A Silicon Compiler for Random Logic

by

Christian Schneider

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Electrical Engineering

Winnipeg, Manitoba, 1985
© Christian Schneider, 1985

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN Ø-315-33861-X

A SILICON COMPILER FOR RANDOM LOGIC

BY

CHRISTIAN SCHNEIDER

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

© 1986

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis. to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The University of Manitoba requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

The random logic silicon compiler (named Nebula) developed translates a transistor net-list specification of a circuit into an integrated circuit (IC) layout. The IC layouts generated by the present version of the compiler require approximately 2.8 times as much IC silicon area as a human-designed layout of the same circuit.

The compiler consists of three sections: the layout optimization module (LOM), the layout database manager (LDM), and the high level layout module (HLLM). The layout optimization module uses a hierarchical clustering algorithm to find an arrangement of the transistors in the net-list which results in an efficient layout. The layout database manager performs all layout operations and maintains all information about the present state of the layout, and the high level layout module supervises the layout process.

To accommodate the rapid changes occurring in IC fabrication technology, the Nebula compiler was designed to allow easy adaptation to new IC fabrication processes. Many fabrication process changes simply require the modification of six technology description files used by the compiler. The present version of the compiler is intended for a single-metal CMOS process. Future extensions to other fabrication technologies are anticipated.

The Nebula compiler currently uses a one-dimensional layout architecture; as a result the layouts tend to be long and narrow. Future versions of the compiler will utilize a two-dimensional layout scheme, which will further improve layout quality.

Acknowledgements

I would like to thank my advisor, Prof. H.C. Card for his assistance and encouragement throughout this project. I would also like to thank my brother, Roland Schneider, for providing encouragement by expressing greater confidence in the eventual success of this project than I had myself.

Financial support from the Natural Sciences and Engineering Research Council of Canada and equipment loans from the Canadian Microelectronics Corporation are gratefully acknowledged.

Table of contents

Abstract iv

Acknowledgements v

List of figures viii

Chapter 1: Introduction 1

1.1 Automated IC design approaches 3

1.1.1 Approaches which improve efficiency of human designer 4

1.1.2 Highly automated IC design 6

1.2 Research work: the Nebula silicon compiler 11

Chapter 2: Overview of the Nebula compiler 12

2.1 Basic design goals 12

2.2 The present system: An overview 13

Chapter 3: Layout optimization 16

3.1 One-dimensional versus two-dimensional placement 16

3.2 Kernighan-Lin partitioning algorithm 18

3.3 Hierarchical clustering algorithm 20

3.4 Comparison of placement algorithms 22

3.5 Operation of layout optimization module 23

3.5.1 Input to layout optimization module 23

3.5.2 One-dimensional placement 24

3.6 Summary 25

Chapter 4: Layout database manager 26

4.1 Layout manager design goals 26

4.2 Layout manager design methodology 27

4.3 Layout manager structure 27

4.4 Technology description files 30

4.4.1 Miscellaneous and color files 30

4.4.2 Layer file 30

4.4.3 Object descriptor file 31

4.4.4 Wire description file 33

4.4.5 Design rule file 34

4.5 Summary of Layout manager functions 36

4.5.1 Object creation 36

4.5.2 Wiring of Objects 37

4.5.3 Deletion of objects and wires 37

4.5.4 Object movement 37
4.6 Summary 38

Chapter 5: High level layout module 40

5.1 Layout architecture of high level layout module 40
5.2 Layout procedure 42
5.2.1 One-dimensional ordering of nodes 42
5.2.2 Track assignment 42
5.2.3 Layout of design 44
5.2.4 Vertical compaction 46
5.2.5 Output of layout 46
5.3 High level layout module code requirements 47
5.4 Results of layout tests 47

Chapter 6: Future work 49

6.1 Short term changes and additions 49
6.1.1 Essential changes 49
6.1.2 Optional additions 50
6.2 Long range development 51
6.2.1 Additions to Nebula 52
6.2.2 Changes to Nebula 53

Chapter 7: Conclusion 55

Appendix I: Technology files for NT CMOS 1B 57

References 58

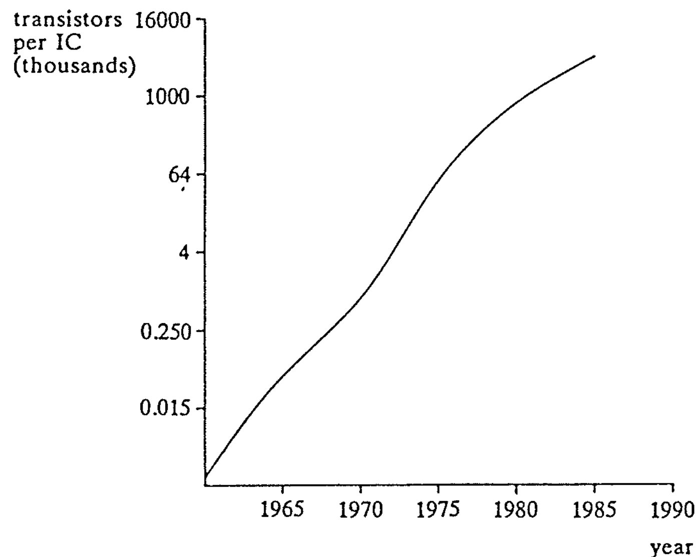
List of figures

1. *Growth of IC complexity* 1
2. *Design methodologies* 4
3. *Target architecture of the MacPitts data path compiler* 8
4. *Silicon compiler variations* 11
5. *Nebula compiler modules* 14
6. *Nebula program code by module* 15
7. *One-dimensional versus two-dimensional placement* 17
8. *Nets and Nodes* 18
9. *Kernighan-Lin swapping algorithm for n nodes in node_list* 19
10. *Clustering of nodes* 21
11. *The hierarchical clustering algorithm* 21
12. *Net-list of a NAND gate* 23
13. *Control flow in the optimization module* 25
14. *Optimization module program code* 25
15. *Objects, wires and ports* 28
16. *Northern Telecom CMOS IB process layers* 31
17. *Description of N-channel MOS transistor* 32
18. *Scaling of objects* 33
19. *Definition of N-diffusion wire* 34
20. *Format of design rule file* 35
21. *Moving objects* 38
22. *Layout manager code requirements* 39
23. *High level layout module architecture* 41
24. *Track assignment algorithm* 43
25. *Function of track assignment heuristics* 44
26. *Portion of an IC layout* 45
27. *Lines of code in high level layout module* 47
28. *Results of layout tests* 47
29. *Summary of short term changes and additions* 51
30. *Summary of long term changes and additions* 54

Chapter 1: Introduction

Integrated circuit (IC) complexity has been doubling every year; this implies that each year it is possible to put twice as many transistors on a single IC. The advance of fabrication technology has been going on at this rate for the past two decades. In the mid 1960's the number of transistors that could be placed on a single IC was limited to about 16, while today the most complex circuits contain from 500 000 to 1 000 000 transistors (see figure 1). This phenomenal advance in semiconductor technology caused the computer revolution of the 1970's and 1980's.

Figure 1. *Growth of IC complexity* [8]



One remarkable aspect of this industry, which has been synonymous with change and advance, is something which has remained fundamentally unchanged for the past two decades: the methodology that is used to specify, or design, the contents of an IC. Certainly there have been enormous advances in the equipment that is

used in the design process. Pencil and paper have given way to sophisticated computer-based color graphics workstations. However, until recently, the workstations have been used as a "better pencil and paper"; the computers were used to improve the productivity of the human designer without changing the basic IC design approach.

By the beginning of this decade it was becoming obvious to industry and university experts that the old approach to circuit design was being overwhelmed by the challenge of designing IC's with hundreds of thousands of transistors. The design cost of an IC is approximately proportional to the number of transistors it contains; thus the cost of designing a state of the art IC has been doubling every year. Presently it costs tens of millions of dollars to design a single complex IC when traditional IC design techniques are employed. As a result of this enormous design cost, the state of the art IC technology has only been applied to designs such as microprocessors and memories, which have a huge production volume over which the design cost may be amortized.

The enormous cost of IC design using traditional methods precludes the use of advanced fabrication technology for more specialized applications, where the total production volume might be a few hundred, or at most a few thousand IC's. The design of specialized low production volume IC's is now being addressed by new, radically different approaches to IC design. Typically these new design methodologies are heavily dependent upon computer processing power; in other words significant portions of the design effort are being shifted from the human designer to the computer. These new design techniques are fundamentally different from the traditional one, where the computer was relegated to the role of the "better pencil and paper".

The principal advantage of the new, highly automated IC design approaches is that the design cost is cut drastically; what would have taken many man-months using traditional IC layout techniques may be accomplished in only a few hours of computer time. As well, the computer generated design is less likely to contain errors which will result in a non-functional circuit. The price that is paid for these advantages is twofold: the computer designed circuit will occupy more silicon area because the layout will not be as clever as a human designed circuit, and secondly the resulting circuit will tend to have a lower maximum operating speed, again because the computer has a tendency to design less well than a human designer. However, neither of these problems outweigh the advantages of this method for the design of specialized, low production volume ICs.

The major criticism of computer designed ICs is that they require too much silicon area, as compared to a functionally identical human IC design. This argument is fallacious. As mentioned previously, IC complexity, or the number of components that may be placed on a single IC, has been doubling every year, as has the cost of designing the circuit to fill the expanding silicon area. However, the cost of manufacturing the IC, excluding design costs, has remained roughly constant for many years. The result is that the manufacturing cost *per transistor* has fallen drastically in the last decade, since the manufacturing cost per IC has remained roughly constant and the number of transistors per IC has increased a hundred-fold. This argument shows that highly automated area inefficient IC design will become more and more widespread as integration density continues to increase. An area penalty of two or four is equivalent to designing using the leading edge technology of one or two years earlier.

Interestingly, the design time of a large IC using traditional design methods is also one or two years; this implies that regardless of the design approach employed, the IC will be one to two years behind the leading edge of technology by the time it is ready for use. When viewed in this light, the penalty due to increased IC area required when automated layout is employed is no longer so dramatic. An additional advantage of automated layout is that the IC is ready for use much sooner than is possible when conventional design techniques are employed. For these reasons highly automated IC design is the key to exploiting ever improving fabrication technology for specialized, limited volume applications.

1.1. Automated IC design approaches

Myriad approaches to highly automated IC design have been explored; to date no single methodology has emerged which is clearly superior to its competitors. As a result, this is currently a field of active research. Some approaches are only modest departures from traditional labour intensive design, while others attempt to keep human intervention to an absolute minimum. Examples of the former include standard cell layout and the designer's assistant expert system approaches to IC design [7]. The latter category is typified by what is referred to as a silicon compiler, a computer program which takes a high level specification of an IC and generates the silicon layout, with limited human intervention [1],[4],[5],[12],[17]. Each of these design methods has

some merit; the choice of one over the other depends on the particular application that the designer must address. The following sections will give an overview of the competing approaches, with particular emphasis on the methods that are closest to the approach developed as part of this thesis work.

Figure 2. *Design methodologies*

approach	advantages	disadvantages	comments
traditional	compact, fast IC	design very costly	most commonly used IC design technique
standard cell	lower design cost than traditional	lower speed, less compact, restrictive	commonly used for medium sized designs
expert system	shorter design time, good quality design	complex software	research area
silicon compiler	very short design time	inefficient layout, complex software	research area, many variations

1.1.1. Approaches which improve efficiency of human designer

The basic precept of this methodology is that the human designer's participation is paramount at all stages of the design process; as a consequence the contribution of the computer to the design process is relatively small. As with the traditional techniques discussed previously, these approaches are quite labour intensive.

Standard cell design

Standard cell design involves the placement of previously defined standard cells. Each cell is a block of circuitry that was created by a human designer using traditional layout techniques. Examples of typical cells include simple logic functions, such as NAND and NOR gates, and also more complex cells, such as *flip-flops* and *adders*. The reason that these cells are referred to as *standard* cells is that they must

conform to a rigid set of guidelines, including restrictions on the height and width of the cell, where input and output connections can be made, and so on. The reason for the set of rigid cell design rules is that these restrictions simplify the task of the computer program which is used to create the interconnections between the cells, according to the specifications set forth by the human designer.

The advantage of standard cell design is that once a library of standard cells has been created, relatively little designer effort is required to design an IC, since computer software is used to do the interconnection, or wiring, between the standard cells which comprise the design. The principal difficulty with standard cell design is that it is not suitable for a broad range of applications. Further, standard cell design is probably not suitable for very large scale integration (VLSI), because of the rigid constraints involved in the design process. For example, a typical standard cell design system restricts the placement of cells to rows, with the inter-row space serving as a *wiring channel*, where the wires which connect the cells are placed. As a consequence of this restrictive layout methodology, standard cell design is not appropriate for VLSI applications.

Standard cell IC design has become commonplace in the last decade and has been accepted as a good approach for medium volume, moderate complexity IC design.

Expert system IC design

Expert system IC design may be viewed as the next logical step in the evolution of traditional design methods. The basic principle is to further improve the productivity of the human designer by applying artificial intelligence expert systems concepts to IC development. An *expert system* is a computer software system which has knowledge in a limited and application specific discipline. Thus, an IC design expert system attempts to mimic the expert IC designer [7]. An expert system IC design workstation should allow, at least in theory, a less experienced designer to achieve results which previously required an expert with many years of experience. The expert system could make suggestions to improve design, and warn the designer if the choices he is making are in some way contradictory. An expert using this type of "designer's assistant" would also benefit, since the software expert system would continually check the countless details of the design process. Generally, a well

constructed expert system will improve productivity and reduce the chance of designer error.

Expert system IC design has been making modest inroads into the market in recent years. However, this field is by no means mature. Substantial research effort is still required before systems are developed that are suitable for large scale IC design.

1.1.2. Highly automated IC design

This approach is a radical departure from conventional IC design, insofar as the human designer is being replaced by a computer and the associated software. Ideally the only human input required in the design of a new IC using this approach is at a very high level; typically the human designer would specify the function of the circuit. This *functional specification* is then translated into an IC design, without further human intervention. The process of translating from the functional specification to an IC design is referred to as *silicon compilation* and the software that performs the translation is called a *silicon compiler*. The term *compiler* stems from the rough analogy between computer software compilers and the IC function to layout translator. In both cases a high level language, or description, is translated into a low-level form.

In spite of the superficial similarities between the processes of software compilation and silicon compilation, the task of the silicon compiler is far more challenging. A silicon compiler must deal with the physical placement of the components which comprise the design. The software compiler does not have an analogous optimization task to perform. Because silicon compilation is very significantly different than software compilation it is currently an area of very active research. The computer program written as part of the thesis addresses a specific aspect of the silicon compilation process.

The first step in discussing silicon compilation it to define what is and what is not a silicon compiler. For the purposes of this discussion, a silicon compiler will be defined as follows:

> A silicon compiler is a software system which takes a description of a circuit that has little or no information about the appropriate physical location of the objects which comprise the design and translates it into a

physical layout suitable for fabrication, with little or no human intervention.

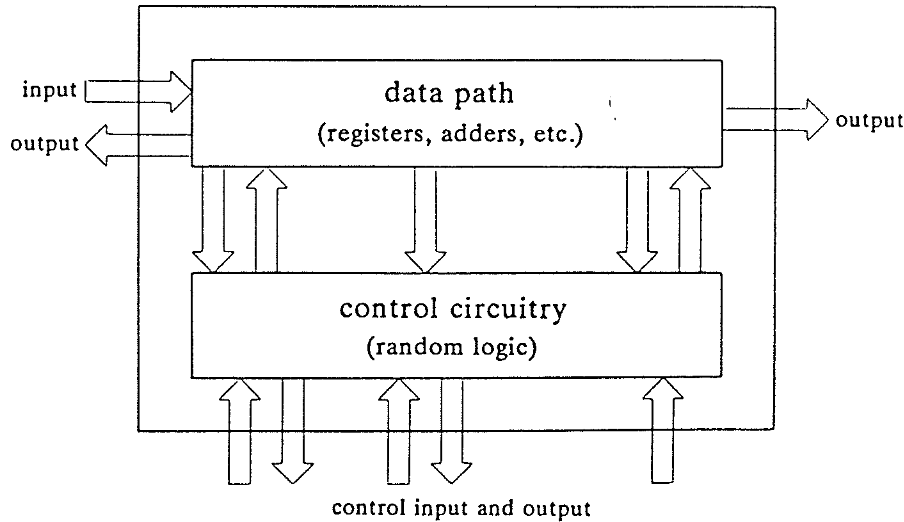
This definition is deliberately vague so that it encompasses all software systems which could conceivablely be considered silicon compilers. The difficulty is that this definition includes such a wide variety of IC design tools that sub-categories must be created. The arcane arguments as to what properties a true silicon compiler must possess will not be considered here.

Three categories of silicon compilers will be discussed in the following sections. Again there are many possible subdivisions, other than the ones chosen; however these three give a fair impression of the diversity of approaches to silicon compilation currently under investigation. The three categories are: the data path compiler, the special purpose compiler, and the random logic compiler. Each approach is suitable for certain applications. Some compilers presently under development employ a combination of these three methodologies. Examples will be used in the following sections to compare and contrast the features of these three approaches.

The data path compiler

Generally, to deal with the enormous complexity of unconstrained IC layout, silicon compilers are limited to a particular implementation strategy, or *target architecture*. The target architecture of a typical data path silicon compiler, the MacPitts compiler by Siskind, Southard and Crouch [12], is shown in figure 3. A competing data path compiler, Apollon [4], employs a similar architecture.

Figure 3. *Target architecture of the MacPitts data path compiler*



This architecture is similar to that of many general-purpose microprocessor ICs. As a result the range of applications for which the MacPitts compiler and its competitors are suitable is quite broad. However regardless of the application, the general structure of the IC will follow the architecture of figure 3.

Input to the MacPitts compiler is in a high level LISP-like language. The circuit behavior is described in terms of data path register operations, such as addition and subtraction. As well, condition tests may be used to control the operation of the IC. Another feature of the MacPitts compiler allows the designer to specify which operations are to be performed concurrently; thus the performance of the IC may be improved at the expense of additional silicon area.

The layout generated by MacPitts consists of two sections: the data path and the control section. The data path consists of adders, registers and other similar circuitry. The circuitry in the data path is constructed from single bit wide adders, registers, etc. joined together to form a complete circuit which is one binary word wide. The single bit building blocks are called *organelles*. Their structure must be specified by a human designer before the compiler can be used. However, many applications require the same set of organelles, so once a comprehensive library has been created no further human intervention is required. The control portion of the design is implemented using the Weinberger [16] array design technique.

IC layout tests with an early version of the MacPitts compiler have been encouraging, even though the resulting design occupied ten times the silicon area required by a comparable circuit designed using traditional IC design techniques [3].

In one case the total design time was reduced from six man-months to five man-days.

MacPitts and its competitors show considerable promise for implementing algorithms that can be represented as a data path with control circuitry, especially for low production volume applications.

Specialized silicon compilers

The data path architecture presented above is suitable for a fairly wide range of applications. Other silicon compilers restrict their architecture to a greater degree; as a result the compiler software is suitable for a very restrictive set of applications, but the design quality is improved. The FIRST [1] (fast implementation of real-time signal transforms) compiler is restricted to digital signal processing (DSP) applications. FIRST uses bit-serial operations to implement DSP functions. DSP applications do not require conditional connection of points within the circuit, so FIRST does not implement this feature, whereas the more general architecture implemented by MacPitts does require conditionals. Specialization enables the FIRST silicon compiler to outperform the MacPitts compiler for the limited set of problems for which it is suitable. This rule appears to be quite general; more specialized compilers tend to generate superior layouts in their particular field. Thus there is a tradeoff between the number of applications for which a compiler is suitable and the efficiency of the resulting layout.

Random logic compiler

A third variation on automated IC layout, the random logic compiler, encompasses a wide variety of approaches. This class of automated design software is typically used to automate the layout of unstructured parts of an IC design. For example, a microprocessor architecture typically consists of registers for data storage, read-only micro-sequencer program memory, and a significant body of unstructured, or random, control circuitry. The random logic compiler is intended for the latter aspect of IC design. Several competing approaches have been developed in recent years; two representative efforts are discussed below.

Shirakawa et al [11] developed a random logic layout system which translates Boolean logic expressions into an IC layout. The process consists of several steps: first, the Boolean expressions are minimized to reduce the number of transistors required to implement the logic function. Secondly, ordering of the transistors is performed to minimize the silicon area required for interconnection. In this scheme, the components which comprise the design are placed in a one-dimensional array. As a result, the IC layouts generated by this program tend to be quite long and narrow. The last phase of the design procedure consists of wiring the components together and then compacting the design.

Wolf et al [17] have also addressed the random logic compilation problem. The circuit description required by this compiler[1] consists of a net-list, instead of the Boolean functions used by Shirakawa el al, and the circuit layout is not limited to a one dimensional array. Another significant difference is that this compiler does not generate the IC layout directly, rather it generates a topological specification of the layout which includes information about the relative positions of the components which comprise the design but not their absolute position. The output of this program serves as input to another program, called a *compactor*, which takes the relative position information generated by the compiler and uses it to create the IC layout.

Summary

The preceding sections show that there are many approaches to silicon compilation and that there is no single preferred method; the tools employed must be suitable for the problem at hand. Figure 4 is a summary of automated layout approaches and applications for which each is suitable.

[1] The compiler developed by Wolf et al is called *Dumbo*.

Figure 4. *Silicon compiler variations*

Category	Examples	Applications	Comments
data path	MacPitts, Apollon	applications suited to microprocessor style architecture	fairly general
special purpose	FIRST	limited to particular application	better layout than data path in field of specialty
random logic	Nebula (this work), Dumbo	random, or unstructured logic of IC	arbitrary circuit layout

1.2. Research work: the Nebula silicon compiler

The current version of the Nebula silicon compiler developed as part of this thesis work best fits into the category of random logic compilers. However, an attempt has been made whenever possible to design the software in a highly flexible manner, so that future enhancements may be added which incorporate some of the concepts of the data path and special purpose silicon compilers.

Work on this project was begun in February 1985, with the development of general design goals. By September a preliminary working version of the compiler had been completed; October and November were spent improving the program's performance. All design, coding and testing was performed by the author.

Chapter 2: Overview of the Nebula compiler

2.1. Basic design goals

Four fundamental design goals were developed at an early stage in the design process:

1. The compiler should be able to lay out an arbitrary design.
2. The modules which comprise the compiler should be as independent as possible.
3. The compiler should be able to handle different fabrication technologies.
4. A preliminary working version of the compiler should be ready as soon as possible.

The first goal, the ability to generate layouts for an arbitrary design, was the most important consideration. Most of the silicon compilers described in the previous sections do not have this feature, the only exception being the random logic compiler by Wolf et al [17]. A substantial penalty in layout area was anticipated as a result of this decision. However, a silicon compiler which is suitable for use at a university requires considerably more flexibility than one used in industry for the layout of fairly mundane designs. University research involves experimentation with radical new architectures for which a data path compiler such as MacPitts is inappropriate.

A second design consideration was the structure and connectedness of the modules which make up the compiler. It was established early in the development of this system that because of the experimental nature of this program, design errors would occur which would require complete redesign of some aspect of the layout

program. To minimize the trauma associated with major shifts in approach after months of work, the program was deliberately designed as several loosely coupled parts. As a result major changes affected only one module, thus reducing the work required to implement new ideas. The loosely coupled structure has one other advantage; future additions to the compiler may simply involve adding another module, without major reworking of existing code.

The third design goal was to make the compiler as technology independent as possible. This is a crucial goal in an industry where more changes occur in one year than would in a decade in most industries. All the silicon compilers discussed in chapter 1 already suffer from technological obsolesence, since all are restricted to the NMOS fabrication process. Newer versions of these compilers may be suitable for a wider range of fabrication technologies. The current version of Nebula is intended for a CMOS process with a single level of metal interconnect. This fabrication technology was chosen because it is the process currently being used at the University of Manitoba. However the program has been designed in such a way that a switch to another fabrication technology, such as single-metal NMOS, could be accomplished with ease. Support of other technologies, including double-metal processes and bipolar technologies is also possible without major program redesign.

The final design constraint was the total time available for the completion of this project. The field of silicon compiler design is by its nature very open-ended. There is always the possibility of adding one more feature which will improve layout efficiency or make the program easier to use. Due to time constraints the current version of Nebula is a minimum system; substantial improvements are planned for the future. Chapter 6 is a discussion of proposed future development.

2.2. The present system: An overview

The current version of Nebula requires a net-list description of the circuit for which the layout is to be generated. A *net-list* is a description of how the components which comprise the layout, in this case transistors, are interconnected. There is no information in the input to this program as to what arrangement of the transistors will result in an efficient layout.

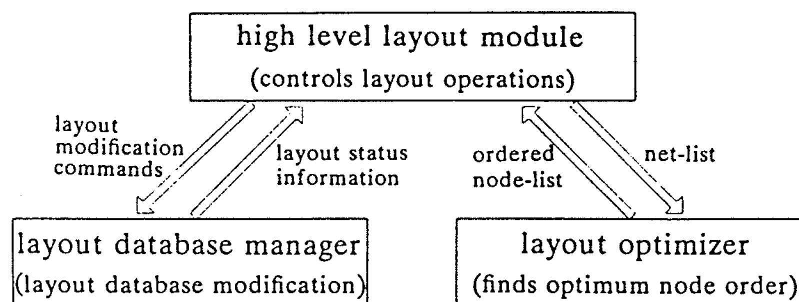
The output of the Nebula compiler is a Caltech Intermediate Form (CIF) description of the layout, an output format which is widely accepted by fabrication

facilities. The layout may also be displayed on a color graphics terminal for visual inspection. The current version of the program consists of three parts:

1. The layout optimizer module (LOM)
2. The layout database manager (LDM)
3. The high level layout module (HLLM)

The layout optimization module takes the net-list description of the circuit and uses the net-list connection information to establish the appropriate relative locations of the transistors which comprise the design to minimize wire area in the layout. The layout database manager is the portion of the program that deals with the details of the particular technology being used. Information about the spacing required between objects which make up the design is handled by this part of the program. Lastly, the high level layout module uses the the other two modules to perform the actual layout. The inter-relationship between these three modules is illustrated in figure 5.

Figure 5. *Nebula compiler modules*



The Nebula random logic compiler was written in the C programming language. C is well suited to this application and is readily portable to many computer systems. The current version consists of approximately 14 000 lines of program code and 450 lines describing the CMOS technology being used. Figure 6 is a breakdown of the code by functional module.

Figure 6. *Nebula program code by module*

Program Module	Lines of code
layout optimizer module	3111
layout database manager	8570
high level layout module	2399
total	14080

As these figures indicate, even a modest venture in the automated IC design field requires a substantial effort. A complete silicon compiler suitable for commercial use would probably require two to three times this amount of code.

The following chapters will describe in detail the functions of the three modules which make up the Nebula compiler: chapter 3 deals with layout optimization, chapter 4 with the layout database manager, and chapter 5 with the high level layout module.

Chapter 3: Layout optimization

The layout optimization module was the first part of the random logic compiler to be developed. The reason for this is simple; without an adequate method of ordering the components which make up the design, layout is not possible. The general problem of finding an optimum arrangement of objects which are interconnected in some arbitrary way is crucial in the automated layout field. Consequently it has been the subject of a large amount of research activity in the past few years.

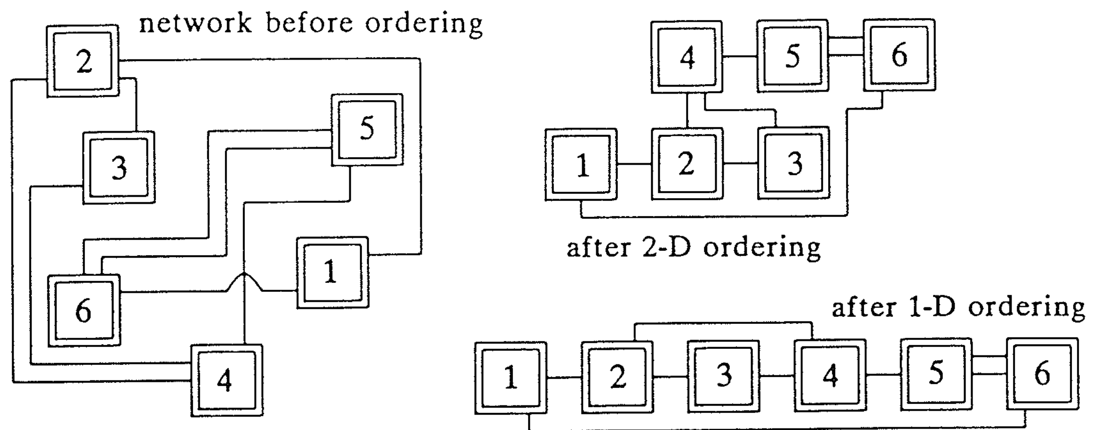
Finding the optimum arrangement of interconnected objects has been shown to be an NP-complete problem [6]. Obviously the brute force approach of trying all possible arrangements is not feasible. Even a relatively modest problem consisting of 30 components would require the services of all computers on Earth for a period of time longer than the present age of the Universe, since problem complexity increases as n factorial, where n is the number of components. Clearly this application requires an efficient heuristic algorithm which yields reasonable results for interconnections of components that occur in real IC designs.

3.1. One-dimensional versus two-dimensional placement

The first design decision was whether to pursue one-dimensional or two-dimensional placement optimization. One-dimensional placement involves placing the components which make up the design in a linear array in an order that minimizes some cost function. Intuitively, the optimum arrangement would tend to have strongly connected components close together in the one-dimensional array. Two-dimensional placement algorithms also tend to group strongly connected components together. However, the optimization problem is far more complex, and therefore computation intensive than the comparable one-dimensional problem, because of the

extra degree of freedom. Figure 7 illustrates one and two-dimensional placement.

Figure 7. *One-dimensional versus two-dimensional placement*



Although two-dimensional placement is in general superior to one-dimensional placement, for simplicity the current version of Nebula is restricted to one-dimensional placement. Future extension to two-dimensional placement is anticipated. Two one-dimensional placement algorithms have been implemented: the Kernighan-Lin partitioning method [14], and a heuristic clustering method developed as part of this thesis project.

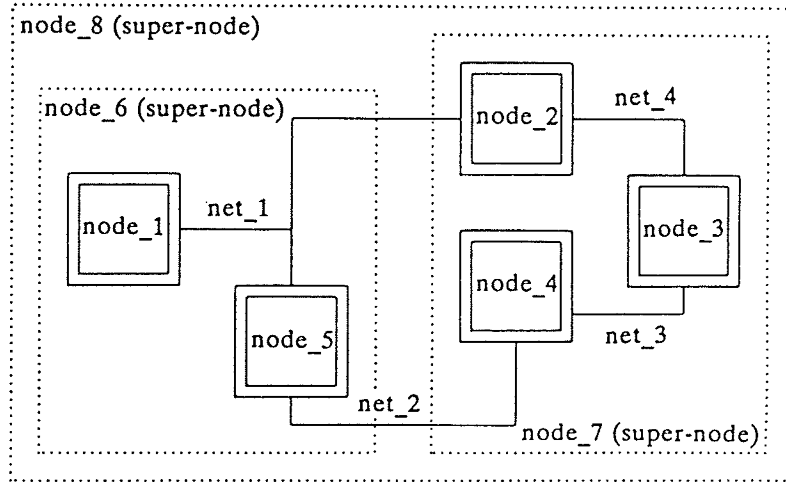
Before these two placement algorithms can be described, some terminology must be defined:

Net: In the following discussion a *net* is a set of wires that connects together all points in a circuit that are supposed to be electrically connected.

Node: A *node* is any component which is part of the design. A node may consist of a single transistor, a group of transistors or a group of groups of transistors.

Figure 8 illustrates these definitions.

Figure 8. *Nets and Nodes*



In figure 8, four nets and eight nodes are illustrated. Note that *net_1* connects three low-level nodes. In general, a net may connect any number of nodes. The nodes of figure 8 fall into two categories: primitive, or low-level nodes, and super-nodes which may consist of several primitive nodes, as well as other super-nodes. Thus the concept of nets and nodes is quite general, and is not simply a direct representation of the net-list that the program received as input. The *hierarchical clustering method* employed in this thesis makes use of the generalized concept of nets and nodes.

3.2. Kernighan-Lin partitioning algorithm

This algorithm was originally used in LGEN [14], an early automated placement program developed by Bell Labs in the United States. The algorithm was adapted for this application and incorporated into the Nebula compiler. The basic algorithm is presented in pseudo-code form in figure 9.

Figure 9. *Kernighan-Lin swapping algorithm for n nodes in node_list*

```
swap_order_nodes:
    {
    divide node_list into left and right sides, with n/2 nodes on each side
    do
        {
        copy node_list into save_list
        i = 1
        unmark all nodes in node_list
        while i <= n/2
            {
            find pair of unmarked nodes whose exchange maximizes swap_benefit
            set node_pair equal to pair of nodes found
            do trial swap of node_pair
            save node_pair and swap_benefit in swap_list
            mark pair of nodes as swapped
            i = i + 1
            }
        copy save_list into node_list
        find j, such that total_swap_benefit = Σ(i=1..j) swap_benefit_i is maximized.
        if total_swap_benefit > 0, perform first j swaps from swap_list on node_list
        } while total_swap_benefit > 0
    }
```

The algorithm listed in figure 9 takes a list of n nodes in *node_list* and swaps node pairs from the right and left sides of the list such that *swap_benefit* is maximized. Since this is a one-dimensional ordering of the nodes which comprise the design, *swap_benefit* is the net reduction in the number of crossings over the mid-point, or pivot point, between the left and right sides of *node_list*. Thus, two nodes are swapped if this results in a net decrease in the number of wires crossing the pivot point. Intuitively, the number of wires crossing any point in the one-dimensional array is a reasonable measure of the quality of a given order. The algorithm presented in figure 9 must be applied recursively by further dividing the left and right portions of the list into two, until the number of wires crossing all points in the linear array have been minimized.

An interesting feature of this algorithm is the trial swap phase, before the permanent swap of two nodes is performed. A more straightforward algorithm would simply evaluate all the possible swaps, and then perform the one that provides the largest decrease in the number of wires crossing the pivot point. Exchanging would continue until no further benefit is achieved by swapping two nodes. The problem with this simplified algorithm is that it tends to "get stuck" at local minima in the function being optimized. The trial swap is an attempt to

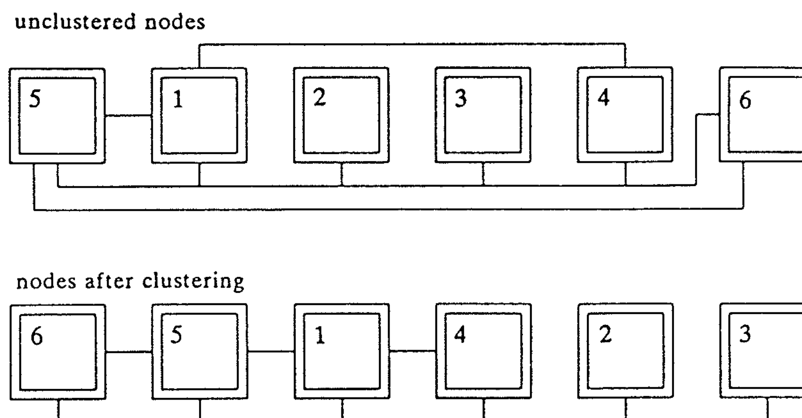
avoid getting stuck in local minima during the search for the global minimum.

This algorithm makes intuitive sense, and as a result one would expect it to do an adequate job of ordering the nodes efficiently. Preliminary tests established that this algorithm works well with small problems, where the number of nodes is at most a few dozen. However, when a design with 200 nodes was processed the quality of the ordered node list was no better than that of a randomly ordered list of the same nodes. The precise reason for the algorithm's failure is rather subtle; it appears that the notion of dividing the nodes into two arbitrary groups and then swapping the nodes to reduce the number of wire crossings between the two groups is not sophisticated enough to untangle a large intricate network of nodes. The problem of ordering the nodes was finally solved in the present work by developing a *hierarchical clustering* algorithm as an alternative to the Kernighan-Lin algorithm. The next section describes this algorithm.

3.3. Hierarchical clustering algorithm

The hierarchical clustering algorithm attempts to emulate, in a crude way, the process that a human designer would use to find the optimum order of a linear array of nodes. The basic principle upon which this algorithm is based is the following: nodes that are joined by nets that connect to few other nodes should be kept together in the linear array. Figure 10 illustrates this principle.

Figure 10. *Clustering of nodes*



Application of the clustering heuristic results in the grouping of *node_6*, *node_5* and *node_1* since each of these nodes is connected to another node by a net with only two nodes connected to it. Clearly, attempting to cluster the nodes on the basis of connections to *net_6*, which connects to all six nodes, would be absurd, since this net must be connected to all six nodes in any case, and must therefore run the full length of the linear array. The complete algorithm is presented below.

Figure 11. *The hierarchical clustering algorithm*

```
cluster_order_nodes:
    {
    i = 1
    while number of nodes > 1
        {
        group nodes joined by nets connecting i nodes
        combine each node group into one super-node
        store new super-nodes at level i
        i = i + 1
        }
    while i > 0
        {
        find optimum order of sub-nodes of nodes at level i
        i = i - 1
        }
    }
```

Figure 11 shows that this optimization scheme consists of two phases: in the first phase the nodes are grouped into larger and larger groups until all the

nodes have been grouped into one super-node. Note that nodes connected by local nets, that is nets that are only required by a few nodes, are grouped first, followed by nodes that require nets that are used more globally, and so on.

After the first phase has been completed, the order of the sub-nodes of each super-node is optimized. This process begins with the highest level super-node, and works its way down to the low-level nodes. The reason that phase two begins at the top is that this results in a global ordering of large blocks of the design, followed by more localized ordering. The criteria that is used for ordering the sub-nodes at each level is based upon the layout strategy that is ultimately to be used to do the wiring between the nodes. Thus the ordering of the sub-nodes may be considered optimal in a limited sense.

3.4. Comparison of placement algorithms

Tests on designs ranging in size from 4 to 600 transistors have shown that the hierarchical clustering algorithm is as good or better than the Kernighan-Lin algorithm in all cases. Furthermore, the quality of the placement generated by the Kernighan-Lin algorithm tends to depend on the order of the input data, whereas the hierarchical ordering algorithm shows no such dependency. No IC design has been found to date for which the clustering algorithm breaks down in the way the Kernighan-Lin algorithm does for large designs. Intuitively it seems unlikely that this problem could arise with the clustering algorithm, no matter how large the problem size. A final advantage of the clustering algorithm is that the degree of optimization is adjustable; this feature makes it possible to reduce the optimizer run time for large designs by relaxing the optimization criteria. This is an important feature if the program must handle large designs. The Kernighan-Lin algorithm, besides becoming ineffective when applied to large problems, also requires many hours of CPU time. As mentioned previously, placement of layout components has been actively researched for many years. Some typical approaches are discussed in [2], [11], [14], and [17]. Comparison of the newly developed hierarchical clustering algorithm with these competing schemes was not pursued. Since the hierarchical clustering algorithm did a satisfactory job of layout optimization, further investigation in this area was not required.

In future versions of the Nebula compiler, the one-dimensional clustering algorithm presented here could be extended to two dimensions. Note, in this connection, that the clustering algorithm of figure 11 does not require the nodes to be in a linear array. This is because the clustering algorithm as presented is essentially dimension independent. The difficulty in extending the clustering algorithm to two dimensions instead arises *after* the clustering operation has grouped nodes that should be placed near one another. In the two-dimensional case, there are a tremendous number of layout alternatives. Dealing with this complexity is a challenging problem.

3.5. Operation of layout optimization module

3.5.1. Input to layout optimization module

As mentioned previously, input to the layout optimization module is in the form of a net-list. A sample net-list is presented in figure 12.

Figure 12. *Net-list of a NAND gate*

d	*net_Vdd*	i				
d	*net_GND*	i				
d	*net_a*	i				
d	*net_b*	i				
d	*net_out*	o				
p	*net_b*	*net_Vdd*	*net_out*	5	11	*trans_0*
n	*net_b*	*net_out*	*net_6*	5	11	*trans_1*
n	*net_a*	*net_6*	*net_GND*	5	11	*trans_2*
p	*net_a*	*net_out*	*net_Vdd*	5	11	*trans_3*

There are two types of lines in this input file: transistor description lines and net description lines. The format of the transistor description lines is:

transistor_type gate_net source_net drain_net length width name

The *n* and *p* refer to N-channel and P-channel transistors respectively. The transistor length and width are entered in the current physical design units. The net description lines have the following format:

d *net_name characteristics*

where the currently supported characteristics are *input (i)*, *output (o)* and *internal (x)*. The net description line is used to specify the characteristics of a particular net in the circuit.

The choice of a simple net-list as an input format may appear to be a poor choice for a silicon compiler: certainly it is not a glamorous interface to present to the user. Nevertheless the net-list approach has several advantages. The most important reason that net-list input was chosen is that a net-list is a very simple interface, and yet allows the user to specify any arbitrary design, since any interconnection of transistors may be represented as a net-list of the form in figure 12. Some compilers use Boolean expressions as an input language, while data path compilers, such as MacPitts, use a high level functional description language. Although these approaches may be more convenient in some applications, neither is as general as the humble net-list. Creating a net-list specification of a circuit is not particularly difficult; typically a schematic editor would be used to interactively enter a net-list. Alternatively, the net-list could be derived from a hardware description language functional specification of the circuit [12].

3.5.2. One-dimensional placement

After input, the one-dimensional ordering algorithms described previously are applied to the net-list circuit specification. The optimization algorithms may be used singly or together to generate an ordered node list. The ordered node list gives the sequence in which the transistors should be placed to minimize layout area. At this stage, the layout is still fabrication technology independent: nowhere in the optimization process was technology-dependent information required.

3.6. Summary

A block diagram of the operation of the placement optimization module is given below.

Figure 13. *Control flow in the optimization module*

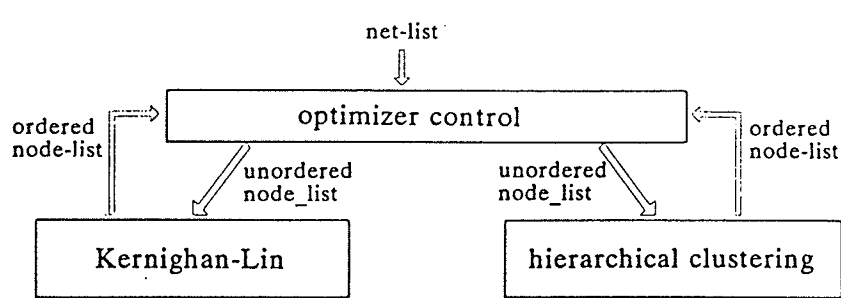


Figure 14 is a summary of the C code required for each of the sections of this module

Figure 14. *Optimization module program code*

Module section	Lines of code
net-list input	148
Kernighan-Lin optimizer	586
hierarchical clustering optimizer	464
general	1913
total	3111

The output of the optimization module is passed to the high-level layout module for translation into an IC layout. Chapter 4 deals with the layout database manager, and chapter 5 describes the high level layout module.

Chapter 4: Layout database manager

The layout database manager is responsible for maintaining all layout information, including the type, position and size of all objects in the layout. All changes to the layout are made through calls to the database manager functions. Similarly, information concerning the present state of the layout comes from the database manager. The reason the approach of an independent standalone layout manager was adopted is twofold: this approach made it possible to design and test the layout manager separately from the rest of the program, and secondly, having distinct modules whose interactions are not tightly coupled reduces the overall complexity of the program.

4.1. Layout manager design goals

The primary objective in designing the layout manager was to develop a highly versatile layout database manager module. As a result the layout manager is not limited to one-dimensional layout, so when future versions of Nebula graduate to two-dimensional placement, the layout manager will not require extensive reworking. The same is true for future changes in fabrication technology; the present version is sufficiently general to deal with NMOS and CMOS, single or double-metal processes, and bipolar processes such as I^2L. Naturally the layout manager is also able to handle the variations between fabrication processes employed by different IC manufacturers. In all these cases, the layout manager can be configured for the new fabrication process through a set of six files which describe the characteristics of a particular technology. The contents of these technology descriptor files is described below.

4.2. Layout manager design methodology

When designing a layout system, either for manual or automatic layout, one of two common approaches is normally used. The first approach deals with the physical layout problem at the mask level: in other words the design is represented as a collection of different types of polygons. When all the polygons of a particular type are combined, the result is a description of one mask in the IC fabrication process. In a typical CMOS process, approximately ten such mask layers are employed.

The second approach to the representation of an IC layout is more sophisticated. Instead of dealing with the design at the mask level, and dealing with polygons as the objects that are manipulated, this second approach deals with complete descriptions of transistors, contacts, and the other items which comprise the layout [10]. Clearly the second approach is more natural, both for the human designer in the case of a manual layout system, and for the high level portions of an automated layout system. The second approach was employed in the layout database manager developed for Nebula.

4.3. Layout manager structure

Before further discussion of the layout manager is possible, a few terms must be defined. Basically, the layout manager deals with three entities: objects, wires and ports.

Objects:

> *Objects* are the parts from which a design is constructed. Examples of objects include transistors and contacts; each object has the appropriate mask layers associated with it to create a functional transistor or contact. Figure 15 illustrates these concepts.

Wires:

> *Wires* are used to make the connections between objects. Each wire has all the appropriate layers associated with it to make a functional wire. Depending on the particular fabrication process, wires come in several varieties. Figure 15 illustrates these concepts. In the present version of the layout manager, wires always

run either vertically or horizontally: no bends are allowed in a wire. If a *jog* is required in the path connecting two objects, a special object called a *pin* is placed at the corner where the bend is required and two separate wires are used to make the connection. If multiple jogs are necessary, additional pins are used. Thus the restriction that wires must either run horizontally or vertically does not restrict design freedom.

Ports:

Each object defined for a particular technology has *ports* defined on its surface where a particular type of wire is allowed to connect. Wires can only connect to an object at a port intended for that type of wire.

Figure 15. *Objects, wires and ports*

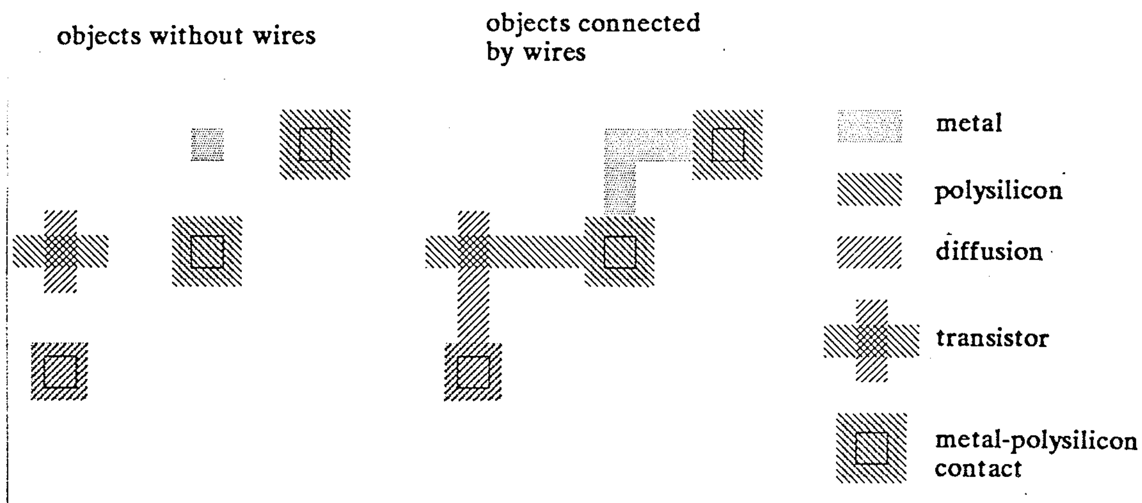


Figure 15 illustrates the basic concepts of objects, wires and ports. In this example, the transistor, metal-polysilicon contact and the metal-diffusion contact are *objects*, and the diffusion and polysilicon paths that join them are *wires*. The points where the wires are connected to the objects are *ports*. This example illustrates the importance of the concept of ports which are restricted to a particular type of wire: otherwise it would be possible to connect the polysilicon gate to the metal-polysilicon contact with a wire of type *diffusion*. The result would be a non-functional circuit. Disallowing invalid operations, such as connecting objects with the wrong type of wire, is an example of the powerful general principle of correctness by construction. Stated simply, *correctness by construction* means that any design created based on a set of rules precludes the chance of creating a non-

functional design[2]. The alternative to this approach is to create the design first, and then verify that no rules have been broken afterwards. Generally correctness by construction is the better approach for automated IC layout, due to the extreme difficulty of repairing an error once it has been incorporated into a large layout.

The basic operations that the layout manager handles are the placement of objects and the connection of objects with wires. All other layout manager operations are built upon this simple foundation.

Besides simply keeping track of the physical locations of objects and wires, the layout manager keeps complete connectivity information, based upon how objects are interconnected with wires. In the example of figure 15, the polysilicon gate of the transistor, the polysilicon and metal wires, the metal-polysilicon contacts, and the metal pin are all electrically connected. Similarly, the metal-diffusion contact, the diffusion wire, and the lower diffusion portion of the transistor are all connected. Connectivity information is vital in the automated layout process, since the minimum allowable spacing between objects and wires depends on whether they are electrically connected.

The final basic function of the layout manager is to assure that the layout operations requested by the high level layout module do not violate any of the design rules of the fabrication technology being used for the current layout. *Design rules* are a set of guidelines which must be followed when laying out an IC, either manually or automatically. These rules vary considerably from one IC manufacturer to another. Design rules specify such things as the minimum allowable spacing between objects which make up the design, as well as specifying how objects such as transistors and contacts are constructed. For example, in the Northern Telecom CMOS 1B technology currently being used at the University of Manitoba, the minimum spacing between two unconnected areas of polysilicon is 5 microns. If a smaller spacing is used, then inaccuracies in the manufacturing process may result in an accidental connection between the two polysilicon areas. The spacing rules are applied by the layout manager during all layout operations.

---

[2] Of course, the chip may fail due to a fabrication fault or defect. Designs which have provisions to survive these "errors" are called *fault tolerant*.

4.4. Technology description files

Six files are used to describe the fabrication technology to the layout manager: the *layer*, *color*, *object*, *wire*, *design rule*, and *miscellaneous* files. Each of these files describes some aspect of the fabrication technology. The following sections discuss the contents of these files. Appendix 1 gives the complete set of technology description files for the Northern Telecom CMOS 1B fabrication process.

4.4.1. Miscellaneous and color files

The *miscellaneous* file contains general information about the technology, including its name, and the scaling factors to be used in interpreting the information contained in the other files. The *color* file specifies what colors are to be used for the various masks, when the layout is displayed on a graphics terminal.

4.4.2. Layer file

The *layer file* is a description of each of the process masks of the technology being defined. Figure 16 gives a partial listing of the layers for the Northern Telecom CMOS 1B process.

Figure 16. *Northern Telecom CMOS 1B process layers*

layer	{	metal	1	001	transparent	CM	}
layer	{	polysilicon	2	002	transparent	CP	}
layer	{	n+	16	020	transparent	CNP	}
layer	{	p+	8	010	transparent	CPP	}
layer	{	device_well__(n)	4	004	transparent	CF	}
layer	{	device_well__(p)	4	004	transparent	CF	}
layer	{	contact_cut	42	077	opaque	CC	}
layer	{	n_guard	44	077	opaque	CNG	}
layer	{	p_guard	45	077	opaque	CPG	}
layer	{	p_well	46	077	opaque	CPW	}
layer	{	passivation	41	077	opaque	CG	}

Each line in the file defines one of the process masks. The format is:

layer { *layer_name draw_info_1 draw_info_2 draw_info_3 CIF_code* }

where *draw_info* is information required for displaying on a color graphics terminal, and *CIF_code* is the code that is used by the IC manufacturer to identify this process mask layer.

4.4.3. Object description file

Each object that is to be used in the layout process has an entry in the *object file*, specifying the mask layers from which the object is composed, the position and characteristics of the object's ports, the function of the object, its name, and its minimum size. A typical description of an N-channel transistor is given in figure 17.

Figure 17. *Description of N-channel MOS transistor*

```
                port_type { metal        metal              }
                port_type { polysilicon  polysilicon        }
                port_type { n_diffusion  n_device_well__(n) }
                port_type { p_diffusion  p_device_well__(p) }
                port_type { p_well       p_well             }

object
    {
    function { n_transistor          }

    port     { polysilicon          poly         0   0 -5  150 0    0  5 -150 }
    port     { n_diffusion          diff_top    -5 150  5 -110 5 -150  5 -110 }
    port     { n_diffusion          diff_bottom -5 150 -5  110 5 -150 -5  110 }

    layer    { device_well__(n)     diff_top    -5 150  5 -150 5 -150  5 -110 }
    layer    { device_well__(n)     diff_bottom -5 150 -5  110 5 -150 -5  150 }
    layer    { device_well__(n)     @none       -5 150 -5  150 5 -150  5 -150 }
    layer    { polysilicon          poly        -5 100 -5  150 5 -100  5 -150 }
    layer    { transistor           @none       -5 150 -5  150 5 -150  5 -150 }
    layer    { p_well               @none       -5  80 -5   80 5  -80  5  -80 }
    layer    { p_guard              @none       -5  30 -5   30 5  -30  5  -30 }
    layer    { n_guard              @none       -5   0 -5    0 5    0  5    0 }

    name     { transistor_well__(n) }
    size     { 350  350 }
    offset   { 150  150  150  150 }
    }
```

Note that three ports are defined, one for each of the *gate*, *source*, and *drain*. The N-channel transistor defined in figure 17 is composed of eight rectangles consisting of various process masks. Each of the layer specification lines defines a rectangle of a specific size consisting of a particular mask layer. The eight numbers that make up the layer specification serve two purposes: they give the relative positions of the eight rectangles which make up the transistor, and secondly they define how the rectangles grow when a non-minimum sized object is used in a design. Figure 18 illustrates the principle of object scaling.

Figure 18. *Scaling of objects*

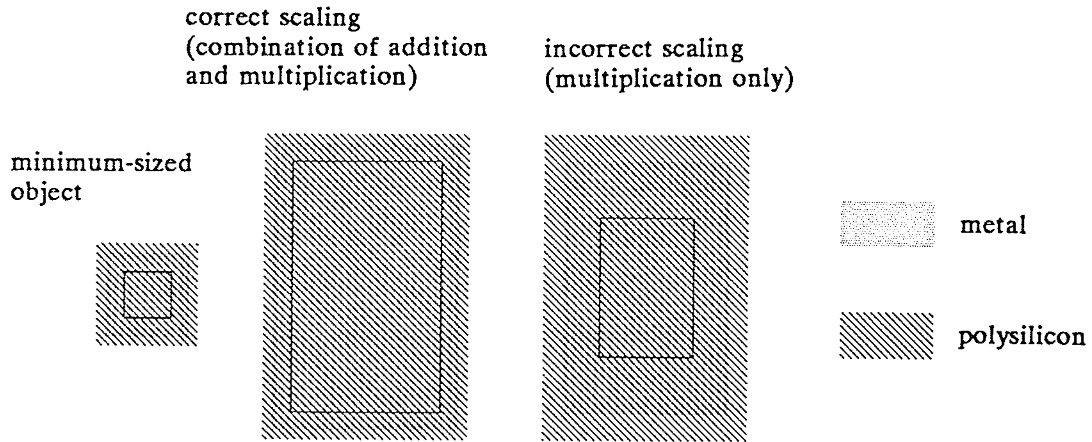


Figure 18 shows that increasing the size of the object by multiplying all its coordinates by a scaling factor is not the correct way to increase the size of an object. Rather, the scaling involves both multiplication and addition.

4.4.4. Wire description file

The *wire description file* is similar in format to the object definition file. Figure 19 is the definition of an N-diffusion wire from the Northern Telecom CMOS 1B process.

Figure 19. *Definition of N-diffusion wire*

```
wire
    {
    function { n_diffusion        }
    name     { n_device_well__(n) }
    width    { 270                }
    offset   { 110                }

    layer    { p_guard            30  @unconnected }
    layer    { n_guard             0  @unconnected }
    layer    { p_well             80  @unconnected }
    layer    { device_well__(n)  110  @connected   }
    }
```

As with the object specification of figure 17, the diffusion wire is composed of layers, and the function of the wire is specified in the description. The comments made above about the scaling of objects also apply to the scaling of wires.

4.4.5. Design rule file

The *design rule file* specifies the spacing of the mask layers which make up the objects and wires. Two cases exist: layers that are electrically connected, and layers that are not electrically connected. Typically, connected layers are allowed to be closer together than unconnected layers. Figure 20 illustrates the format of the design rule file.

Figure 20. *Format of design rule file*

```
unconnected
          {
          design_rule { metal              metal              60  }
          design_rule { polysilicon        polysilicon        50  }
          design_rule { polysilicon        device_well__(n)   40  }
          design_rule { polysilicon        device_well__(p)   40  }
          design_rule { n+                 device_well__(n)   40  }
          design_rule { p+                 device_well__(n)   70  }
          design_rule { device_well__(n)   device_well__(n)   70  }
          design_rule { device_well__(p)   device_well__(p)   70  }
          design_rule { device_well__(n)   device_well__(p)   170 }
          }
connected
          {
          design_rule { polysilicon        device_well__(n)   40  }
          design_rule { polysilicon        device_well__(p)   40  }
          design_rule { n+                 device_well__(n)   40  }
          design_rule { p+                 device_well__(n)   70  }
          design_rule { device_well__(n)   device_well__(p)   170 }
          }
```

The design rules are specified as minimum spacings between the layers defined for the technology. Since all objects and wires are made up of these layers, these design rules define the minimum spacing between all objects and wires.

By setting up the information in these six files, the layout manager can be programmed for a wide variety of technologies. Typically one set of technology description files would be created for each fabrication facility that is going to be used. Then the designer selects one of the technologies, and Nebula reads in the corresponding technology description file.

4.5. Summary of Layout manager functions

The preceding discussion has outlined how the layout manager is structured internally, without touching on the subject of how the layout manager is used by the rest of the layout program. This section will describe some of the functions that the layout manager provides.

In the present version of Nebula, the layout manager stores all information pertaining to the position of objects in the layout, and all wires which connect the objects together. Because of the unified way in which wires and objects are being handled by the layout manager, details regarding the particular fabrication technology do not affect the operation of the high level layout module. In fact the majority of the layout manager module is also not concerned about the shape or composition of the objects with which it deals; all the necessary information is provided by the position of the ports on the objects, the list of wires which may connect to each port, and the design rules concerning how close together objects and wires may be placed. The technology independent nature of the majority of the layout manager greatly simplified its design and implementation.

The layout manipulation commands available from the present version of the layout manager may be broken down into the following categories:

1. Object creation
2. Wiring of objects
3. Deletion of objects and wires
4. Movement of objects

4.5.1. Object creation

The layout manager has functions for creating objects anywhere in the design, with arbitrary size, rotation and transposition. When an object is created, it is inserted into a set of linked lists which link together all objects and wires in the layout. These linked lists are used to find the nearest neighbors of a particular object for operations such as design rule verification.

4.5.2. Wiring of Objects

Many ways are provided to wire objects together; however, the basic operations are always the same. The layout manager is given two objects, the type of wire to use to connect them, and the port on each object to use for the wire. After the connection operation is performed, the connectivity information is updated in the affected portions of the design. Variations on object wiring include allowing the layout manager to choose the type of wire to use for the connection of two objects. The choice of wire type will then depend on whether ports exist on the objects for a particular wire type, and whether using that type of wire will cause a design rule violation. Another variation involves connecting an object to any point in the layout that is electrically connected to a particular object. These high level wiring functions are essential to the efficient implementation of the high level layout module.

4.5.3. Deletion of objects and wires

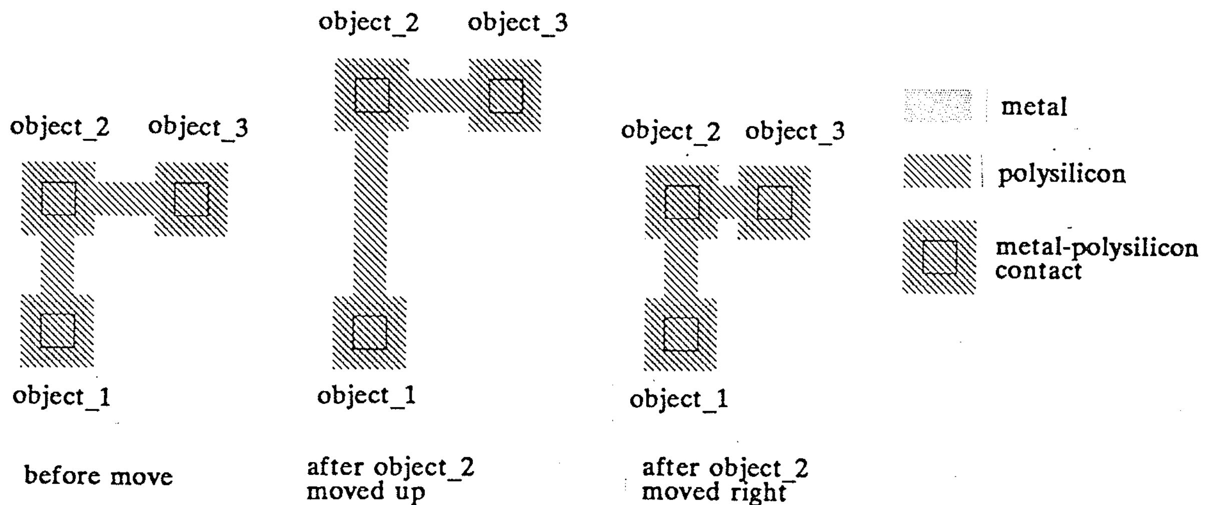
Any wire or object in the design may be deleted from the layout. When an object is deleted, all wires connected to the object will also be deleted, since a wire only exists for the purpose of connecting two objects. Again, when an object or a wire is deleted, connectivity information throughout the affected portions of the layout is updated.

4.5.4. Object movement

The layout manager has a number of functions for moving objects, or groups of objects. Wire movements are not allowed, since wires are not independent of the objects that they are connecting. When an object is moved, all connected wires must be modified to maintain the connections. If the object is being moved in the same direction that the wire runs in, the wire will stretch to maintain the connection. If the move is in the other direction, the object at the other end of the wire will be forced to move to maintain the connection. The movement of the second object may force a third to be moved, and so on. Figure 21 illustrates object movement. A

variation on the move function allows for a group of objects to be moved as far as possible in a specified direction without causing a design rule violation. This particular feature is essential for design compaction operations, where the objects which make up the design are pushed as close together as possible.

Figure 21. *Moving objects*



4.6. Summary

The present version of the layout manager provides a strong foundation which the high level layout module can use for automated IC layout. Future enhancements to the layout manager are discussed in chapter 6. The program code requirements for the layout manager are summarized in figure 22.

Figure 22. *Layout manager code requirements*

Module section	Lines of code
design rule processing	1002
object creation and deletion	434
object movement	669
object wiring	1373
technology input	1361
general	3731
total	8570

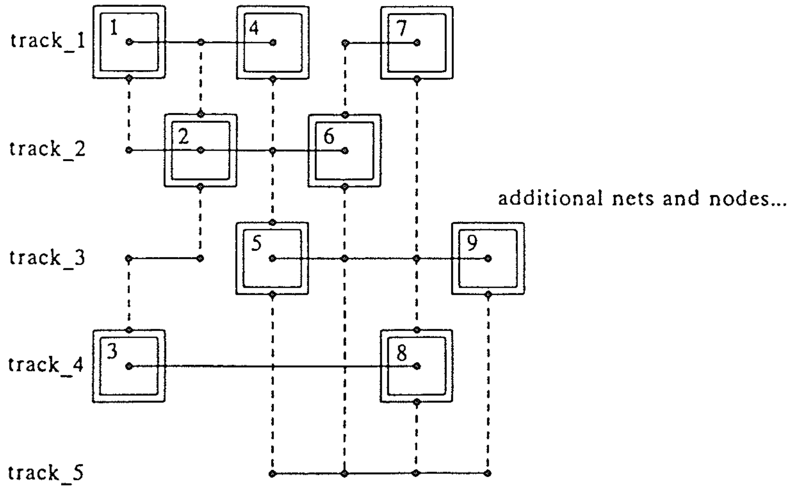
Chapter 5: High level layout module

The high level layout module uses the functions of the layout optimizer and the layout manager to create the layout. The high level layout module does not deal with fabrication technology details; however this module is currently tied to one particular layout strategy, or architecture. In the future, it is conceivable that new versions of the high level layout manager will be developed for other architectures, but it seems unavoidable that each high level layout module will be restricted to a single architecture, or at least a narrow range of architectures. This limitation is also apparent in the silicon compilers described in chapter 1.

5.1. Layout architecture of high level layout module

The current high level layout module uses the architecture of figure 23.

Figure 23. *High level layout module architecture*



This architecture is one-dimensional in nature; in other words the layout consists of a long horizontal chain of nodes, connected by wires that run horizontally. The horizontal wires run in fixed strips called *tracks*, where a track may contain any number of separate unconnected wires, as long as the nodes requiring these wires do not overlap. The layout of figure 23 consists of five tracks, with a total of seven wires, or nets, in these five tracks. To minimize the size of the layout in the vertical direction, it is advantageous to use as few tracks as possible. Typically as the number of nodes in a layout increases the number of tracks required rises quickly at first and then tends to remain constant for the remainder of the layout. The reason that track requirements tend to level off is that most of the nets are required by only a few nodes, and the one-dimensional layout optimizer has grouped the nodes such that nodes that require the same nets are close together in the one-dimensional array.

Figure 23 illustrates another important point; although the optimizer found a one-dimensional order of the nodes which comprise the design, this does not preclude placing adjoining nodes underneath one another. This is fortunate, for in the case of the Nebula compiler the nodes are individual transistors, and stringing them out in a long line would result in a very poor layout.

The present Nebula architecture is quite restrictive: it is limited to one-dimensional layout, and interconnections are made by horizontal wires running in predefined tracks. However, despite these rigid restrictions, this architecture is suitable for laying out any arbitrary design. Naturally it will tend to be more efficient for certain types of designs, but given a sufficiently large silicon area, *any* design may be implemented in this fashion.

Another concern is that this design methodology will tend to result in long, thin layouts, because of the one-dimensional nature of the architecture being used. Since most IC dies are roughly square, it appears that one-dimensional layout results in very inefficient silicon area usage. However this problem is easily remedied by *snaking*, or zig-zagging the long, thin layout back and forth across the IC die. The total increase in design area due to the snaking would not be significant.

5.2. Layout procedure

The layout procedure consists of the following steps:

1. Call optimizer module to do one-dimensional ordering of nodes.
2. Assign nets to tracks in an efficient manner.
3. Perform layout.
4. Compact resulting layout.
5. Output design.

5.2.1. One-dimensional ordering of nodes

The one-dimensional ordering of the nodes which make up the design is accomplished by passing the user specified net-list to the one-dimensional ordering module. In the current version of Nebula, the nodes are the set of transistors that make up the design. The ordering module returns a list of the nodes, ordered to minimize interconnecting wire.

5.2.2. Track assignment

The next step in the layout procedure is to assign tracks to the nets of the circuit. Tracks must be assigned to nets so that any node that requires a particular net has access to it. For example, if a net is required by the nodes in positions 2, 5 and 15 in the node list, then this net must be assigned to a track that is currently unused

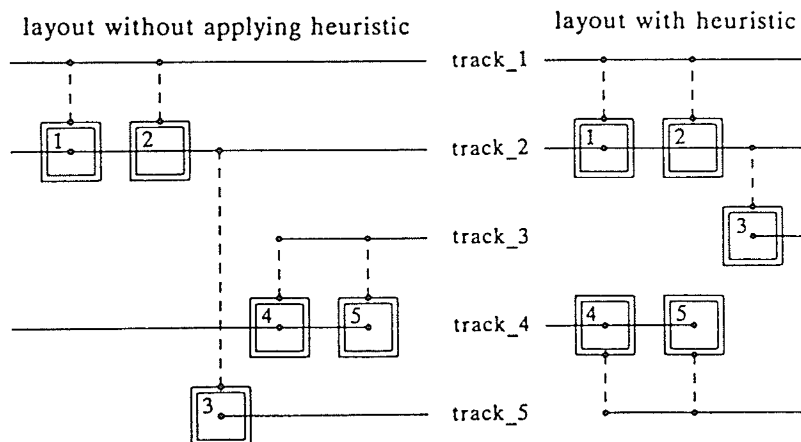
from node 2 through to node 15. The track assignment procedure is repeated for each net in the design. Any time that no track can be found which is free in the range required by a net, a new track is created. The track assignment is performed by the algorithm of figure 24.

Figure 24. *Track assignment algorithm*

```
assign_tracks:
    {
    find range of nodes over which each net is required
    current_node = left-most node
    while not all nodes processed
        {
        current_net = first unassigned net used by current_node
        while unassigned nets in current_node
            {
            find lowest_track used by nets connected to current_node
            find highest_track used by nets connected to current_node
            attempt to find free track for current_net between
                lowest_track and highest_track
            if no free track, attempt to find track outside but close to range
                lowest_track to highest_track
            if no free track, create new track
            assign current_net to track found
            current_net = next unassigned net used by current_node
            }
        current_node = next node to right in node_list
        }
    }
```

The objective of this algorithm is two-fold: first, it attempts to minimize the total number of tracks by re-using existing tracks whenever possible; secondly, it tries to use tracks as close as possible to tracks already assigned to nets that are required by the node being processed. The reason for the first objective is to minimize the number of tracks and thereby the vertical height of the layout. The second heuristic attempts to limit the vertical spread of the current node's connections by having all the nets that the node must connect to grouped as closely together as possible. This heuristic tends to reduce the horizontal size of the layout, since it increases the likelihood that two or more nodes may be placed under one another, instead of after one another. Figure 25 illustrates these concepts.

Figure 25. *Function of track assignment heuristics*



In the design where the track assignment heuristic was not applied, the net that started at *node_3* was assigned to *track_5* instead of *track_3*, and as a result it was not possible to place *node_4* and *node_5* under *node_1* and *node_2*. Design 2 shows the track assignment that results if the heuristics of the algorithm in figure 24 are applied. In this example, the length of the layout was reduced, without increasing its width, by applying these heuristics.

Further improvements to the track assignment algorithm would likely result in more efficient layout. Some possible improvements are discussed in chapter 6.

5.2.3. Layout of design

At this stage, the order in which the nodes are to be laid out has been established and each net has been assigned to a track. Thus layout consists of placing the nodes, wiring them to the appropriate nets, and compacting the result. The present version of Nebula employs a straightforward layout strategy. The nodes are placed, starting with the left-most node in the ordered node list and proceeding to the right, where each transistor in the net-list is treated as a separate node. The transistors are placed so that the transistor's *gate* may be connected directly to the appropriate net. Thus the position of each transistor is determined by the track that was assigned to the net connected to its *gate*. The transistor's *source* and *drain* are then connected to the appropriate track by diffusion wires running vertically. Figure 26 illustrates this process.

Figure 26. *Portion of an IC layout*

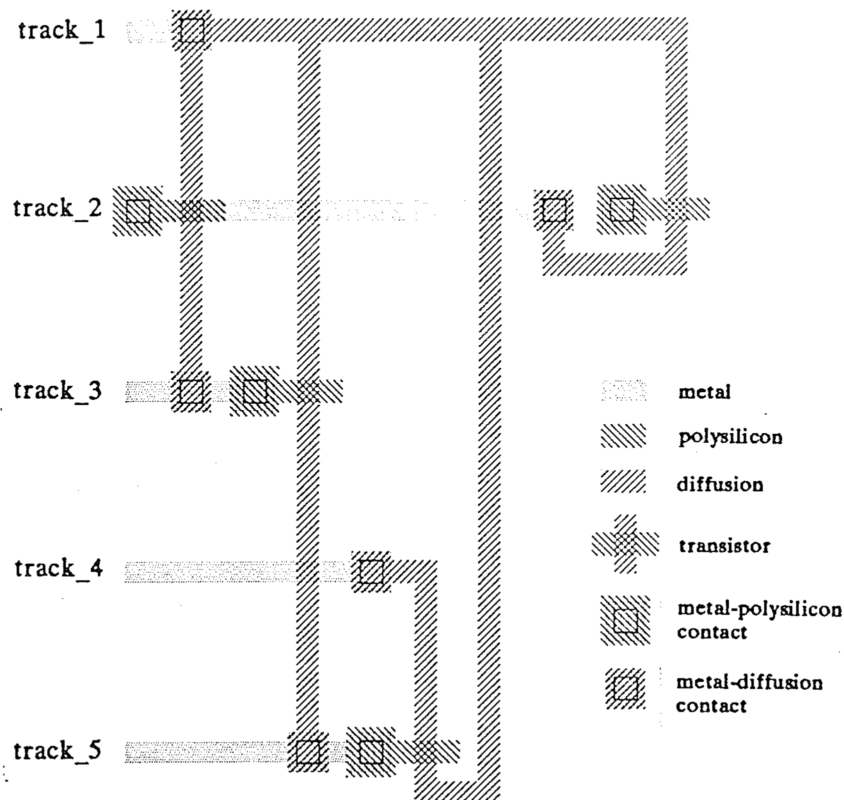


Figure 26 demonstrates why it is important to have the *source, gate* and *drain* nets as close together as possible; otherwise the wires connecting to the diffusion nets will waste layout space.

After the transistor has been connected to the appropriate nets, the complete group consisting of the transistor and the associated wires is moved as far to the left as possible without violating any design rules. To further improve the layout efficiency of the design, *jogs* are introduced in the wires connecting the diffusion ports of the transistor to the appropriate net. This allows the wires to closely follow the profile of the existing design, thereby reducing wasted space. Figure 26 shows a portion of a layout where jogging of vertical wires was not performed; it is apparent from this layout that significant savings would result if each of the vertical wires followed the profile of the existing design closely. The process of connecting a transistor and then pushing it to the left is repeated for each transistor in the design, with the order of placement determined by the order of the nodes in the ordered list returned by the one-dimensional optimizer module.

Unfortunately the layout portion of the high level layout module is technology dependent; in the present version it is assumed that the layout is intended for a single-metal CMOS process, such as the Northern Telecom CMOS 1B process used at the University of Manitoba. A limited degree of technology dependence seems to be unavoidable in a silicon compiler, as evidenced by the technology dependence of the Nebula compiler, and the compilers discussed in chapter 1. This is analogous to the machine-dependent portions of high-level language compilers: most parts of a software compiler may used for *any* computer, but a small portion is always machine-dependent. However, the fundamental principles remain unchanged for double-metal, NMOS or even bipolar processes, so writing a new high level layout module for these technologies would not be difficult.

5.2.4. Vertical compaction

After all the transistors have been placed, an optional vertical compaction may be performed. The vertical compaction is a straightforward but CPU intensive process; it simply consists of taking the objects which comprise the design and pushing them as close together in the vertical direction as possible. Experience has shown that vertical compaction results in a 15 to 30 per cent reduction in layout area. Figure 26 illustrates a layout before vertical compaction is performed; the large unused spaces between the tracks would be eliminated by the vertical compaction phase.

5.2.5. Output of layout

The last stage of the layout process is to output the finished design. The current version of Nebula outputs the layout information in CIF (Caltech Intermediate Form), a widely accepted IC layout interchange format. Adding code to generate a Calma GDS-II layout description, or any other format, would require very little effort.

5.3. High level layout module code requirements

Figure 27 gives the breakdown of the number of lines of code required for each of the sections of the high level layout module.

Figure 27. *Lines of code in high level layout module*

Module section	Lines of code
net track assignment	405
layout wiring	999
vertical compaction	194
CIF output	144
general	657
total	2399

5.4. Results of layout tests

Figure 28. *Results of layout tests*

circuit	transistors	layout efficiency (compiler / human)	CPU time optimizer	CPU time layout	CPU time compactor
two input nand	4	1.44	00:00:01	00:00:08	00:00:09
two input xor	6	2.00	00:00:01	00:00:14	00:00:19
flip-flop	16	4.24	00:00:01	00:01:03	00:01:24
full adder	18	3.83	00:00:03	00:01:13	00:01:42
6 bit adder	120	3.11	00:00:36	00:13:09	00:16:35
shift register	28	1.67	00:00:03	00:01:36	00:02:20
8 bit shift reg	230	3.40	00:00:59	00:50:05	no compact

Figure 28 lists layout test results for a variety of circuits. The present version of the Nebula compiler generates layouts which are about 2.8 times as large as a human designed layout of the same circuit. Clearly Nebula has to be improved considerably before it rivals human IC layout for efficiency; however, these results show that there is some promise in the layout methodology adopted. An early version of the MacPitts silicon compiler discussed in chapter 1 created layouts that occupied *ten times* as much area as a comparable human layout. There are several short-term changes which are planned for the Nebula compiler which should result in further layout area reductions. These are discussed in chapter 6.

The most encouraging aspect of the layout test results tabulated in figure 28 is that there seems to be no tendency for Nebula's layout efficiency to decrease vis a vis human layout design efficiency for large designs. This is of critical importance, for the advantage of using a silicon compiler *increases* as design size is increased. Only rarely would it be worthwhile to use a silicon compiler to lay out a few tens of transistors, if a four times increase in silicon area results. A four times increase in layout area when implementing a design consisting of several thousand transistors is more tolerable, because this makes the design of low production volume IC's commercially viable by reducing design cost at the expense of layout efficiency.

Although the Nebula compiler tends to become less efficient as the layout problem size is increased, the same is true for the human designer. The increase in inefficiency with increased design size seems to be comparable for human and compiler layouts, and thus the ratio of human to compiler design layout area remains roughly constant.

Chapter 6: Future work

Although the present version of Nebula is fully functional, this type of project tends by its nature to be very open-ended. No matter how satisfactory the results are, there are always new features whose implementation would further improve program performance. Interestingly, one of the most fruitful ways of improving the performance of the Nebula compiler was to monitor its operation during the layout process, keeping track of design decisions which resulted in an inefficient layout. This chapter deals with some potential areas for future work. The first section discusses improvements that are short-term in nature, and section two discusses long range objectives in the development of Nebula.

6.1. Short term changes and additions

The short range changes discussed in this section are relatively minor in nature, requiring at most a week or two of work. Most of these changes must be made before Nebula can actually be used to lay out an IC for fabrication.

6.1.1. Essential changes

The present version of Nebula does not handle input and output from the circuit: it does not bring I/O nets to the edge of the layout so that external connections may be made. Several parts of the program already have provisions for input and output, so adding this feature will not be difficult. Clearly provisions have to be made for input and output before Nebula can be used for IC design. It is not anticipated that adding input and output capabilities would significantly increase the size

of the layout, especially for large designs.

A second feature that should be added is a heuristic algorithm for CMOS fabrication processes which results in grouping of *P-wells* or *N-wells* and then connects the resulting wells to the power supply. Again, this work should be done before the program is used to implement a design. The amount of effort required is quite small, and the increase in layout size should be quite modest.

6.1.2. Optional additions

Aside from the required additions listed above, there is a long list of improvements which would tend to increase layout efficiency. Some of these changes are discussed below.

The present version of Nebula uses a substantial amount of CPU time for the layout process. Although layout is by its nature quite time consuming, substantial speedup should be possible by analyzing the program and establishing which parts of the layout process consume most of the CPU time. Because of the experimental nature of the development of this program, some portions of the code were repeatedly modified and as a consequence are not very efficient.

Another potentially fruitful area to investigate is the present method of assigning nets to tracks, and also the practise of placing transistors so that their *gates* can be connected directly to the appropriate net. More sophisticated approaches in both these areas would undoubtedly lead to more efficient layout. The present approaches were chosen for expediency, as well as layout efficiency. Further reductions in layout area could be achieved if nets are allowed to change tracks; in the present version a net will utilize the same track along its entire run. Unfortunately it is very difficult to predict which of these changes would result in the greatest gain, with the least effort. It seems that the only way to approach this type of problem is to try an idea, and if it is successful, incorporate it into the program.

To make this program useful for designs of over 500 transistors, steps must be taken to reduce memory usage. This presents no great problem: the most reseasonable approach would be to write the parts of the design that are complete into a file. Implementing this feature would be quite straightforward, and would eliminate memory usage as a constraint on the size of design that could be processed.

Finally, it would be beneficial to perform grouping operations on the transistors in the net-list before calling the one-dimensional ordering module. Potentially useful groupings include grouping of series and parallel connected transistors. The reason this type of grouping could be useful is that series and parallel connections of transistors may be laid out efficiently if it is known beforehand that a particular transistor is part of a series or parallel chain. Code has already been incorporated in several parts of the program for this purpose; thus the addition of this feature is quite straightforward.

Figure 29. *Summary of short term changes and additions*

task	benefit	priority	comments
handle I/O	get signals to and from circuit	required	not difficult
group, connect wells	prevent latch-up, increase layout efficiency	required	moderately difficult
optimize program code	shorter run-time	low	moderately difficult
better track assignment algorithm	increased layout efficiency	moderate	moderately difficult
better transistor placement	increased layout efficiency	moderate	moderately difficult
reduce memory requirements	increase maximum problem size	high	not difficult
transistor grouping	increased layout efficiency	low	not difficult

6.2. Long range development

There are many possible directions which the present work may take in the future. The decision to pursue one path and to neglect another must be considered very carefully, since early design decisions affect the character of the final product. At this stage, no firm conclusions have been reached as to what future development plan is most suitable; thus the discussion below will present a number of ideas, without attempting to evaluate their relative merit. The long range development

plans for Nebula may be divided into two classes: additions to the present system, and fundamental changes to the present approach. Additions to the present system are discussed below.

6.2.1. Additions to Nebula

The development of one or more modules which take a high level functional specification of an IC, and translate it into a net-list for layout, would make the compiler more convenient to use for many applications. One such module could accept Boolean logic expressions as input. The Boolean expressions would be minimized, and then a net-list would be generated, where the currently selected fabrication technology would determine whether an NMOS, CMOS or bipolar net-list is created. This module would be fairly straightforward to develop, and could be used for small designs, such as special purpose controller IC's.

Adding a general hardware description language [12] (HDL) to net-list translator would make it possible, at least in theory, to specify the complete operation of an IC in a few succinct pages. The MacPitts compiler uses this approach. Advantages include rapid, relatively error-free design. Potential problems include the difficulty of designing and implementing a good hardware description language, and the inevitable loss of layout efficiency. Layout efficiency will tend to be reduced because the human designer will no longer have direct control over the form of the net-list that will be used to implement a particular function. However, the anticipated designer productivity gains make this extension worth considering. The layout architecture used by the Nebula compiler may not be suitable for all hardware description languages, so the present layout strategy may require modification before a hardware description language module can be added.

Extending Nebula to other fabrication technologies, such as NMOS and bipolar processes, would increase the compiler's flexibility considerably. Some technologies, for example a single-metal NMOS process, could be added with very little effort. Double-metal fabrication processes would require somewhat more effort to implement: however changes would be limited to the high level layout module. An additional advantage of developing a high level layout module for double-metal processes is that the resulting layouts would be considerably more compact than the present single-metal layouts.

6.2.2. Changes to Nebula

Apart from the additions to Nebula discussed above, there are also a number of changes to the present layout approach which should be investigated. As mentioned previously, it is very difficult to predict *a priori* whether a particular modification will result in a better compiler. Thus, a certain amount of trial and error is inevitable. Some possible directions are discussed below.

The present version of Nebula is limited to one-dimensional layout; developing some sort of two-dimensional layout scheme could improve program performance. There are endless variations on two-dimensional layout which must be considered. One approach, which retains much of the philosophy of the present system, would involve finding a satisfactory two-dimensional arrangement of the nodes which comprise the design, picking a starting point, and then placing the components in a spiral fashion outwards from the starting point. The location of each component would be chosen to minimize the layout area. Note how this procedure is a direct extension of the one-dimensional architecture presently employed; in the present system layout begins at the left and proceeds to the right, while the two-dimensional version begins at the center and radiates outwards in all directions.

Another feature which has deliberately been excluded from the present system is any form of *hierarchy* in the layout process. Many silicon compilers employ hierarchy, insofar as they deal with larger objects which are composed of many transistors. Typically, these high-level objects, or *cells*, are developed by human designers. The advantage of this approach is that it makes use of the human designer's innate ability to create compact layouts; the problem is that adopting this approach tends to reduce the range of applications for which the compiler is suitable, without additional human intervention.

Perhaps the best solution would be to move away from the fundamental precept of silicon compilation: limited human intervention in the layout process. There are some global high-level decisions involved in designing an IC which a human designer can make without difficulty, but which are exceptionally difficult to incorporate into a software system. In this type of system, the compiler would ask the human designer for *advice* when it is confronted with a difficult design decision.

The preceding discussion only scratches the surface of the myriad possibilities for future development of the Nebula compiler. The choice of one direction over another is very difficult. An encouraging thought: there is no single *right* solution to the problem of silicon compilation and there is also no *wrong* solution. Whatever approach is adopted, the resulting system will find a niche somewhere in the field of automated IC design.

Figure 30. *Summary of long term changes and additions*

task	benefit	priority	comments
Boolean logic module	make program easier to use	low	not very difficult
HDL module	make program easier to use	low	difficult
add other technologies	increase flexibility	low	moderately difficult
two-dimensional layout	increased layout efficiency	moderate	difficult
hierarchy in layout	increased layout efficiency	low	difficult
human interaction module	increased layout efficiency	low	difficult

Chapter 7: Conclusion

At the outset of the Nebula silicon compiler project, several hypotheses were used to establish fundamental design directions. The most important of these hypotheses dealt with the role of hierarchy in IC design.

When an IC is being developed using traditional approaches, the design tends to be highly hierarchical in nature, particularly for large designs. This implies that the human designer starts by constructing a number of general-purpose building blocks which will be used throughout the design. Combinations of building blocks are in turn used to make further, more complex building blocks. This process is known as hierarchical design. The purpose of hierarchical design is to deal with the tremendous number of objects which make up an IC design by partitioning the design into manageable hierarchical pieces.

The interesting thing is that hierarchical design serves no purpose, other than to manage the complexity of IC design. The price that is paid with the hierarchical design methodology is in terms of layout efficiency; standard building blocks must be used which are not quite ideal for a particular application, and thus will tend to use more layout area. At least in theory, eliminating design hierarchy can improve layout efficiency. This is the philosophy of the Nebula compiler: eliminate design hierarchy and let the computer deal with the tremendous complexity of non-hierarchical design. The underlying hypothesis was that the compiler's ability to handle greater complexity than the human designer would offset its inability to make individual layout decisions that are as good as those made by a human designer. Thus each IC design methodology attempts to capitalize on its strengths: the manual IC design approach limits complexity through hierarchy and then gives the human designer free rein to assemble hierarchical blocks in an optimum way, while the Nebula compiler eliminates hierarchy entirely, makes more simplistic design decisions, and capitalizes on the computer's ability to deal with a great number of objects simultaneously.

In the present version of the Nebula compiler the human-hierarchical approach wins out over the computerized non-hierarchical approach. However, this is an early version of the compiler, employing very restrictive and simplistic design strategies. Even so, respectable layout efficiency has been achieved. When one-dimensional layout is replaced with two-dimensional layout, and more sophisticated heuristics are employed for the placement and interconnection of components, it is conceivable that non-hierarchical automated IC layout will rival human-hierarchical IC layout in efficiency. Further research is required to establish conclusively whether the layout efficiency of this approach to automated IC layout can potentially reach the efficiency of human IC layouts.

Appendix A: Technology files for NT CMOS 1B

```
/*

COLOR FILE

*/


/* the low 5 bits map metal  polysilicon  d-well   p+  and n+
 *  41: passification
 *  42: contact cut
 *  43: invisible
 *  44: n-guard
 *  45: p-guard
 *  46: p-well
 */

color {  0   0   0   0}  /* 0:                                      */
color {  1   0   0 255}  /* 1:                               metal */
color {  2 223   0   0}  /* 2:                      polysilicon     */
color {  3 150  20 150}  /* 3:                      polysilicon+metal */
color {  4   0 255   0}  /* 4:          d-well                      */
color {  5   0 160 160}  /* 5:          d-well   +            metal */
color {  6 180 130   0}  /* 6:          d-well   +polysilicon       */
color {  7 120  60 120}  /* 7:          d-well   +polysilicon+metal */
color {  8 255 190   6}  /* 8:       p                              */
color {  9 100 100 200}  /* 9:       p+                       metal */
color { 10 255 114   1}  /* 10:      p+             polysilicon     */
color { 11  70  50 150}  /* 11:      p+             polysilicon+metal */
color { 12 180 255   0}  /* 12:      p+d-well                       */
color { 13  40 160 160}  /* 13:      p+d-well    +            metal */
color { 14 200 180  70}  /* 14:      p+d-well    +polysilicon       */
color { 15  60  60 130}  /* 15:      p+d-well    +polysilicon+metal */
color { 16 170 140  30}  /* 16: n   +                               */
color { 17  60  80 200}  /* 17: n   +                         metal */
color { 18 220 130   0}  /* 18: n   +               polysilicon     */
color { 19 120  20 120}  /* 19: n   +               polysilicon+metal */
color { 20 156 220   3}  /* 20: n   +  d-well                       */
color { 21   0 120 120}  /* 21: n   +  d-well    +            metal */
color { 22 170 170   0}  /* 22: n   +  d-well    +polysilicon       */
color { 23  35  50 120}  /* 23: n   +  d-well    +polysilicon+metal */
color { 24 200 160  20}  /* 24: n   +p                              */
color { 25  80  50 230}  /* 25: n   +p+                       metal */
color { 26 255 100  50}  /* 26: n   +p+             polysilicon     */
color { 27 130  20 130}  /* 27: n   +p+             polysilicon+metal */
color { 28  60 190   0}  /* 28: n   +p+d-well                       */
color { 29  30 110 100}  /* 29: n   +p+d-well    +            metal */
color { 30 150  90   0}  /* 30: n   +p+d-well    +polysilicon       */
color { 31 110  80 140}  /* 31: n   +p+d-well    +polysilicon+metal */
color { 32   0   0   0}  /* 32: background color (same as color 0) */
color { 33 120   0   0}  /* 33: window border color                */
color { 34   0 255   0}  /* 34: highlighted window border color    */
color { 35 130 180 130}  /* 35: menu border color                  */
color { 36 255 255   0}  /* 36: highlighted menu border color      */
color { 37 255 128   0}  /* 37: menu text color                    */
color { 38 175 100   0}  /* 38: menu glyph color                   */
color { 39 255 255 255}  /* 39: cursor color                       */
color { 40 200 200 100}  /* 40: cell name  outline  port name color*/
color { 41 255 160 255}  /* 41: passification                      */
color { 42  90  90  90}  /* 42: contact cut                        */
color { 43 160 160 160}  /* 43: invisible                          */
color { 44 150 100 100}  /* 44: n-guard                            */
color { 45 100 170 100}  /* 45: p-guard                            */
color { 46 255 255   0}  /* 46: p-well                             */
color { 47 128 128 128}  /* 47-63: unused                          */
color { 64 200 200 200}  /* 64-127: contact cut  boxes  text  grid */
color {128 255 255 255}  /* 128-255: highlight box                 */
```

```
/*

WIRE FILE

*/

wire
        {
        function { metal }
        name { metal }
        width { 50 }
        offset { 0 }
        layer { metal 0 @connected }
        }

wire
        {
        function { polysilicon }
        name { polysilicon }
        width { 50 }
        offset { 0 }
        layer { polysilicon 0 @connected }
        }

wire
        {
        function { n_diffusion }
        name { n_device_well__(n) }
        width { 270 }
        offset { 110 }
        layer { p_guard 30 @unconnected }
        layer { n_guard 0 @unconnected }
        layer { p_well 80 @unconnected }
        layer { device_well__(n) 110 @connected }
        }

wire
        {
        function { p_diffusion }
        name { p_device_well__(p) }
        width { 190 }
        offset { 70 }
        layer { device_well__(p) 70 @connected }
        layer { p+ 30 @unconnected }
        layer { n+ 0 @unconnected }
        }

wire
        {
        function { p_well }
        name { p_well }
        width { 260 }
        offset { 160 }
        layer { p_well 80 @unconnected }
        layer { p_guard 30 @unconnected }
        layer { n_guard 0 @unconnected }
        }
```

```
/*

OBJECT FILE

*/


port_type { metal               metal                                           }
port_type { polysilicon         polysilicon                                     }
port_type { n_diffusion         n_device_well__(n)                              }
port_type { p_diffusion         p_device_well__(p)                              }
port_type { p_well              p_well                                          }

object
        {
        function{ metal_poly_contact }

        port { metal        metal_1         -5  10  -5  10    5 -10   5 -10 }
        port { polysilicon  poly_1          -5   0  -5   0    5   0   5   0 }

        port_connect { @all }

        layer { metal         metal_1   -5  10  -5  10    5 -10   5 -10 }
        layer { polysilicon   poly_1    -5   0  -5   0    5   0   5   0 }
        layer { contact_cut   @none     -5  30  -5  30    5 -30   5 -30 }

        name { metal_polysilicon_con }
        size { 110 110 }
        offset { 30 30 30 30 }
        }

object
        {
        function { metal_p_diffusion_contact }

        port { metal       metal         -5  70  -5  70    5 -70   5 -70 }
        port { p_diffusion p_diffusion   -5  70  -5  70    5 -70   5 -70 }

        port_connect { @all }

        layer { metal              metal          -5  70  -5  70    5 -70  5 -70 }
        layer { device_well__(p)   p_diffusion    -5  70  -5  70    5 -70  5 -70 }
        layer { p+                 @none          -5  30  -5  30    5 -30  5 -30 }
        layer { n+                 @none          -5   0  -5   0    5   0  5   0 }
        layer { contact_cut        @none          -5  90  -5  90    5 -90  5 -90 }

        name { metal_p_dev_well_con__(p) }
        size { 230 230 }
        offset { 90 90 90 90 }
        }

object {
        function { metal_n_diffusion_contact }

        port { metal         metal          -5  110  -5  110    5 -110   5 -110 }
        port { n_diffusion   n_diffusion    -5  110  -5  110    5 -110   5 -110 }

        port_connect { @all }

        layer { metal              metal          -5  110 -5  110    5 -110 5 -110 }
        layer { device_well__(n)   n_diffusion    -5  110 -5  110    5 -110 5 -110 }
        layer { p_well             @none          -5  80  -5  80     5 -80  5 -80 }
        layer { p_guard            @none          -5  30  -5  30     5 -30  5 -30 }
        layer { n_guard            @none          -5  0   -5  0      5   0  5   0 }
        layer { contact_cut        @none          -5  130 -5 130     5 -130 5 -130 }

        name { metal_n_dev_well_con__(n) }
        size { 310 310 }
        offset { 130 130 130 130 }
        }

object
        {
        function { p_transistor }

        port { polysilicon  poly          0   0  -5  110     0   0   5 -110 }
        port { p_diffusion  diff_top     -5  120  5  -70     5 -120  5  -70  }
        port { p_diffusion  diff_bottom  -5 120  -5   70     5 -120 -5   70  }
```

```
        layer { device_well__(p) diff_top      -5 120   5 -110   5 -120  5 -70 }
        layer { device_well__(p) diff_bottom   -5 120  -5  70    5 -120 -5 110 }
        layer { device_well__(p) @none         -5 120  -5  110   5 -120  5 -110 }
        layer { polysilicon      poly          -5  70  -5  110   5 -70   5 -110 }
        layer { p+               @none         -5  80  -5  30    5 -80   5 -30 }
        layer { p+               @none          5 -80  -5  70    5 -30   5 -70 }
        layer { n+               @none         -5  50  -5  0     5 -50   5  0 }
        layer { n+               @none          5 -50  -5  40    5  0    5 -40 }
        layer { transistor       @none         -5 120  -5 110    5 -120  5 -110 }

        name { transistor__(p) }
        size { 290 270 }
        offset { 120 110 120 110 }
        }

object
        {
        function { n_transistor }

        port { polysilicon  poly          0    0  -5  150    0    0    5 -150 }
        port { n_diffusion  diff_top     -5  150   5 -110    5 -150    5 -110 }
        port { n_diffusion  diff_bottom  -5  150  -5  110    5 -150   -5  110 }

        layer { device_well__(n)  diff_top     -5  150  5 -150 5 -150  5  -110 }
        layer { device_well__(n)  diff_bottom  -5  150 -5 110  5 -150 -5   150 }
        layer { device_well__(n)  @none        -5  150 -5 150  5 -150  5  -150 }
        layer { polysilicon       poly         -5  100 -5 150  5 -100  5  -150 }
        layer { transistor        @none        -5  150 -5 150  5 -150  5  -150 }
        layer { p_well            @none        -5  80  -5  80  5 -80   5   -80 }
        layer { p_guard           @none        -5  30  -5  30  5 -30   5   -30 }
        layer { n_guard           @none        -5   0  -5   0  5   0   5     0 }

        name { transistor_well__(n) }
        size { 350 350 }
        offset {150 150 150 150 }
        }

object
        {
        function { metal_n_split_contact }

        port { metal        metal_1   -5 110   -5 110   5 -110  -5  110 }
        port { n_diffusion  diff_1    -5 110    5 -135  5 -110   5 -135 }

        port_connect { @all }

        layer { metal              metal_1  -5  110  -5  110    5 -110   5 -110 }
        layer { device_well__(n)   metal_1  -5  110  -5  110    5 -110   5 -110 }
        layer { p+                 @none    -5   70  -5   70    5 -70    0    0 }
        layer { n+                 @none    -5   40  -5   40    5 -40    0    0 }
        layer { p_well             @none    -5   80  -5   80    5 -80    5  -80 }
        layer { p_guard            @none    -5   30  -5   30    5 -30    5  -30 }
        layer { n_guard            @none    -5    0  -5    0    5   0    5    0 }
        layer { split_contact__(p) @none    -5    0  -5    0    5   0    0    0 }
        layer { contact_cut          @none  -5  130  -5  130    5 -130   5 -130 }

        name { metal_n_split_con__(n) }
        size { 310 380 }
        offset {130 130 130 130 }
        }

object
        {
        function { metal_p_split_contact }

        port { metal        metal_1  -5  70  -5  70  5  -70   5 -70 }
        port { p_diffusion  diff_1   -5  70  -5  95  5   70  -5  95 }

        port_connect { @all }

        layer { metal               metal_1   -5  70  -5   70   5 -70   5  -70 }
        layer { device_well__(p)    metal_1   -5  70  -5   70   5 -70   5  -70 }
        layer { p+                  @none     -5  30  -5   30   5 -30   0    0 }
        layer { n+                  @none     -5   0  -5    0   5   0   0    0 }
        layer { split_contact__(p)  @none     -5  30   0    0   5 -30   5  -30 }
        layer { contact_cut         @none     -5  90  -5   90   5 -90   5  -90 }

        name { metal_p_split_con__(p) }
        size { 230 300 }
        offset { 90 90 90 90 }
        }
```

```
object
        {
        function { metal_p_substrate_contact }

        port { metal      metal      -5 110   -5 110   5 -110  5 -110 }
        port { p_well     p_well     -5  80   -5  80   5  -80  5  -80 }

        port_connect { @all }

        layer { metal                metal    -5  110  -5  110   5 -110   5 -110 }
        layer { device_well_sc_(n)   metal    -5  110  -5  110   5 -110   5 -110 }
        layer { p+                   @none    -5   70  -5   70   5  -70   5 -130 }
        layer { n+                   @none    -5   40  -5   40   5  -40   5 -130 }
        layer { p_well               @none    -5   80  -5   80   5  -80   5  -80 }
        layer { p_guard              @none    -5   30  -5   30   5  -30   5  -30 }
        layer { n_guard              @none    -5    0  -5    0   5    0   5    0 }
        layer { split_contact__(p)   @none    -5    0  -5    0   5    0   5 -130 }
        layer { contact_cut          @none    -5  130  -5  130   5 -130   5 -130 }

        name { p_well_substrate_con__(n) }
        size { 310 320 }
        offset { 130 130 130 130 }
        }

object
        {
        function { metal_n_substrate_contact }

        port { metal     metal      -5 40  -5 40    5 -40    5 -40 }

        port_connect { @all }

        layer { metal               metal    -5  40  -5  40    5 -40   5 -40 }
        layer { device_well__(p)    metal    -5  40  -5  40    5 -40   5 -40 }
        layer { split_contact__(p)  @none    -5   0  -5  60    5   0   5   0 }
        layer { contact_cut         @none    -5  60  -5  60    5 -60   5 -60 }

        name { n_substrate_con__(p) }
        size { 170 180 }
        offset { 60 60 60 60 }
        }

object
        {
        function { metal_pin }

        port { metal         metal_1   -5  0  -5  0    5  0  5 0 }

        port_connect { @all }

        layer{ metal  metal_1   -5  0  -5  0   5  0   5  0 }

        name {metal_pin }
        size { 50 50 }
        offset { 0 0 0 0 }
        }

object
        {
        function { polysilicon_pin }

        port { polysilicon  poly_1         -5    0 -5    0   5    0  5    0 }

        port_connect { @all }

        layer { polysilicon poly_1   -5    0 -5    0   5    0  5    0 }

        name { polysilicon_pin }
        size { 50 50 }
        offset { 0 0 0 0 }
        }

object
        {
        function { n_diffusion_pin }

        port { n_diffusion  diff_1  -5   110 -5   110   5  -110  5  -110 }

        port_connect { @all }
```

```
        layer { device_well__(n) diff_1    -5   110  -5   110   5 -110  5   -110 }
        layer { p_well           @none     -5    80  -5    80   5  -80  5    -80 }
        layer { p_guard          @none     -5    30  -5    30   5  -30  5    -30 }
        layer { n_guard          @none     -5     0  -5     0   5    0  5      0 }

        name { n_device_well_pin__(n) }
        size { 270 270 }
        offset { 110 110 110 110 }
        }

object
        {

        function { p_diffusion_pin }

        port { p_diffusion  diff_1  -5   70 -5   70   5  -70  5  -70 }

        port_connect { @all }

        layer { device_well__(p)  diff_1   -5   70  -5   70  5   -70  5   -70 }
        layer { p+                @none    -5   30  -5   30  5   -30  5   -30 }
        layer { n+                @none    -5    0  -5    0  5     0  5     0 }

        name { p_device_well_pin__(p) }
        size { 190 190 }
        offset { 70 70 70 70 }
        }

object
        {
        function { p_well_pin }

        port { p_well       p_well_1  -5   80 -5   80   5  -80  5  -80 }

        port_connect { @all }

        layer { p_well   p_well_1  -5   80 -5   80   5  -80  5  -80 }
        layer { p_guard  @none     -5   30 -5   30   5  -30  5  -30 }
        layer { n_guard  @none     -5    0 -5    0   5    0  5    0 }

        name { p_well_pin }
        size { 260 260 }
        offset { 80 80 80 80 }
        }
```

```
/*

MISC FILE

*/

lambda{100}
name{ntcmos}
scale{10}
```

```
/*

LAYER FILE

*/

layer { metal                  1   001   transparent   CM      }
layer { polysilicon            2   002   transparent   CP      }
layer { n+                    16   020   transparent   CNP     }
layer { p+                     8   010   transparent   CPP     }
layer { device_well__(n)       4   004   transparent   CF      }
layer { device_well__(p)       4   004   transparent   CF      }
layer { device_well_sc_(n)     4   004   transparent   CF      }
layer { contact_cut           42   077   opaque        CC      }
layer { n_guard               44   077   opaque        CNG     }
layer { p_guard               45   077   opaque        CPG     }
layer { p_well                46   077   opaque        CPW     }
layer { passivation           41   077   opaque        CG      }
layer { invisible             43   077   opaque        @none   }
layer { transistor             0   000   opaque        @none   }
layer { split_contact__(p)     0   000   opaque        @none   }
layer { split_contact__(n)     0   000   opaque        @none   }
```

```
/*

DRC FILE

*/

unconnected
        {
        design_rule { metal                     metal                 60 }
        design_rule { polysilicon               polysilicon           50 }
        design_rule { polysilicon               device_well__(n)      40 }
        design_rule { polysilicon               device_well__(p)      40 }
        design_rule { polysilicon               device_well_sc_(n)    40 }
        design_rule { n+                        device_well__(n)      40 }
        design_rule { n+                        split_contact__(p)     0 }
        design_rule { p+                        device_well__(n)      70 }
        design_rule { p+                        split_contact__(p)     0 }
        design_rule { contact_cut               contact_cut           50 }
        design_rule { contact_cut               transistor            50 }
        design_rule { p_guard                   device_well__(p)      90 }
        design_rule { device_well__(n)          device_well__(n)      70 }
        design_rule { device_well__(p)          device_well__(p)      70 }
        design_rule { device_well__(n)          device_well__(p)     170 }
        design_rule { device_well__(n)          split_contact__(n)     0 }
        design_rule { device_well__(n)          device_well_sc_(n)    70 }
        design_rule { device_well__(p)          device_well_sc_(n)    70 }
        design_rule { device_well__(p)          device_well_sc_(n)   170 }
        design_rule { split_contact__(n)        device_well_sc_(n)     0 }
        design_rule { device_well_sc_(n)        device_well_sc_(n)    70 }
        }

connected
        {
        design_rule { polysilicon               device_well__(n)      40 }
        design_rule { polysilicon               device_well__(p)      40 }
        design_rule { polysilicon               device_well_sc_(n)    40 }
        design_rule { n+                        device_well__(n)      40 }
        design_rule { n+                        split_contact__(p)     0 }
        design_rule { p+                        device_well__(n)      70 }
        design_rule { p+                        split_contact__(p)     0 }
        design_rule { contact_cut               contact_cut           50 }
        design_rule { contact_cut               transistor            50 }
        design_rule { p_guard                   device_well__(p)      90 }
        design_rule { device_well__(n)          device_well__(p)     170 }
        design_rule { device_well__(n)          split_contact__(n)     0 }
        design_rule { device_well__(p)          device_well_sc_(n)   170 }
        design_rule { split_contact__(n)        device_well_sc_(n)     0 }
        }
```

References

[1] N. Bergmann et al., "A Case Study of the F.I.R.S.T. Silicon Compiler," *Proceedings Third Caltech Conference on VLSI*, pp. 413-430, 1983.

[2] J. Blanks., "Near-Optimal Quadratic-Based Placement for a Class of IC Layout Problems," *IEEE Circuits and Devices Magazine*, pp. 31-37, September 1985.

[3] J. Fox., "The MacPitts Silicon Compiler: A View from the Telecommunications Industry," *VLSI Design*, pp. 30-37, May/June 1983.

[4] R. Jamier et al., "APOLLON, A Data-Path Silicon Compiler," *IEEE Circuits and Devices Magazine*, pp. 6-14, May 1985.

[5] D. Johannsen., "Bristle Blocks: A Silicon Compiler," *Caltech Conference on VLSI*, pp. 303-310, January 1979.

[6] T.Kashiwabara et al., "NP-completeness of the problem of finding a minimum clique number interval graph containing a given graph as a subgraph," *Proceedings IEEE International Symposium on Circuits and Systems*, 1979, pp. 657-660.

[7] T. Kowalski et al., "The VLSI design automation assistant: Prototype system," *Proceedings ACM IEEE 20th Design Automation Conference*, June 1983, pp. 479-483.

[8] C. Mead et al., "Silicon compilers and foundries will usher in user-designed VLSI," *Electronics*, pp. 107-111, August, 1982.

[9] C. Piguet et al., "A Metal-Oriented Layout Structure for CMOS Logic," *IEEE Journal of Solid-state Circuits*, vol. SC-19, no. 3, pp. 425-436, June 1982.

[10] M.Rubin., *Internals of the ELECTRIC Database: A Guide for Writing Analysis Aids*, Fairchild Laboratory for Artificial Intelligence Research, 1983.

[11] I. Shirakawa et al., "A Layout System for the Random Logic Portion of an MOS LSI Chip," *IEEE Transactions on Computers*, vol. C-30, no. 8, pp. 572-581, August 1981.

[12] J. Southard et al., "MacPitts: An Approach to Silicon Compilation," *Computer*, pp. 74-82, December 1983.

[13] T. Uehara et al., "Optimal Layout of CMOS Functional Arrays," *IEEE Transactions on Computers*, vol. C-30, no. 5, pp. 305-312, May 1981.

[14] J. Ullman., *Computational Aspects of VLSI.* Rockville: Computer Science Press, 1984.

[15] P. Wallich., "On the horizon: fast chips quickly," *IEEE Spectrum*, pp. 28-34, March 1984.

[16] A. Weinberger., "Large Scale Integration of MOS Complex Logic: A Layout Method," *IEEE Journal of Solid-State Circuits*, SC-2, no. 4, pp. 182-190, December 1967.

[17] W. Wolf et al., "Dumbo, A Schematic-to-layout Compiler," *Proceedings Third Caltech Conference on VLSI*, pp. 379-391, 1983

[18] J. Young., "IC-Design Automation Strides into Silicon-Compiler Era," *Electronics*, pp. 58-63, June 1985.