1

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No. 2)

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

AUTOMATIC GENERATION OF CMOS STANDARD CELL LAYOUT

FROM LOGIC LEVEL DESCRIPTIONS

by

Martin C. Lefebvre

A thesis submitted to the Faculty of Graduate
Studies and Research in partial fulfillment
of the requirements for the degree of

Master of Engineering

Department of Electronics
Faculty of Engineering
Carleton University
Ottawa, Ontario
April 1986

© copyright

1986, Martin Lefebvre

1 INTRODUCTION

1.1 General Overview

Integrated Circuits (IC) Technology is preparing to outgrow the frontiers of Very Large Scale Integration (VLSI). Monolithic designs of 100,000 transistors or more have become common, and the rate of growth is still very high. How well we harness that fast paced technology, however, depends on the Computer Aided Design (CAD) tools that we put at our own disposal. The development of such tools has consistently trailed that of fabrication technology in recent years. This situation has caused a chronic delay in the availability of new technologies and has inhibited the proliferation of Application Specific IC's (ASIC's) by contributing to high development costs.

This thesis addresses the issue of computer aided design in the context of IC synthesis. It is based on the development and implementation of a software tool for the automatic synthesis of CMOS standard cells. The proposed system is aimed at facilitating the design and maintenance of complete libraries, and will considerably reduce the time required to accommodate changes in process technology.

The results presented herein reflect the conclusions drawn from experimentation as well as an extensive literature survey.

The undersigned recommend to the Faculty of Graduate Studies
and Research the acceptance of the Thesis:

"Automatic Generation of CMOS Standard Cell Layout
from Logic Level Descriptions"

submitted by Martin C. Lefebvre in partial fulfillment of
the requirements for the degree of Master of Engineering.

Prof. A.R. Boothroyd
Chairman,
Department of Electronics

Prof. C.H. Chan
Thesis supervisor

Carleton University

April, 1986

ABSTRACT

Standard cells have become widely used in the design of Integrated Circuits. They are a collection of low level building blocks that are pre-designed and stored in a cell library for subsequent use by designers. This thesis presents a cell synthesizer that is capable of generating layout geometries for CMOS standard cells, from logic descriptions. The synthesis process begins with a mapping of the cell logic description onto an approximately optimal transistor interconnect. Transistor placement is based on the creation of "clusters" and adheres to the "line of diffusion" layout style. Metal connections as well as contact windows are created to complete the cell netlist. The output from the cell synthesizer is a description of the cell layout in Caltech Intermediate Form.

ACKNOWLEDGEMENTS

The author wishes to thank his fellow graduate students and the entire staff of the department of Electronics for creating a superb work environment. Special thanks to my advisor, Dr. C.H. Chan, to Dr. J.P. Knight for his much appreciated feedback and advice and to Dr. E.F. Girczyc for stirring my interest for design automation.

This research was made possible by the financial assistance of NSERC and Bell-Northern Research. The contribution of Digital Equipment Corporation, who provided the VAX-11/750 computer used for prototype development, is also acknowledged.

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	General Overview	1
1.2	Review Of Standard Cells Systems	2
1.3	Motivation For The Research	5
1.4	Organization Of The Thesis	6
2	RELATED WORK	7
2.1	Survey Of Relevant Literature	7
2.1.1	TOPOLOGIZER	7
2.1.2	TALIB	9
2.1.3	DUMBO	9
2.1.4	BTL Semi Manual Method	10
2.1.5	NEC Gate Array System	11
2.1.6	Graph Theoretic Approach	13
2.1.7	CALMA Project	15
2.1.8	Primitive Based Cell Design	15
2.2	Summary Of Survey	17
3	OUTLINE OF A TURNKEY STANDARD CELL DESIGN SYSTEM	18
3.1	Review Of Process Independence	18
3.1.1	Existing Paths To Process Independence	19
3.1.2	Analysis Of Design Rules	19
3.2	Other Attributes Of A Turnkey System	20
3.3	Complete Compilation Process	21
3.4	"Turnkey" System Architecture	23
3.4.1	Expert CMOS Designer	23
3.4.2	Optimizer	25
3.4.3	Placer-Router	25
3.4.4	Layout Compiler	25
3.4.5	Feature Extraction	26
3.4.6	Simulation Expert	26
4	OVERVIEW OF PROTOTYPE CELL COMPILER	27
4.1	Layout Style	27
4.2	Translation Of Logic Descriptions Into Transistor Structures	31
4.2.1	The Graph Model	31
4.2.2	Logic Gate To Transistor Interconnects Translator	35
4.2.3	Compilation Of Other Structures	37

4.3	Mapping Of Transistor Structures Onto Cell Layout	37
4.3.1	Placement Technique	39
4.3.2	Routing Technique	39
4.3.3	Generation Of CIF Code	41
5	PROTOTYPE CELL COMPILER	42
5.1	General System Organization	42
5.2	Cluster Formation	42
5.2.1	Or-into-nand Clusters	44
5.2.2	And-into-nor Clusters	44
5.2.3	Transmission Gate Clusters	45
5.2.4	Single Gate Clusters	45
5.3	Cluster Expansion	46
5.3.1	Expansion Of Two-level Logic Clusters	46
5.3.2	Expansion Of Single-level Logic Clusters	49
5.3.3	Expansion Of Clusters Of Type "tg-mux"	51
5.4	Detection Of Metal Nets	51
5.5	Cluster Placement Algorithm	52
5.6	Formation Of Layout Pieces	54
5.7	Piece Placement Algorithm	55
5.8	Formation Of Slices	55
5.9	Routing Algorithm	57
5.10	Layout Compilation	60
6	ILLUSTRATIVE EXAMPLES	62
6.1	Complete Compilation Process Of A Five Gate Cell	62
6.2	Effect Of Primary Cell Outputs On Layout Compilation	70
6.3	Compilation With Transmission Gates	75
6.4	Compilation Of Complex Cells	78
7	CONCLUSIONS	80
7.1	Summary	80
7.2	Suggestions For Future Work	81
	APPENDICES	84
	BIBLIOGRAPHY	100

LIST OF FIGURES

Number	Title	Page
1.1	Typical Standard Cell Floor Plan	4
2.1	TOPOLOGIZER Placement Strategy	8
2.2	NEC Graph Representation	12
2.3	Graph Mode Proposed by [NgJo]	14
2.4	Hsu's Circle Graph	16
3.1	Block Diagram of a "Turnkey" Synthesis System	24
4.1	Line of Diffusion Layout Style	28
4.2	Example Cell Structure	30
4.3	Graph Model Proposed by [UevC]	32
4.4	Graph Without Single Euler Cover	34
4.5	Complex Graph with Single Euler Cover	36
4.6	Example Transmission Gate Structures	38
4.7	Virtual Grid Used by Routing Algorithm	40
5.1	Rambo Software Org-Chart	43
5.2	Example of Cluster Expansion	47
5.3	Expansion of Two Level Clusters	50
5.4	Formation of Layout Slices	56
6.1	Simple Compilation Example	63
6.2	Effect of Primary Outputs on Compilation	71
6.3	Compilation with Transmission Gates	76
6.4	Compilation of JK flip-flop with Scan Inputs	79

LIST OF APPENDICES

Appendix	Title	Page
A	Rambo Data Structures	84
B	FormClusters Program Listing	85
C	ExpandClusters Program Listing	89
D	Glossary of Terms	99

1 INTRODUCTION

1.1 General Overview

Integrated Circuits (IC) Technology is preparing to outgrow the frontiers of Very Large Scale Integration (VLSI). Monolithic designs of 100,000 transistors or more have become common, and the rate of growth is still very high. How well we harness that fast paced technology, however, depends on the Computer Aided Design (CAD) tools that we put at our own disposal. The development of such tools has consistently trailed that of fabrication technology in recent years. This situation has caused a chronic delay in the availability of new technologies and has inhibited the proliferation of Application Specific IC's (ASIC's) by contributing to high development costs.

This thesis addresses the issue of computer aided design in the context of IC synthesis. It is based on the development and implementation of a software tool for the automatic synthesis of CMOS standard cells. The proposed system is aimed at facilitating the design and maintenance of complete libraries, and will considerably reduce the time required to accommodate changes in process technology.

The results presented herein reflect the conclusions drawn from experimentation as well as an extensive literature survey.

1.2 Review Of Standard Cells Systems

Standard Cells have become widely used in the design of VLSI circuits. They consist of the low level building blocks that IC designers dispose of when creating circuits. Such cells are normally pre-defined and stored in a fixed cell library. Their complexity normally ranges from that of a two transistor inverter cell up to approximately 50, transistors, as would be the case for a JK flip-flop with scan inputs. More complex cells such as counters or Arithmetic Logic Units (ALU) are normally created hierarchically and are termed Macrocells. Larger scale components such as memory blocks and Programmable Logic Arrays (PLA) are also possible.

The use of standard cells largely facilitates logic designs over the full custom approach, at the cost of reduced area efficiency. They are normally designed with uniform features (i.e. pitch, vdd, vss and i/o positions etc.) so as to achieve high packing density and ease of use. Such cells also lend themselves easily to automatic layout methods, yielding fast turnaround for most designs.

In a typical Standard Cell system, cells are first placed in rows by abutment, according to some placement scheme and objective function. Many such placement techniques have been presented in the literature: [Ueda], [DuKe], [Laut], [Rive], [Huds].

The resulting rows of cells are separated by rectangular routing channels which can also be handled

automatically. The channel routing problem has been the object of many publications: [Lee], [Yosh], [RiFi], [Reed], [BuPe], [JoSi].

For larger designs, it may be useful to partition the overall circuit into functional blocks which may be laid out individually and then assembled hierarchically on a pad frame. This introduces a series of new problems that very often need to be addressed jointly. First, the floorplanning step consists of positioning the functional blocks and of determining their aspect ratio. In the global routing phase [Dai] [Cies], channels are created for inter block routing. Nets also have to be assigned to specific channels where a choice exists. A typical standard cell floorplan is shown in Figure 1.1.

A modern standard cell system is a very complex collection of design automation software, graphics support and simulation programs. It seems almost surprising that one of the very few steps of the design process that has yet to be automated is the design of the standard cells themselves. This document addresses that issue in particular by stating the problems that it involves and proposing potential solutions.

Figure 1.1 Typical Standard Cell Floor Plan

1.3 Motivation For The Research

The concept of automating the design of Standard Cells has been recognized to present the following advantages:

- o Design Portability - This is certainly the predominant factor. It refers to the ability to accommodate variations in design rules. These variations may range from a simple quantitative reduction in feature dimension to more fundamental changes such as the addition of a second metal layer.
- o Correctness by Construction - Machine generated cells promote correctness by construction, by limiting user involvement to the specification of certain functional and electrical parameters.
- o Documentation and Maintenance - Important reduction in manual entry for purposes of documentation will be achieved as a result of automating the design process.
- o Ease of Characterization - Performance evaluation can be performed using readily available information. This will eliminate the need for feature extraction.

o Flexibility - Circuits or systems designers may create new cells as the need arises without the assistance of an IC expert. Furthermore, by clustering "loose" gates into leaf cells at design time, much of the routing between them can be implemented within the active cell area.

o Parameterization - An automatic synthesis method is easily amenable to the generation of parameterized cells.

1.4 Organization Of The Thesis

The main objective of this thesis is to investigate the concept of Automatic Standard Cell Synthesis. The results of an extensive literature survey of related work are presented in chapter 2. In chapter 3, the building blocks of a "turnkey" Standard Cell synthesis system are defined and analysed in terms of complexity of implementation as well as feasibility. Finally, a prototype cell compilation system is described in detail with examples, to illustrate the most important steps of the compilation process. Chapter 4 is an overview of the system whereas chapters 5 and 6 present a description of each building block and a series of illustrative synthesis examples respectively. Conclusions and recommendations for future work are the object of chapter 7.

2 RELATED WORK

2.1 Survey Of Relevant Literature

Research on certain aspects of Standard Cell design automation has been reported in the literature. A sample of the most important papers, presenting either an algorithm or a complete system is reviewed.

2.1.1 TOPOLOGIZER - TOPOLOGIZER is a rule-based automatic synthesis system for leaf cell design [KoWe1] [KoWe2]. Input to TOPOLOGIZER consists of a transistor connection list and an environment description which contains the desired cell aspect ratio and external signal placement. Following a random initial transistor placement, a 20-rule expert system attempts to find a good placement, by representing each transistor as a tile. Each tile has three sections corresponding to the source, gate and drain of the transistor, which makes the placement process analogous to the game of dominoes. The objective is to maximize the number of adjacent domino nodes that match. Figure 2.1 illustrates that placement strategy, for the case of a 5/2 cell aspect ratio.

After completing the placement, TOPOLOGIZER invokes the routing expert. Again, a correct but wasteful initial solution is found and then improved upon iteratively. The result is a symbolic description of the desired cell to be passed on to the MULGA symbolic layout system for final layout compilation [West1].

2.1-1 Initial placement

2.1-2 Final placement

————— p-type transistors
 - - - - - n-type transistors

Figure 2.1 TOPOLOGIZER Placement Strategy

The techniques used by TOPOLOGIZER are guaranteed to yield a solution, the quality of which depends on that of the placement and routing expert systems. However, the disjoint nature of these two operations represents a fundamental limitation. Furthermore, placement based on individual transistors may not be practical for very large cells with fixed pitch constraints.

2.1.2 TALIB - TALIB is a knowledge-based expert system for cell layout, tailored for a single metal layer nMOS process and a Manhattan geometry [KIMS]. It capitalizes on the fact that typical circuits are built out of a number of commonly occurring substructures. Domain specific knowledge is used to recognize certain patterns and manipulate them to determine the relative placement of individual transistors. The user specifies transistor interconnects as well as a number of layout constraints (terminal position, feedthrough nets, routing layer and design rule information) and TALIB produces final layout masks by manipulating a certain number of "layout models". Performance of the system has been found to be dependent on the number of transistors in the input description as well as the boundary conditions specified by the user. The main drawback of the TALIB system is the use of predefined "layout models". Such models severely limit the flexibility of a layout system.

2.1.3 DUMBO - DUMBO constitutes some of the earlier work in logic circuit compilation, and is focussed on nMOS

technology [WNMD]. Cells are described to the compiler in terms of electrical information (transistor interconnect) and structural information, suggesting the desired layout. Various component templates exist for each element. User hints may be used to influence the choice that the system will make among these templates. The compilation process is comprised of five algorithmic steps: Placement, Orientation, Expansion, Wiring and Routing. These produce a stick diagram that can be compacted into a layout. Experience has shown that layouts generated by DUMBO are 120% larger than hand designs. Its creators have identified deficiencies in the placement and wiring phases as probable causes.

2.1.4 BTL Semi Manual Method - This method is based on a CMOS layout style which lends itself to automation and design rule updatability [LeCJ]. The design style consists of a double row of transistors located on a horizontal grid of equal spacing, with pFETs on top and nFETs at the bottom. This is also referred to as the "line of diffusion" layout style. Input description is in a symbolic code which fully describes all transistors as well as all the metal nets, including their positions on an orthogonal grid. Hence, the only automation as such is the layout compilation process. Technology parameters are contained in a separate file for ease of modification. The mask description is merged with a user defined command file and used as input of a characterization and verification system.

2.1.5 NEC Gate Array System - This program is based on placement and routing algorithms for automatic layout of functional blocks for CMOS gate arrays [Noda]. The basic cell consists of two p-channel transistors connected through a common diffusion and two n-channel transistors also sharing a common diffusion. Two metal layers are available for interconnect. Circuit data for the blocks is composed of transistors, external terminals and a set of nets between them. The main objectives of the system are (i) to reduce the number of cells employed by the block and (ii) to reduce the amount of interconnect outside the active cell area and on the second metal layer.

In the placement phase, a graph representation of the circuit, shown in Figure 2.2, is used to assign PN pairs to cells. This is achieved by coupling PN transistor pairs which may be packed into a basic cell (common sources or drains). Then, the couples are assigned to cells according to signal flow within the cell. This initial placement is improved upon by pairwise interchanges in order to minimize the maximum number of interconnections crossing vertical cut lines.

In the routing phase, a channel router is used that treats both the p-channel and n-channel areas as routing space. The aim is to achieve all wirings with the primary use of the first metal layer and using the least number of tracks.

2-2-1 Circuit Schematic

2-2-2 Graph Representation

2-2-3 PN pairs

Figure 2.2 NEC Graph Representation

The system shows impressive results where the final layout is as good as manual layouts in most cases. Even though the objective of the placement technique is to minimize the number of gate-array basic cells used, which is irrelevant in the case of CMOS standard cells, the principle of grouping transistors, according to a certain affinity, before placement, is directly applicable to more general problems. It should therefore be considered when conceiving a placement technique in the context of standard cell synthesis.

2.1.6 Graph Theoretic Approach - A second graph model to represent transistor interconnections has been proposed [NgJo], which differs from that of section 2.1.5, and is shown in Figure 2.3. In this model, a net is represented as a vertex in the graph. Hence a transistor is defined implicitly as a pair of directed edges terminating on the net vertex to which the gate is connected.

Such a graph representation reduces the layout problem to that of embedding a graph in a set of planes. The layout that results is impractical however due to the fact that most connections are implemented in either polysilicon or diffusion, and the layout style yields an irregular cell topology. Further, the technique was developed with an nMOS technology in mind and would not take advantage of the structural regularity of CMOS.

2.3-1 Circuit Schematic

2.3-2 Circuit Graph

T	S	D	GateNode
n1	2	1	out
n2	5	3	A
n3	6	4	B

2.3-3 Internal Representation

Figure, 2.3 Graph Model Proposed by [NgJo]

2.1.7 CALMA Project - The CALMA Project [NoHe] is aimed at constructing CMOS cell layouts from transistor netlists. It uses a diffusion line tracing method, similar to that used in manual layout, to achieve an efficient transistor placement. "Loose" wiring is performed by representing the routing space as a switchbox routing problem in the form of a circle graph [Hsu]. Figure 2.4 illustrates the concept of the circle graph, and its linear equivalent. Optimization of this loose wiring is accomplished by compaction.

2.1.8 Primitive Based Cell Design - As the name implies, this manual method consists of using several transistor level primitives to construct cells [ToSa]. For example, a "and-into-nor" gate may be constructed out of the "nand" primitive and the "output" primitive. These primitives are of constant height and may therefore be abutted and connected. A certain degree of design portability is achieved by having to re-design the primitives only when certain changes occur in the target process. At the time of publication (1983), the layout process was still a manual process, but it would certainly lend itself to automation.

2.4-1 Hsu's Circle Graph

2.4-2 Linear Representation

Figure 2.4 Hsu's Circle Graph

2.2 Summary Of Survey

The combined knowledge encompassed by these papers represents the current state of affairs in the field of Standard Cell Design Automation, as far as published results are concerned. Here are some of the specific shortcomings exhibited by these publications:

- o Complex input descriptions - The user must define a transistor net list sometimes involving aspect ratios, using a hardware description language. Hence, the problem of finding optimal transistor net lists is not addressed.
- o Impractical~~er~~ Layout Styles - Generated cells exhibit non-uniform pitches and inefficient compaction due to the absence of 45 degree bends. There are no provisions made for split contacts or feedthrough nets.
- o Cell Characterization - Automatic cell characterization has not been addressed in the context of cell synthesis.

3 OUTLINE OF A TURNKEY STANDARD CELL DESIGN SYSTEM

After having established the usefulness of a synthesis tool for automatic design and maintenance of standard cells, this chapter discusses the various attributes of an idealized "turnkey" design system. The key features that such a system should possess are reviewed with the emphasis being put on Design Portability. The complete cell compilation process is also described which leads to the presentation of a tentative architecture for the implementation of the system.

3.1 Review Of Process Independence

Process independence is certainly the most important argument in favour of the development of an automatic synthesis system for standard cells. In order to achieve process independence however, a sufficiently high level of abstraction has to be used in the internal representation of the cell, at each compilation step, in order to keep it as remote as possible from any specific technology. Furthermore, the algorithms used have to be implemented in their most general form possible to facilitate their adaptation to future technologies.

A superb review of Process Independence exists in the literature [RoHB], and is summarized in the following two sections.

3.1.1 Existing Paths To Process Independence - The "Stick Diagram" [Will] was the first attempt at achieving some degree of process independence, by representing a circuit layout in a form which is not bound to any particular process. However, fleshing and compaction are difficult and computationally time consuming tasks and tend not to yield an acceptable degree of density.

A later attempt was referred to as "lambda-based rules" [MeCo] which have become extremely popular. Their effectiveness in terms of process independence is limited by the fact that design rules, in most cases, do not scale linearly. Also, the minimum dimensions of features on individual layers vary widely and interactions between layers are very complex. Finally, and most importantly, the different rules do not hold their relations over different processes.

3.1.2 Analysis Of Design Rules - Differences between sets of process design rules can be classified as follows:

- o Quantitative changes affecting widths, spacing and overlaps which can normally be handled by compaction.
- o Qualitative changes in the form of existing rules between different processes e.g. differences between p-well, n-well and twin-tub processes.

- o Fundamental differences - By far the most difficult case to handle since provisions have to be made for future enhancements. Some examples of fundamental features are whether or not polysilicon is allowed to run alongside metal (reflexion rules), whether or not 45 degree bends may exist, and whether or not contacts to polysilicon over active area are permitted (anticipated).

3.2 Other Attributes Of A Turnkey System

In addition to process independence, a Turnkey Standard Cell Design System should have a number of other attributes, described in this section.

- o Ease of Use - Use of the system should require very little knowledge of integrated circuits if any, so as to allow logic designers to create cells independently.
- o Simple Input Description - Input at the logic level is much preferred over transistor netlist. The family of logic primitives should be augmented with special symbols such as transmission gates to enhance flexibility. This implies that the mapping of logic level descriptions into transistor net lists becomes a task of the cell compiler.
- o Ability to handle constraints - The user should be permitted to specify performance parameters for the cells as well as special layout features such as feedthrough

nets and split contacts. In addition, the layout style should be programmable to allow construction of cell libraries with uniform features. The use of a cell template is one possible way to achieve such programmability.

- o Possibility of User Intervention - User intervention should be allowed, but not required, at various stages of the compilation process, to facilitate manual optimization.
- o Sufficient Output Data - The output data produced by the system should be sufficient for the cells to be used immediately. This includes complete layout as well as characterization information, both in terms of electrical and functional behaviour.

3.3 Complete Compilation Process

The complete cell compilation process may be described as a collection of seven sub-tasks.

1. Mapping of logic level input description into an approximately optimal transistor netlist. This corresponds to a many-to-one mapping, and the choice of the best solution must be based on the objective placement strategy.

2. Grouping of transistors into clusters that can share diffusion areas. This has been found to yield considerable area savings for CMOS.
3. Transistor Placement that takes into account signal flow within the cell, the objective routing strategy, as well as the target layout style. The placement algorithm should operate on clusters formed in 2. for the initial phase but allow smaller objects for optimization.
4. Routing of metal connections within the active cell area. Due to the inherent complexity of the problem, it should be performed in two phases:
 - Algorithmic method for simple nets,
 - Heuristic method for backtracking and to complete complex nets.

Provisions may have to be made to allow modification of placement in the course of routing, in order to guaranty a solution.
5. Layout Compilation/Compaction including the modification of layout features, to achieve high efficiency. This is the critical phase for Design Portability and should be implemented with future developments in mind as much as possible.
6. Simulation file extraction for both electrical and functional characterization. Cell characterization is an important aspect of their design and should not be left out of a "turnkey" system.

7. Automatic generation of simulation control command file.
This may require a certain degree of user supplied information in order to achieve efficiency.

These sub-tasks represent problems which need to be addressed and the order in which they are presented does not necessarily reflect data flow through a compilation system. For example, the issues of placement and routing are intimately related and a specific placement may be altered during the routing phase.

3.4 "Turnkey" System Architecture

Figure 3.1 shows a block diagram of a complete standalone Standard Cell Synthesis system that would support all the features discussed in previous sections. Each block is now discussed in detail.

3.4.1 Expert CMOS Designer - The Expert CMOS Designer is a program which is capable of recognizing certain structures such as and-into-nor or or-into-nand and group them to maximize the number of common diffusion connections. Further, it attempts to form pairs of transmission gates which will be implemented using a simple structure. It should also be capable of a certain level of circuit critiquing to correct anomalous logic designs. For example, it is ill-advised to drive a storage node with transmission gates due to increased susceptibility to noise coupling.

Figure 3.1 Block Diagram of "Turnkey" Cell Synthesis System

Such a situation should be at least flagged if not corrected. The output of the Expert CMOS Designer is an approximately optimal transistor net list corresponding to the input logic description. The output description also contains information regarding transistor clusters, that may facilitate placement.

3.4.2 Optimizer - The purpose of the optimizer is to combine the output of the Expert CMOS Designer with user supplied performance parameters to determine transistor aspect ratios. This program should also attempt to vary the sizes of transistors according to their positions in a series structure or according to internal capacitive loading.

3.4.3 Placer-Router - As the name implies, this module performs placement and interconnect. It generates a metal cover for the cell in order to complete routing of all the nets that have not been fully implemented in polysilicon or diffusions.

3.4.4 Layout Compiler - This module combines information about placement, routing, transistor sizes and special layout features to form a complete set of masks. Design rules and layout style (cell template) are read in at this point only, so as to minimize the amount of re-work required should any changes become necessary.

3.4.5 Feature Extraction - Since the circuit net list is already available to the machine at this point, only parasitic capacitance and wire resistance information needs to be extracted from the layout.

3.4.6 Simulation Expert - The Simulation Expert analyzes the circuit to generate a simulation command file necessary to drive the simulation for both electrical and functional characterization.

4 OVERVIEW OF PROTOTYPE CELL COMPILER

Rambo is a prototype cell compiler that was implemented to investigate the feasibility of the "turnkey" system presented in chapter 3. Although its current implementation does not support all the attributes outlined, its architecture is such as to promote system development towards that end. Rambo accepts three files as input data: (i) a logic description of the cell, comprised of logic gates, transmission gates, etc., (ii) a description of the cell structure, based on the target layout style, and (iii) a technology file containing design rule parameters. The system then generates an approximately optimal transistor netlist, performs placement and interconnect, and compiles a complete set of layout masks, represented as CIF data.

4.1 Layout Style

The "line of diffusion" layout style, shown in Figure 4.1, has become extremely popular for CMOS standard cells. As the name implies, it consists of two parallel lines of diffusion for p-channel and n-channel devices respectively. Polysilicon gates run perpendicular to the diffusions whereas metal connections run parallel to them. This layout style yields uniform pitch cells which is an important feature when automatic placement and routing tools are to be used. The cells may be alternatively arranged in double rows sharing one of the two power busses.

Figure 4.1 Line of Diffusion Layout Style

For the purpose of illustration, a sample cell structure with five metal lines in the p-region and three in the n-region is shown in Figure 4.2.

This layout style promotes global optimization of transistor sizes. A model has been proposed [Kang] to calculate the width of p-channel and n-channel devices in order to minimize the product of propagation delay and chip area while allowing respectable noise margins. This model is based on the following four effects of increasing transistor sizes:

1. increased drive capability; minimum dimension transistors have been found to produce unsatisfactory results.
2. increased capacitive loading for each internal net.
3. increased resistance due to longer poly runs.
4. increased area.

However, the proposed synthesis system would capitalize on the increase in active area by increasing the amount of interconnect within cell area. This is expected to counterbalance at least partially the increase in cell area due to increasing transistor sizes.


```

*****
FILE NAME:    cellstruct.lsp
MODULE NAME:  Cell structure
CONTENTS:     This file contains a set of cell structure parameters
               for the physical realization of the cell.
*****
(setq num-p-lines 5)
(setq num-n-lines 3)
(setq num-lines 10)

(setq p-l 5)
(setq g-l 4)
(setq n-l 3)

(setq border-spa 50)
(setq v-edge-end-spa 9)
(setq v-edge-poly-spa 17)
(setq end-end-spa 32)
(setq end-gate-spa 20)
(setq end-poly-spa 23)
(setq gate-mid-spa 20)
(setq poly-poly-spa 20)

(setq n-top-pos 116)
(setq n-bot-pos 50)
(setq p-top-pos 264)
(setq p-bot-pos 151)
(setq pol-top-pos 289)
(setq pol-bot-pos 25)
(setq vdd-top-pos 274)
(setq vdd-bot-pos 254)
(setq vss-top-pos 40)
(setq vss-bot-pos 20)

(setq ytabs-vect (make-array (list num-lines)
                             :initial-contents
                             '(30 59 83 107 133 159 163 207 231 255)))

```

Figure 4.2 Sample Cell Structure

4.2 Translation Of Logic Descriptions Into Transistor Structures

This section addresses the problem of mapping logic circuit descriptions into transistor interconnects. A graph model attributed to T. Uehara and W.M. vanCleemput is presented, which may be used to represent two-level logic functions for implementation with CMOS [UevC]. Conclusions drawn from analysis of the graph model have been used to develop an algorithm to generate transistor interconnects directly from logic diagrams. This method is described in section 4.2.2. Other structures, such as transmission gates are handled as special cases.

4.2.1 The Graph Model - The objective of the graph model is to maximize the use of common diffusion connections in the implementation of logic gates. The model requires that the topology of the p-MOS and n-MOS sides of the circuit be each other's dual. Figure 4.3 illustrates the model which is now summarized.

The graph is separated into a p-side and an n-side representing p-MOS and n-MOS portions of the circuit respectively. Every source/drain potential is represented as a vertex and labelled with the corresponding signal name. Every transistor is represented by an edge connecting the source and drain vertices and is labelled with the gate signal name.

4.3-1 Logic schematic

4.3-2 Graph representation

4.3-3 Outline of the layout

Figure 4.3-Graph Model proposed by [UevC]

The result is that if two edges x and y are adjacent in the graph model, then it is possible to place the corresponding transistors side by side and to connect them via a common diffusion. The key point is that if there exists an Euler path, i.e. a sequence of edges that contains all the edges of the graph model, then all the transistors can be chained by a diffusion area. If there is no Euler path, then the graph can be decomposed into several subgraphs which have Euler paths. The problem consists of finding a sequence which constitutes a Euler path in both the p-side and the n-side of the graph to achieve maximum packing density with a "line of diffusion" layout style. Whether a Euler path exists or not for a given graph, depends on whether or not it is possible to traverse each edge in sequence, exactly once, starting from an arbitrary position.

There are two problems with this approach that need to be addressed for this method to be used successfully.

o Multiple Valid Graphs - Given a logic diagram, there may be many ways to draw the corresponding graph, some of which may yield optimal solutions whereas some others will not. Figure 4.4 shows an example of a graph representation of a circuit equivalent to that of Figure 4.3, but which does not have a single Euler cover.

o Exhaustive Search - To find an optimal solution given a "good" graph requires an exhaustive enumeration of all possible decompositions of the graph.

4.4-1 Logic Schematic

4.4-2 Graph representation

4.4-3 Outline of the Layout

Figure 4.4 Graph Without Single Cover

The graph model however has inspired the compilation scheme described in the next section.

4.2.2 Logic Gate To Transistor Interconnects Translator.

Implementation of the new proposed translator is based on the following conclusion drawn from the graph model: Considering any two-level logic function of type and-into-nor or or-into-nand, the corresponding graph model will have a single Euler cover if the number of first level gates with an even number of inputs is not greater than two, and they are located at opposite ends of the graph. The reason for this simple but important result is that a first level gate with an odd number of inputs will be reflected in the graph by the same odd number of edges between two points. An infinite number of those may be traversed in sequence by traversing each edge only once. On the other hand, a first level gate with an even number of inputs will be reflected in the graph by the same even number of edges. Attempting to traverse such a group of edges will inevitably yield coinciding start and finish positions. At most two such "self-loops" may be allowed provided that they occur either at the beginning or at the end of the traversal, i.e. at either end of the graph. An example is shown in Figure 4.5. Hence, the compilation process consists of forming clusters out of the gates that meet the above criteria and to implement them by aligning the transistors on two parallel lines of diffusion. This process is presented in detail in Chapter 5.

Figure 4.5 Complex Graph with Single Euler Cover

4.2.3 Compilation Of Other Structures - It may also be advantageous to form clusters out of gates that are not so directly related. For example two transmission gates may be coupled and implemented as shown in Figure 4.6. These two structures are shown here simply to illustrate how it may be advantageous to merge two transmission gates into a cluster. They are example structures that would result from further processing and are not stored for use as layout models.

Furthermore, two unrelated gates may be merged to share common diffusions connected to Vdd and Vss. However, there exists no formal method for this merger and whether or not any benefits would result greatly depends on the specific routing situation. Hence, this operation may be performed at the translation level or may be postponed until the placement or routing phases. It is not currently supported.

4.3 Mapping Of Transistor Structures Onto Cell Layout

Following compilation of a transistor netlist corresponding to the cell logic description, the mapping of that intermediate description onto final layout masks consists of three tasks: (i) transistor placement, consistent with the target layout style, (ii) routing of internal connections and (iii) compilation of abstract circuit representation into GIF data.

4.6-1

4.6-2

Figure 4.6 Example of Tg-mux Structures

4.3.1 Placement Technique - Transistor placement is performed on a "cluster" basis. Clusters created in the course of transistor netlist compilation are placed in such a way as to facilitate internal connections to gates.

The rationale for the technique used resides in the fact that internal connections to gates constitute the major layout bottleneck. The layout style is such that there can only be one metal line between the p-region and the n-region and hence only one possible contact location for each polysilicon gate. Therefore, the key objective of the placement algorithm is to facilitate such gate connections. In order to achieve this goal, the clusters are placed according to signal flow through the cell from left to right.

Following this initial placement, the clusters are broken down into their smallest resolvable "pieces" without breaking diffusion strips. Although not currently implemented, an interactive improvement phase is envisaged that would manipulate these pieces in order to optimize the transistor placement.

4.3.2 Routing Technique - The routing algorithm is based on a two-dimensional virtual routing grid, as illustrated by Figure 4.7. Each horizontal metal line within the cell boundaries constitutes a row in the routing grid. The number of rows is therefore a function of the cell structure. The number of columns however will depend on the cell being created. The routing task then consists of

generating a series of two-point connections for the metal layer, corresponding to the cell transistor netlist. The virtual grid is used to mark off locations already covered with metal during the progression of the routing phase as well as to indicate contact location. The algorithm attempts to complete gate connections first since they constitute the major routing bottleneck. Further, the p-region is favoured over the n-region when a choice exists due to its potentially larger number of tracks. The use of a virtual routing grid will make it possible for future enhancements of the routing program to allow modification of transistor placement, whenever required by a particular routing problem.

Figure 4.7 Virtual Grid Used by Routing Algorithm

4.3.3 Generation Of CIF Code -

The final phase of the cell synthesis process is the transformation of a two-dimensional grid circuit representation to mask geometries. First, the virtual grid points are mapped onto physical layout locations. In the current implementation, this mapping is a non-linear scaling of axis, where two points that have the same x-coordinate on the virtual grid will also have the same x-coordinate on the physical grid.

Following grid mapping, the compilation process is concluded by fleshing polysilicon gates, two-point metal connections and contact windows.

5 PROTOTYPE CELL COMPILER

5.1 General System Organization

Figure 5.1 presents a software organization chart for the complete prototype. Each rectangular box represents a software module whereas directed edges refer to data transfers between modules. In the following sections, each of the software modules will be described in detail.

5.2 Cluster Formation

The purpose of the "FormClusters" module is to cluster gates that may potentially be implemented in the same device wells. The first step is to look for two-level clusters such as "and-into-nor" or "or-into-nand". The next step is to couple transmission gates into simple multiplexer structures. Finally, all remaining gates in the input description are assigned to single-gate clusters of the appropriate type. Refer to Appendix B for a listing of the Lisp implementation of the "FormClusters" module.

Figure 5.1 Rambo Software Organization Chart

5.2.1 Or-into-nand Clusters - An attempt to create an "or-into-nand" cluster is made whenever a gate of type "nand" is encountered in the input description. The algorithm is:

- create a new instance of the "cluster" data structure
(see Appendix A for data structures)
- consider all "nand" gate input signals
 - if signal is the output of a gate of type "or" and signal is not a member of the cell outputs then add "or" gate to cluster gate list
- if "nand" gate is the only gate of cluster gate list then set cluster type to "nand" else set cluster type to "or-into-nand"
- in the case of "or-into-nand" clusters only, add a dummy buffer for each input of the "nand" gate not coming from a gate within the same cluster.

The reason why primary cell outputs are to be excluded from a two-level cluster is that signals from the first level of logic of a two-level cluster become implicit signals and are not physically available in the final structure. Refer to sections 6.1 and 6.2 for examples of how primary cell outputs affect the final layout structure.

5.2.2 And-into-nor Clusters - These are created in the same way as or-into-nand by exchanging the roles of "nor" and "and" gates with "nand" and "or" respectively.

5.2.3 Transmission Gate Clusters - A cluster of type "tg-mux" is created whenever a transmission gate is found in the input description. It is assumed that two transmission gates with complementary gate signals and one common diffusion connection can always be found. This restricts the use of transmission gates to two-to-one multiplexers, which is quite reasonable considering the fact that allowing single transmission gates would violate the layout style that has been assumed and would represent the same area cost as a two-to-one multiplexer. The algorithm is:

- create a new instance of the "cluster" data structure
- set cluster type to "tg-mux"
- find other "tg" in the input description by matching output signals.

5.2.4 Single Gate Clusters - It is necessary for proper operation of subsequent modules that all the gates of the input description be assigned to a cluster. Hence, single gate clusters need to be formed as follows:

- a gate of type "inv" forms a cluster of type "inv".
- a gate of type "and" a cluster of type "nand-into-inv".
- a gate of type "or" a cluster of type "nor-into-inv".

5.3 Cluster Expansion

The purpose of the "ExpandClusters" module (Appendix C) is to produce transistor interconnects for each cluster. The result is a set of three ordered lists of signals corresponding to the p-diffusion and n-diffusion areas as well as gate signals, as shown in Figure 5.2.

5.3.1 Expansion Of Two-level Logic Clusters - In the expansion of two-level logic clusters, and-into-nor clusters and or-into-nand clusters are treated in the exact same way. Differentiation between the two types is performed at the final step only.

Each input to the second level logic gate corresponds to a "stage". During preprocessing, each stage is assigned a parity (either odd or even) according to the number of cluster inputs associated with it. Also, a "weight" attribute is computed for each stage equal to the number of gate connections within that stage. A gate connection has to be made whenever the gate signal is not a primary cell input. Figure 5.3-1 (page 50) illustrates the concepts of weight and parity. The reason for the weight assignment comes from the subsequent placement algorithm, which is based on signal flow through the cell from left to right. It is therefore advantageous to place connected gates as far to the left as possible during cluster expansion. The position of unconnected gates is of lesser importance unless a specific i/o ordering has to be observed.

5.2-1 Logic symbol

pdiff-signals = (C vdd C)

gate-signals = (A B)

ndiff-signals = (vss i0 C)

5.2-2 Internal representation of expanded cluster

5.2-3 Implied layout structure

Figure 5.2. Example of Cluster Expansion

Following weight and parity assignments, the expansion process is carried out according to the number of even stages. The method used is based on the analysis of the graph model and has been outlined in section 4.2.2. (It consists of finding an ordering for the first level stages that will yield a single Euler cover. Such a situation will exist only if the number of even stages is less than or equal to two. In cases where the number of even stages is greater than two, sets of stages have to be formed that will conform to that restriction. Set formation is equivalent to splitting the graph into sub-graphs that each have a single Euler cover. The detailed algorithm is:

- if no even stage exists

- then sort odd stages according to weight
 - and form single set of stages for further processing.

- if one even stage exists

- then sort odd stages according to weight
 - and form single set of stages for further processing
 - placing even stage in front of the list if its weight exceeds the highest weight of odd stages, and placing it at the back otherwise.

- if two even stages exist

- then sort odd stages according to weight
 - and sort even stages according to weight
 - and form single set of stages for further processing
 - placing the first even stage at the front of the list
 - and the second even stage at the back of the list.

-if more than two even stages exist
 then sort odd stages according to weight
 and sort even stages according to weight
 and form one set of stages for further processing
 placing all odd stages and two of the even stages
 as for the case of two even stages
 and form another set of stages comprised of the remaining
 stages.

The next step consists of compiling two diffusion strips from the description of "stages". The first strip contains transistors arranged in parallel and associated with each input signal of the stage and the second a series combination with the same gate signals. For clusters with more than two even stages, it is necessary to leave a gap between the two sets of stages. This is reflected in the second level of nesting used to describe each strip.

In the wrap-up phase, the two diffusion strips are bound to the appropriate side (either p- or n-) depending on the nature of the cluster (and-into-nor or or-into-nand). Figure 5.3 illustrates the expansion process of two-level clusters.

5.3.2 Expansion Of Single-level Logic Clusters -
 Single-level logic clusters are "nand", "nor",
 "nand-into-inv" and "nor-into-inv" clusters. Their
 expansion is carried out in much the same way as two-level
 logic clusters without consideration for logic stages.

5.3-1 Cluster logic schematic

stages = ((A i0) (i1))

5.3-2 Formation of stages

pdiff-signals = (vdd i2 B vdd)
gate-signals = (A i0 i1)
ndiff-signals = (i3 vss i3 B)

5.3-3 Internal representation of expanded cluster

5.3-4 Implied layout structure

Figure 5.3 Expansion of Two-level Logic Clusters

The first step consists of implementing the logic function (either nand or nor) by simply constructing one series arrangement and one parallel arrangement of the appropriate number of transistors. The two diffusion strips hence created are then assigned to the p-side or the n-side depending on the logic function required, and an inverter stage is appended if required.

5.3.3 Expansion Of Clusters Of Type "tg-mux" - Expansion of the "tg-mux" cluster is a mere mapping of the appropriate signals onto a predefined structure. The structure associates signal names with diffusion areas and polysilicon lines and does not impact on signal routing at all.

5.4 Detection Of Metal Nets

The role of the "FindNets" module is to identify all the metal connections that will have to be made in the final layout. The technique is to apply the following rule to all the clusters after expansion;

- if a given signal x appears more than once in a cluster
- and that signal is not a cell input
- and a net does not already exist for it
- then create a new instance of the "net" data structure
- and set the net signal to be signal x.

Primary cell inputs are excluded since they are assumed to be connected externally. Note also that limiting the search to single clusters will always yield the correct solution for CMOS since any signal common to many clusters will appear twice in at least one of them. The only exception to this rule is the case of primary cell inputs connected directly to a single diffusion area. This is currently handled by special processing.

5.5 Cluster Placement Algorithm

As was discussed in Chapter 4, initial transistor placement is performed on a "cluster" basis. Furthermore, it is based on signal flow through the cell from left to right.

The first step consists of creating instances of relations between clusters that share at least one net. These relations define the relative placement of two clusters x and y in a one dimensional space by placing x to the left of y and y to the right of x . The algorithm to create a list of relations is:

- do for each pair of clusters
 - if clusters share at least one net
 - and their relation does not violate existing relation
 - and their relation does not already exist
 - then create new instance of the "relation" data type
 - and propagate relation if necessary.

Relation propagation is used to eliminate the possibility of cycles in the relation list. It simply means that if a cluster x has been placed to the right of cluster y , new relations should be created to force x to be to the right of all clusters that are to the left of y .

The placement algorithm as such, utilizes these relations to determine cluster placement. The algorithm is:

- do for each relation
 - if neither right or left cluster of relation are placed then place both clusters side-by-side at the beginning of the array.
 - if only the left cluster is placed then place right cluster to its right and go to clean-up step
 - if only the right cluster is placed then place left cluster to its left and go to clean-up step
 - if both clusters have been placed then go to clean-up step
 - clean-up step:
 - if placement of clusters affected by current relation creates a violation of another relation
 - then interchange clusters

The result of the "PlaceClusters" module is an ordered list of clusters.

5.6 Formation Of Layout Pieces

The purpose of the "FormPieces" module is dual. First, clusters that have been implemented as two sets of stages may now be split into two "active pieces". This will greatly enhance the flexibility of the placement program by allowing mirroring on a piece basis as opposed to a cluster basis. The second purpose served by this module is the addition of so called "poly pieces". Such additional polysilicon lines are necessary to bring cell outputs outside the cell area into the routing channel, and occasionally to carry cell inputs into the cell area in the case of cell inputs not already routed through polysilicon. For the case of cell outputs, the algorithm is:

- for each cell primary output
- if signal x has no "sink"
- then create a new instance of the "piece" data structure
- and set the piece type to be "poly"
- and set the piece gate signal to signal x.

For the case of cell inputs:

- for each cell primary input
- if signal x is connected to a transmission gate
- and signal x is not a gate signal
- then create a new instance of the "piece" data structure
- and set the piece type to be "poly"

and set the piece gate signal to signal x.

5.7 Piece Placement Algorithm

In its current implementation, the "PlacePieces" module performs placement of "active" pieces in a way consistent with the placement of clusters from which they were derived. In addition to this, the position of "poly" pieces is defined so as to minimize the span of the nets to which they are connected.

5.8 Formation Of Slices

The "FormSlices" module decomposes layout pieces into vertical "slices". The term "slice" refers to any layout region to which a distinct connection can be made. Slices may be of four different types:

"poly"; vertical polysilicon wire over field oxide

"gate"; vertical polysilicon wire over active region

"enddiff"; diffusion region between field oxide and a gate

"middiff"; diffusion region between two gates.

Figure 5.4 illustrates the formation of slices for a two input nand gate.

5.4-1 logic symbol

5.4-2 Layout slices

Figure 5.4 Formation of Layout Slices

The algorithm for slice formation is:

- for each piece do
 - as long as current piece not complete do
 - create a new instance of the slice data structure
 - set slice-type and slice-signal fields
 - for each signal that belongs to a net
 - add current slice to list of slices for that net

In addition to compiling a slice list for each net, the "FormSlices" module also determines net types. This is accomplished as follows:

- for each net do
 - if net contains a connection to a gate
 - then set type to be "gate"
 - else if net contains both p-region and n-region points
 - then set type to be "cross"
 - else if net contains p-region connections
 - then set type to be "pdiff"
 - else set net type to be "ndiff"

5.9 Routing Algorithm

The "RouteMetal" module is perhaps the most complex of the system. The routing algorithm is based on a two-dimensional virtual grid, which is an abstract representation of the layout space for the cell. Two arrays

are used as internal representation of the virtual grid: one for the metal layer and the other for contact windows. The number of rows of these arrays is equal to the number of horizontal metal lines within the cell and is therefore a cell structure parameter. The number of columns however, is equal to the number of layout slices for the current cell and will vary from one design to the other.

The contact array stores information about contact locations and will be used directly by the layout compiler for mask creation, by performing a mere ~~fleshing of~~ contact locations to the appropriate dimension. On the other hand, the metal array is only used as a guide to mark grid locations where the metal layer is already being used. An explicit description of connections serves in the generation of mask data for the metal layer. Connections are described internally by using instances of the "conn" data structure, which is capable of representing straight-line two-point connections on the virtual grid. More complex metal connections may be represented by cascading several such instances.

For each instance of the "net" data structure, corresponding connections are created by first checking for a path using the metal array as a guide. The search method for such paths depends on the current net being processed as well as the target cell structure.

The general operation of the routing algorithm is based on the following steps:

1. Mark Vdd and Vss connections - Both metal and contact arrays are marked in positions where a Vdd or Vss connection exists.
2. Mark gate connections - As in 1. for grid locations where a connection exists to a gate.
3. Find and route crossing position of gate nets - Since a gate net will inevitably involve connections to both the p-region and the n-region, it is necessary to compute an appropriate position for a vertical metal segment joining the p-sub-net and the n-sub-net.
4. Route cross nets - Cross nets are nets that have both p-region and n-region connections but no gate connection.
5. Route pdiff and ndiff nets - These nets are routed as far away from the center of the cell as possible. Only straight line connections are attempted at this stage.
6. Complete gate nets - At this point, routing between crossing segment and gate connections is completed. The remaining part of the gate net is split into its pdiff and ndiff components.

7. Route pdiff and ndiff nets created in 6. - This time these nets are routed as close to the center as possible.
8. Combine metal contact arrays into a single "cover" array.

5.10 Layout Compilation

A layout mask representation of the circuit in the Caltech Intermediate Form is generated by the "CompileCell" module. Mapping of virtual grid coordinates onto physical layout locations is achieved through the use of two non-linear mapping vectors. Each entry in the mapping vector corresponds to a physical location and virtual grid positions are used as indices. The vertical spacing vector is a function of the cell structure whereas the horizontal spacing vector has to be computed by considering spacing between each of the various layout "slices". Spacing between the various types of slices should be computed automatically using information from the "design rules" file, in order to insure design portability.

The layout compilation process may be summarized by the following steps:

1. Initialize CIF file.
2. Compute horizontal spacing
3. Compile device wells
 - p-devices
 - n-devices
4. Compile polysilicon layer
 - polysilicon lines
 - polysilicon covers over contacts
5. Compile metal layer
 - metal connections
 - metal covers over contacts
 - connections to vdd and vss
 - vdd and vss busses
6. Compile contact windows

The resulting set of polygons is stored into a file which has the same name as the input file with the suffix "cif".

6 ILLUSTRATIVE EXAMPLES

This chapter presents four examples of cells that were successfully compiled using RAMBO.

6.1 Complete Compilation Process Of A Five Gate Cell

Figure 6.1 illustrate the complete compilation process of a simple five-gate cell. Table 6.1 presents a summary of the figure.

Figure	Description
6.1-1	Logic Schematic
6.1-2	Logic Netlist
6.1-3	List of Clusters
6.1-4	List of Pieces
6.1-5	List of metal Nets
6.1-6	List of Slices
6.1-7	Cover array
6.1-8	List of Connections
6.1-9	Cell Layout

Table 6.1 Summary of Example 6.1

Figure 6.1-1 Logic Schematic

```
example1 #s(leaf-cell
           :inputs (in1 in2 in3 in4 in5 in6 in7)
           :outputs (out1)
           :size ( ) )
x-1 #s(gate
      :type inv
      :output sig-1
      :inputs (in4))
x-2 #s(gate
      :type and
      :output sig-2
      :inputs (in1 in2))
x-3 #s(gate
      :type and
      :output sig-3
      :inputs (in3 sig-1))
x-4 #s(gate
      :type and
      :output sig-4
      :inputs (in5 in6))
x-5 #s(gate
      :type nor
      :output out1
      :inputs (sig-2 sig-3 sig-4 in7))
end
```

Figure 6.1-2 Logic Netlist

```

|c-1| #S(CLUSTER
      :TYPE INV
      :GATES (X-1)
      :GATE-SIGNALS ((IN4))
      :PDIFF-SIGNALS ((VDD SIG-1))
      :NDIFF-SIGNALS ((VSS SIG-1))
|c-0| #S(CLUSTER
      :TYPE AND-INTO-NOR
      :GATES (|x-0| X-2 X-3 X-4 X-5)
      :GATE-SIGNALS ((IN3 SIG-1 IN7 IN5 IN6) (IN1 IN2))
      :PDIFF-SIGNALS ((|sig-1| VDD |sig-1| |sig-3|
                        |sig-4| |sig-3|)
                        (OUT1 |sig-4| OUT1))
      :NDIFF-SIGNALS ((VSS |sig-2| OUT1 VSS |sig-5| OUT1)
                        (VSS |sig-6| OUT1))
END

```

Figure 6.1-3 List of clusters

```

|p-13| #S(PIECE
      :TYPE POLY
      :GATE-SIGNALS (OUT1)
      :PDIFF-SIGNALS NIL
      :NDIFF-SIGNALS NIL
      :SLICES NIL)
|p-12| #S(PIECE
      :TYPE ACTIVE
      :GATE-SIGNALS (IN1 IN2)
      :PDIFF-SIGNALS (OUT1 |sig-4| OUT1)
      :NDIFF-SIGNALS (VSS |sig-6| OUT1)
      :SLICES NIL)
|p-11| #S(PIECE
      :TYPE ACTIVE
      :GATE-SIGNALS (IN3 SIG-1 IN7 IN5 IN6)
      :PDIFF-SIGNALS ((|sig-1| VDD |sig-1| |sig-3|
                        |sig-4| |sig-3|)
                        :NDIFF-SIGNALS (VSS |sig-2| OUT1 VSS |sig-5| OUT1)
                        :SLICES NIL)
|p-10| #S(PIECE
      :TYPE ACTIVE
      :GATE-SIGNALS (IN4)
      :PDIFF-SIGNALS (VDD SIG-1)
      :NDIFF-SIGNALS (VSS SIG-1)
      :SLICES NIL)
END

```

Figure 6.1-4 List of Pieces

```

|n-10| #S(NET
      :TYPE CROSS
      :SIGNAL-NAME OUT1
      :CROSSING-SLICE NIL
      :POLY-SLICES (|s-32|)
      :PDIFF-SLICES (|s-31| |s-27|)
      :GATE-SLICES NIL
      :NDIFF-SLICES (|s-31| |s-26| |s-20|)
|n-9| #S(NET
      :TYPE PDIFF
      :SIGNAL-NAME |sig-4|
      :CROSSING-SLICE NIL
      :POLY-SLICES NIL
      :PDIFF-SLICES (|s-29| |s-24|)
      :GATE-SLICES NIL
      :NDIFF-SLICES NIL)
|n-8| #S(NET
      :TYPE PDIFF
      :SIGNAL-NAME |sig-3|
      :CROSSING-SLICE NIL
      :POLY-SLICES NIL
      :PDIFF-SLICES (|s-26| |s-22|)
      :GATE-SLICES NIL
      :NDIFF-SLICES NIL)
|n-7| #S(NET
      :TYPE PDIFF
      :SIGNAL-NAME |sig-1|
      :CROSSING-SLICE NIL
      :POLY-SLICES NIL
      :PDIFF-SLICES (|s-20| |s-16|)
      :GATE-SLICES NIL
      :NDIFF-SLICES NIL)
|n-6| #S(NET
      :TYPE GATE
      :SIGNAL-NAME SIG-1
      :CROSSING-SLICE NIL
      :POLY-SLICES NIL
      :PDIFF-SLICES (|s-15|)
      :GATE-SLICES (|s-19|)
      :NDIFF-SLICES (|s-15|)

```

END

Figure 6.1-5 List of Metal Nets

```

|s-13| #S(SLICE
      :TYPE ENDDIFF
      :PDIFF-SIGNAL VDD
      :GATE-SIGNAL NIL
      :NDIFF-SIGNAL VSS
      :POSITION 0)
|s-14| #S(SLICE
      :TYPE GATE
      :PDIFF-SIGNAL NIL
      :GATE-SIGNAL IN4
      :NDIFF-SIGNAL NIL
      :POSITION 1)
|s-15| #S(SLICE
      :TYPE ENDDIFF
      :PDIFF-SIGNAL SIG-1
      :GATE-SIGNAL NIL
      :NDIFF-SIGNAL SIG-1
      :POSITION 2)
|s-16| #S(SLICE
      :TYPE ENDDIFF
      :PDIFF-SIGNAL |sig-1|
      :GATE-SIGNAL NIL
      :NDIFF-SIGNAL VSS
      :POSITION 3)
|s-17| #S(SLICE
      :TYPE GATE
      :PDIFF-SIGNAL NIL
      :GATE-SIGNAL IN3
      :NDIFF-SIGNAL NIL
      :POSITION 4)
      .
      .
      .
|s-31| #S(SLICE
      :TYPE ENDDIFF
      :PDIFF-SIGNAL OUT1
      :GATE-SIGNAL NIL
      :NDIFF-SIGNAL OUT1
      :POSITION 18)
|s-32| #S(SLICE
      :TYPE POLY
      :PDIFF-SIGNAL NIL
      :GATE-SIGNAL OUT1
      :NDIFF-SIGNAL NIL
      :POSITION 19)

```

END

Figure 6.1-6 List of Slices

```

" 1 2 0 0 0 0 0 0 0 2"
" 1 0 0 0 0 0 0 0 0 0"
" 1 0 0 2 1 2 0 0 0 0"
" 1 2 0 0 1 0 0 0 2 0"
" 1 0 0 0 1 0 0 0 1 0"
" 1 0 0 0 1 0 0 0 1 2"
" 1 0 0 0 2 0 0 0 1 0"
" 1 0 2 0 0 0 0 0 2 0"
" 1 0 1 0 0 0 0 0 0 0"
" 1 2 1 0 0 0 0 0 2 0"
" 1 0 1 0 0 0 0 0 1 0"
" 1 0 1 0 0 0 0 2 1 0"
" 1 0 1 0 0 0 0 1 1 0"
" 1 0 2 0 0 0 0 1 2 0"
" 1 2 1 0 0 0 0 1 2 0"
" 1 0 1 0 0 0 0 1 1 0"
" 1 0 1 0 0 0 0 2 1 0"
" 1 0 1 0 0 0 0 1 0"
" 1 0 2 2 1 2 0 0 2 0"
" 1 0 0 0 2 0 0 0 0 0"

```

2 = Metal and Contact
1 = Metal only

Figure 6.1-7 Cover Array

```

|conn-54| #S(CONN :START (2 4) :END (6 4))
|conn-53| #S(CONN :START (7 2) :END (18 2))
|conn-52| #S(CONN :START (11 7) :END (16 7))
|conn-51| #S(CONN :START (9 8) :END (13 8))
|conn-50| #S(CONN :START (3 8) :END (7 8))
|conn-49| #S(CONN :START (14 8) :END (18 8))
|conn-48| #S(CONN :START (18 4) :END (19 4))
|conn-47| #S(CONN :START (18 3) :END (18 5))
|conn-46| #S(CONN :START (2 3) :END (2 5))
END

```

Figure 6.1-8 List of Connections

Figure 6.1-9 Cell Layout

6.2 Effect Of Primary Cell Outputs On Layout Compilation

This section presents an example which is aimed at illustrating the effect of including additional signals as primary cell outputs. The example cell is essentially the same as that of section 6.1 with the exception that "sig-2" has now been included in the list of primary cell outputs. This addition has caused the creation of an additional cluster of type "nand-into-inv" and has had a significant impact on the final cell layout. Table 6.2 summarizes the figures associated with this example.

Figure	Description
6.2-1	Logic Schematic
6.2-2	Logic Netlist
6.2-3	List of Clusters
6.2-4	Cell Layout

Table 6.2 Summary of Example 6.2

Figure 6.2-1 Logic Schematic

```

example2 #s(leaf-cell
           :inputs (in1 in2 in3 in4 in5 in6 in7)
           :outputs (out1 sig-2)
           :size () )
x-1 #s(gate
       :type inv
       :output sig-1
       :inputs (in4))
x-2 #s(gate
       :type and
       :output sig-2
       :inputs (in1 in2))
x-3 #s(gate
       :type and
       :output sig-3
       :inputs (in3 sig-1))
x-4 #s(gate
       :type and
       :output sig-4
       :inputs (in5 in6))
x-5 #s(gate
       :type nor
       :output out1
       :inputs (sig-2 sig-3 sig-4 in7))
end

```

Figure 6.2-2 Circuit Netlist

```

|c-2| #S(CLUSTER
      :TYPE INV
      :GATES (X-1)
      :GATE-SIGNALS ((IN4))
      :PDIFF-SIGNALS ((VDD SIG-1))
      :NDIFF-SIGNALS ((VSS SIG-1)))
|c-1| #S(CLUSTER
      :TYPE NAND-INTO-INV
      :GATES (X-2)
      :GATE-SIGNALS ((IN1 IN2) (|sig-2|))
      :PDIFF-SIGNALS ((|sig-2| VDD |sig-2|) (VDD SIG-2))
      :NDIFF-SIGNALS ((VSS |sig-3| |sig-2|) (VSS-SIG-2)))
|c-0| #S(CLUSTER
      :TYPE AND-INTO-NOR
      :GATES (|x-1| |x-0| X-3 X-4 X-5)
      :GATE-SIGNALS ((IN3 SIG-1 SIG-2 IN7 IN5 IN6) NIL)
      :PDIFF-SIGNALS ((|sig-4| VDD |sig-4| |sig-6|
                        |sig-7| OUT1 |sig-7|)
                        NIL)
      :NDIFF-SIGNALS ((VSS |sig-5| OUT1 VSS OUT1 |sig-8| VSS)
                        NIL))
END

```

Figure 6.2-3 List of Clusters

IN4 IN1 IN2 IN3 SIG-1 SIG-2 IN7 IN5 IN6 OUT1

Figure 6.2-4 Cell Layout

6.3 Compilation With Transmission Gates

The purpose of this example is to illustrate how RAMBO handles the specific layout problems associated with transmission gates. Figure 6.3-1 presents the logic schematic of a simple latch and Figure 6.3-2 the compiled layout. Two things are interesting to note:

- o The additional poly-line used to connect the "IN" signal directly to a diffusion area.
- o The fact that the output stage of inverter "x-6" is used to complete the routing associated with the "tg-mux" cluster. This connection would otherwise be blocked.

Figure 6.3-1 Logic Schematic

Figure 6.3-2 Cell Layout

6.4 Compilation Of Complex Cells

This section presents an example which is believed to be at the limit of feasibility for the current implementation of RAMBO. Figure 6.4-1 presents the logic schematic of a Master-Slave JK flip-flop with scan inputs. The cell is comprised of 36 transistors and is further complicated by the presence of four transmission gates as well as a feedback net.

In order to complete the routing successfully, a minor manual modification had to be performed to the outcome of the placement phase, which consisted of the interchange of two clusters of type "inv". The results are shown in Figure 6.4-2.

Figure 6.4-1 Logic Schematic for JK flip-flop

Figure 6.4-2 Cell Layout for JK flip-flop

7 CONCLUSIONS

7.1 Summary

This thesis has addressed the issue of Standard Cell Synthesis in the context of complete chip design systems. Motivation for the research was based on recognition of several advantages such as design portability, correctness by construction, documentation and maintenance, ease of characterization and flexibility.

A survey of related literature was presented, that outlined some of the most important synthesis methods reported to date. It was concluded from the survey that the current state of affairs in the field of Standard Cell Design Automation exhibited some serious shortcomings such as complex input description and impractical layout styles. The issue of translation of logic schematics into transistor interconnects as well as cell characterization were found to be absent from current synthesis schemes.

In order to further solidify our bases before experimentation, much thought was given to providing design specifications for a "turnkey" standard cell design system. Such a system was outlined in terms of its key attributes and a high level block diagram.

A prototype cell synthesizer, RAMBO, was implemented in order to demonstrate the feasibility of a software tool for cell design. The prototype is capable of generating layout geometries from a cell description at the logic gate level. Its operation is characterized by the creation of

"clusters" for the purpose of mapping logic descriptions onto transistor interconnects as well as for initial placement. Functionality of the system was demonstrated by the construction of a JK flip-flop with scan inputs, comprised of 38 transistors.

RAMBQ was implemented in LISP as a series collection of single-function modules to facilitate future enhancements. It is the first milestone in the development of a "turnkey" standard cell synthesis system.

7.2 Suggestions For Future Work

The formation of "clusters" is currently limited to two levels of logic, although it is theoretically possible to consider higher orders. A direct system enhancement would be to accommodate a third level of logic which is a rather common occurrence. Alternatively, a new method could be developed to handle an arbitrary number of levels.

The placement and routing algorithms are also subject to improvement. Placement optimization could be achieved by interchanging and mirroring layout pieces in order to minimize the density of the routing channels within the p-region and n-region. Allowing interaction between placement and routing would also increase the routing completion rate.

Optimization of the final layout could be implemented as a post-processing step. Operations such as contact movement, routing backtracking, feature bending and

horizontal compaction should be envisaged. Provisions should also be made for split contacts, for the system to become commercially viable.

APPENDICES

APPENDIX A RAMBO DATA STRUCTURES

```

*****
FILE NAME:    datastruct.lsp
MODULE NAME:  Datastructure
CONTENTS:     STRUCTURE          FUNCTION
              leaf-cell          basic data structure for cell
              signal             electrical signal
              gate                logic gate within a cell
              cluster            logically related gates
              piece               common diffusion areas of the cell
              slice              vertical slices of clusters
              net                 signal requiring metal interconnect
              conn                metal connection between two points
              relation            placement relation
*****

(defstruct leaf-cell
  inputs
  outputs
  size )
(defstruct signal
  source
  sinks
  net )
(defstruct gate
  type
  output
  inputs )
(defstruct cluster
  type
  gates
  output-signal
  gate-signals
  pdiff-signals
  ndiff-signals )
(defstruct piece
  type
  gate-signals
  pdiff-signals
  ndiff-signals
  slices )
(defstruct slice
  type
  pdiff-signal
  gate-signal
  ndiff-signal
  position )
(defstruct net
  type
  signal-name
  crossing-slice
  poly-slices
  pdiff-slices
  gate-slices
  ndiff-slices )
(defstruct conn
  start
  end )
(defstruct relation
  left-cluster
  right-cluster )

```

APPENDIX B "FORMCLUSTERS" LISP LISTING

```

*****
FILENAME:      formclusters.lsp
MODULENAME:    Formclusters
CONTENTS:      FUNCTIONS      DESCRIPTION
               formclusters    main program for formclusters module
               findnandclusters find or-into-nand clusters
               findnorclusters  find and-into-nor clusters
               findtgclusters    finds tg-mux clusters
               clustergate       assigns other gates to clusters
*****
*****
*****
FUNCTION NAME: formclusters
INPUTS:        cellname.....cell name
DESCRIPTION:    Invokes 'readcell' to read cell description from file.
                Forms clusters by merely listing out the gates contained
                "in each of the clusters found in 'gatelist'".
                Invokes 'writeclr' to write cluster list to a file.

(defun formclusters (cellname)
  (readcell cellname)
  (setq clusters nil)
  (mapc #'findnandclusters gatelist)
  (mapc #'findnorclusters gatelist)
  (mapc #'findtgclusters gatelist)
  (mapc #'clustergate gatelist)
  (writeclr cellname))

```



```

*****
FUNCTION NAME:  findnandclusters

INPUTS:        seed.....name of gate from which to start searching
                for 'nand' or 'or-into-nand' clusters.

DESCRIPTION:    Creates a new cluster if the seed gate is of type 'nand'.
                If the seed gate has inputs that are not cell outputs and
                that are outputs of 'or' gates a 'or-into-nand' cluster
                is created. Otherwise, a cluster of type 'nand' is created.
                Function findorinputs is used to search the list of inputs
                to the seed gate for 'or' gates.
                Function add-dummy-inputs is used to add buffers for
                inputs to the seed gate not coming from a gate within the
                same cluster, for the case of or-into-nand clusters.

(defun findnandclusters (seed)
  (cond ((equal (gate-type (eval seed)) 'nand) \
    (setq gatelist (remove seed gatelist))
    (setq newcluster (gentemp "c-"))
    (set newcluster (make-cluster :type 'nand))
    (setq clusters (cons newcluster clusters))
    (setf (cluster-output-signal (eval newcluster))
      (gate-output (eval seed)))
    (setf (cluster-gates (eval newcluster))
      (cons seed (cluster-gates (eval newcluster))))
    (findorinputs (gate-inputs (eval seed)))
    (cond ((equal (cluster-type (eval newcluster)) 'or-into-nand)
      (setq sig-list (mapcar #'(lambda (x)
        (gate-output (eval x)))
        (cluster-gates (eval newcluster))))
      (setq seed-gate seed)
      (mapc #'add-dummy-inputs (gate-inputs (eval seed)))))))

(defun findorinputs (list)
  (cond ((cdr list) (findorinputs (cdr list))))
  (cond ((equal (gate-type (eval (car (signal-source (eval (car list))))) 'or)
    (cond ((not (member (car list) (leaf-cell-outputs (eval cell))))
      (setq gatelist
        (remove (car (signal-source (eval (car list))))
          gatelist))
      (setf (cluster-type (eval newcluster)) 'or-into-nand)
      (setf (cluster-gates (eval newcluster))
        (cons (car (signal-source (eval (car list))))
          (cluster-gates (eval newcluster)))))))))

(defun add-dummy-inputs (in-sig)
  (cond ((member in-sig sig-list) nil)
    (t (setq newsignal (gentemp "sig-"))
      (setf (gate-inputs (eval seed-gate))
        (cons newsignal (remove in-sig
          (gate-inputs (eval seed-gate)))))
      (setq newgate (gentemp "x-"))
      (set newgate (make-gate :type 'buff :output newsignal))
      (setf (gate-inputs (eval newgate)) (list in-sig))
      (setf (cluster-gates (eval newcluster))
        (cons newgate (cluster-gates (eval newcluster))))
      (setq signallist (cons newsignal signallist)))))

```

%%%

FUNCTION NAME: findnorclusters

INPUTS: seed.....name of gate from which to start searching
for 'nor' of 'and-into-nor' clusters.

DESCRIPTION: Creates a new cluster if the seed gate is of type 'nor'.
If the seed gate has inputs that are not cell outputs and
that are outputs of 'and' gates a 'and-into-nor' cluster
is created. Otherwise, a cluster of type 'nor' is created.
Function findandinputs is used to search the list of inputs
to the seed gate for 'and' gates.
Function add-dummy-inputs is used to add buffers for
inputs to the seed gate not coming from a gate within the
same cluster, for the case of and-into-nor clusters.

```
(defun findnorclusters (seed)
  (cond ((equal (gate-type (eval seed)) 'nor)
    (setq gatelist (remove seed gatelist))
    (setq newcluster (gentemp 'c-'))
    (set newcluster (make-cluster :type 'nor))
    (setq clusters (cons newcluster clusters))
    (setf (cluster-output-signal (eval newcluster))
      (gate-output (eval seed)))
    (setf (cluster-gates (eval newcluster))
      (cons seed (cluster-gates (eval newcluster))))
    (findandinputs (gate-inputs (eval seed)))
    (cond ((equal (cluster-type (eval newcluster)) 'and-into-nor)
      (setq sig-list (mapcar #'(lambda (x)
        (gate-output (eval x)))
        (cluster-gates (eval newcluster))))
      (setq seed-gate seed)
      (mapc #'add-dummy-inputs (gate-inputs (eval seed))))))
    (t))))
```

```
(defun findandinputs (list)
  (cond ((cdr list) (findandinputs (cdr list))))
  (cond ((equal (gate-type (eval (car (signal-source (eval (car list))))))
    'and)
    (cond ((not (member (car list) (leaf-cell-outputs (eval cell))))
      (setq gatelist
        (remove (car (signal-source (eval (car list))))
          gatelist))
      (setf (cluster-type (eval newcluster)) 'and-into-nor)
      (setf (cluster-gates (eval newcluster))
        (cons (car (signal-source (eval (car list))))
          (cluster-gates (eval newcluster))))))
    (t))))
```

2

OF/DE

2

MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS
STANDARD REFERENCE MATERIAL 1010a
(ANSI and ISO TEST CHART No 2)

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

FUNCTION NAME: findtgclusters

INPUTS: seed.....seed gate from which to search for cluster.

DESCRIPTION: A new cluster of type 'tg-mux' is created if the seed gate is of type 'tg'. The matching transmission gate is found by searching for another source for the seed gate output. This operation restricts the use of transmission gates outside of the 2-to-1 tg-mux context.

```

(defun findtgclusters (seed)
  (cond ((and (equal (gate-type (eval seed)) 'tg)
              (member seed gatelist))
        (setq gatelist (remove seed gatelist))
        (setq newcluster (gentemp 'c-))
        (set newcluster (make-cluster :type 'tg-mux))
        (setq clusters (cons newcluster clusters))
        (setf (cluster-output-signal (eval newcluster))
              (gate-output (eval seed)))
        (cond ((equal seed
                      (car (signal-source (eval (gate-output (eval seed))))))
              (setq othertg
                    (cadr (signal-source (eval (gate-output (eval seed))))))
              (t (setq othertg
                      (car (signal-source
                            (eval (gate-output (eval seed)))))))
              (setf (cluster-gates (eval newcluster)) (list seed othertg))
              (setq gatelist (remove othertg gatelist))))))

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

FUNCTION NAME: clustergate

INPUTS: victim.....gate with which to form single gate cluster.

DESCRIPTION: A single gate cluster is created with the appropriate cluster type.

```

(defun clustergate (victim)
  (setf newcluster (gentemp 'c-))
  (set newcluster (make-cluster))
  (setq clusters (cons newcluster clusters))
  (setf (cluster-output-signal (eval newcluster))
        (gate-output (eval victim)))
  (setf (cluster-gates (eval newcluster))
        (cons victim (cluster-gates (eval newcluster))))
  (cond ((equal (gate-type (eval victim)) 'inv)
        (setf (cluster-type (eval newcluster)) 'inv)))
  (cond ((equal (gate-type (eval victim)) 'and)
        (setf (cluster-type (eval newcluster)) 'nand-into-inv)))
  (cond ((equal (gate-type (eval victim)) 'or)
        (setf (cluster-type (eval newcluster)) 'nor-into-inv)))

```

APPENDIX C "EXPANDCLUSTERS" LISP LISTING

```

*****
FILENAME:      expandclusters.lsp
MODULE NAME:   expandclusters
CONTENTS:      FUNCTIONS      DESCRIPTION
                expandclusters  calling function
                expandcluster   main routine
                expand2level    expand 'and-into-nor' 'or-into-nand'
                expand1level    expand 'and' 'or' 'nand' 'nor'
                expand0level    expand 'inv'
                expandtgmux     expand 'tg-mux'
*****
*****
FUNCTION NAME: expandclusters
INPUTS:        cellname
DESCRIPTION:    This function is the calling function for the
                expand clusters module.
(defun expandclusters (cellname)
  (mapc #'expandcluster clusters))

```

```

*****
FUNCTION NAME:  expandcluster
INPUTS:        clustername
DESCRIPTION:    This function is the main program of the 'expandclusters'
                module.  It calls the appropriate function depending on
                cluster type of clustername.

(defun expandcluster (clustername)
  (setq vdd (make-signal))
  (setq vss (make-signal))
  (setq polylist nil)
  (setq pdiffstrip nil)
  (setq ndiffstrip nil)
  (cond ((equal (cluster-type (eval clustername)) 'and-into-nor)
        (expand2level clustername))
        ((equal (cluster-type (eval clustername)) 'or-into-nand)
        (expand2level clustername))
        ((equal (cluster-type (eval clustername)) 'nand)
        (expand1level clustername))
        ((equal (cluster-type (eval clustername)) 'nor)
        (expand1level clustername))
        ((equal (cluster-type (eval clustername)) 'nand-into-inv)
        (expand1level clustername))
        ((equal (cluster-type (eval clustername)) 'nor-into-inv)
        (expand1level clustername))
        ((equal (cluster-type (eval clustername)) 'inv)
        (expand0level clustername))
        ((equal (cluster-type (eval clustername)) 'tg-mux)
        (expandtgmux clustername)))
  (setf (cluster-gate-signals (eval clustername)) polylist)
  (setf (cluster-pdiff-signals (eval clustername)) pdiffstrip)
  (setf (cluster-ndiff-signals (eval clustername)) ndiffstrip))

*****
FUNCTION NAME:  gensig
INPUTS:        none
DESCRIPTION:    This function is used to create signal names during
                cluster expansion.  It returns a signal name which is
                an instance of the 'signal' data structure.

(defun gensig ( )
  (setq sig (gentemp 'sig-))
  (set sig (make-signal))
  (eval 'sig))

```

FUNCTION NAME: expand2level

INPUTS: clustername

DESCRIPTION: This function expands clusters of the type 'and-into-nor' and 'or-into-nand' or so called two-level logic clusters. 'out-stage' is the gate of the second level of logic and 'stages' are the gates at the first level of logic. Even and odd stages refer to the number of inputs of their respective gate. Sets refer to gates that may be implemented by two un-broken lines of diffusion. Implementation is based on the fact that there can be at most two even stages per set. See thesis for details.

```
(defun expand2level (clustername)
  (setq evenstages nil)
  (setq oddstages nil)
  (mapc #'weightstage (remove-if #'(lambda (x)
                                     (member (gate-type (eval x)) '(nor nand)))
                                (cluster-gates (eval clustername)))))
  (setq out-stage (car (remove-if-not #'(lambda (x)
                                          (member (gate-type (eval x))
                                                    '(nor nand)))
                                      (cluster-gates (eval clustername))))))
  (setq out-sig-name (gate-output (eval out-stage)))
  (cond ((equal (length evenstages) '0)
        (weightsort oddstages)
        (setq stages (list sortedlist)))
        ((equal (length evenstages) '1)
        (weightsort oddstages)
        (cond ((> (get (car evenstages) 'weight) maxweight)
              (setq stages (list (cons (car evenstages) sortedlist))))
              (t (setq stages (list (append sortedlist evenstages)))))
        ((equal (length evenstages) '2)
        (weightsort oddstages)
        (setq stages sortedlist)
        (weightsort evenstages)
        (setq stages (list (append (cons (car sortedlist) stages)
                                   (cdr sortedlist)))))
        ((> (length evenstages) '2)
        (weightsort oddstages)
        (setq stages sortedlist)
        (weightsort evenstages)
        (setq stages (list
                        (append (cons (car sortedlist) stages)
                                (list (cadr sortedlist)))
                        (caddr sortedlist)))))
```

```

;define diffusion strips for each stage
(cond ((equal (cluster-type (eval clustername)) 'and-into-nor)
      (setq raila 'vdd) (setq railb 'vss))
      (t
       (setq raila 'vss) (setq railb 'vdd)))
(setq stripa nil)
(setq stripb nil)
(setq polya nil)
(setq stripc nil)
(setq stripd nil)
(setq polyb nil)
;first stage of first set
(setq firstset (car stages))
(setq polya (gate-inputs (eval (car firstset))))
(cond ((or (cdr firstset) (cdr stages))
      (setq newsignal (gensig)))
      (t (setq newsignal out-sig-name)))
(cond ((equal (length polya) '1)
      (setq stripa (list raila newsignal))
      (setq stripb (list railb out-sig-name)))
      ((equal (length polya) '2)
      (setq stripa (list newsignal raila newsignal))
      (setq stripb (list railb (gensig) out-sig-name)))
      ((equal (length polya) '3)
      (setq stripa (list raila newsignal raila newsignal))
      (setq stripb (list railb (gensig) (gensig)
                           out-sig-name)))
      ((equal (length polya) '4)
      (setq stripa (list newsignal raila newsignal raila newsignal))
      (setq stripb (list railb (gensig) (gensig)
                           (gensig) out-sig-name))))
;second stage of first set
(cond ((cdr firstset) (setq firstset (cdr firstset))
      (setq cursignal newsignal)
      (cond ((or (cdr firstset) (cdr stages))
            (setq newsignal (gensig)))
            (t (setq newsignal out-sig-name)))
      (setq newpoly (gate-inputs (eval (car firstset))))
      (cond ((equal (length newpoly) '1)
            (setq stripa (append stripa (list newsignal)))
            (setq stripb (append stripb (list railb))))
            ((equal (length newpoly) '2)
            (setq stripa (append stripa (list newsignal cursignal)))
            (setq stripb (append stripb (list (gensig) railb))))
            ((equal (length newpoly) '3)
            (setq stripa (append stripa
                                   (list newsignal cursignal newsignal)))
            (setq stripb (append stripb
                                   (list (gensig)
                                       (gensig) railb))))
            ((equal (length newpoly) '4)
            (setq stripa
              (append stripa (list newsignal cursignal newsignal
                                   cursignal)))
            (setq stripb (append stripb (list (gensig) (gensig)
                                               (gensig) railb))))))
      (setq polya (append polya newpoly))

```



```

;third stage of first set
  (cond ((cdr firstset) (setq firstset (cdr firstset))
        (setq cursignal newsignal)
        (cond ((or (cdr firstset) (cdr stages))
              (setq newsignal (gensig)))
              (t (setq newsignal out-sig-name)))
        (setq newpoly (gate-inputs (eval (car firstset))))
        (cond ((equal (length newpoly) '1)
              (setq stripa (append stripa (list newsignal)))
              (setq stripb (append stripb (list out-sig-name))))
              ((equal (length newpoly) '2)
              (setq stripa (append stripa (list newsignal cursignal)))
              (setq stripb (append stripb (list (gensig)
                                                out-sig-name))))
              ((equal (length newpoly) '3)
              (setq stripa
                (append stripa (list newsignal cursignal
                                      newsignal)))
              (setq stripb
                (append stripb (list (gensig)
                                      (gensig)
                                      out-sig-name))))
              ((equal (length newpoly) '4)
              (setq stripa
                (append stripa (list newsignal cursignal newsignal
                                      cursignal)))
              (setq stripb
                (append stripb (list (gensig)
                                      (gensig)
                                      (gensig)
                                      out-sig-name))))))
  (setq polya (append polya newpoly))

```

```

;fourth stage of first set
(cond ((cdr firstset) (setq firstset (cdr firstset))
      (setq cursignal newsignal)
      (cond ((cdr stages) (setq newsignal (gensig))
            (t (setq newsignal out-sig-name)))
      (setq newpoly (gate-inputs (eval (car firstset))))
      (cond ((equal (length newpoly) '1)
            (setq stripa (append stripa (list newsignal)))
            (setq stripb (append stripb (list railb))))
            ((equal (length newpoly) '2)
            (setq stripa
              (append stripa (list newsignal cursignal)))
            (setq stripb
              (append stripb (list (gensig)
                                   railb))))
            ((equal (length newpoly) '3)
            (setq stripa
              (append stripa (list newsignal cursignal
                                   newsignal)))
            (setq stripb
              (append stripb (list (gensig)
                                   (gensig)
                                   railb))))
            ((equal (length newpoly) '4)
            (setq stripa
              (append stripa (list newsignal cursignal
                                   newsignal
                                   cursignal)))
            (setq stripb
              (append stripb (list (gensig)
                                   (gensig)
                                   (gensig)
                                   railb))))))
      (setq polya (append polya newpoly))))))

```

```

;first stage of second set
(cond ((cdr stages) (setq secondset (cdr stages))
      (setq polyb (gate-inputs (eval (car secondset))))
      (setq cursignal newsignal)
      (cond ((cdr secondset) (setq newsignal (gensig))
            &t (setq newsignal out-sig-name)))
      (cond ((equal (length polyb) '1)
            (setq stripc (list cursignal newsignal))
            (setq stripd (list railb out-sig-name)))
            ((equal (length polyb) '2)
            (setq stripc (list newsignal cursignal newsignal))
            (setq stripd (list railb (gensig) out-sig-name)))
            ((equal (length polyb) '3)
            (setq stripc (list cursignal newsignal cursignal newsignal))
            (setq stripd (list railb (gensig) (gensig)
                               out-sig-name)))
            ((equal (length polyb) '4)
            (setq stripc (list newsignal cursignal newsignal
                               cursignal newsignal))
            (setq stripd (list railb (gensig) (gensig)
                               (gensig) out-sig-name))))))

;second stage of second set
(cond ((cdr secondset) (setq secondset (cdr secondset))
      (setq cursignal newsignal)
      (setq newsignal out-sig-name)
      (setq newpoly (gate-inputs (eval (car secondset))))
      (cond ((equal (length newpoly) '1)
            (setq stripc (append stripc (list newsignal)))
            (setq stripd (append stripd (list railb))))
            ((equal (length newpoly) '2)
            (setq stripc (append stripc (list newsignal cursignal)))
            (setq stripd (append stripd
                               (list (gensig) railb))))
            ((equal (length newpoly) '3)
            (setq stripc
                  (append stripc
                           (list newsignal cursignal newsignal)))
            (setq stripd
                  (append stripd
                           (list (gensig) (gensig) railb))))
            ((equal (length newpoly) '4)
            (setq stripc
                  (append stripc
                           (list newsignal cursignal newsignal
                               cursignal)))
            (setq stripd
                  (append stripd
                           (list (gensig)
                               (gensig)
                               (gensig) railb))))))
      (setq polyb (append polyb newpoly))))))

```

```

;wrap-up
(cond ((equal (cluster-type (eval clustername)) 'and-into-nor)
      (setq pdiffstrip (list stripa stripc))
      (setq ndiffstrip (list stripb stripd)))
      (t
       (setq pdiffstrip (list stripb stripd))
       (setq ndiffstrip (list stripa stripc))))
(setq polylist (list polya polyb)))

(defun weighstage (gatename)
  (setf (get gatename 'weight)
        (length (remove-if #'(lambda (x)
                                (member x (leaf-cell-inputs (eval cell))))
                            (gate-inputs (eval gatename))))))
  (cond ((evenp (length (gate-inputs (eval gatename))))
        (setf (get gatename 'parity) 'even)
        (setq evenstages (cons gatename evenstages)))
        (t (setf (get gatename 'parity) 'odd)
         (setq oddstages (cons gatename oddstages)))))

(defun weightsort (startlist)
  (setq sortedlist startlist)
  (setq sortedlist (sort sortedlist #'weight-p))
  (setq maxweight (get (car sortedlist) 'weight))
  (defun weight-p (x y)
    (> (get x 'weight) (get y 'weight)))

```

```

*****
FUNCTION NAME:  expandllevel
INPUTS:        clustername
DESCRIPTION:    This function expands clusters of the type 'and' 'or'
                'and-into-inv' or 'or-into-inv' ; so called single level
                logic clusters.

(defun expandllevel (clustername)
  (setq stripa nil)
  (setq stripb nil)
  (setq polya nil)
  (setq stripc nil)
  (setq stripd nil)
  (setq polyb nil)
  (setq out-sig-name
    (gate-output (eval (car (cluster-gates (eval clustername))))))
  (setq poly
    (gate-inputs (eval (car (cluster-gates (eval clustername))))))
  (cond ((member (cluster-type (eval clustername)) '(nand nand-into-inv))
    (setq raila 'vdd) (setq railb 'vss))
    (t
      (setq raila 'vss) (setq railb 'vdd)))
  ; nand or nor gate stage
  (cond ((member (cluster-type (eval clustername)) '(nand nor))
    (setq newsignal out-sig-name)
    (t (setq newsignal (gensig))))
    (cond ((equal (length poly) '2)
      (setq stripa (list newsignal raila newsignal))
      (setq stripb (list railb (gensig) newsignal)))
      ((equal (length poly) '3)
      (setq stripa (list raila newsignal raila newsignal))
      (setq stripb (list railb (gensig) (gensig)
        newsignal)))
      ((equal (length poly) '4)
      (setq stripa (list newsignal raila newsignal raila newsignal))
      (setq stripb (list railb (gensig) (gensig)
        (gensig) newsignal))))
  ; inverter stage
  (cond ((member (cluster-type (eval clustername)) '(nand nor)) nil)
    (t (setq polyb (list newsignal))
      (setq stripc (list raila out-sig-name))
      (setq stripd (list railb out-sig-name)))
  ; wrap up
  (cond ((member (cluster-type (eval clustername)) '(nand nand-into-inv))
    (setq pdiffstrip (list stripa stripc))
    (setq ndiffstrip (list stripb stripd))
    (t
      (setq pdiffstrip (list stripb stripd))
      (setq ndiffstrip (list stripa stripc)))
    (setq polylist (list polya polyb)))

```

```

*****
FUNCTION NAME:  expand0level
INPUTS:        clustername
DESCRIPTION:    At present, this function expands clusters of the type 'inv'
                only.

(defun expand0level (clustername)
  (setq polylist
    (list (gate-inputs (eval (car (cluster-gates (eval clustername)))))))
  (setq out-sig-name
    (gate-output (eval (car (cluster-gates (eval clustername))))))
  (setq pdiffstrip (list (list 'vdd out-sig-name)))
  (setq ndiffstrip (list (list 'vss out-sig-name)))
  )

*****

FUNCTION NAME:  expandtgmux
INPUTS:        clustername
DESCRIPTION:    This function expands clusters of the type 'tg-mux'

(defun expandtgmux (clustername)
  (setq out-sig-name
    (gate-output (eval (car (cluster-gates (eval clustername))))))
  (setq sig-a
    (car (gate-inputs (eval (car (cluster-gates (eval clustername)))))))
  (setq sig-b
    (car (gate-inputs (eval (cadr (cluster-gates (eval clustername)))))))
  (setq polylist
    (list (cdr (gate-inputs (eval (car (cluster-gates
                                          (eval clustername)))))))
    )
  (setq pdiffstrip (list (list sig-a out-sig-name sig-b)))
  (setq ndiffstrip (list (list sig-b out-sig-name sig-a)))
  )

```

APPENDIX D GLOSSARY OF TERMS

cluster: Group of logic gates formed to facilitate the generation of a transistor netlist as well as initial placement.

gate: Logic input description primitive.

net: Refers to a connection implemented with the metal layer.

piece: Smallest partition of the cell layout that can be manipulated by the placement algorithm.

relation: Relative placement of two clusters.

signal: Electrical signal.

slice: Smallest partition of the cell layout that can be handled by the routing algorithm.

stage: For two-level logic clusters, a stage corresponds to an input of the second level gate.

strip: Refers to a connection implemented with the metal layer.

BIBLIOGRAPHY

[BuPe]Burstein, M., and Pelavin, R., "Hierarchical Wire Routing", IEEE Transactions on CAD of IC's and Systems, October 1983, pp. 223-234.

[Cies]Ciesielski, M.J., "Two-Dimensional routing for the SILC Silicon Compiler", IEEE Transactions on CAD, CAD-4, July 1985, pp. 198-203.

[Dai]Dai, W.M., et al, "Routing Region Definition and Orderign Scheme for Building-Block Layout", IEEE Transactions on CAD, CAD-4, July 1985, pp. 189-197.

[DuKe]Dunlop, A.E., and Kernighan, B.W., "A Procedure for Placement of Standard Cell VLSI Circuits", IEEE Trans. on CAD for IC's and Systems, CAD-4, No. 1, January 1985, pp. 92-98.

[Huds]Hudson, J.A., et al, "Module Positioning Algorithms for Rectilinear Macrocell Assemblies", Proc. 21st Design Automation Conf., 1984, pp. 672-675.

[JoSi]Joobbanni, R., and Siewiorek, "WEAVER: A Knowledge-Based Routing Expert", Proc. 22nd Design Automation Conf., 1985, p.266.

[Kang]Kang, S.M., "A Design of CMOS Polycells for LSI Circuits", IEEE Trans. on Circuits and Systems, vol. CAS-28, no. 8, August 1981, pp. 838-843.

[KiMS]Kim, J.H., McDermott, J., Siewiorek, D.P., "Exploiting Domain Knowledge in IC Cell Layout", IEEE Design and Test, August 1984, pp. 52-64.

[KoWe1]Kollarritsch, P.W., Weste, N.H.E., "A Rule-Based Symbolic Layout Expert" (TOPOLOGIZER), VLSI Design, August 1984, pp. 62-66.

[KoWe2]Kollarritsch, P.W., Weste, N.H.E., "TOPOLOGIZER: An Expert System Translator of Transistor Connectivity to Symbolic Cell Layout", IEEE Journal of Solid-State Circuits, vol. SC-20, no. 3, June 1985, pp. 799-804.

[Laut]Lauther, U., "A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation", Journal of Digital Systems, Volume IV, (1980), pp. 21-34.

[LeCJ]Lee, C.M., Chawla, B.R., Just, S., "Automatic Generation and Characterization of CMOS Polycells", Proc. 18th Design Automation Conf., June 1981, pp. 220-224.

[Lee]Lee, C.Y., "An Algorithm for Path Connections and its Application", IRE Transactions on Electronic Computers, September 1961, pp. 346-365.

[MeCo]Mead, C.A., and Conway, L., "Introduction to VLSI Systems", Addison-Wesley Publishing Company, October 1980.

[NgJo]Ng, T.K., Johnsson, S.L., "Generation of Layouts from MOS Circuit Schematics: A Graph Theoretic Approach", Proc. 22nd Design Automation Conf., June 1985, pp. 39-45.

[Noda]Noda, S., et al, "Automatic Layout Algorithms for Function Blocks of CMOS Gate Arrays", Proc. 22nd Design Automation Conf., June 1985, pp. 46-52.

[NoHe]Nogatch, J.T., Hedges, T., "Automated Design of CMOS Leaf Cells", VLSI Systems Design, vol. VI, no. 11, November 1985, pp. 66-78.

[PoEl]Powel, P.A.D., Elmasry, M.I., "Cell Design Using ICEWATER", Proc. 1985 Canadian Conf. on VLSI, November 1985, pp. 188-191.

[Reed]Reed, J. et al, "A New Symbolic Channel Router: YACR2", IEEE Trans. on CAD, CAD-4, July 1985, pp. 208-219.

[RiFi]Rivest, R.L., and Fiducia, C.M., "A Greedy Channel Router", Proc. 19th Design Automation Conf., 1982, 418-242.

[Rive]Rivest, R.L., "The 'PI' (Placement and Interconnect) System", Proc. 19th Design Automation Conf., 1982, pp. 475-481.

[RoRD]Rogers, C.D., Rosenberg, J.B., Daniel, S.W., "MCNC's Vertically Integrated Symbolic Design System", Proc. 22nd Design Automation Conf., June 1985, pp. 62-68.

[RoHB]Royal, N., Hunter, J., Buchanan, I., "A Case Study in Process Independence", Proc. 22nd Design Automation Conf., June 1985, pp. 591-596.

[ToSa]Tosuntikool, N., Saxe, C.L., "Automated Design of Standard Cells", Proc. of Custom IC Conf., 1983, pp. 110-114.

[Ueda]Ueda, K., et al, "CHAMP: Chip Floor Plan for Hierarchical VLSI Layout Design", IEEE Trans. on CAD for IC's and Systems, CAD-4, No. 1, January 1985, pp. 2-22.

[UeVC]Uehara, T., vanCleemput, W.M., "Optimal Layout of CMOS Functional Arrays", IEEE Trans. on Computers, vol. c-30, no. 5, May 1981, pp. 305-312.

[West]West, N., "Virtual Grid Symbolic Layout", Proc. 18th Design Automation Conf., 1981, pp. 225-233.

[Will]Williams, J.D., "STICKS: A Graphical Compiler for High Level LSI Design", Proc. of the National Computer Conference, 1978, pp. 289-295.

[WNMD]Wolf, W., Newkirk, J., Matheus, R., Dutton, R., "DUMBO, a Schematic to Layout Compiler", in R. Bryant (ed), Proc. 3rd Caltech Conf. on VLSI, Computer Science Press, 1983, pp. 379-393.

[Yosh]Yoshimura, T., "An Efficient Channel Router", Proc. 21st Design Automation Conf., 1984, pp. 28-44.

END

08.08.87

FIN