inside n-doped tub
EXTENSION[<ND_TUB>,<ND_DIFF>] := 3.0;
# outside n-doped tub
SPACING[<ND_TUB>,<ND_DIFF>]    := 7.0;
SPACING[<ND_DIFF>,<ND_TUB>]    := 7.0;
# outside n-doped tub
SPACING[<ND_TUB>,<PD_DIFF>]    := 5.0;
SPACING[<PD_DIFF>,<ND_TUB>]    := 5.0;
# inside p-doped tub
EXTENSION[<PD_TUB>,<ND_DIFF>] := 4.0;
# inside p-doped tub
EXTENSION[<PD_TUB>,<PD_DIFF>] := 2.0;
# outside p-doped tub
SPACING[<PD_TUB>,<PD_DIFF>]    := 8.0;
```

```
SPACING[<PD_DIFF>,<PD_TUB>]      := 8.0;
# outside p-doped tub
SPACING[<PD_TUB>,<ND_DIFF>]      := 6.0;
SPACING[<ND_DIFF>,<PD_TUB>]      := 6.0;

# guard diffusion
WEIGHT[<GUARD>]                  := 8;
WIDTH[<GUARD>]                   := 2.0;
# n+ guard ring
OVERLAY[NGUARD,<ND_TUB>]         := 1.0;
OVERLAY[<ND_TUB>,NGUARD]         := 1.0;
# outside p-doped tub
SPACING[NGUARD,<PD_TUB>]         := 2.0;
SPACING[<PD_TUB>,NGUARD]         := 2.0;
# p+ guard ring
OVERLAY[PGUARD,<PD_TUB>]         := 0.0;
OVERLAY[<PD_TUB>,PGUARD]         := 0.0;
# outside n-doped tub
SPACING[PGUARD,<ND_TUB>]         := 1.0;
SPACING[<ND_TUB>,PGUARD]         := 1.0;

# transistor diffusion
WEIGHT[<TR_DIFF>]                := 10;
WIDTH[<TR_DIFF>]                 := 3.0;

# diffusion (without transistor)
WEIGHT[NDIFF]                    := 8;
WEIGHT[PDIFF]                    := 8;
WIDTH[NDIFF]                     := 2.0;
WIDTH[PDIFF]                     := 2.0;
```

Fig.5.3 Fragment of a technology description, illustrating the use of design layers, and the specification of the (layer dependent) compaction weights.

The separation of the technology data from the generic topological objects is essential in the TILT system. It is even impossible to know which technology is actually running inside a generator (although we could define a variable *name* which is instantiated by the technology). This property helps to describe technologies in terms of common characteristics (such as one-or two-metal, single or twin well etc.), instead of technology specific items.

An advantage of this strict separation is the possibility to modify technology data without recompiling the generators. Also, new technologies may be added after the design of a generator has been finished. If a new technology is very similar to an already existing one, the generator is very likely to produce correct layout without modification.

Generators may access technology data only through a generic environment interface. Such an interface defines the type of the data which will be instantiated during run time. An example of this encapsulation is given in the next paragraph.

The size of a technology object is mainly determined by the number of design rules. The definition of these rules is largely simplified using the type *set*, which allows to group layers in a single identifier. This is illustrated by the fragment of the cmn20 process definition describing some of the rules for the diffusion layers.

As a result, technology objects remain clear and small. The processes which are defined in the CMOS and the bipolar library contain 150-200 lines of code, including comment.

Environment

The principal role of the environment objects is to define a set of symbols to ease and clarify the programming effort. All symbols which should become visible outside the environment must be declared in the *EXPORT()* command.

For instance, the object *LAYERS* defines an identifier for each layer. These identifiers are known if the name "*LAYERS*" is specified in the command *ENVIRONMENT()*.

Another example concerns the environment *DEF_RESS*. This object defines constants to ease the use of the library objects, such as *left*, *right*, and *MIN_XWIRE*, *SAME_LEFT* etc. Many of these identifiers are used as resources to the library functions.

```
TYPE("environment");
ENVIRONMENT("LAYERS");
TECHNO();

SCALE           : float;    EXPORT(SCALE);
WIDTH[]         : float;    EXPORT(WIDTH);
SPACING[]       : float;    EXPORT(SPACING);
EQUIPOT[]       : float;    EXPORT(EQUIPOT);
EXTENSION[]     : float;    EXPORT(EXTENSION);
OVERLAY[]       : float;    EXPORT(OVERLAY);
MAX_WIDTH[]     : float;    EXPORT(MAX_WIDTH);
WEIGHT[]        : integer;  EXPORT(WEIGHT);
```

Fig.5.4 Header of the environment RULES which defines the technology interface to the design rules.

An environment object gets a special semantics if the command *TECHNO()* is defined in its header. Now, the exported identifiers will be instantiated with the values provided by the technology. It is the only way to access to technology specific data.

The object *RULES* in the library *CMOS* implements this interface to the technology data. The header of this object is given above; its body is empty, but could have been defined to contain generic rules to be used if no technology was given at all.

Geometry

The role of the geometry objects is to provide a convenient interface to the basic building blocks of CMOS circuitry. All of these objects are highly user configurable through their resources. As a result, none of the applications given later on in this chapter access the low-level commands.

All geometry objects are implemented as function type objects, which are generally overloaded. This allows the user to provide a minimum amount of resources, using defaulting whenever possible.

For example, all types of metal-to-layer contacts can be created with the object *ADDCONT*. The optional resource *res* allows to specify the contact properties in detail, if omitted, a minimum fixed contact is created. An ordinary user only needs to consult the header (available as a manual page) to understand its functionality. The return value of the function type object is a record with 5 fields: *contact*, *metal*, *poly*, *smetal*, and *diffusion*. Some of these fields will be empty, dependent on the generated contact type.

Different implementations are possible to realize the object *ADDCONT*. Three extreme cases are presented here: a *high level al2*, a *low level al2*, and a *C*-implementation. They differ in design complexity and run time efficiency and should be chosen with care. The complete source code of these three examples is given in Annex C.

different implementations of the ADDCONT object				
implementation	**#al2 instr**	**#C lines**	**size[1]**	**CPU [s][2]**
high level al2	29	224	7088	40.6
low level al2	38	331	12804	18.3
C	-	149	5392	14.8

The high level al2 description provides a very concise and compact solution. It uses the CMOS library objects *ADDRECT* and *EXTEND*, and it does not access any of the technology data. It yields a very readable source code and takes a minimum time to write.

Another possibility is to access the lowest level al2 functions *RECT()* and *EXTENSION()* directly. This implementation benefits from the fact that inside the *ADDCONT* object, the elements to be created are exactly known. Now, the technology data (*WIDTH* and *EXTENSION*) is accessed directly and passed to the al2 built-in functions. However, the development time of this flattened approach is longer and tedious to read.

The last method is to write the object in C. From the execution-time point of view, this solution is the best. As the source code is not generated automatically, clever optimizations are possible. On the contrary, the development time of such a solution is much longer. It obliges the programmer to know the internal interface to the TILT library, and he needs to know C as well. This (tedious) solution should only be applied when really necessary: either in the case where run time is critical, or in the case where al2

[1] of the object in bytes on a SUN4/50
[2] creation time of a table of 1000 contacts without compaction on a SUN4/50 with 32Mb of central memory

does not offer the desired functionality. In the former case, the C equivalent of a low level al2 description might serve as a starting point for an optimized C version.

A comparison of the three solutions is given in the table above. It appears that many lines are required to obtain a C-implementation of the (simple) ADDCONT object. This is primary due to the bad correspondence between the functionality described in al2 and the semantics of C. A C++ based version of the TILT system would reduce this amount considerably and yield better readable code.

More elaborated geometry objects in the CMOS and bipolar library concern complete transistors of different topologies. Again, these objects are highly configurable through their resources. Examples of transistors for analog applications are presented later on in this chapter.

Topology

The main interest of the topology objects is to provide a high level interface to the most common positioning of geometrical objects. At this level, the technology details are completely hidden; there is no direct access to any of the technology parameters.

The role of these objects is essential: they enable to capture the spatial knowledge in the design. It is somewhat surprising that 6 objects (*ABUT*, *ALIGN*, *ATTACH*, *CONNECT*, *EXTEND*, and *PLACE*) are sufficient to achieve this.

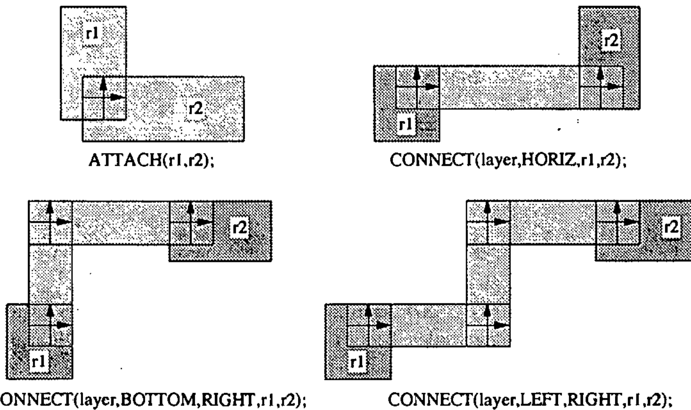


Fig.5.5 Illustration of the functionality of ATTACH and CONNECT.

ABUT is an object that places the cell's abutment box. Abutment of cells is especially useful in hierarchical applications, as they allow for an efficient

placement of the cells. A concrete example of abutment is given with the presentation of the RAM generator.

Two identical sides (left, right, bottom, or top) are aligned with the object *ALIGN*, which inserts a fixed constraint between its arguments. It may be used to obtain a precise configuration at the cost of a reduced cell flexibility.

An *ATTACH* corresponds to an electrical connection between two elements. As such, an attachment can only be made among elements of the same layer. It assures a sufficient overlap between the two geometries to obtain a physical connection, however, without inserting any new element.

The semantics of *CONNECT* is similar to *ATTACH*, except for the fact that it creates new geometries. Depending on the resource values, the two geometries are connected with a single (horizontal or vertical) wire, a horizontal and a vertical wire, or even three wires (two vertical and a horizontal, or vice-versa). It offers a very flexible way to interconnect elements in a cell.

The primary use of *EXTEND* is to create contacts, and transistors. It inserts a constraint between the same sides of two geometries (two left sides, two bottom sides etc.). The object can also be used to cover a geometry explicitly with another, for instance to obtain a uniform well if multiple transistors are used.

One of the most frequently used objects is *PLACE*. It positions a geometry beside another in the horizontal or vertical direction. 16 different calls (overloaded) are recognized, which may be classified in 4 principal groups.

All objects require a resource to be passed, indicating how the placement is to be done. This resource contains 4 fields:

- *direction*, a set which contains the relative position of the first geometry relative to the second.
- *value*, a float number which contains the amount of space to be respected. If no value is specified, the technology variable *SPACING* is used to determine the clearance.
- *cross*, a boolean flag which determines whether the placement is to be performed from each member of the first geometry to each member of the second geometry, or only between members with the same layers. By default, cross is assumed to be *TRUE*.
- *fix*, also a boolean that indicates whether the constraint is a minimum placement (default), or a fixed placement.

Various common resource values are predefined in the environment object *DEF_RESS*. For example *LEFT*, *RIGHT*, *BOTTOM*, and *TOP* define the direction field only. The resulting action is a minimum placement between all layers applying the technology values.

```
TYPE("function");
ENVIRONMENT("RULES","DEF_RESS");

FUNCTION(pl0,res,rec1,rec2);
FUNCTION(pl0,res,rec1,tab2);
FUNCTION(pl0,res,tab1,rec2);
FUNCTION(pl0,res,tab1,tab2);

FUNCTION(pl1,res,lay1,rec1,rec2);
FUNCTION(pl1,res,lay1,rec1,tab2);
FUNCTION(pl1,res,lay1,tab1,rec2);
FUNCTION(pl1,res,lay1,tab1,tab2);

FUNCTION(pl2,res,rec1,lay2,rec2);
FUNCTION(pl2,res,rec1,lay2,tab2);
FUNCTION(pl2,res,tab1,lay2,rec2);
FUNCTION(pl2,res,tab1,lay2,tab2);

FUNCTION(pl3,res,lay1,rec1,lay2,rec2);
FUNCTION(pl3,res,lay1,rec1,lay2,tab2);
FUNCTION(pl3,res,lay1,tab1,lay2,rec2);
FUNCTION(pl3,res,lay1,tab1,lay2,tab2);

pl0            : record;
pl1            : record;
pl2            : record;
pl3            : record;

rec1           : record;
rec2           : record;
tab1[]         : record;
tab2[]         : record;

res.direction  : set;
res.value      : float;    #default  SPACING[layer1,layer2]
res.cross      : boolean;  #default  TRUE (cross)
res.fix        : boolean;  #default  FALSE (stretch)

lay1           : integer;
lay2           : integer;
```

Fig.5.6 Header of the topology object PLACE.

```
rect.name   : string;
rect.layer  : integer;
r1 = SEARCH(rec1,rect,"well","BOX");
t1 = SEARCH(tab1,rect,"well","BOX");
r2 = SEARCH(rec2,rect,"well","BOX");
t2 = SEARCH(tab2,rect,"well","BOX");
g1 = SCAN(r1,t1);
g2 = SCAN(r2,t2);

spa0 = integer|
   (GETFIRST(res.value,SPACING[g1.layer,g2.layer]).*SCALE);
sty0 = (NOT cross AND (g1.layer == g2.layer)) OR cross;
IF (sty0)
   pl0 -> SPACING(g1.name,g2.name,dir,bind,spa0);
```

Fig.5.7 Fragment of the PLACE body illustrating the use of pattern searching. Note the compact notation, combining 4 cases.

Moreover, all of the function calls contain the parameters *rec1* or *tab1* and *rec2* or *tab2*. These arguments are of the type *record* or *table of records*, and contain the geometries to be placed. As constraints can only be inserted between rectangles, the routine *PLACE* scans the record argument in a search

for the typical rectangle pattern. This scanning is performed through the tables, as well as through the records. It is this LISP-like feature which allows to add new geometrical structures (= new record structures), without modifying the *PLACE* object.

The pattern search may also be completed with an ignorance frame. This is for example used to split the wells from a transistor in order to ignore well placement if they are of the same type.

The parameters *layer1* and *layer2* are used to select a specific layer. It allows, for instance, to place an element with respect to a CMOS transistor's gate only. The inverse effect is obtained if the layer is preceded by a minus sign.

Summary

The CMOS and bipolar library provide a set of basic interface objects to a technology class. These objects hide all of the low-level geometries and constraints and offer an abstraction level in terms of the global topology and elements such as wires, contacts and transistors. In fact, this is the essential part of the layout design; it corresponds to the topological knowledge described by the user.

The pattern search facilities, offered by the language al2, play an important role to achieve this behavior. Even if a table of geometries is passed as an argument, the individual rectangles may be recovered, and the context may be preserved. This encapsulation explains the high reusability of these objects and limits their number to the strict minimum.

5.3 Standard cells

An early version of TILT was used in an industrial environment for the description of low-power standard cells [MAS91]. At that moment, the language did not yet support all of the features it does support today, and the compactor used a heuristic slack distribution algorithm instead of the weighted optimum solution which is provided now.

Without a high-level library, the size of the descriptions became large and difficult to manage. Still, the density of the cells was near to hand-crafted results under equal conditions (i.e. no 45 degrees). Some of the cells have been reformulated in the al2 to test whether the current version of tool would be able to better.

Floor planning

In the world of standard cells, a common approach is to place the transistors in 2 lines, one which contains the N-transistors, the other the P-

transistors [UEH81]. Whereas a pure graphical layout system is often constructed on top of a cut and paste mechanism, a procedural approach allows to parameterize and to reuse such a floor plan.

For example, consider a 3-input AND gate:

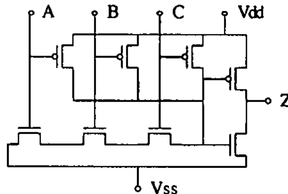


Fig.5.8 Schematics of a 3-input AND gate.

Using a special function to create this floor plan, only 10 instructions are required to create the basic topology. This function defines the position of the abutment box, the two transistor regions, and the power buses. The transistor sizes, types and positions are programmed by means of a resource. If desired, the frame may also define a pitch on its port positions. Many commercial systems impose the standard cell ports to be on a grid, in order to be able to guarantee correct routing.

The resource programming is simplified with the convenience function *SETFRAME()*. Dependent on the type and number of its parameters, it constructs a record containing the transistor type, a flag indicating whether it should be aligned to the abutment box, and the fanout of the transistors. Eventual poly-type output wires are declared by setting the type to *WIRE*.

```
nframe[1] := SETFRAME(TRUE,DRAIN_LEFT);
nframe[2] := SETFRAME(TRUE,CHANNEL);
nframe[3] := SETFRAME(TRUE,VSOURCE_RIGHT);
nframe[4] := SETFRAME(FALSE,VSOURCE_DRAIN,fanout);
nframe[5] := SETFRAME(TRUE,WIRE);

pframe[1] := SETFRAME(TRUE,DRAIN_VSOURCE);
pframe[2] := SETFRAME(TRUE,VSOURCE_DRAIN);
pframe[3] := SETFRAME(TRUE,DRAIN_VSOURCE);
pframe[4] := SETFRAME(FALSE,VSOURCE_DRAIN,fanout);
pframe[5] := SETFRAME(TRUE,WIRE);

t := ADDFRAME(nframe,pframe,5,5);
```

Fig.5.9 Programming the standard cell frame in the body of a 3-input AND gate.

Once the frame resource is programmed, it is passed to the function object *ADDFRAME()*. If desired, the width of the power buses may be specified as well, by default, twice the minimum metal width is assumed.

Internal interconnections

After declaring the frame, the basic transistor interconnections are made.

Again, specialized functions have been defined to ease this task. They require the frame and the indices of the n- and p-transistors to be connected. The interconnections are correctly placed between the transistor rows, while respecting the design rules. A similar approach is applied to the drain interconnections, and the poly-to-metal contacts.

```
p1 := GATECONNECT(t,1,1);
p2 := GATECONNECT(t,2,2);
p3 := GATECONNECT(t,3,3);
p4 := GATECONNECT(t,4,4);
p5 := GATECONNECT(t,5,5);

m1 := DRAINCONNECT(t,1,1);
m2 := DRAINCONNECT(t,1,2);
m3 := DRAINCONNECT(t,1,3);
m4 := DRAINCONNECT(t,4,4);

c1 := FRAMECONT(t,p3,NONE,p4,m3);
c2 := FRAMECONT(t,p4,NONE,p5,m4);

PLACE(LEFT,c1,c2);
PLACE(BOTTOM,p1.wire.horiz,p2.wire.horiz);
```

Fig.5.10 Definition of the connectivity in the body of the 3-input AND gate.

After the declaration of all the basic interconnections, some additional topological constraints may be needed to respect all the design rules. This phase can be considered as the *trimming* of the design.

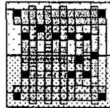


Fig.5.11 One-metal implementation of a 3-input AND gate (technology: sacmos2u, abutment box: 22.0μmx26.0μm, scale 1)

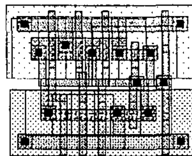


Fig.5.12 One-metal implementation of a 3-input AND gate (technology: cmn20a, abutment box: 43.5μmx38.0μm, scale 1)

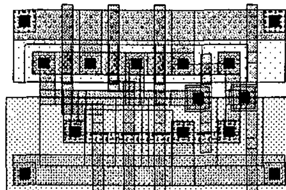


Fig.5.13 One-metal implementation of a 3-input AND gate (technology: mietec2u, abutment box: 66.0μmx49.0μm, scale 1)

Three examples of this cell are given in figures 5.11 to 5.13. Each cell is drawn at the same scale. The abutment box passes through the bulk contacts at the left and right sides and fits the polysilicon ports at the top and at the bottom.

External interconnections

Of course, this method works well with simple circuitry, as illustrated with the 3-input AND-gate. Only 25 instructions are sufficient to describe a technology independent layout with a hand-crafted equivalent density. However, in practice, somewhat more complicated gates will get stucked because of surrounding interconnections which may not be crossed.

Instead of growing the cell size, a better solution is to charge the standard cell router with this task. Now, the elements which cannot be routed inside the cell, are made available as ports, and the router is informed that these ports are to be interconnected. Note, that this solution allows to keep the average height of the cells low. The routing channel will not suffer very much from these interconnections as they are very local.

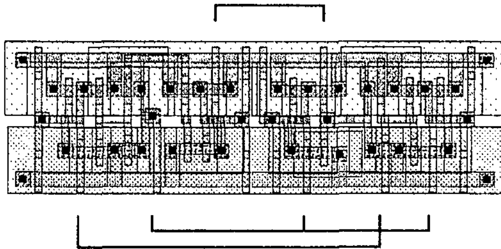


Fig.5.14 External interconnections can be submitted to the router instead of increasing the cell size.(example: 2-to-4 line decoder, technology: cmn20a, abutment box: 121.5μmx38.0μm, scale 1)

Using the second metal

One of the first questions which arises when speaking about technology independent layout is the behavior with respect to a second metal. If a generator wants to be independent on the technology, it must be able to handle this feature.

The two-metal standard cell floor plan differs from the single metal one by the fact that all ports are available in the second metal, instead of in poly. This property causes the topologies to be basically different when comparing single-metal with double-metal implementations. Instead of treating the technologies with one or two metals in a single description, the maintenance of a generator becomes much easier if two separate descriptions are used.

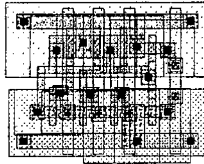


Fig.5.15 two-metal implementation of a 3-input AND gate (technology: cmn20a, abutment box: 43.5μmx38.0μm, scale 1)

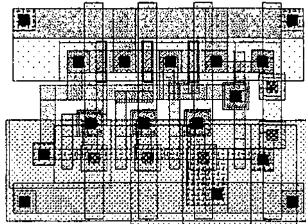


Fig.5.16 two-metal implementation of a 3-input AND gate (technology: mietec2u, abutment box: 70.0μmx49.0μm, scale 1)

The complexity of a second metal description is similar to a single metal version. In fact, the second metal is primarily used for the inputs and outputs of the cells and, again, special functions are defined to achieve this. As a comparison, the complete code of a two-metal implementation of the 3-input AND gate is given here.

```
nframe[1]  := SETFRAME(DRAIN_LEFT,1,GATE_LEFT);
nframe[2]  := SETFRAME(CHANNEL,1,GATE_LEFT);
nframe[3]  := SETFRAME(ESOURCE_RIGHT,1,GATE_LEFT);
nframe[4]  := SETFRAME(ESOURCE_DRAIN,fanout);

pframe[1]  := SETFRAME(DRAIN_VSOURCE);
pframe[2]  := SETFRAME(VSOURCE_DRAIN);
pframe[3]  := SETFRAME(DRAIN_VSOURCE);
pframe[4]  := SETFRAME(VSOURCE_DRAIN_NC,fanout);

t := ADDFRAME(nframe,pframe,4,4);

p1 := GATECONNECT(t,1,1);
p2 := GATECONNECT(t,2,2);
p3 := GATECONNECT(t,3,3);
p4 := GATECONNECT(t,4,4);

m1 := DRAINCONNECT(t,1,1);
m2 := DRAINCONNECT(t,1,2);
m3 := DRAINCONNECT(t,1,3);
m4 := SDRAINCONNECT(t,4,4);

c1 := FRAMECONT(t,t.port[3].poly,m4,p4,m3);

PLACE(BOTTOM,p1.wire.horiz,p2.wire.horiz);
PLACE(BOTTOM,t.port,p3.wire.horiz);
PLACE(BOTTOM,t.port,m1.wire.horiz);
PLACE(BOTTOM,t.port,m2.wire.horiz);
PLACE(BOTTOM,t.port,m3.wire.horiz);
```

Fig.5.17 Body of the two-metal implementation of a 3-input AND gate.

The use of a second metal will not automatically lead to smaller cells: this will depend on the detailed design rules of the technology. On the contrary, cells with a small number of gates may become even somewhat larger than the equivalent single-metal ones, due to the additional vias.

Variable fanouts

Now that the standard cells are generated, it becomes possible to create the cells such that the fanout capacity of the output transistors fits exactly with the number of gates it drives. On the other hand, it may be interesting to maximize the fanout, while not increasing the minimum cells size, for instance for elements on the critical path. Note as well that the fanout can be coupled to the technology, thus providing a technology dependent optimization.

If the generators execute sufficiently fast, an approach of *supply and demand* can be reasonable: a layout is constructed whenever needed, but the cells as such are not stored in a library. Current standard cell approaches do not support this customization because it would require many alternative circuits to be stored in a library. Instead, a few variants are provided, or specific driver circuitry is to be added separately.

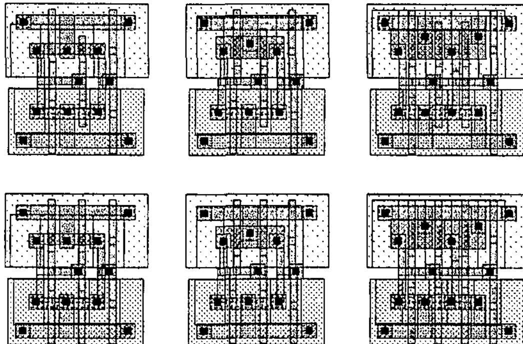


Fig.5.18 *Layouts of a buffer generated with a fanout of 0 (minimum transistors), 1 (symmetrical transistors), and 4 (parallel transistors). The top range has no intermediate output, contrary to the bottom range (technology: cmn20, scale 1).*

Now, it becomes feasible to provide basic gates with integrated output drivers, thus reducing the external routing. Furthermore, this approach helps to increase the design transparency because groups of cells with the same functionality may be put together, forming hence the beginning of an object oriented approach to standard cells. In this way, libraries with 400-600 different cells would become more manageable.

As an example, a non-inverting buffer generator is given. A fanout of 0 means that all transistors are minimum. Higher fanout values cause transistor sizing and possibly the creation of additional parallel branches. Figure 5.18 illustrates the layouts for 3 classes: minimum transistors, a fanout of 1, and a fanout of 4, each with and without an intermediate output. All of these layouts are produced with the same generator.

It should be mentioned that this kind of parametrization only makes sense if the corresponding models may support this feature. If so, it provides an interesting solution to reduce the storage size and to increase the flexibility of the created cells.

Further parametrization

So far, the parametrization of the cells was limited to the fanout and the width of the power lines. Of course, it is possible to introduce more parameters such as a topology choice or desired position of the terminals.

In particular, it is possible to parameterize the type of the gate in a single description. For instance, it is possible to combine a 2-input, a 3-input and a 4-input NAND gate. Adding a second parameter would allow to select for an AND-gate, and so on.

Power consumption, or delays may be supplementary options to be integrated into a description. A low-power cell might minimize the gate sizes, whereas a high-speed cell would increase the transistor sizes in order to reduce the delays.

Still, the pros and cons should be weighted. Concentrating the definitions of different gate types in a single description may complicate the design phase considerably. Here, a trade-off between flexibility and complexity has to be made. Even if different descriptions are used, much of the source code may be reusable.

Furthermore, different topologies for the same cell may be defined. Especially, if a wide range of technologies is chosen, it is possible that the smallest solution differs from one technology to another. This means that the best choice is technology dependent; the only way to obtain the smallest result is to try all of the alternatives, and to choose the best. In this manner, technology independent design will not suffer from worst-case decisions, and may compete with technology specific approaches.

```
topo1 := example(par1);
topo2 := example(par2);
topo3 := another_example();
best := MINSELECT("width",topo1,topo2,topo3);
instance := CELL(best);
```

Fig.5.19 Example illustrating how the separation of the cell activation and the cell instantiation allows to select properties at run time.

Al2 allows to choose among different classes because activation and instantiation of calls are separated. The activation of a cell causes it to be solved, and a cell model is created. At this stage, the minimum size and positions of the ports are known, but a cell activation as such cannot be drawn, it is only a template. The instantiation takes a physical copy of the cell model, which can be drawn on the screen since it will have a position associated with it.

The feature described above allows to evaluate different possibilities, and to choose the best one to be instantiated. It is a transparent and powerful way to select optimum solutions.

Performances

The standard cell descriptions are well adapted to measure the non-hierarchical efficiency of the compaction algorithms. All of the cells are implemented in two levels: the frame at the lowest level, and the interconnections at the top level.

The sizes of the circuits for 3 different 2μ technologies are summarized in the table below. Whenever possible, both the data for the single- and the double-metal implementation is given. The data is compared with the hand-crafted layout of the CSEL_LIB library, which uses 45 degrees, and the same kind of cell floor-plan. However, CSEL_LIB imposes an 8 micron grid on the locations of the terminals to assure compatibility with a grid-based router. This condition has not been imposed on the cells generated with the TILT generators, in order to determine the minimum possible size if no such constraints need to be imposed.

typical TILT cell dimensions in micron[1]							
			one metal			*two metals*	
cell name	**#gates**	**#instr**[2]	**cmn20a**	**sacmos2u**	**mietec2u**	**cmn20a**	**mietec2u**
INV	2	10/8	19.5x38.0	10.0x26.0	29.0x49.0	20.0x38.0	26.5x55.0
BUFFER	4	46[3]/13	27.5x38.0	16.0x26.0	41.0x49.0	27.5x38.0	42.0x55.0
NAND2 / NOR2	4	15/15	25.5x38.0	14.0x26.0	41.0x49.0	24.0x38.0	39.0x55.0
NAND3 / NOR3	6	19/20	32.5x38.0	18.0x26.0	53.0x49.0	31.5x38.0	51.5x55.0
AND2 / OR2	6	20/18	32.5x38.0	20.0x26.0	53.0x49.0	36.5x38.0	56.0x55.0
AND3 / OR3	8	25/24	43.5x38.0	22.0x26.0	66.0x49.0	43.5x38.0	70.0x55.0
DEC2TO4	20	59/63	113.5x38.0	64.0x26.0	182.0x49.0	113.5x38.0	168.0x55.0
DFLIPFLOP	22	79/94	96.5x38.0	60.0x26.0	143.0x49.0	110.5x38.0	169.5x55.0

Note in particular how similar the cell areas are with respect to the hand-crafted CSEL_LIB reference. It illustrates well that a procedural approach is able to compete with it.

[1] each cell has been generated using the same topology for each technology. The power buses equal twice the minimum metal width.

[2] one metal implementation/two metal implementation

[3] the instruction count is somewhat larger because the fanouts are parameterized over a large range (see fig.5.16)

typical CSEL_LIB cell dimensions in micron[1]			
		one metal	*two metals*
cell name	**#gates**	**sacmos2u**	**cmn20a**
INV	2	16.0x36.0	16.0x35.0
BUFFER	4	24.0x36.0	24.0x35.0
NAND2 / NOR2	4	24.0x36.0	24.0x35.0
NAND3 / NOR3	6	32.0x36.0	32.0x35.0
AND2 / OR2	6	32.0x36.0	32.0x35.0
AND3 / OR3	8	40.0x36.0	40.0x35.0
DEC2TO4	20	60.0x36.0	88.0x35.0
DFLIPFLOP	22	56.0x36.0	88.0x35.0

The table below shows that the execution of the al2 description takes much more time than the compaction phase (which includes both the graph construction and result generation time). This property can be explained by the high vertex reduction factor, as well as by the extensive use of high-level al2 functions. It appears to be a property common to most descriptions: *the compaction is faster than the constraint creation.* Of course, someone might say that this means that the creation phase is non-optimal, but it should not be forgotten that it is the al2 code which takes care of the technology independence.

The size of the executables depends on the complexity of the cell, and on the type of the binding (see Annex A). Static links result in executables of approximately 800Kb, dynamic links require only 50Kb per generator[2].

typical run time figures (in seconds) for a one- and two-metal implementation in the cmn20 technology[3]										
cell name	**execution**		**compaction**		**#rectangles**		**#constraints**		**reduction**	
INV	4.7	4.1	0.3	0.4	82	85	1034	994	0.18	0.16
BUFFER	9.9	8.0	0.8	0.7	134	140	2199	1943	0.16	0.15
NAND2 / NOR2	7.8	8.0	0.7	0.8	113	143	1752	1981	0.18	0.15
NAND3 / NOR3	12.0	12.9	0.8	1.1	144	192	2651	3214	0.18	0.15
AND2 / OR2	14.3	12.7	1.0	1.0	164	176	3161	3120	0.16	0.15
AND3 / OR3	19.7	19.3	1.2	1.4	195	238	4310	5129	0.16	0.14
DEC2TO4	96.2	73.2	7.3	5.6	485	478	22299	18698	0.15	0.13
DFLIPFLOP	75.2	76.5	5.8	6.7	439	496	19032	19866	0.12	0.13

The typical design effort to describe a cell in al2 depends very much on the complexity of the cell, the experience of the designer, and the computational power of the platform. In general it will vary between 1 hour and 2 days, including the visual inspection of 2 or 3 different technologies. A final design rule check should be added for each cell variant in each technology.

[1] the table contains the values for the low power cells.

[2] object sizes obtained with an ANSI C compiler under SunOS Release 4.1.

[3] mean values of single and double metal implementations on a SUN4/50 IPX with 32 MB of memory.

Summary

In conclusion, we may state that a TILT based standard cell description can compete with hand-crafted layout, with a similar development time. This comparison does not take into account the time to establish a set of convenience functions. However, the procedural approach is parameterized and technology independent and source code may be reused to implement resembling layout. The bottleneck in the design cycle is not formed by the compaction, but by the compilation phase and the al2 execution. An interpreted version of al2 would be a valuable solution here.

5.4 A static RAM generator

The hierarchical properties of the system may be well illustrated on the example of a memory generator. It allows to measure some of the performances and limitations of the actual implementation. Only the topological aspects are treated here. The electrical analysis and the sizing of the transistors is assumed to be performed in an earlier stage.

A memory created during the CERS 2168.1[1] project [WAT92] has served as a reference. This allowed us to concentrate on the topological aspects and parametrization. This low-power memory was created to be used in conjunction with a watch processor. All the leaf cells were designed by hand, and assembled on the Compass CAD system.

First, we will see how the memory matrix is created using a 6 transistor static memory bit. Two approaches are implemented: one which describes the complete cell in al2, another which defines a blackbox that is replaced by an optimized cell afterwards. Next, the line decoder is presented in somewhat more detail. It illustrates the pitch-matching facility and top-down cell programming. Finally, some typical figures of the complete application as well as comparisons are given.

The memory matrix

The core of a memory is formed by its memory matrix; a two dimensional array of memory bit cells. The final size of the memory is primarily determined by the density of this matrix, especially if it concerns large memories. Angular geometries are often used in this context, and design rules are sometimes violated.

In this example, a static read/write memory cell with 6 transistors has been implemented; one for a single-metal technology, and one for a double-metal technology. It consists of a latch, whose state may be read or set through two

[1] CERS = Federal Commission for the Encouragement of Scientific Research

transmission gates. In this configuration, the sizing of the transistors is critical and influences both the speed and power consumption.

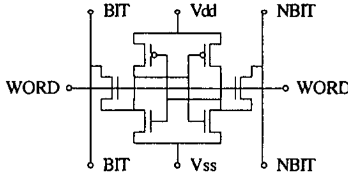


Fig.5.20 Schematics of a 6-transistor static read/write memory cell.

Then, the memory cell is repeated in the horizontal and vertical direction. It is important to foresee how the resources may be shared amongst neighbor cells in order to obtain the highest possible density.

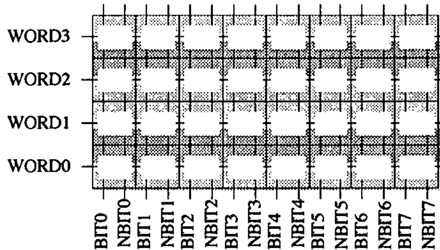


Fig.5.21 Floor plan of a memory matrix where the control lines traverse the cells horizontally, and the data lines vertically. Subsequent rows are flipped so that they may share power supply resources.

The al2 description of the memory matrix is largely simplified with the use of an abutment box. Contrary to the bounding box (the smallest rectangular region which envelops the whole cell), the abutment box is placed explicitly by the user. It serves for design purposes only., and indicates a rectangular region which can be used to abut neighbor cells at a higher hierarchical level without violating design rules. For instance, the bit lines in the memory cell become adjacent to the bit lines in a neighbor cell when we create a row of memory cells. The abutment box in the memory cell is placed such that the minimum design rule is respected when two memory cells are abutted. In a similar way, resources may be shared if they are centered on a side of the abutment box.

Using the abutment box principle, the al2 code to create a row of memory cells becomes very dense. After the instantiation of the cells, their abutment boxes are simply aligned. Then, the ports are interconnected, new ports are defined, and the abutment box of the row of memory cells is placed. The same mechanism is applied to create an array of rows, thus achieving the complete matrix.

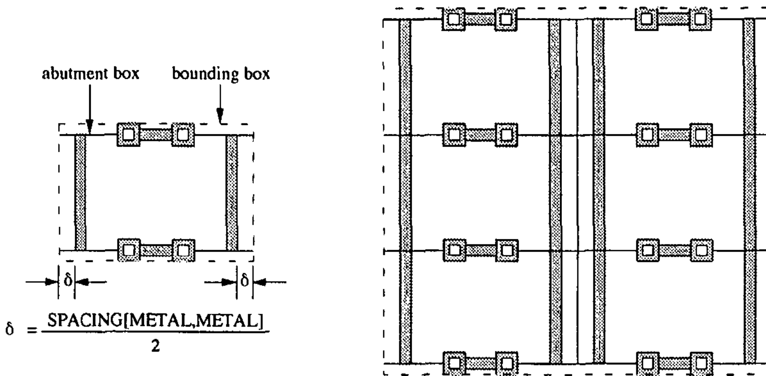


Fig.5.22 Illustration of the use of the abutment box. Appropriate spacing is guaranteed such that this box can be aligned with its neighbors.

Nevertheless, the obtained density will remain below to the one which may be reached with cells using non-rectangular structures. Indeed, the tool should be able to reuse existing layout, paving the way for a more general layout assembler. Economically speaking, such a feature is crucial; nobody would use a new tool if he would have to enter every single layout from scratch.

```
memcall := mem6(memdim);              # call memory cell

n := RANGE(0,memdim.nb_col);          i = <n>;
mem[i] := CELL(memcall,0);            # activate memory cells

# place the cells relative to their abutment boxes
PLACE(FIX_LEFT,mem[i].BOX[BUTTING],mem[NEXT(i)].BOX[BUTTING]);

# declare ports
WORD := ADDRECT(MIN_XWIRE,POLY);
VDDL[i] := mem[i].VDDL;     VDDR[i] := mem[i].VDDR;
VSSL[i] := mem[i].VSSL;     VSSR[i] := mem[i].VSSR;
BIT[i]  := mem[i].BIT;      BIT_[i] := mem[i].BIT_;

# interconnect word lines and power supplies
ATTACH(WORD,mem[i].WORD);
vss[i] := CONNECT(NDIFF,HORIZ,mem[i].VSSR,mem[NEXT(i)].VSSL);
vdd[i] := CONNECT(PDIFF,HORIZ,mem[i].VDDR,mem[NEXT(i)].VDDL);

# place abutment box of the current cell
ALIGN(LEFT,mem[0].BOX[BUTTING],$[BUTTING]);
ALIGN(RIGHT,mem[memdim.nb_col-1].BOX[BUTTING],$[BUTTING]);
ALIGN(VERT,mem[i].BOX[BUTTING],$[BUTTING]);
ALIGN(VERT,mem[i].BOX[BUTTING],$[BUTTING]);
```

Fig.5.23 Assembling a row of cells through abutment.

However, the concept of assembling technology specific leaf cells is somewhat contradictory with repsect to the philosophy of a technology independent layout tool. The technology dependency is complete, and the generic character of the al2 descriptions is lost. Therefore, it is recommended to use this option exceptionally and to provide an al2 alternative in any case.

Instead, a transparent seed cell mechanism, compatible with a data-base extension, is proposed. The idea is that the design time cycle may be shortened if intermediate layouts may be preserved. When executing a generator, it first checks whether a layout for the specific cell call with the appropriate parameters and technology exists. If so, the underlying cell description is not executed, but the layout from the data-base is returned.

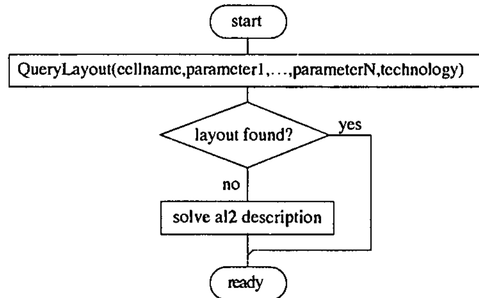


Fig.5.24 Flow diagram of a link between a layout data-base, and an al2 description.

Now, existing layouts will be used if they are installed in this data-base. Only a translation of the original layout format (c.f. *CIF*), to the internal TILT format (the *image*) is required. New, external layouts may be added without any modification of the generator itself; the data-base version will prime at run time.

An alternative is to describe a black-box of the layout we want to use. This box consists of the definition and the placement of the cell ports only. It is a general applicable way to reuse external leaf cells. The final layout is obtained by replacing the black-box in the geometrical output, with the actual leaf cell layout.

The black-box approach is not as flexible as the use of a data-base. The former requires the al2 description to foresee the existence of the leaf cell, and the introduction of new leaf cells implies the edition of the al2 source code. However, the layout density becomes equal to the one obtained in the technology specific environment. It is actually the best solution to import critical layout in the current TILT version.

The line decoder

Whereas the memory matrix is extremely regular, the peripherals are less. Still, there is a lot of structure which may be exploited by a procedural approach. As an example, the line decoder is presented here, because it provides good examples of parametrization and hierarchical compaction.

The role of the line decoder is to select one line out of N from the memory matrix. A current approach here is to use AND gates with dynamic logic, where the inputs of the gate are connected to the address to be decoded. It has the advantage of using only n-type transistors for the effective decode operation.

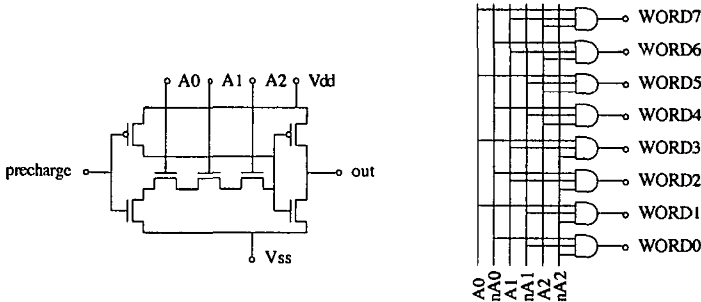


Fig.5.25 Schematics of a dynamic 3-input AND-type decoder, and the logic diagram of the complete line decoder.

The layout topology of the gate follows almost directly from the schematics. An address bus traversing the decoder cells, while connecting the appropriate gates is a good solution. However, each decoder will be connected to the address bus in a unique way. Instead of describing a new cell for each address, the connection may be programmed in al2.

The programming of the bus inside the line decoder leaf cell has the advantage that the bus connections are treated at the lowest level. The address to be decoded is given as a parameter to the cell, and the address bus is exported as a port. However, this solution is inefficient from the computational point of view. As each cell call has a different parameter value, a new cell will be created for each address to be decoded. Now, more than half of the total memory run time will be required to create the decoder.

A better alternative is to place the address bus at the same level as the instantiation of the decoder cells. Now, only *one* line decoder call is to be made, and each gate is connected with the appropriate bus line. The decoder graph model is used to position the ports, according to which the internal structures of the decoder cells are matched. Using the compactor to solve an hierarchical alignment problem in this way provides dense layout and allows for local optimizations.

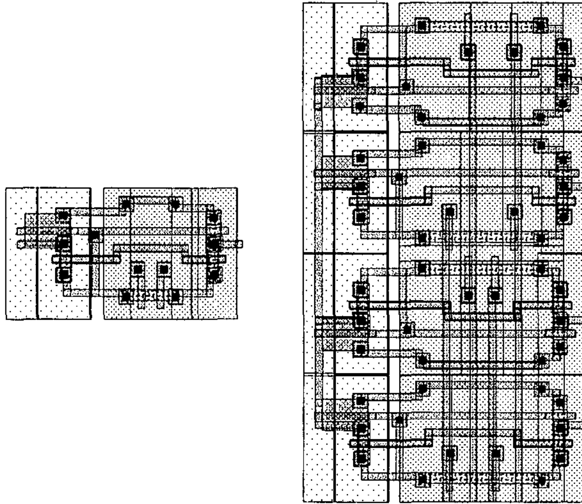


Fig.5.26 Illustration of the pitch matching of a single line decoder on the left, and the same line decoder after programming at the upper level on the right (technology: cmn20, scale 1).

This flexi-cell mechanism is a very powerful one. First of all, it discharges the designer from having to foresee the absolute position of all the ports in a design or to use a router to achieve this. Second, it eases the parametrization and the reusability of the individual cells. As their port positions will be matched automatically, this parameterization may concentrate on transistor sizes etc.

The complete generator

A complete memory does not only consist of a core matrix and a line decoder. In practice, a row decoder, precharge amplifiers, read-write circuitry and control logic form an integrate part of the memory [MAR93, WAT92]. Some typical figures of such a complete application are presented in this section.

One of the first decisions during the design of a memory generator is to determine its floor plan. As we are interested to compare the results with the alternatives mentioned in [WAT92], the same floor plan was adopted. It consists of two memory matrix sections, separated by a line decoder, and completed with the bit multiplexer and read/write logic at the bottom. Note that the generator could easily be expanded to allow to control the memory organization in more detail.

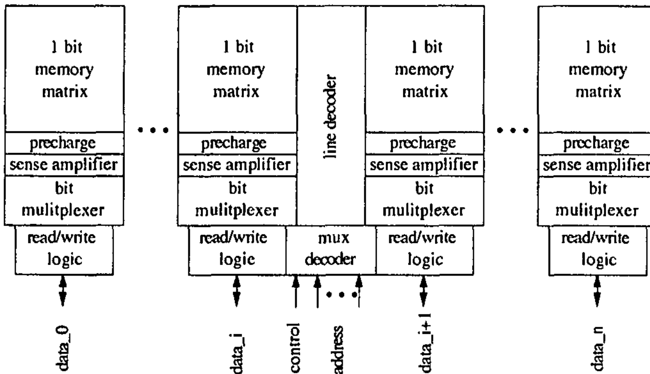


Fig.5.27 Floor plan of the memory generator.

From this floor plan, the following minimum set of generator parameters was extracted (default values in brackets):

- total number of words [64]
- number of bits per word [2]
- preferred form ratio [1]
- use of second metal [FALSE]
- name of black box core cell [""]

The word count, together with the word width, determine the actual number of bits in the memory. The number of columns in the memory will always be a multiple of the word size, which may be different from a power of 2. The number of lines result from the number of bits in the memory divided by the column width.

The form factor is a preference value; it serves as a guideline only, because the number of columns should always be a multiple of the number of bits per word. A value of 1 indicates a square memory, a larger value tends to increase its height and to lower the width, whereas values smaller than 1 widen the memory base, while decreasing the height.

The second metal is not chosen automatically when a technology with two metals is given. Instead, a boolean option is provided which allows to make honest comparisons between technologies using a single metal.

Finally, the name of a blackbox core cell can be given. The dimensions of the core cell used by [WAT92] has been used here.

It can be seen from the table that the TILT generator outperforms slightly the hand-crafted version (≈10% less area), though the latter uses oblique structures. The explanation for this improvement is the different implementation of the leaf cells, and the performance of the compactor. Beside the

technology independence, the generated version has two important advantages over the hand crafted version: it allows to generate the minimum number of required bits (which is not always a power of 2), and the form factor allows for a wide variety of possible sizes. The last remark states that the area can be better adapted to the surrounding circuitry on the chip.

typical memory dimensions. all examples concern 8 bits/word									
configuration		*256 words 32 lines 64 columns*		*128 words 32 lines 32 columns*		*128 words 16 lines 64 columns*		*64 words 16 lines 32 columns*	
technology	[1]	**dimension**	**area**	**dimension**	**area**	**dimension**	**area**	**dimension**	**area**
mietec2u	1	3.895x2.042	7.95	2.039x2.033	4.15	3.877x1.186	4.60	2.021x1.177	2.38
sacmos2u	1	1.644x1.111	1.83	1.296x1.460	1.89	1.636x0.647	1.06	0.868x0.637	0.55
cmn20a	1	2.448x1.461	3.58	0.876x1.101	0.96	2.436x0.845	2.06	1.284x0.844	1.08
mietec2u	2	2.492x2.076	5.17	1.340x2.058	2.76	2.474x1.220	3.02	1.322x1.202	1.59
cmn20a	2	1.776x1.418	2.52	0.960x1.409	1.35	1.764x0.834	1.47	0.948x0.825	0.78
cmn20a with blackbox	2	1.583x1.386	2.19	0.863x1.377	1.19	1.571x0.818	1.29	0.851x0.809	0.69
cmn20a from [WAT92]	2	1.612x1.481	2.39	0.892x1.475	1.32	1.600x0.913	1.46	0.880x0.907	0.80

typical figures for the single and double metal version in the cmn20 technology[2]								
configuration	*256 words 32 lines 64 columns*		*128 words 32 lines 32 columns*		*128 words 16 lines 64 columns*		*64 words 16 lines 32 columns*	
property	**single**	**double**	**single**	**double**	**single**	**double**	**single**	**double**
al2 execution [s]	62.6	51.4	61.7	50.6	47.8	44.4	49.3	45.2
#rectangles	11438	6575	9295	5524	6632	4137	5385	3470
#constraints	147307	107044	125825	93806	85310	64759	71508	55105
total compaction[3] [s]	170.9	116.8	153.1	112.8	67.3	46.0	59.3	42.0
vertex fusion[4]	39%	39%	42%	38%	31%	35%	33%	33%
centering	49%	47%	52%	45%	41%	44%	41%	41%
compaction algorithms	60%	63%	65%	62%	52%	57%	52%	55%
graph construction	15%	14%	14%	13%	18%	18%	17%	17%

The run time figures of the memory depend highly on the organization of the memory. This is primarily due to the different number of stretchable decoder cells as a function of the number of lines in the memory. It is the slack centering which takes about 50% of the total compaction time. The number of constraints is much larger than the minimum required. This is due to the extensive use of high-level positioning objects.

The limitations of this generator concern the amount of memory that is required to compute a solution. In the current version, the compactor keeps all the graphs in memory. Considering the amount of rectangles and constraints, this means that the maximum size is reached rapidly.

[1] number of used interconnection metals
[2] CPU times are measured on a SUN 4/50 IPX station with 32Mb of internal memory
[3] the compaction includes the graph construction, its resolution, and the creation of the output data
[4] the vertex fusion time includes calls made during the graph construction and compaction phase

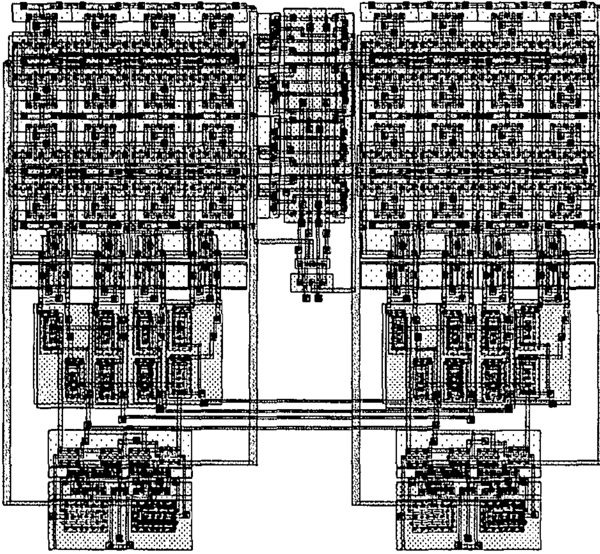


Fig.5.28 Layout generated with the following parameters: 2 bits per word, 16 words, a form factor of 0.5, and one interconnection metal (technology: mietec2u, scale 4).

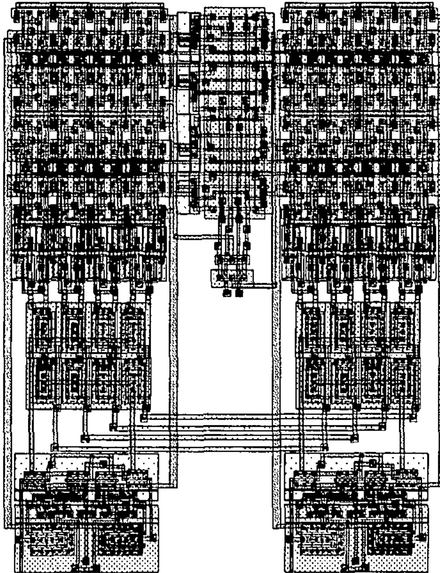


Fig.5.29 As fig 5.28, but with two interconnection metals (scale 4).

In particular the fact that each rectangle is identified with a unique name charges the system considerably. A more sophisticated and efficient name management mechanism would be needed to solve this. Perhaps, larger structures could be generated if the different sections would be generated individually. However, black boxes would be required to allow for correct port matching.

Summary

It has been shown that regular structures may be described in a fairly straightforward manner in al2. The stretchability of the instances allows to reduce the effort on pitch-matching and increases the reusability of the cells. Nevertheless, a non-negligible effort is needed to connect the ports between cells. The role of the abutment box is essential here. It allows to align cells while respecting the design rules automatically.

The major drawback of strongly hierarchical application in the current TILT implementation is the fact that all the intermediate results are recomputed for every run of the generator. This is particularly inefficient during the design cycle. An incrementally interpreted implementation and the adjunction of a data-base could drastically reduce the development time.

The possibility to include existing layout in the TILT environment seems an interesting extension. It would allow to integrate optimized layout in a procedural environment without loosing any previous design effort. Actually, the black-box approach offers a satisfactory solution here. The same density is obtained as in a conventional system, but the language al2 is exploited to complete the design.

5.5 Analog building blocks

Whereas the accent during the design of layout for digital circuitry is usually miniaturization, the priority in the analog domain is more on performance. When designing an analog circuit, one has to be aware of the sensitivity of analog devices to thermal gradients and orientation with respect to axes of the cristal. Placement of the devices should be done with care instead of packing towards highest density. The layout with the highest density does not automatically give the best performing circuit.

Remark that not only the design of active elements, but also the design of passive devices is important. As passive devices are also used in the processing of analog signals, small changes in the layout may have large overall effects.

A common approach towards automated layout generation of analog circuitry is to determine the device orientations and placement with an analog circuit generator [KAY88, RIJ88, MEY93]. This circuit generator creates a set

of constraints which are to be fulfilled by the layout generator. The placement and shape of devices that are not fixed by the circuit generator is left to the layout generator.

All of these approaches use a structure generator to create the layout of the individual devices. In order to ensure flexibility, these generators need to be highly parameterized. They must be able to produce different alternatives for the same set of specifications.

This section presents different types of transistor generators that have been realized in al2. These generators can either be used as a back-end facility in a larger design system or they may be reused to compose a real analog circuit.

CMOS transistor generators

Whereas the transistors used in the digital domain are chosen as to be as small as possible, the transistor sizing in the analog domain is a much more delicate affair. The range of gate widths and lengths is much larger, which allows for many different implementations. The CMOS library contains 5 types of generators, each of which create transistors with a specific shape:

- *MOS*, creates a rectangular transistor with optionally a contact on the drain and source area
- *BIGMOS*, creates a rectangular transistor with optionally a row of contacts on the drain and source area
- *UMOS*, creates a transistor with an U-shaped gate with either the source or the drain in the middle
- *SNAKEMOS*, creates a transistor with its gate has the form of a snake and configurable placement of the source and drain contacts
- *SQUAREMOS*, creates a rectangular transistor with a gate which forms a closed rectangle

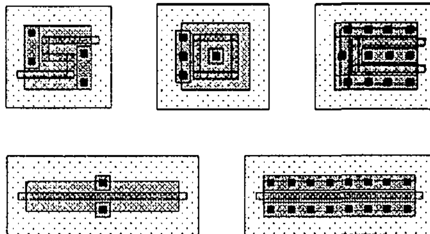


Fig.5.30 Different implementations of a p-type MOS transistor (scale 1).

An illustration of the layouts that can be created with these different generators is given in figure 5.30. Run times vary from 0.5s to 1.8s[1]. All examples implement a transistor with the same gate width and length.

[1] on a SUN 4/50 with 32 Mb of memory

For the *SQUAREMOS* generator, the number of gate segments may be set by the user, thus influencing the final shape of the transistor. The figure below illustrates the effect of this parameter.

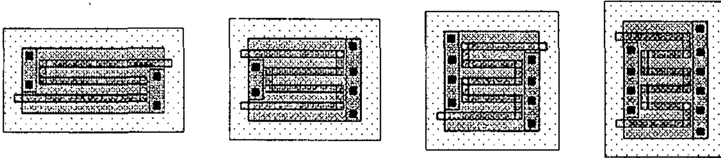


Fig.5.31 Illustration of the variation of the number of gate segments while preserving the same transistor size (scale 1).

The high degree of parametrization makes the transistor generators generally applicable. As a matter of fact, the descriptions capture the know-how of the designer because he was forced to parameterize the structure of the transistor, not to find a single solution. As such, they are good examples of reusable software.

All transistor generators are realized as a couple of two objects: a *class* type object which contains the actual implementation, and a *function* type object which takes care of the interface using the overloading mechanism.

The class header defines all the parameters that are of interest for the specific generator. The class body describes how the transistor will be generated using these parameters. The technology independence is implemented here. Notice that only one orientation is considered by the class implementation.

The function header defines the different possible calls of the generator. Here, overloading and defaulting play an important role to limit the required set of parameters. The function body creates the complete set of variables that is needed to activate the class (class type objects do not support overloading), calls the class, creates the cell instance, and returns.

An indication on the complexitiy of the generators may be found in Annex A, whereas Annex C contains the code of a complete MOS transistor generator. It accesses the technology rules for example to compute the number of contacts that may be placed on the source and drain area. Notice that the design effort of the generators is largely paid by the high reusability of the design.

An operational transconductance amplifier

An example of an analog application which uses most of the transistor generators presented in the previous paragraph is provided with an ota.

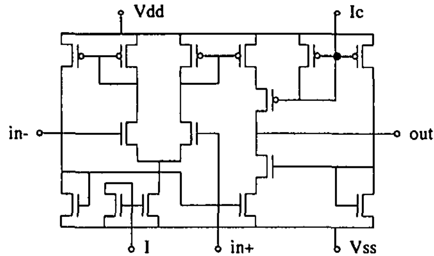


Fig.5.32 Netlist of a high voltage gain CMOS operational transconductance amplifier.

The transistor type of the input stage can be chosen at the invocation of the generator. All placements and interconnections are explicitly specified in the description. As they may largely influence the performance of the circuit, this specification needs to be very precise, resulting in a somewhat longer description: approximately 300 lines of al2 code were required to generate the layouts given below.

Observe that the density of the circuit is again similar to the one which would be obtained with a hand-crafted layout. However, the same topology might be reused for another ota with slightly different transistor lengths and widths.

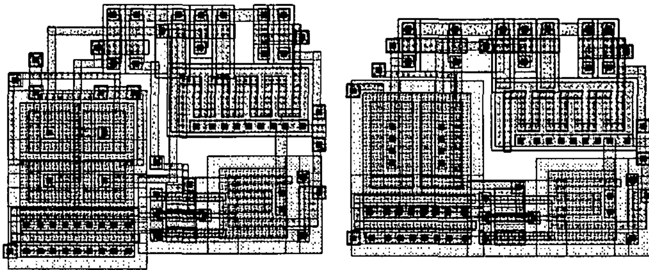


Fig.5.33 Two ota implementations with different input stage transistors (technology: mietec2u, scale 2).

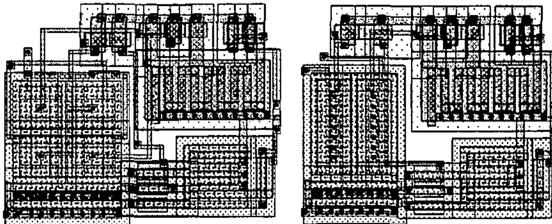


Fig.5.34 Two ota implementations with different input stage transistors (technology: cmn20, scale 2).

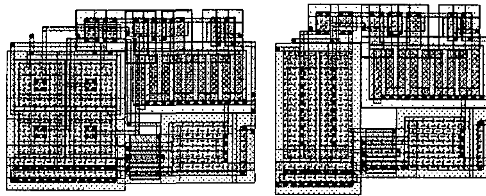


Fig.5.35 Two ota implementations with different input stage transistors (technology: sacmos2u, scale 2).

Bipolar transistor array

As the tool TILT acts on the geometrical level, it has no dependence of any kind on the technology families. To illustrate this, the tool was used to describe a bipolar variant of a gate-array [DEC92]. Groups of 10 hybrid transistors which may be used both as a npn or as a pnp type are surrounded by resistor groups and interconnect areas.

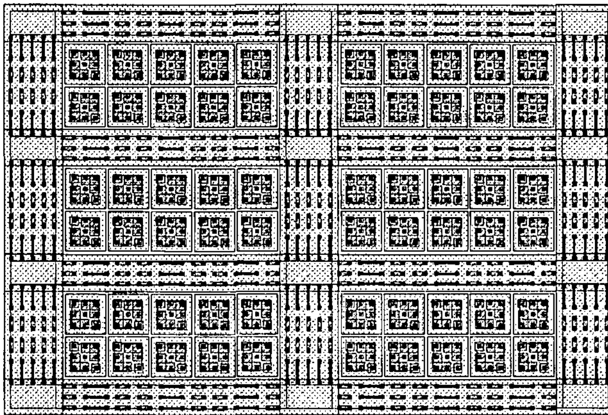


Fig.5.36 Bipolar gate array with 60 hybrid transistors (technology: WP2, scale 5).

The generator illustrates the interest of the automatic cell recognition. Similar to the CMOS transistor generators, all devices are declared with a call to a function type object. Inside this function, the transistor or resistor class is activated. The administration of earlier class activations avoids the regeneration of the device for every call.

Summary

Several types of transistor generators were presented and combined in an ota application. Their high degree of parametrization allows to control the created geometry in detail, and to create different alternative solutions with

comparable electrical characteristics. The generators execute sufficiently fast to be interesting for on-line implementations.

Not even 200 lines of al2 code, organized in 4 objects (1 transistor block, 2 resistor blocks and top-level assembling), were sufficient to describe a parameterized bipolar transistor array.

5.6 A switched capacitor silicon compiler

The final application is an example of a complete silicon compiler implemented with the TILT tool. It gives an illustration of the open character of the system. New, application specific functionalities have been added on to be able to create a complete filter, given the values of its capacitors. The generated layouts are optimal for the different technologies that were considered, and second metal possibilities has been explored.

Laker biquads

A special category of biquad filter structures is formed by the Laker biquads. Many classes of these filters exist, which differ in the number and position of capacitors and switches. The generic netlist of one filter class is given below; it will be used as a reference for the examples in this section.

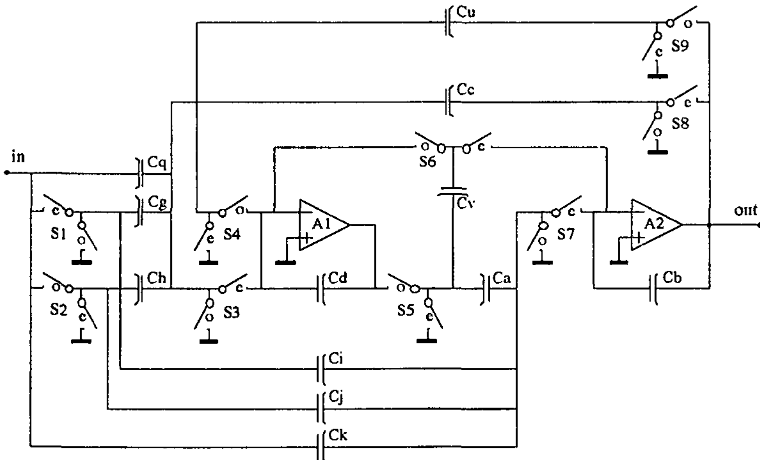


Fig.5.37 Generic netlist of a Laker biquad filter class.

Like most silicon compiler approaches, a fixed floor plan is used to generate the layout. This floor plan consists of five parts: a capacitor field, a switch area, an amplifier area, and two routing channels. It allows for an efficient implementation of cascaded biquad filter sections.

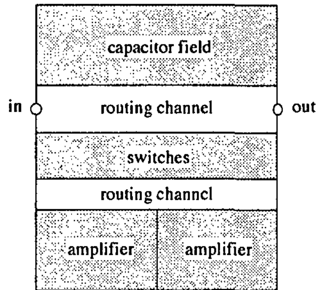


Fig.5.38 Floor plan of the switched capacitor filter silicon compiler.

The TILT system does not provide any automatic placement and routing tool. Therefore, three external functionalities were added: one to determine the orientation of the switches and to compute the optimum capacitor sequence, another to create the capacitor field, and finally a channel router for analog signals.

All of these functionalities were directly implemented in C, because the semantics of al2 is less suited for these purely algorithmic implementations. The interfacing with al2 is straightforward and consists of different data structure transformations (see also 4.7).

Capacitor ordering and switch orientation

Every capacitor and amplifier is connected to at least one switch. The ordering of the capacitors and the orientation of the switches influences both the total wire length used for the interconnections, and the number of interconnection-line cross-overs.

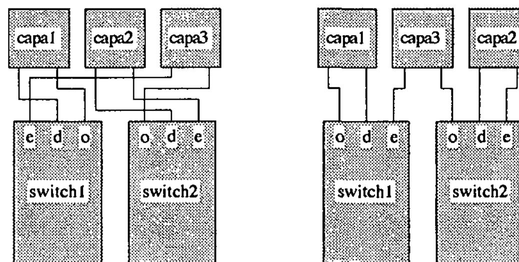


Fig.5.39 The orientation of the switches, and the ordering of the capacitors influences both the wire length and the number of cross-overs.

The (somewhat exaggerated) example in figure 5.39 shows well that a simple permutation may reduce both cross-overs and wire length considerably. If the switch order is fixed, a backtracking approach can be used to find the

optimum solution. As the number of elements is small, only a few seconds are required to select the best arrangement.

Capacitor field

The choice of a filter contains the selection of the biquad structure and the dimensioning of the capacitor values. In practice, every filter needs a specific set of capacitor values, which asks for an automatic approach. Due to the properties of a switched capacitor filter in general, the parasitic effects of a physical implementation may be minimized if the capacitors are realized as a multiple of the smallest capacitor, chosen equal to the unit capacitor. The parts of a capacitor values that are smaller than the unit capacitor, are implement with a rest-capacitor.

According to the floor plan, an algorithm has been developed which fills in a matrix of unit capacitors in an efficient way. It takes a table of capacitor values as its input, and it creates an al2 data structure which contains the filling of the matrix. The maximum height of the capacitor field can be given as a parameter, which allows for a matching of subsequent biquad sections. If this parameter is omitted, a square solution is created. The output data is used to control the capacitor field layout driver.

Routing

After all the blocks have been generated, they may be interconnected. However, no router is available inside the TILT system. As we were interested in a complete silicon compiler solution, a channel router had to be added on explicitly. The same router is used to interconnect the capacitors with the switches, and the switches with the amplifiers.

Again, the routing algorithm does not directly create layout, but a data structure which is used by a router layout driver implemented in al2. The use of constraint costs provides an efficient solution for the automatic minimization of highly resistive layers.

Examples

Two examples of layouts generated with this compiler are given in the figures 5.40 and 5.41. They are members of the class given in figure 5.37, with the height of the capacitor field limited to 5. The orientation of the switches can just be distinguished, and the dense matrix filling is apparent. Note that the polysilicon interconnections are globally minimized in the one-metal implementation.

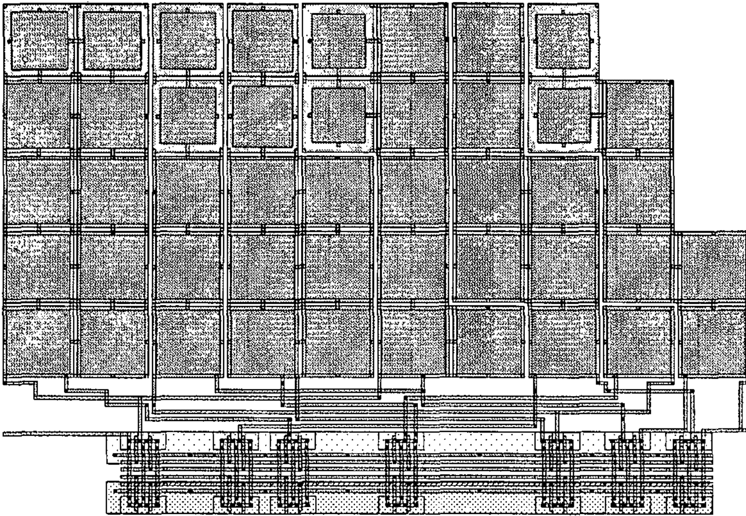


Fig.5.40 Detail of a biquad filter of the class given in fig.5.37. Ca=4.5 Cb=6.4 Cd=7.8 Cg=9.6 Ci=7.4 Cu=7.0 Cv=1.0, a maximum matrix height of 5, and one interconnection metal (technology: sacmos2u, scale 3).

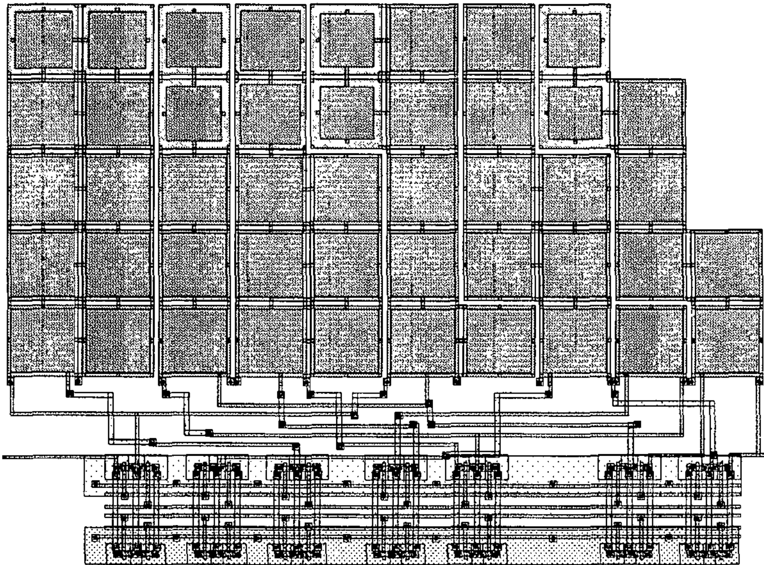


Fig.5.41 As above, but using two interconnection metals (technology: cmn20a, scale 3).

Summary

It is possible to implement a complete silicon compiler with the TILT tool using the facilities to add external functionalities. In this solution, they are responsible for the placement of the switches, creation of the capacitor matrix, and the routing between the different modules.

However, the development effort of such a compiler has been important. Especially the investment to realize the routing algorithm was large, whereas the benefit from the viewpoint of technology independence is relatively low. Instead, it may be more efficient to generate the different blocks (capacitor field, switches, and amplifiers) separately, and to use a commercial system to perform the module placement and routing.

5.7 Conclusions

This chapter has shown that low level design rules may be hidden in a few, high level objects, without loss of generality. This abstraction helps the design to concentrate on the topological intent, not on the process specific properties. As such, design become portable, and the user may forget about technology details (information hiding, encapsulation).

The definition of parameterized application specific functions which establish the design floor plan reduces the top level complexity of standard cell generators. Only a few instructions are sufficient to capture the essence of the design for a wide range of technologies. One and two metal descriptions are separated because this improves the readability of the design.

Large regular structures can be generated efficiently due to the support of hierarchy and instance matching. The memory generator illustrates the blackbox approach for the inclusion of external cells, as well as the advantages of the pitch matching facility. The latter is also of interest for the description of data path generators [MAN88].

The tool is well suited for the description of basic structure generators. The highly parameterized generators allow to generate a large variety of solutions, which may be of interest for analog design.

Missing features may be added on easily. Still, the lack of integrated place and route tools make it a tedious task to implement complete silicon compilers.

5.8 References

[BER91] M. Berkhout, *"SCSC a Switched Capacitor Silican Compiler, Valume I: Design Manual"*, internal report, IMT University of Neuchâtel, Switzerland, March 1991.

[DEC92] M. J. Declercq, P. Duchêne, O. Buset, *"Design & Performance of Analog Arrays far Semicustom*

IC Design", Proc. Eurochip workshop on VLSI training, Grenoble, France, 1992.

[LOG92] R.A. Logtmeijer, "*Analysis of a Layout Generator for Switched Capacitor Filters*", report 328 PE EC 09/92 at the Institute of Microtechnology, Neuchâtel, Switzerland, September 1992.

[MAN88] H. de Man, J. Rabaey, J. Vanhoof, G. Goossens, P. Six, and L. Claessen, "*CATHEDRAL II - a Computer Aided Synthesis System for Digital Signal Processing VLSI Systems*", Computer Aided Engineering Journal, pp. 55-66, April 1988.

[MAR93] J. Margairaz, "*Réalisation d'un générateur de mémoire RAM à l'aide de TILT*", Diploma work at the Institute of Microtechnology, Neuchâtel, Switzerland, March 1993.

[MAS91] J.M. Masgonty, C. Arm, C. Piguet, "*Technology- and Power-Supply-Independent Cell Library*", Centre Suisse d'Electronique et de Microtechnique CSEM SA, IEEE 1991, Custom Integrated Circuits Conference CICC'91, May 12-15, San Diego CA, USA, 1991.

[MEY93] V. Meyer zum Bexten, C. Moraga, R. Klinke, W. Brockherde, K-G. Hess, "*ALSYN: Flexible Rule-Based Layout Synthesis for Analog IC's*", IEEE Journal of Solid State Circuits, Vol. 28, No.3, pp.261-267, March 1993.

[KAY88] M. Kayal, M. Declercq, S. Piguet, E. Zysman, "*SALIM: A Layout Generation Tool for Analog ICs*", proc. CICC, Rochester, May 1988.

[RIJ88] J. Rijmenants, T. Schwarz, J. Litsios, R. Zinszner, "*ILAC: An automated layout tool for analog CMOS circuits*", CICC, 1988.

[UEH81] T. Uehara & W.M. vanCleemput, "*Optimal layout of CMOS functional arrays*", IEEE Trans. on Computers, Vol. C-30, pp.305-312, May 1981.

[WAT92] J.-P. Wattenhofer, "*Mémoire RAM pour processeur PUNCH*", Project CERS 2168, report no.799, Switzerland, October 1992.

1190

6 Conclusions

in my rear view mirror the sun is going down
sinking behind the bridges in the road
and I think of all the good things
that we have left undone

Pink Floyd

6.1 Technology independent layout design

Technical viewpoint

It has been shown that it is possible to create high quality layout for different technologies with a single generator. To achieve this, technologies need to be defined as members of specific classes, such as the CMOS, or the bipolar class. Furthermore, it may be advantageous to use separate descriptions for processes with different interconnection layers.

Technology independent layout design does not need to imply a performance penalty [RIE92]. The approach presented in this thesis shows that equal or close to hand-crafted densities may be obtained with technology independent generators.

The cost of the process tolerance is reflected in the increased complexity of the design system, and the longer development time. A design has to be verified for multiple technologies, in principle after every modification of the generator. The technology independence requires a large degree of parameterization, which makes it more difficult to guarantee the correctness of the generator.

It is essential to position the objects as a whole, not by selecting a specific part of it. Instead, the basic library functions access the object's attributes, thus concentrating the positioning actions. This does not only improve the portability of the design, but it also captures more of the (reusable) design intent.

Economical interest

A priori, the creation of a generator is a time-intensive activity. This can be explained from the large number of possible parameters in the topological, electrical, physical or even functional domain, which do often interact. Furthermore, a complete generator should provide, preferably, not only the layout, but also the netlist and a model!

Therefore, most commercial technology independent generators are quite rigid. They are usually based on a single rigid floor plan, and support only a limited set and range of parameters. Moreover, they are generally *virtually* technology independent, i.e. they are build on top of technology specific leaf-cells. The addition of a new technology requires the preliminary design of these cells.

It is true that the generator's counterpart, which is a library, has one very important advantage over a generator: each cell can be guaranteed to work properly, because it can be tested before a library is released. This property has contributed to the fast progress which has been achieved over the last few decades. For sure we would never have had portable PCs on the market without the use of libraries!

It should also be noted that only a few companies need to support generators for more than, lets say, three technologies. And if they do so, it is rare that this is known from the beginning. At short term, it is generally less expensive to adapt a technology specific generator to a new technology.

We may assume that the two principle motivations to achieve a design in a technology independent way are first the possibility to select the cheapest foundry at any moment, and second the ability to reuse the design for different processes. Taking this and the previous remarks into consideration, it may be stated that real technology independent generators are only of interest for those companies which need to support *themselves* many different technologies. Unfortunately, only a few large CAD vendors, plus some multinationals are the possible candidates here. And even, many large companies have based their production on their own, in-house, technology, which reduces the need for the ability to change the target process rapidly.

Still, the concept of generators is more flexible than the library approach. It would, for instance, be impossible to store all the possible variants of a memory block in a library. Also, generators may be used to create specific library cells (e.g. standard cells [MAS91]), thus concentrating the library maintenance, and reducing the time-to-market for new technologies.

Nevertheless, an exception has to be made for basic structure generators, in particular in the analog domain [MEY93]. Contrary to the digital domain,

analog building blocks are less suited to be frozen and put in a library because the designs are often more specific and precise. Many different structures exist to implement an operational amplifier, and every transistor in a design needs to be sized correctly. Providing technology independent generators for different transistor types, resistors, current mirrors, etc., may really help to reduce the design effort here.

6.2 TILT

The approach

The combination of a predictable compactor with an object oriented language provides a powerful and flexible environment to create technology independent layout generators, both for analog and digital circuitry. Compared with commercial mainframes, TILT is complementary: it only performs a very small part of the design task, but in a technology independent way, with near or equal to hand-crafted densities for Manhattan type layout.

These results are due to the use of a technology specific graph based compaction. Compared with fixed or virtual grid approaches [PRE92], only the absolutely needed constraints are inserted. All these constraints are controlled by the user.

The use of a constraint cost is essential to achieve predictable results. This requires a cost minimization phase which provides an optimum distribution of the non-critical vertices. A specific TILT-feature is that costless vertex groups are centered within their free margin. The resulting distribution is uniform and may have a positive effect on the yield.

Compared to a graphical entry, a language is much more expressive and powerful, but less user-friendly. Al2 was developed to cope with the technology independence in a geometrical environment. Its object oriented character combined with pattern search facilities allows to encapsulate the access to technology specific data. As a result, al2 descriptions allow to capture the design intent and tend to increase the reusability of the code.

Applications

The complete system has been tested on a variety of applications. It appears that it is well adapted to achieve a high degree of parametrization. Especially the possibility to perform an automatic pitch-matching makes it interesting for the creation of regular modules. However, routing inside TILT is tedious, compared to the facilities offered in commercial systems.

In particular, the tool is useful to create basic structure generators, for example for different transistor types, resistors, etc. These elements do not require any routing, and "ask" for a parameterized approach.

Still, the design effort to write a technology independent generator brings into question the investment it requires. Therefore, I believe that only an interactive optimized implementation, which can directly incorporate existing designs, could be of commercial interest.

Implementation

The actual TILT version (3.0), has been implemented in ANSI-C and has been installed on SUN and HP type of workstations. The basic configuration consists of 25.000 lines of source code organized in 85 modules, which needs less than one megabyte of object code. Additional space is needed to store the different application libraries and manual pages.

An important improvement could be made if the core of the system would be rewritten in C++, because this language is much closer to the al2 semantics than C. Such an adaptation would not only reduce the amount of source code drastically, but it would also improve the performance of the generators, and ease the implementation of an interpreter.

The *bootstrap* implementation of the compiler may be recommended because of its multiple advantages. The translation of the source language into an intermediate high-level language does not only allow for a rapid prototyping, but it also provides a large portability of the system, as well as an easy addition of external modules (open system).

Most of the algorithms present in the compactor are close to optimum with regard to their structure. However, I believe that an additional gain of at least a factor of 2 is possible with a more efficient choice of the data structure, in particular with respect to vertex fusion. Furthermore, the implementation of the centering algorithm is far from the optimum.

Limitations

One of the ideas behind TILT was that it should be able to produce layout equivalent to its hand-crafted counterpart in a technology tolerant environment. Therefore, the approach does not rely on any automatic placement algorithms, but only on the knowledge of the designer, combined with a constraint solver (the compactor). An essential condition for such a system to be useful, is that it should produce predictable results. As a consequence, no exclusive or inter-dimensional constraints should be allowed as they introduce some kind of randomness. On the contrary, the addition of active constraints for the definition of symmetry [OKU89] seems to be an interesting extension.

A similar explanation can be given with regard to the absence of oblique elements [MAR88]. Again, it would become very difficult to foresee the result, even locally. The placement of neighbor elements would rapidly become tedious. Instead, it is suggested to implement non-rectangular elements, such as circles and triangles, with a rectangular envelop. This envelop allows to refer to these elements as if they were rectangles.

Moreover, it should be noted that the tool does not guarantee the construction of design rules error free layout, because all elements are placed by the user himself. If the system would add on the necessary constraints automatically (using the technology file), this drawback could be avoided. However, this would introduce the inconvenience that for example overconstraint reports could contain a mixture of user and system constraints.

The experience that the number of library functions is small, would allow for the addition of a graphical interface on top of a technology class library. In this way, the structural information could be entered graphically, whereas the algorithmic data would be entered as text.

Finally, the automatic creation of a netlist would be a valuable extension. Actually, the only way to obtain the netlist, is to extract it from the layout. Instead, it could be constructed by inserting some additional commands into the library functions.

6.3 Suggestions

Technology independent design

Personally, I believe that technology independent layout design by construction can only be justified for a very limited set of applications. The return on investment is simply too low for most situations.

Instead, there might be a future for translation style tools [CON92], which would allow a layout to be translated into another technology. The penalty in terms of density loss can be kept low if such a method would first extract the basic building blocks in terms of objects, followed by the determination of the essential topological information in between these objects.

In the ideal case, such a tool could generate an al2 description which would be able to regenerate the original layout, as well as the layout for a set of resembling technologies. In this case, no direct intervention of the user would be needed anymore, which would make the run time performance of the tool a secondary issue.

Data flow

The existence of both *cell* and *class* type objects is highly related with the implementation. A cell object can never be called by another object, and therefore, produces a fixed layout. A class object is in intermediate object, and may possibly be stretched by caller constraints, which obliges it to extract a substitute graph.

However, a more general and flexible solution would eliminate the cell type object, and call a class object directly. In fact, only the data format would be important, not the implementation of the objects. Each object would manage its own data, and could be coupled to a library or a data-base. It could be the base for a complete main frame where all layout would be provided by class type objects.

On the other hand, new object types could be added to implement specific features. In particular, this would be useful to define alternative solutions in an analog design environment [KAY88]. Here, the use of a shape function may help to explore the design space thoroughly.

A mixed approach

It is true that a textual approach is more flexible than a method based on a graphical entry. However, the latter is very well suited to capture topological information, and may eliminate the problem of syntax errors, just to mention a few advantages. Taking the low number of topological library objects into account, an elegant and efficient solution would be to combine the two approaches [EES91, GAU92]: the textual one to describe the parameterized objects, and the graphical one to create cells based on these basic objects.

Netlist

Unfortunately, TILT provides no direct link with the netlist of the generated structure. Still, the netlist is an excellent low level, technology independent representation of the circuit to be realized. It contains both device and connectivity information, but nothing more.

Therefore, it seems interesting to use the netlist as a starting point for cell definitions. The missing topological information could be annotated, or extracted from the device's placement and orientation in a planar representation. An intelligent (hierarchical) schematic editor, which allows to move devices freely while preserving the full connectivity, would be the main entry of such a system.

If a layout design tool was to be developed based on the principles mentioned above, I would start with a technology *independent* template netlist,

but use a technology *specific* instance to complete the design. The *devices* themselves, could be generated with a tool like TILT.

6.4 References

[CON92] B. Conq, R. Etienne, T. Perez-Segovia, "*Design Library Portobility: A Case Study*", WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, March 12-13 1992.

[DUF92] J.C. Dufourd, J.F. Naviner & F. Jutand, "*PREFORM: a process independent symbolic layout system*", Proc. ICCAD, 1990.

[EES91] R.A. Eesley & M.A. Tarsi, "*ACAP - A System for the Interactive Graphical Capture of Module Generotors*", Proc. CICC, 1991.

[GAU92] E. Gautrin, L. Perraudeau, "*MADMACS: o tool for the layout of regular arrays*", WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, March 12-13 1992.

[KAY88] M. Kayal, M. Declercq, S. Piguet, E. Zysman, "*SALIM: A Layout Generation Tool for Analog ICs*", proc. CICC, Rochester, May 1988.

[MAR88] D. Marple, M. Smulders, H. Hegen, "*An efficient compactor for 45° layout*", Proc. 25th DAC, pp.396-402, June 1988.

[MAS91] J.M. Masgonty, C. Arm, C. Piguet, "*Technology- and Power-Supply-Independent Cell Library*", Centre Suisse d'Electronique et de Microtechnique CSEM SA, IEEE 1991, Custom Integrated Circuits Conference CICC'91, May 12-15, San Diego CA, USA, 1991.

[MEY93] V. Meyer zum Bexten, C. Moraga, R. Klinke, W. Brockherde, K-G. Hess, "*ALSYN: Flexible Rule-Based Layout Synthesis for Analog IC's*", IEEE Journal of Solid State Circuits, Vol. 28, No.3, pp.261-267, March 1993.

[OKU89] R. Okuda, T. Sato, H. Onodera, and K. Tamuru, "*An efficient algorithm for layout compaction problem with symmetry constraints*", Proc. ICCAD '89, pp.148-151, 1989.

[RIE92] R. Riem-Vis, G. Maliki, & F. Pellandini, "*TILT, a Technology Independent Layout Tool*", WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASIC Design, Grenoble, March 12-13, 1992.

Annex A: Technical data

A.1 Utilities and libraries

The table below lists all of the utilities that are available in the TILT tool. The extensive use of the yacc tool limits the size of the source codes considerably.

utility	# modules	# lines C	size in KB
precompiler	1	150	40
compiler	5	6000	187
compactor	5	5500	156
monitor	2	2100	236
image to CIF converter	1	300	49
image to PostScript® converter	1	600	49
CIF to image converter	2	900	74

An overview of the library sizes illustrates the compactness of al2 code. The module count does not include the cell type objects.

library	# modules	# lines al2	# lines C	size in KB
system	77	-	15000	268
cmos	25	3500	-	1048
bipolar	17	1600	-	499
standard cell	13	800	-	262
RAM	21	2200	-	670
switched capacitor	10	900	4400	390

A.2 Al2 compiler options

The al2 compiler is built on top of the C language. It performs a complete syntax check of the al2 source code, but it creates a C file which is passed to an

on site ANSI C compiler to create the object code. The al2 compiler recognizes only a few options himself; the remaining ones are passed to the C compiler and can be used to optimize the code, create debugging information, etc. A list of the options and their meaning is given below.

-Idir	add dir to the search path to look for al2 definition files
-P	run only the preprocessor, and leave the result in a .c file

A few environment variables are used to define the following:

AL2_CC	name of the ANSI C compiler to be used
AL2_INC	include path of system include files
AL2_LIB	link path of the system library
AL2_PC	full path of the precompiler
AL2_TMP	directory were to place temporary files (default /tmp)

A.3 Compactor options

The application of the slack distribution algorithm and the implementation of fixed bound constraints can be set externally by means of the following environment variables:

TILTMODE	determines the algorithms to be applied for the slack distribution, should be one of the following: none reduce critical path 0 reduce critical path 1 reduce zero cycles 2 reduce both critical path and zero cycles 3 no reduction at all
TILTFIX	determines how fixed bound constraints are implemented. If this variable equals "parallel" two parallel lower bound constraints are inserted, otherwise, vertex fusion is applied.

Annex B: Formats

B.1 Al2 syntax

The complete syntax of the language al2 is given below. It is a context-free grammar which has been implemented in yacc using 120 rules, including the rules to capture syntax errors.

```
<al2 program> ::=          <al2 object>
<al2 object> ::=           <definition> <body>
<definition> ::=           <cell def> |
                           <class def> |
                           <environment def> |
                           <function def> |
                           <technology def>
<cell def> ::=             "TYPE" "(" "cell" ")" ";" [<environment spec>)
<class def> ::=            "TYPE" "(" "class" ")" ";" <resource spec> <port
                           spec> <proto spec> [<environment spec>]
<environment def> ::=      "TYPE" "(" "environment" ")" ";" [<export spec>
                           [<proto spec> [<environment spec>] [<techno spec>]
<function def> ::=         "TYPE" "(" "function" ")" ";" <function spec> <proto
                           spec> [<environment spec>]
<technology def> ::=       "TYPE" "(" "techno" ")" ";" [<export spec> <proto
                           spec>] [<environment spec>]
<environment spec> ::=     "ENVIRONMENT" "(" <environment list> ")" ";"
<resource spec> ::=        "INPUT" "(" <identifier> ")" ";"
<port spec> ::=            "OUTPUT" "(" [<identifier list>] ")" ";"
<proto spec> ::=           <identifier> {"[]" <field>} ":" <type> ";" |
                           <identifier> {<field> "[]"} ":" <type> ";" |
                           <identifier> "[]" {<field> "[]"} ":" <type> ";" |
                           <identifier> <field> {"[]" <field>} ":" <type> ";"
<export spec> ::=          "EXPORT" "(" [<identifier list>] ")" ";"
<techno spec> ::=          "TECHNO" "(" ")" ";"
<function spec> ::=        "FUNCTION" "(" [<identifier list>] ")" ";"
<environment list> ::=     <string> [", " <string>]
<identifier list> ::=      <identifier> [", " <identifier>]

<body> ::=                 {<instruction>}
```

```
<instruction> ::=       <empty instr> |
                        <conditional instr> |
                        <assignment instr> |
                        <reference instr> |
                        <indirection instr> |
                        <link instr> |
                        <instruction block> |
                        <comment>
<empty instr> ::=       ";"
<conditional instr> ::= "IF" "(" expression ")" <instruction>
<assigmment instr> ::=  ( <identifier> ":=" <expression> ";" ) |
                        ( <identifier> <index> ":=" <expression> ";" ) |
                        ( <identifier> <field> ":=" <expression> ";" ) |
                        ( <identifier> <field> <index> ":=" <expression> ";" )
<reference instr> ::=   <identifier> "=" <expression> ";"
<indirection instr> ::= <identifier> "::" <expression> ";"
<link instr> ::=        <identifier> "->" <expression> ";"
<instruction block> ::= "{" {<instruction>} "}"
<comment> ::==          "#" {<ascii>}
<expression> ::=        <boolean> |
                        <integer> |
                        <real> |
                        <string> |
                        <identifier> |
                        <identifier> <index> |
                        <identifier> <field> |
                        <unary operator> <expression> |
                        <expression> <binary operator> <expression> |
                        "<" <expression> ">" |
                        <function object> "(" [<parameter list>] ")" ";" |
                        <class object> "(" <expression> ")" ";" |
                        "(" <expression> ")"
<index> ::=             "[" <parameter list> "]"
<field> ::=             "." <identifier> {"." <identifier>}
<parameter list> ::=    <expression> {"," <expression>}
<identifier> ::=        <alpha> {<alpha> | <digit> | "_"}
<type> ::=              "boolean" |
                        "integer" |
                        "real" |
                        "string" |
                        "set"
<boolean> ::=           "FALSE" |
                        "TRUE"
<integer> ::=           <digit> {<digit>}
<real> ::=              <integer> "." <integer> |
                        "." <integer> |
                        <integer> "."
<string> ::=            """ {<ascii>} """
<unary operator> ::=    "-" | "NOT"
<binary operator> ::=   "AND" | "OR" | "<" | ">" | "<=" | ">=" | "<>" | "=="
                        | "+" | "-" | "*" | "MOD" | "/" | "DIV" | "**" | "|"
<alpha> ::=             "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
                        "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" |
                        "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" |
                        "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
                        "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
                        "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
<digit> ::=             "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
                        "9"
<ascii> ::=             any printable ascii character
```

B.2 Layout image

TILT uses its own specific layout format because each geometry is identified with a name. This binary format can be translated rapidly into CIF, PostScript® or visualized on the screen with an X-Windows based monitor. The definition file of this format is given below.

```
typedef int        TimgTtrans[10];

typedef struct {
   int          x,y;
} TimgTpoint;

typedef struct {
   char         *name;
   unsigned     length;
   int          layer,x,y;
   unsigned     width,height;
} TimgTport;

typedef struct {
   char         *name;
   unsigned     length;
   int          x,y;       /* left,bottom */
   unsigned     width,height;
   int          orientX,orientY;
} TimgTrect;

typedef struct {
   char         *name;
   unsigned     length;
   int          pointCount;
   TimgTpoint   *pointArray;
} TimgTpoly;

typedef struct {
   char         *name;
   unsigned     length;
   int          x,y;
   unsigned     width,height;
   int          angle1X,angle1Y,angle2X,angle2Y;
} TimgTarc;

typedef struct {
   int          layer;
   int          rectCount,polyCount,arcCount;
   TimgTrect    *rectArray;
   TimgTpoly    *polyArray;
   TimgTarc     *arcArray;
} TimgTslice;

typedef struct {
   char         *name,*sourceName;
   unsigned     length,sourceLength;
   unsigned     width,height;
   int          x,y;
   TimgTtrans   trans;
   int          portCount;
   TimgTport    *portArray;
} TimgTcell;

typedef struct {
   char         *name;
   unsigned     length;
   int          multScale,divScale;
```

```
    int          x,y;
    unsigned     width,height;
    int          sliceCount,cellCount,portCount;
    TimgTslice   *sliceArray;
    TimgTcell    *cellArray;
    TimgTport    *portArray;
} TimgTimage;
```

Annex C: Application code

C.1 The UNION function

The implementation of the function UNION illustrates the modularity of the al2 system operators and functions. All of them are implemented according to the same principle. The type checking is performed in the *SEMANT* mode, and the code of the intermediate target language (C) is created with a two pass process. Different macros are used to increase the readability of the code.

The overloading of the functions is simulated with the use of different internal procedure names. A unique letter, representing the data type of each input argument allows for a simple mapping.

C code

```
#include "Tilt.h"
#include "Tsym.h"
#include "Tlib.h"
#include "Tclc.h"
#include "Terr.h"

TclcProto(      UNION_i)
BEGIN
  TiltTset      result = (TiltTset)TclcGetResult;

  if(TclcGetData(0))
    TclcInsertSet(&result,TclcGetInteger(0));
  TclcPutSet(result);
END

TclcProto(      UNION_e)
BEGIN
  TiltTset      result = (TiltTset)TclcGetResult;

  TclcUnionSet(&result;result,TclcGetSet(0));
  TclcPutSet(result);
END

TsymProto(UNION)
```

```
{
  register int    i;

  switch(mode)
  {
    case SEMANT :
      if(argCount)
      {
        for(i=0; i<argCount; i++)
          switch(TsymGetType(i))
          {
            case TiltCinteger :
            case TiltCset :
              break;
            default : /* integer or set type expected */
              TsymError(_N126,_C126);
            case TiltCundef :
              return;
          }
        TsymPutType(TiltCset);
        TsymPutScope(TsymCtmp);
      }
      else
        TsymError(_N9,_C9); /* at least one argument expected */
      break;

    case PASS1 :
      if(id == NULL)
        TsymInstallTemp((TiltPsymbol)expr);
      for(i=0; i<argCount; i++)
      {
        switch(TsymGetType(i))
        {
          case TiltCinteger :
            TsymInstallRoutine("UNION_i");
            break;
          case TiltCset :
            TsymInstallRoutine("UNION_e");
            break;
        }
        TsymPass1(TsymGetArg(i),NULL);
      }
      break;

    case PASS2 :
      TsymSetBlock(expr);
      for(i=0; i<argCount; i++)
        TsymPass2(TsymGetArg(i),NULL);

      TsymInstallDelete((TiltPsymbol)expr);
      for(i=0; i<argCount; i++)
      {
        TsymPrintExpr((TiltPsymbol)expr);
        switch(TsymGetType(i))
        {
          case TiltCinteger :
            TsymPrint("=UNION_i(1");
            break;
          case TiltCset :
            TsymPrint("=UNION_e(1");
            break;
        }
        TsymPrintArg(TsymGetArg(i));
        if(i)
          TsymPrintArg((TiltPsymbol)expr);
        else
          TsymPrint(",NULL");
        TsymPrint(");\n");
```

```
        }

        if(id == NULL)
          TsymInstallFree((TiltPsymbol)expr);
        TsymResetBlock(expr);
        break;
    }
}
```

C.2 ADDCONT

In section 5.2 it was shown that the implementation of an object may have a large influence on its performance. Three extreme cases were presented: a high level al2 implementation, a low level al2 implementation, and a C implementation. All of them use the same header, but they have a different degree of abstraction, efficiency and complexity.

Al2 header

```
TYPE("function");
ENVIRONMENT("DEF_RESS","LAYERS","RULES");
FUNCTION(cont1, layer);        #create a minimum fixed contact

cont1.metal        : record;
cont1.contact      : record;
cont1.diffusion    : record;
cont1.poly         : record;
cont1.smetal       : record;
layer              : integer;
```

High level al2 body

A high level al2 description is the fastest to develop and the easiest to read. These advantages are paid with a performance penalty.

```
dif    :: ADDRECT(MIN_STR,layer);
pol    :: ADDRECT(MIN_STR,POLY);
met    :: ADDRECT(MIN_STR,METAL);
smet   :: ADDRECT(MIN_STR,SMETAL);
conl   :: ADDRECT(MIN_FIX,conlay);

cont1.contact := conl;
cont1.metal := met;
cont1->EXTEND(FIX_COVER,met,conl);

IF(layer == NDIFF) {
  cont1.diffusion := dif;
  cont1->EXTEND(FIX_COVER,dif,conl);
  conlay := CDIFF;
}
IF(layer == PDIFF) {
  cont1.diffusion := dif;
  cont1->EXTEND(FIX_COVER,dif,conl);
  conlay := CDIFF;
}
IF(layer == POLY) {
  cont1.poly := pol;
  cont1->EXTEND(FIX_COVER,pol,conl);
  conlay := CPOL;
```

```
}
IF(layer == SMETAL) {
  contl.smetal := smet;
  contl->EXTEND(FIX_COVER,smet,conl);
  conlay := VIA;
}
```

Low level al2 body

A low level al2 description is somewhat longer to develop but may offer a remarkable performance gain. The implementation below runs two times faster than the previous one.

```
conmin := integer|(SCALE*WIDTH[conlay]);
difmin := integer|(SCALE*WIDTH[layer]);
polmin := integer|(SCALE*WIDTH[POLY]);
metmin := integer|(SCALE*WIDTH[METAL]);
smetmin := integer|(SCALE*WIDTH[SMETAL]);

difext := integer|(SCALE*EXTENSION[layer,conlay]);
polext := integer|(SCALE*EXTENSION[POLY,conlay]);
metext := integer|(SCALE*EXTENSION[METAL,conlay]);
smetext := integer|(SCALE*EXTENSION[SMETAL,conlay]);

conl :: RECT(conlay,conmin,conmin,conmin,conmin,0,0);  # fixed
dif :: RECT(layer,difmin,difmin,0,0,0,0);
pol :: RECT(POLY,polmin,polmin,0,0,0,0);
met :: RECT(METAL,metmin,metmin,0,0,0,0);
smet :: RECT(SMETAL,smetmin,smetmin,0,0,0,0);

contl.contact := conl;
contl.metal := met;
contl->EXTENSION(met.rectangle.name,
               conl.rectangle.name,alldir,2,metext);

IF(layer == NDIFF) {
  contl.diffusion := dif;
  contl->EXTENSION(dif.rectangle.name,
               conl.rectangle.name,alldir,2,difext);
  conlay := CDIFF;
}
IF(layer == PDIFF) {
  contl.diffusion := dif;
  contl->EXTENSION(dif.rectangle.name,
               conl.rectangle.name,alldir,2,difext);
  conlay := CDIFF;
}
IF(layer == POLY) {
  contl.poly := pol;
  contl->EXTENSION(pol.rectangle.name,
                  conl.rectangle.name,alldir,2,polext);
  conlay := CPOL;
}
IF(layer == SMETAL) {
  contl.smetal := smet;
  contl->EXTENSION(smet.rectangle.name,
                  conl.rectangle.name,alldir,2,smetext);
  conlay := VIA;
}
```

C version

A C implementation is close to the internal TILT structure and allows for the optimum speed. However, the development time is much longer than a low

level al2 description, and the method is far from error prone. Furthermore, the additional performance gain, compared to the low level al2 description, is only 25%. The example below is a manually optimized version of the intermediate code that was created for the low level al2 description.

```
#include "Tclc.h"
extern void free(void* pointer);
extern TclcProto(equal_ii);
extern TclcProto(LAYERS_SMETAL);
extern TclcProto(EXTENSION_sseii);
extern TclcProto(point_g);
extern TclcProto(RECT_iiiiiii);
extern TclcProto(cast_ir);
extern TclcProto(multiply_ir);
extern TclcProto(RULES_SCALE);
extern TclcProto(content_g);
extern TclcProto(RULES_WIDTH);
extern TclcProto(assign_g);
extern TclcProto(record_g);
extern TclcProto(LAYERS_VIA);
extern TclcProto(LAYERS_POLY);
extern TclcProto(LAYERS_CPOL);
extern TclcProto(LAYERS_PDIFF);
extern TclcProto(LAYERS_CDIFF);
extern TclcProto(LAYERS_NDIFF);
extern TclcProto(DEF_RESS_alldir);
extern TclcProto(RULES_EXTENSION);
extern TclcProto(LAYERS_METAL);
TclcProto(CONTC)
{
char *_list=(void*)&_aList,_buffer[BUFSIZ];
jmp_buf _exception;
TclcRstack _pile[100],*_base=_pile,*_head=_pile;
static int  _ti[2]={0,2,};
TiltTdata   EXTENSION,_id2,SCALE,cont1,METAL,metmin,metext,
       conlay,layer,_id3,alldir,layext,_id0,rect,
       conmin,laymin,_id1,con1,WIDTH,met,recordfield;
SEED(5);
TiltTdata _ex[5];
switch(argCount) {
case 0 : {
layer=TclcGetData(0);
START;
cont1=UNDEF;
layer=_OK(layer);

if(layer == NULL)
  return(NULL);

_id0=equal_ii(2,layer,LAYERS_NDIFF(0,NULL));
_id1=equal_ii(2,layer,LAYERS_PDIFF(0,NULL));
_id2=equal_ii(2,layer,LAYERS_POLY(0,NULL));
_id3=equal_ii(2,layer,LAYERS_SMETAL(0,NULL));

/* if ( layer == NDIFF ) */
if(_TEST(_id0)) {
conlay=LAYERS_CDIFF(0,NULL);
recordfield = (TiltTdata)"diffusion";
}
/* if ( layer == PDIFF ) */
else if(_TEST(_id1)) {
conlay=LAYERS_CDIFF(0,NULL);
recordfield = (TiltTdata)"diffusion";
}
/* if ( layer == POLY ) */
```

```
else if(_TEST(_id2)) {
conlay=LAYERS_CPOL(0,NULL);
recordfield = (TiltTdata)"poly";
}
/* if ( layer == VIA ) */
else if(_TEST(_id3)) {
conlay=LAYERS_VIA(0,NULL);
recordfield = (TiltTdata)"smetal";
}
else
  return(NULL);

METAL=LAYERS_METAL(0,NULL);
WIDTH=RULES_WIDTH(0,NULL);
alldir=DEF_RESS_alldir(0,NULL);
EXTENSION=RULES_EXTENSION(0,NULL);
SCALE=RULES_SCALE(0,NULL);

{
static int _ty0[5] = {9,9,9,9,9,};
static char *_la0[5] =
{"metal","diffusion","poly","contact","smetal",};
TclcCreateRecord((TiltTrecord*)&cont1,5,_la0,_ty0,0);
}

/* compute width and extension */
_ex[0]=content_g(2,conlay,WIDTH);
_ex[1]=multiply_ir(2,SCALE,_ex[0]);
_FREE(_ex[1],5);
conmin=cast_ir(2,"integer",_ex[1]);
_ex[0]=content_g(2,layer,WIDTH);
_ex[1]=multiply_ir(2,SCALE,_ex[0]);
_FREE(_ex[1],5);
laymin=cast_ir(2,"integer",_ex[1]);
_ex[0]=content_g(3,layer,conlay,EXTENSION);
_ex[1]=multiply_ir(2,SCALE,_ex[0]);
_FREE(_ex[1],5);
layext=cast_ir(2,"integer",_ex[1]);

/* create contact */
_FIELD("contact");
con1=RECT_iiiiiii(7,conlay,conmin,conmin,conmin,conmin,
   (TiltTdata)&_ti[0],(TiltTdata)&_ti[0]);
cont1=record_g(2,"contact",con1,cont1);

_ex[0]=multiply_ir(2,SCALE,content_g(2,METAL,WIDTH));
_FREE(_ex[0],5);
metmin=cast_ir(2,"integer",_ex[0]);

/* create metal */
_FIELD("metal");
met=RECT_iiiiiii(7,METAL,metmin,metmin,(TiltTdata)&_ti[0],
   (TiltTdata)&_ti[0],(TiltTdata)&_ti[0],(TiltTdata)&_ti[0]);
cont1=record_g(2,"metal",met,cont1);

_ex[0]=multiply_ir(2,SCALE,content_g(3,METAL,conlay,EXTENSION));
_FREE(_ex[0],5);
metext=cast_ir(2,"integer",_ex[0]);

/* extend metal on contact */
_ex[0]=point_g(3,"rectangle","name",met);
_ex[1]=point_g(3,"rectangle","name",con1);
EXTENSION_sseii(5,_ex[0],_ex[1],alldir,(TiltTdata)&_ti[1],metext);

/* create second layer */
_FIELD(recordfield);
rect=RECT_iiiiiii(7,layer,laymin,laymin,(TiltTdata)&_ti[0],
   (TiltTdata)&_ti[0],(TiltTdata)&_ti[0],(TiltTdata)&_ti[0]);
```

```
contl=record_g(2,recordfield,rect,contl);

/* extend second layer on contact */
_ex[0]=point_g(3,"rectangle","name",rect);
_ex[1]=point_g(3,"rectangle","name",conl);
EXTENSION_sseii(5,_ex[0],_ex[1],alldir,(TiltTdata)&_ti[1],layext);
_UPDATE;

_FREE_b(_id0); _FREE_b(_id1); _FREE_b(_id2); _FREE_b(_id3);
_FREE_i(conmin); _FREE_i(metmin); _FREE_i(metext);
_FREE_i(laymin); _FREE_i(layext);
return(contl);
} break;
}
}

TclcProto(CONTC_i)
{
char *_list=(void*)&_aList;
return(CONTC(0,TclcGetData(0)));
}
```

C.3 MOS transistor

Thhe implementation of the very frequently used MOS transistor is given below. Usually, this object is only called indirectly (by the function type object *ADDMOS*) in order to benefit from overloading and to provide a more convenient interface.

AI2 header

```
TYPE("class");
ENVIRONMENT("LAYERS","RULES","DEF_RESS");

INPUT(layer,width,length,cont);

layer              : integer;
width              : float;    #default WIDTH[layer]
length             : float;    #default WIDTH[POLY]
cont.cdrain        : boolean;  #default TRUE
cont.csource       : boolean;  #default TRUE

OUTPUT(drain,source,channel,gate,well);

drain.metal        : record;
drain.contact      : record;
drain.diffusion    : record;
source.metal       : record;
source.contact     : record;
source.diffusion   : record;
channel            : record;
gate               : record;
well               : record;
```

AI2 body

```
IF((layer <> PTR) AND (layer <> NTR))
  WRITE("WARNING  illegal layer specified in MOS call : ",layer);

IF(layer == PTR) { cdiff := PDIFF; cwell := NWELL; }
IF(layer == NTR) { cdiff := NDIFF; cwell := PWELL; }
```

```
len = GETFIRST(length,WIDTH[POLY]);
minwid = WIDTH[CDIFF]+2*EXTENSION[layer,CDIFF];
wid = GETFIRST(width,minwid);

well := ADDRECT(MIN_STR,cwell);

rgt.y.fix    := TRUE;
rgt.y.minvalue := len;
rgt.x.minvalue := wid+2*EXTENSION[POLY,layer];
wpol = GETFIRST(WEIGHT[POLY]+2,1);
gate := ADDRECT(rgt,POLY,wpol);

rch.x.fix := TRUE;
rch.x.minvalue := wid;
channel := ADDRECT(rch,layer);

IF (cont.cdrain) { #declare and position drain contact
   drain.contact := ADDRECT(MIN_FIX,CDIFF);
   drain.diffusion := ADDRECT(MIN_STR,cdiff);
   drain.metal := ADDRECT(MIN_STR,METAL);
   ATTACH(layer,channel,cdiff,drain.diffusion);
   PLACE(TOP,drain.contact,gate);
   EXTEND(HORIZ,channel,drain.contact);
   EXTEND(FIX_COVER,drain.diffusion,drain.contact);
   EXTEND(FIX_COVER,drain.metal,drain.contact);
}
IF (cont.csource) { #declare and position source contact
   source.contact := ADDRECT(MIN_FIX,CDIFF);
   source.diffusion := ADDRECT(MIN_STR,cdiff);
   source.metal := ADDRECT(MIN_STR,METAL);
   ATTACH(layer,channel,cdiff,source.diffusion);
   PLACE(BOTTOM,source.contact,gate);
   EXTEND(HORIZ,channel,source.contact);
   EXTEND(FIX_COVER,source.diffusion,source.contact);
   EXTEND(FIX_COVER,source.metal,source.contact);
}

EXTEND(HORIZ,gate,channel);
EXTEND(VERT,channel,gate);
EXTEND(COVER,well,channel);
EXTEND(COVER,well,source);
EXTEND(COVER,well,drain);
```