NOVELTY IN INTERFACE AND IMPLEMENTATION: "MAGIC" ON THE MACINTOSH AND POWERPC

by

Selim Özgür Yoğurtçu

B.S. Electrical Engineering

Boğaziçi University, 1992

Submitted to the

Institute for Graduate Studies in Science and Engineering

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

Boğaziçi University

1995

ACKNOWLEDGMENTS

I would like to thank my advisors Prof. Dr. Ömer Cerid and Prof. Dr. Sina Balkır for their support and guidance in the preparation of this thesis. Also I would like to mention my appreciation to the first implementors of Magic at the University of California at Berkeley and other developers of Magic in different institutions for the clear and easy to understand source.

ABSTRACT

The complexity and capability of VLSI technology is increasing day by day while design techniques still seem to be inadequate to overcome all the challenges of VLSI. The market potential of innovative design is large but the competition and changing consumer demands, specially with the high equipment requirements of VLSI design tools still lead VLSI technology into some limitations.

The main aim of this study is to bring one of the most advanced VLSI design systems to the desktop, adding further functionality and ease of use. Namely, the project is the implementation of "Magic", an interactive system for creating VLSI circuit layouts, on PC range computers. Magic was formerly available only on UNIX workstations. The program is re-implemented on the Macintosh computers considering the user interface advantages of the MacOS and the computational advantages of the RISC technology brought to the desktop by PowerPC computers. The implementation was carried out using C++ taking advantage of the existing C sources of Magic.

ÖZET

VLSI teknolojisinin gücü ve karmaşıklığı gün geçtikçe artarken, tasarım teknikleri halen tüm güçlükleri yenebilecek kadar güçlü görünmüyor. Yenilikçi tasarımlara büyük bir gereksinim varken, bu alandaki rekabet, değişken tüketici istekleri ve özellikle tasarım araçlarının yüksek donanım gereksinimleri, VLSI teknoloisini kısıtlamakta.

Çalışmanın temel amacı en ileri VLSI tasarım sistemlerinden birini masaüstüne taşımak. Proje, VLSI devre tasarım için etkileşimli bir sistem olan "Magic" yazılımının kişisel bilgisayarlar üzerinde gerçeklenmesi şeklinde tanımlanabilir. Magic bu güne dek sadece UNIX tabanlı iş istasyonlarında bulunmaktaydı. Yazılım Macintosh işletim sisteminin kullanıcı arayüzünden ve PowerPC bilgisayarlarıyla masaüstüne taşınan RISC teknolojisinin sayısal işlem gücünden yararlanılabilmesi için Macintosh ortamı seçilerek yeniden gerçeklendi. Gerçeklenme, iş istasyonlarında çalışmakta olan uygulamanın C program kodları da kullanılarak C++ dilinde yapıldı.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES AND LISTINGS

1. INTRODUCTION

Several events trace the evolution of what we call "The VLSI technology" since the early 1930's, when Lilienfeld and Heil set up a base for the theoretical developments of a field effect transistor (FET). But the practical utilization of those ideas were as late as 1947-1948 because of technological problems. On those years the first bipolar junction transistor (BJT) was introduced at the Bell Laboratories. This was the practical beginning of the micro-electronics industry and was followed by a fast development in the VLSI technology and applications of the VLSI technology on many areas.

Further development of the technology forced the use of computer aided design tools as the new designs tended to increase in complexity. The market forced the designer to faster designs at lower costs and simulation of the circuits before manufacturing has become a must. Today computer aided design tools are used in different phases of VLSI design processes and are the main driving force of the VLSI design technology.

Magic is a widely known and used tool for the design and simulation of VLSI layouts because it simplifies the design process using a simpler design methodology and graphical user interface. Moreover it includes a fast built-in error checker that illustrates the errors on the layout as the designer gets on with the design process. Thus once the design is complete the design is known to be free of any errors related to leakage and manufacturing.

Magic[1] is a freeware software with its full source code and documentation and has been in development since its first release in 1986 from the University of California at Berkeley. This study bases its development upon Magic version 6 which runs on a number of UNIX workstations. Magic version 6 supports X11 as the graphical user interface and thus provides a rather simple interface to the VLSI designer but lacks many features that could be easily implemented with a new interface design.

The purpose of this study is to come up with a new implementation of the Magic VLSI layout design system which runs on personal computers. The user interface is redesigned to take advantage of the new graphical user interface environment and to supply

---

a simple but robust VLSI layout design tool. Through use of palettes and a simpler interface for the design phase, the development aimed to carry the VLSI design tool to the desktop computers as a simple drawing tool. On the other hand, existing features of Magic were not sacrificed for, such as, on the fly error checking. Advanced use of Magic through the command line interface is also supported. The implementation was carried out on the Macintosh environment and the interface design conforms to the look and feel of a native Macintosh application.

Chapter 2 makes a review of the tasks involved in VLSI design giving a basic VLSI information and focuses on the needs of a VLSI designer with basic design methodologies. Then existing Magic implementation which runs on the workstations was discussed in the same chapter.

Then the implementation is briefly examined in Chapter 3 focusing on the tools and environments utilized in the study. This chapter includes the basic source structures with basic program design issues and also goes further through the implementation emphasizing some of the key porting problems depending on the environments.

Chapter 4 takes a look at the resulting application and describes the new user interface with discussions of performance.

Chapter 5 suggests further improvements that could be carried out in future versions of the application.

2. REVIEW

This review starts with a description and general issues in VLSI design. General concepts in VLSI design and VLSI design techniques are covered with an overview of VLSI processes and design rules. Later the UNIX implementation of the VLSI layout design tool, Magic, is examined briefly.

2.1. VLSI Design

The improving technology, and significant financial gains, resulted in a steep development curve in the VLSI area since the beginning of the VLSI technology since the 1930's and today, most integrated circuits (IC) contain a very large number of transistors (up to a million in most designs). Conventional methods for circuit design that involved iteration at the breadboard level proved impractical for designing integrated circuits both due to low designer productivity and high cost of IC fabrication. Methods of handling large data quantities, and accurate transistor models were required. Thereby

> "The tools available to the IC designer are very powerful and dynamic. Most require the use of large computers or, more recently, powerful graphics-intense workstations" (Geiger, Allen & Strader, 1990)

In spite of the sophistication and high cost of both hardware and software to remain competitive in the IC design area, most major contributions today are relatively simple and engineer based, and occur in circuit design, modeling as well as innovations in the CAD tools.

2.1.1. The Design Philosophy

The goal of an IC designer might be stated as follows: design an IC that meets a given set of specifications with minimum labor and physical resources in a limited, and generally short time. The classical approach to circuit design i.e. the breadboard approach, involves too much of iteration and is generally unacceptable for a practically useful IC. It is apparent that much more efficient methods are required with schematic drafting, layout and simulation to cope with the large amounts of data required for any useful IC design.

Lately with the increasing complexity of the VLSI systems, it has been more clear that well-structured design methods, mostly utilization of existing VLSI circuit structures are required for VLSI design. Structured approaches, mostly suggested for large software systems, today, are being used for designing VLSI systems. The black box analogy of

software technology is being used in most VLSI designs. Thus computer aided VLSI design systems are currently supporting such existing cell use. Mostly even they include a pre-designed and tested set of components that may be used as black boxes in new designs. Two basic philosophically identifiable approaches are suggested for IC design;

Bottom-up Approach

The designer starts at the transistor or gate level and designs sub-circuits of increasing complexity, which are tested and simulated as single systems. Then the design is carried out with the interconnection of the sub-circuits to realize the required functionality.

Top-down Approach

The designer repeatedly breaks down the main task to simpler and simpler pieces of tasks. The lowest level tasks are ultimately implemented in silicon either with pre-designed and tested sub-circuits, or with new designs that fit the required specifications. Geiger, Allen and Strader take the software engineering analogy further and state that the top-down approach results in a *silicon compiler* where all blocks are designed automatically with the computer.

Geiger, Allen and Strader (1990) suggest the top-down approach to be more useful and productive for digital systems but not so much fit to analog systems. And generally both approaches should be used successively in different parts of the designs.

The block diagram of a conventional IC design process is depicted in Figure 1.1. The starting point is a set of design specifications which could require a major effort on complicated designs. Preliminary designs are based on simple models of devices or sub-circuits and should be simulated using "good" models, i.e. models that can accurately predict the experimental performance after fabrication.

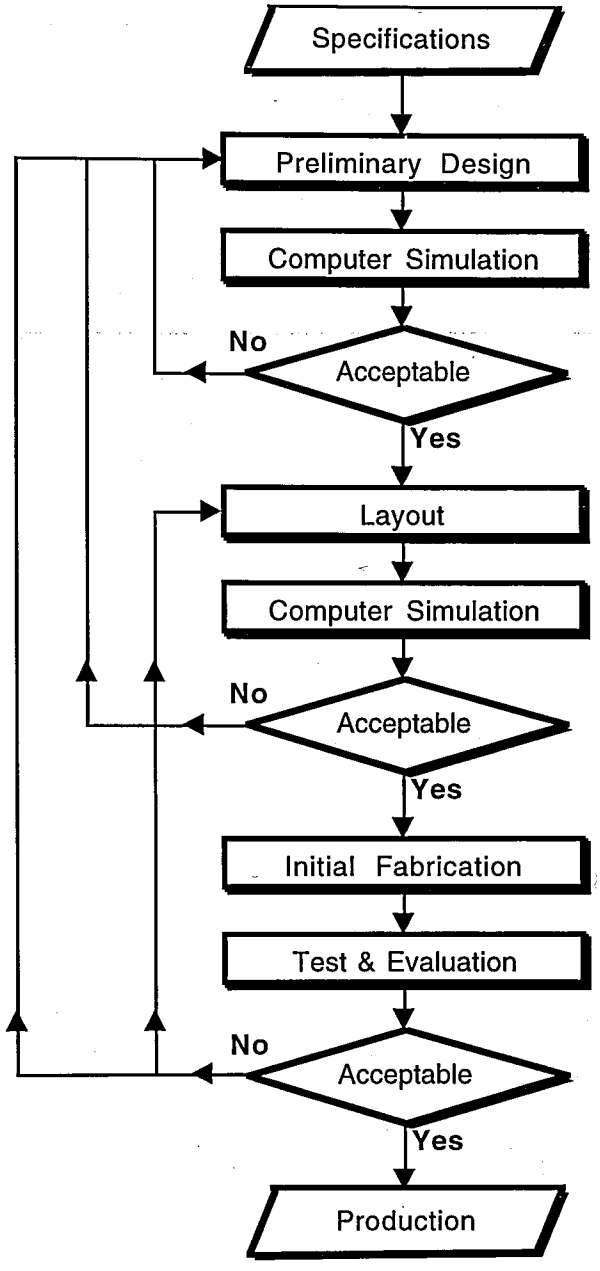


FIGURE 1.1 Block diagram of a conventional IC design.

Once the preliminary design is acceptable, the actual layout starts. A good floor plan should be obtained in the early design phase. More computer simulation must proceed the layout phase since any parasitic effect associated with the layout could cause problems after fabrication. Later comes the fabrication and testing phase which is the most expensive phase of the design process. Iteration at the fabrication level is generally not acceptable because of the high production costs and limited time to get to the market. These facts make the computer simulation phases, and of course computer aided design tools, most important aspects of the whole design process.

2.1.2. Processes

There are currently three processes for the fabrication of monolithic integrated circuits containing active devices. There are the *NMOS*, *CMOS* and *bipolar* processes. The first two are both termed as *MOS* (Metal Oxide Semiconductor) processes. The NMOS and CMOS issues include very similar concepts and are well defined by MOSIS[2]. Another approach combines the mentioned processes into a single process so called as the *Bi-MOS process*. Bi-MOS processes will not be examined in detail for they are simply a combination of the MOS and bipolar processes. Another process for constructing ICs is called the *hybrid process*.

MOS Process

The NMOS (n-channel MOS) and CMOS (Complementary Metal Oxide Semiconductor) processes are quite similar in that both have the FET (or MOSFET - field effect transistor) as the active device. In NMOS processes n-channel MOSFETs are used while in the CMOS processes both n-channel and p-channel MOSFETs are available. One other type of process, namely PMOS process is the dual of NMOS processes and uses p-channel MOSFETs as the active device but generally today NMOS processes are preferred due to higher electron mobility with respect to the hole mobility. In the design point of view, CMOS processes offer advantages when compared to NMOS for they supply much of a design simplicity. Also CMOS offers improvements in power dissipation and performance over MOS.

Bipolar Process

Bipolar processes use the BJT (Bipolar Junction Transistor) as the active device. Higher speeds are available through bipolar processes while they result in considerable internal power dissipation, compared to MOS processes. For logic circuits MOS circuits have a significantly higher component density.

---

[2]The MOSIS (MOS Implementation System) program is sponsored by the U.S. Department of Defense and participated by NFS and aims reducing the cost of MOS process fabrication for experimental and academic purposes. MOSIS assumes responsibility for both mask generation and processing once the design is complete.

Hybrid Process

The hybrid processes combine thin and/or thick passive components over the so known active devices and thereby are rather expensive to fabricate. They could supply practical solutions for applications where precise and temperature-stable passive components are required.

2.1.3. Layouts, Design Rules and Process Parameters

The design method suggested by Carver Mead and Lynn Conway (Mead C.A., Conway L.A., 1980) defines a systematic and much simple approach to NMOS design and simplifies the design rules. This approach later has been extended to include CMOS design rules as well and will be used in the rest of the discussions.

To simplify the subject a bit one can take *VLSI layout design process*, the process of creating appropriate masks to define the sizes and locations of sources, drains, gates and the necessary interconnections. The sophisticated physics behind the design process will be out of the scope of this discussion and if we symbolize each substance with a different color, the process will be nothing more than drawing overlapping color boxes considering some rules that define the minimum width and spacing of the boxes. Of course we should take a brief look at the electronic counterparts of our colored boxes, and consider how a physical component could be created by the *color box* approach and have the notion of design rules that originate from the physical properties of our colored boxes.

Transistor Structures

Diffusions for n-channel and p-channel devices involve different masks, thus they must be distinguishable. NMOS source and drain regions are symbolized by green and PMOS, by brown. Color Plate 1 shows the transistor structures for the discussed n-channel and p-channel devices with their symbolic colors.

The NMOS and PMOS single metal processes are analogous to a three level circuit board in that, they also consist of three conducting layers. Each layer is isolated from each other by silicon dioxide and there exist specific contacts between each layer. The top layer is the *metalization* symbolized by blue, followed by a *polysilicon* layer (*red*) and finally the *diffused* layer (*green* for NMOS, *brown* for PMOS). Contacts are shown in *black*. The three level structures shown in Color Plate 1 can be expanded to two or more metal or polysilicon layers only with a little complication in the fabrication. *Dark blue* might be used for the second metal layer, and a different shade of *red* for the second polysilicon.

Metal layers can cross both of the other layers with no interference but a parasitic capacitance. Thus blue layers can cross the other layers with no connection up the black connecting areas. Polysilicon, crossing a diffused area, results in a FET, i.e. red crossing green results in an n-channel transistor, while red crossing brown, makes up a p-channel transistor. Color Plate 2 shows the layouts, the electronic counterparts, and stick representations[3] for the discussed components.

Design Rules

The object of the design rules is to allow a fast translation of the circuit design, probably in the stick form to an error free circuit layout. One of the most important factors associated with design rules is the achievebility of the rules with the current process line.

The simple lambda (λ) based design rules, set out first by Mead and Conway have been widely used and supported. The idea is to base all rules to a lambda parameter which leads to a simple set of rules for the designer, and possibility to extend the parameter during the fabrication, according to limitations of the process line. In general, λ based design results in layouts of longer lifetime, for they will be up-to-date with current technology even if they were past designs.

A basic set of lambda based design rules are given in Table 1.1 for NMOS processes. Similar but slightly more complex design rules hold for CMOS. Design rules of both NMOS and CMOS processes based on MOSIS is given in Appendices A and B respectively for completeness.

---

[3] Stick diagrams are used to convay layer information through use of color code. This information can relatively easily be turned into mask layouts once the size of the transistor gates is given, which makes this kind of representation useful in layout designs.

TABLE 1.1 Lambda based MOSIS, NMOS design rules.

N^+ diffusion Mask	
Diffusion width	2λ (due to the standard deviation of line widths)
Diffusion spacing	3λ (to keep depletion regions from short circuit)
Implant Mask	
Implant gate overlap	1.5λ (to avoid an enhancement FET around depl. FET)
Implant-enhancement gate spacing	1.5λ (to avoid gate short circuits)
Buried Contact Mask	
Buried contact-active device	2λ (to avoid short circuits)
Overlap in diffusion direction	1λ (to ensure contact)
Buried contact-other poly or diff.	2λ (to avoid short circuits)
Buried contact poly -other poly	2λ (to avoid short circuits)
Poly Mask	
Poly width	2λ (due to the standard deviation of line widths)
Poly spacing	2λ (to avoid short circuits)
Poly-diffusion spacing	1λ (to avoid overlap capacitance)
Poly gate extension beyond diff.	2λ (to avoid shorting the source and drain)
Diffusion to poly edge	2λ (to ensure source and drain regions on required sides)
Contact Mask	
Contact Size	2λ*2λ (to ensure good contact)
Contact-diffusion overlap	1λ (to ensure good contact)
Contact-poly overlap	1λ (to ensure good contact)
Contact-contact spacing	2λ (to avoid shorting circuiting contacts)
Contact-FET channel spacing	2λ (to avoid shorting circuiting gates)
Contact-metal overlap	1λ (to ensure good contact)
Metal Mask	
Metal width	3λ (metal runs over roughest terrain)
Metal spacing	3λ (metal runs over roughest terrain)

2.2. Magic; the UNIX Implementation

Magic version 6 gathers the work done by numerous people at different institutions since the first release of Magic from Berkeley at 1986.

2.2.1. What is Magic

> "Magic is an interactive system for creating and modifying VLSI circuit layouts." (OusterHout, 1989)

The word "interactive" in the definition is what makes Magic more than a simple layer based drawing program. Magic has a built in design rule checker, which runs in the background while the design is being edited. The design rule checker continuously checks the suitability of the layout being edited to the so called *technology file*. In fact the technology file is where the design rules, for the process being used, are defined. So when a layout is complete in Magic, the layout is known to be free of any design rule violations and inconsistencies with the technology used. Moreover Magic knows about connectivity and transistor which enables one to stretch and compact wires, and has a built in circuit extractor. Magic is based on Mead-Conway style of design as discussed in the previous chapter. Thus it uses rather simple design rules which makes Magic's fast design rule checker possible. The drawback is that with Magic only *Manhattan*[4] designs are possible. This results in a 5 to 10 per cent overhead in the circuit size, but this cost is acceptable considering the reduced design time.

Magic runs on UNIX workstations and supports most windowing systems for user interface. Though it could be used without a graphical display, design of a layout, practically, requires a color graphics display and a mouse.

2.2.2. User Interaction in Magic

Commands in Magic can be invoked in three ways. The user invokes certain commands in the most user friendly manner with mouse clicks on the layout window. Commands may be invoked by typing single keystrokes on the keyboard; and finally by typing the whole command (or an abbreviation) on the keyboard. The *box*, a rectangular mark on the layout window created by the user with mouse clicks, guides many of the

---

[4] Manhattan design style is one that supports only horizontal and vertical geometries. The name arises from the similarity between the aerial view of the street layout of New York's Manhattan borough and Manhattan style VLSI layouts.

commands. While mouse supplies a much faster interface for layout design, all commands in Magic can be given in long text form, which makes use of Magic possible even on a text-terminal.

2.2.3. Basic Concepts

Cells

Cell is the fundamental concept in Magic. A layout designed in Magic is a hierarchical collection of cells. Each cell contains three elements;

Paint

Paint is the overall name for the colored boxes existing in the cell. Different colors in paint correspond to different material and masks, so the paint defines the eventual function of the designed layout practically. The other two elements are only available for reducing the design time and aiding the designer.

Labels

Labels are pieces of text attached to paint to provide information to the designer and other tools that will process the circuit. Most labels are only node names for easy reference of nodes in tools such as routers, simulators and timing analyzers.

Subcells

Each cell in the layout (or the layout itself, for it could also be regarded as a cell), might contain subcells which make up the hierarchical structure as mentioned above. In fact the subcells are no more than previously designed pieces of layouts. In any layout you create you can use one or more of pre-designed layouts. This structure makes the ideas of abstraction and black-box approaches possible with Magic and favors more structural designs. Of course the hierarchical structure is only implemented for structural layout designs and any layout created with the this structure might also be designed in a *flat* manner i.e. using paint only in a single level of cell.

Layers

The layer concept in Magic might be a bit confusing at a first glance as a Magic *layer* does not correspond to a physical entity (such as mask layers) as one might expect. Magic layers could be called *abstract layers* because they correspond to constructs like wires and contacts, rather than mask layers. They are also called *logs* because they look like sticks.

Magic has one paint layer for each conducting material (polysilicon, ndiffusion, metal1 etc.), plus one additional paint layer for each kind of contact (pcontact, ndcontact, m2contact etc.). Obviously the layers are dependent of the technology being used. In Magic the designer does not draw implants, wells, buried windows or contact via holes. Instead, only primary conducting layers are drawn and some of their overlapping area is painted with types as n-transistor or polysilicon contact. The necessary polysilicon and diffusion with any implants will be generated automatically by Magic when creating a CIF file for the design. For contacts, the contacting area between the conducting layers must be painted. Rest of the work as creating the mask layers with vias and buried windows will be carried out by Magic. The layer based approach of Magic simplifies the design rules, thereby making the fast operation of the design rule checker possible. Moreover the user does not anymore have to cope with supplying unnecessary information that could be created automatically.

2.2.4. More About Magic

Design Rule Checking

During the editing of a layout in Magic, the system continuously checks for design rule violations even without the notice of the user. The design rule violations are based on the current technology defined in the *technology file*, which will be discussed in the next chapter. Continuous design rule checking gives the designer always an up-to-date picture of problems in his layout. Any change in the paint forces magic to recheck the changed and related areas, the designer can, in no way, force Magic to forget about the change in the layout. Some commands to pause error checking are available but still then the unchecked areas in the layout will be collected into a new *layer* and will be checked once the designer turns on error checking again. The error is presented to the user graphically as white stains on the layout. Furthermore the user has the chance to learn the reason for an error is it is not apparent at a first glance. Magic supplies design rule checking commands to count the errors on a layout, to select the error and to display the rule related to a specific error.

Magic's design rule checker works with cell hierarchies as well. Magic takes the below three steps when working with a hierarchical layout.

- The paint of current cell is checked for any design rule violations without considering any of the subcells.

- The combined paint of the current cell and its subcells are checked for consistency. If any inconstancies are found caused by the interaction of paint in the current cell with any of its subcells, the error is displayed in the current cell.

• Finally, these two approaches are carried out in a recursive manner, and all subcells of current cell are checked for consistency according to the above steps, up to the lowest level in the hierarchy. The inconsistency of any cell in the recursive search is reflected to its parent, so that the errors will be apparent on the highest level, i.e. the current cell.

3. THE IMPLEMENTATION, DESIGN AND METHODOLOGY

The implementation was discussed in this chapter, emphasizing the key points in the design. Tools and environment and the reasoning for their use is given. Basic data structures and source organization, with their counterparts in the new implementation is given with a comparison of the program structures of both.

3.1. Implementation Issues

The implementation of Magic on the MacOS is based on the sources of Magic version 6. Magic version 6 has been designed to cover a range of hardware platforms all of which are UNIX workstations. The implementation mainly aims to carry Magic to a much more available platform, desktop computers. Using the more user friendly Macintosh user interface, the design also aims to supply the user with the notion of a simple drawing software.

One of the key decisions in the software design was maintainability of the software. Magic has been available since the middle of 80's and is being maintained by a large number of software engineers. Besides, future releases of Magic will be supplying the designer with more tools which will probably make the new implementation out of date. Therefore the basic approach in the design was to keep current sources as much as possible and to supply a smooth transition of current sources, maybe adding transparent new routines to the Macintosh operating system. This approach will lead to a fast re-implementation on the Macintosh side, once new versions of Magic are released.

Furthermore, during the implementation, Magic's existing source was completely retained and changes were made only through conditional compilation flags to have the chance to see the required changes at any time.

User interface additions were also designed in a compatible manner, using Magic's existing structures, so that they will not have to be re-implemented in the new versions.

3.2. Tools and Environment

3.2.1. MacOS and PowerPC

Macintosh operating system was selected for the new implementation not only because of its well suitability to design a user friendly interface, but also the smooth transition path from 68K based Macintoshes to RISC based PowerPC[5] computers. The computational power of RISC technology might be more than suitable in both the design and simulation of the design, while simulation might sometimes take unacceptably long on 68K computers. The programming tools used were selected considering both processors, which made it possible to come up with a single FAT[6] application running on both platforms.

3.2.2. Language

Magic's UNIX source is completely written in C. The new implementation was carried out using C++ to make use of object oriented programming features. Use of C++ led to an easy to maintain source and easy access to Magic's exiting routines. There was one major problem with C++. As Magic's headers were not written to handle C++ compilations and they were problematic when used from C++ sources. This arouse the need of rewriting some of the headers. This drawback is well compensated considering the advantages of the object oriented software technology.

3.2.3. Compiler

The implementation was started using MPW 3.3.1[7] development environment and Symantech C++ 7.0.3[8]. Later both speed problems encountered in using MPW and PowerPC porting considerations led to the transition to another compiler. Currently the project is being developed using Metrowerks CodeWarrior C++ version 1.2. Current compiler supports both 68K and PowerPC development with slight changes to the source and has reduced the compilation times drastically.

---

[5] A new line of RISC based personal computers first introduced by Apple Computer in 1994. PowerPC computers are based on the PowerPC microprocessor created by the joint develompent of Apple Computer, IBM and Motorola Company.

[6] FAT applications contain code for both 68K and PowerPC microprocessors and single application will determine the configuration to run on both microprocessors taking full advantage of the processor.

[7] Macintosh Proggrammer's Workshop; Copyright © Apple Computer Inc, 1995 - 1993.

[8] Copyright © 1985-1994 Symantec Corporation

3.2.4. Libraries, Tools

MacApp[9]3.1.1 is being used as the application framework. MacApp is an object-oriented framework, which is designed to automate many of the programming tasks of building a Macintosh application. Thus lets the programmer concentrate on the specifics of the program, rather than spending time grappling with the details of the operating system. More information about MacApp and the structure of a MacApp application will be given later on in this text.

Use of MacApp was one of the key decisions in the implementation because it required a merging of two complete environments to come up with a single application. Besides, this increased the project size drastically. On the other hand, MacApp is obviously a good choice in any software project. It includes a powerful class hierarchy and use of MacApp ensures a consistent software, with a consistent user interface. Furthermore, MacApp is well -supported by Apple Computer, thus any extension in the MacOS will be incorporated into MacApp in no time. This makes MacApp based programs easy to keep up-to-date with the new technologies in system software.

3.3. Design and Methodology; a Comparison

3.3.1. Magic Source and Basic Structures

Summary of Source Structure

This section contains brief summaries of what is in each of the Magic UNIX source sub-directories. Some information is given about what the sub-directory stands for and its position in the new implementation. All sub-directories have been processed and had minor changes because of the Operating System inconsistencies, limitations, and/or optimization purposes as discussed in previous chapters or will be discussed in the current chapter. Such changes will not be discussed for each of the modules and only global changes that are worth mentioning will be explained.

calma

Contains code to read and write Calma Stream-format files. It uses many of the procedures in the *cif* module. Minor changes were made because of file system specifications.

---

[9] Copyright Apple Computer Inc, 1995 - 1993.

cif

Contains code to process the CIF sections of technology files, and to generate CIF files from Magic. Just as the calma section only file system related parts were re-implemented.

cmwind

Contains code to implement special windows for editing color maps. Is not being used in the new implementation.

commands

The procedures in this module contain the top-level command routines for layout commands (commands that are valid in all windows are handled in the windows module). These routines generally just parse the commands, check for errors, and call other routines to carry out the actions. Parsing of the commands and sending of commands to the Magic command queue was re-implemented and enhanced to intake the fake commands created by the user events in the new implementation.

database

This is the largest and most important Magic module. It implements the hierarchical corner-stitched database, and reads and writes Magic files. This module required mostly changes in memory management and also file system routines. The general data structures were not changed to be consistent with the Magic system in the future versions.

dbwind

Provides display functions specific to layout windows, including managing the box, re-displaying layout, and displaying highlights and feedback. Most of this module is replaced or changed as is basically one of the most graphical intense parts.

debug

There's not much in this module, just a few routines used for debugging purposes. They are not used at all, in the new implementation.

drc

This module contains the incremental design-rule checker. It contains code to read the *drc* sections of technology files, record areas to be rechecked, and recheck those areas in a hierarchical fashion.

ext2sim

The ext2sim directory isn't part of Magic itself. It's a self-contained program that flattens the hierarchical *.ext* files generated by Magic's extractor into a single file in *.sim* format. See the manual page ext2sim. Only needs an interface change since it is a simple and self contained program with no more user interface than selection of a file.

ext2spice

This is another self-contained program. It converts *.ext* files into single file in spice format. See the manual page *ext2spice.* Same issues as *ext2sim* hold here as well.

extflat

Contains code that is used by the *extract* module and the *ext2...* programs. The module produces a library that is linked in with the above programs. Did not require major changes in the new implementation.

extract

Contains code to read the *extract* sections of technology files, and to generate hierarchical circuit descriptions (*.ext* files) from Magic layouts. Did not require major changes except for the file system related parts.

fsleeper

Like *ext2sim* this directory is a self-contained program that allows a graphics terminal attached to one machine to be used with Magic running on a different machine. Is not used in the new implementation.

garouter

Contains the gate array router from Lawrence Livermore National Labs.

gcr

Contains the channel router, which is an extension of Rivest's greedy router that can handle switch boxes and obstacles in the channels.

graphics

This is the lowest-level graphics module. It was completely replaced for the new environment.

grouter

The files in this module implement the global router, which computes the sequence of channels that each net is to pass through.

irouter

Contains the interactive router written by Michael Arnold at Lawrence Livermore National Labs. This router allows the user to route nets interactively, using special hint layers to control the routing.

macros

Implements simple keyboard macros. Rewritten to account for the different macro mechanism.

magicusage

Like *ext2sim*, this is also a self-contained program. It searches through a layout to find all the files that are used in it. Was ignored in the new implementation.

main

This module contains the main program for Magic, which parses command-line parameters, initializes the world, and then transfers control to *textio*. Of course this module was replaced with similar routines to do the initialization and hold the control in hand.

misc

A few small things that didn't belong anyplace else. They could be used as they were.

mpack

Contains routines that implement the Tpack tile-packing interface using the Magic database. (not supported)

mzrouter

Contains maze routing routines that are used by the irouter and garouter modules.

net2ir

Contains a program to convert a netlist into irouter commands. Not implemented in the current implementation.

netlist

Netlist manipulation routines. Did not require major changes.

netmenu

Implements netlists and the special netlist-editing windows.

parser

Contains the code that parses command lines into arguments.

plot

The internals of the *plot* command. Not implemented in the current implementation.

plow

This module contains the code to support the *plow* and *straighten* commands. Modified to fit the new interface.

prleak

Also not part of Magic itself. Prleak is a self-contained program intended for use in debugging Magic's memory allocator. It analyzes a trace of mallocs/frees to look for memory leaks. This part was not used in the new implementation as memory allocation was completely replaced.

resis

Resis is a module that does better resistance extraction via the :extresis command. Courtesy of Don Stark of Stanford.

router

Contains the top-level routing code, including procedures to read the router sections of technology files, chop free space up into channels, analyze obstacles, and paint back the results produced by the channel router.

select

This module contains files that manage the selection. The routines here provide facilities for making a selection, enumerating what's in the selection, and manipulating the selection in several ways, such as moving it or copying it. This part was modified to fit the new interface.

signals

Handles signals such as the interrupt key and control-Z. UNIX system dependent parts. (not needed)

sim

Provides an interactive interface to the simulator rsim.

tech

This module contains the top-level technology file reading code, and the current technology files. The code does little except to read technology file lines, parse them into arguments, and pass them off to clients in other modules (such as *drc* or *database*). This module is currently used without major changes but will be re-implemented in future versions for optimization purposes.

textio

The top-level command interpreter. This module grabs commands from the keyboard or mouse and sends them to the window module for processing. Also provides routines for message and error printout, and to manage the prompt on the screen. This module was where the Magic's event mechanism worked. In the new implementation this part was re-

implemented preserving the current interface and was one of the main bridges between the Magic's command handlers and the new implementation.

tiles

Implements basic corner-stitched tile planes. This module was separated from *database* in order to allow other clients to use tile planes without using the other database facilities too. Did not face major modifications.

undo

The *undo* module provides the overall framework for undo and redo operations, in that it stores lists of actions. However, all the specific actions are managed by clients such as *database* or *netmenu.* Magic's undo mechanism is supported in the new implementation with the menu based user interface.

utils

This module implements a whole bunch of utility procedures, including a geometry package for dealing with rectangles and points and transformations, a heap package, a hash table package, a stack package, a revised memory allocator, and lots of other stuff. Was partly replaced by new routines as this part included operating system dependent routines.

windows

This is the overall window manager. It keeps track of windows and calls clients (like *dbwind* and *cmwind*) to process window-specific operations such as re-displaying or processing commands. Commands that are valid in all windows, such as resizing or moving windows, are implemented here. Magic implemented all the windowing system routines in its sources. This module was completely replaced in the new implementation.

wiring

The files in this directory implement the *wire* command. There are routines to select wiring material, add wire legs, and place contacts. Did not require major changes except for the new interface.

Tiles

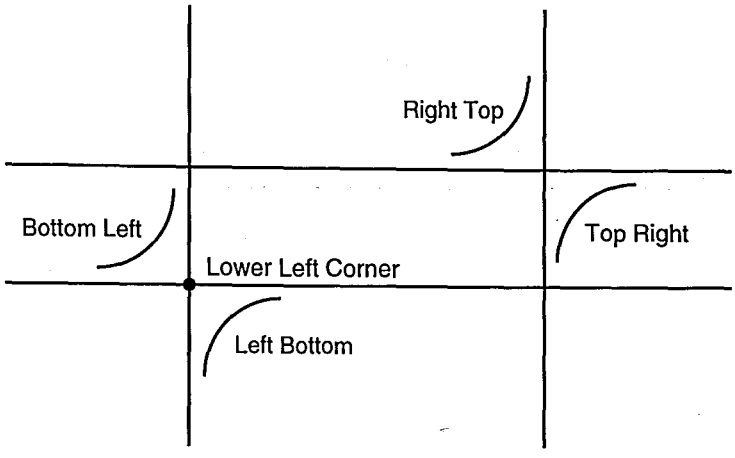


FIGURE 2.3.1 Graphical representation of a Magic *tile*.

Tile is the basic data structure in Magic. A tile is used to represent both space and solid area in a plane. Figure 2.3.1 pictures the basic notion of a tile. A tile structure stores the lower left point of the tile with four pointers to neighboring tiles which define the left-bottom, bottom-left, top-right, right-top corners respectively. In fact this approach defines a rectangular area, with pointers to its four adjacent rectangles, in a more efficient way for Magic's algorithms. Listing 2.3.1 gives the structure of a tile.

LISTING 2.3.1 The tile data structure.

```
typedef struct tile
{
      ClientData   ti_body;   /* Body of tile */
      struct tile       *ti_lb;     /* Left bottom corner stitch */
      struct tile       *ti_bl;     /* Bottom left corner stitch */
      struct tile       *ti_tr;     /* Top right corner stitch */
      struct tile       *ti_rt;     /* Right top corner stitch */
      Point  ti_ll;                 /* Lower left coordinate */
      ClientData   ti_client;       /* Not used yet. */
} Tile;
```

Tile Planes

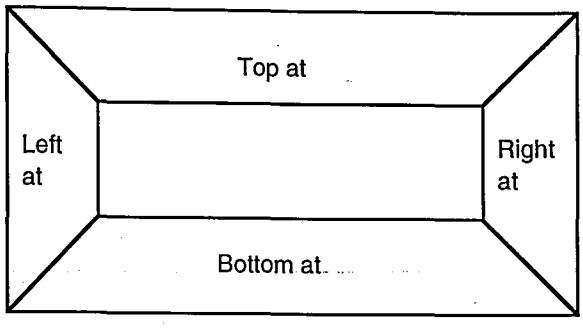


FIGURE 2.3.2 Planes limited by *pseudo-tiles.*

A plane of tiles, consists of the four special tiles needed to surround all internal tiles on all sides. Logically, these tiles appear as in Figure 2.3.2 except for the fact that all are located off at infinity and are non-existing *pseudo-tiles*. Generally speaking they define the limits of a search operation that would go through all tiles in a specific Magic layer. Thus as in Listing 2.3.2, there exist four limiting tiles, with a hint tile, that might be used to give an efficient starting point for the previously mentioned search.

LISTING 2.3.2 The plane data structure.

```
typedef struct
{
      Tile *pl_left;          /* Left pseudo-tile */
      Tile *pl_top;           /* Top pseudo-tile */
      Tile *pl_right;         /* Right pseudo-tile */
      Tile *pl_bottom;  /* Bottom pseudo-tile */
      Tile *pl_hint;          /* Pointer to a "hint" at which to
                              begin searching. */
} Plane;
```

Cell definitions

A cell definition, although will not be examined in detail here, is the fundamental structure for the cell. There exists one cell definition for each cell (or layout) opened in the application. Listing 2.3.3 gives the cell definition structure. This structure stores almost all information about an existing cell. For instance all possible planes are stored in the *cd_planes* element and *cd_types* carries a mask of the planes that are actually used. Furthermore all labels belonging to the cell also do exist in this structure. Anyway this structure is the basic structure used for the drawing of the cell, and any manipulation in the cell directly affects this structure.

LISTING 2.3.3 The cell definition data structure.

```
typedef struct celldef
{
      int cd_flags;      /* Flag definitions for the cell */
      Rect cd_bbox;      /* Bounding box for cell */
      char *cd_file;     /* File containing cell definition */
      char *cd_name;     /* Name of cell */
      struct celluse     *cd_parents;
                         /* NULL-terminated list of all uses */
      Plane *cd_planes[MAXPLANES]; /* Tiles */
      ClientData  cd_client; /* This space for rent */
      int  cd_timestamp;     /* Unique integer identifying last
                             time that paint, labels, or subcell
                             placements changed in this def. */
      Label *cd_labels;      /* Label list for this cell */
      Label *cd_lastLabel;   /* Last label in list for cell. */
      char *cd_technology;   /* Name of technology for this cell
                             (Not yet used.) */
      ClientData cd_props;   /* Private data used to maintain
                             lists of properties. Properties
                             are name-value pairs and provide
                             a flexible way of extending the
                             data that is stored in a CellDef. */
      ClientData cd_filler;  /* UNUSED */
      HashTable cd_idHash;   /* Maps cell use ids to cell uses.
                             Indexed by cell use id; value is a
                             pointer to the CellUse. */
      TileTypeBitMask cd_types;     /* Types of tiles in cell */
} CellDef;
```

Cell Uses

A cell use carries information about each instantiation of the cell. With Magic it is possible to edit a single layout in multiple windows each viewing a different part of the layout in a different scale. The cell use structure given in Listing 2.3.4 is what makes this possible. It holds information about its cell definition and contains a transform from the cell definitions coordinates to the current use coordinates.

LISTING 2.3.4 The cell definition data structure.

```
typedef struct celluse
{
      unsigned cu_expandMask;/* Mask of windows in which
                              this use is expanded. */
      unsigned cu_flags;      /* CURRENTLY UNUSED */
      Transform cu_transform;/* Transform parent coordinates */
      char *cu_id;            /* Unique identifier of this use */
      ArrayInfo cu_array;     /* Arraying information */
      CellDef *cu_def;        /* The definition of the cell */
      struct celluse *cu_nextuse;  /* Next in list of uses of our
                              def, or NULL for end of list.*/
      CellDef *cu_parent;     /* Cell def containing this use */
      Rect cu_bbox;           /* Bounding box of this use, with
                              arraying taken into account,
                              in coordinates of the parent def. */
      ClientData cu_client;   /* This space for rent */
      int cu_delta;           /* Used by snowplow */
} CellUse;
```

Window Related Structures

One more structure in Magic is worth mentioning here, for it was used to some extent and will be talked about in the rest of the discussions. So called, *DBWI* structure is the representation of a window in Magic excluding the graphical aspects of the window. This structure has pointers to both the cell structures, (as discussed above) displayed in the window and to the graphical window object (depends on the graphics display and windowing system). This structure was also used in the new implementation to some extent.

Support Files

Magic requires some support files to run that define some optional parameters in the execution of the system. Some of those files are still required in the new implementation while some are not used any more.

Technology Files

This file is the most important of the support files, for it includes the technology Magic uses, which means all layers, design rules, the layout definitions depend on this file. This file might be changed to run Magic with a completely new technology. The new implementation also uses this file as the source of technology.

Display Styles

The display style file were used to read the graphics driver information in Magic. The new implementation does not use any of the information in this file, but still tries to open and read the file for consistency. Might be completely removed in future versions.

Glyph Files

Glyph files were used to define cursor patterns and are obsolete in the new implementation.

Color Maps

This file defines the real colors used in the layouts. Is still used in the new implementation, but might be replaced by a better way to get the color information in future releases.

3.3.2. MacApp as an Application Framework

MacApp is an object-oriented application framework which was designed to aid a Macintosh programmer with the automatization of the standard tasks that are required for any Macintosh application. MacApps object oriented structure lets the programmer to borrow and customize code from its class library. MacApp classes provide the basic functionality for an application and the programmer should modify that functionality to perform the program specific tasks, by overriding methods of the supplied classes or by building other classes that can be derived from MacApp's classes.

As a rule, MacApp handles events involving the controls—such as the size box and close box—of each window, since these are predictable; and the programmer must handle events involving the content of the windows, since this only depends on the program itself.

Common sense logic of this sort figures out what MacApp does for the programmer. Sometimes MacApp knows when some behavior specific to the program is required. When this is so, MacApp sends specific messages to the code that must be responded by the programmer.

Managing windows and Views

MacApp has a powerful view hierarchy which, definitely, is unavoidable in any application supporting user interface. Manipulating a Macintosh window will be briefly examined here with some information about the view hierarchy, to give the reader a notion

of programming with MacApp. Details of programming with MacApp and a complete class hierarchy will not be given in the scope of this text.

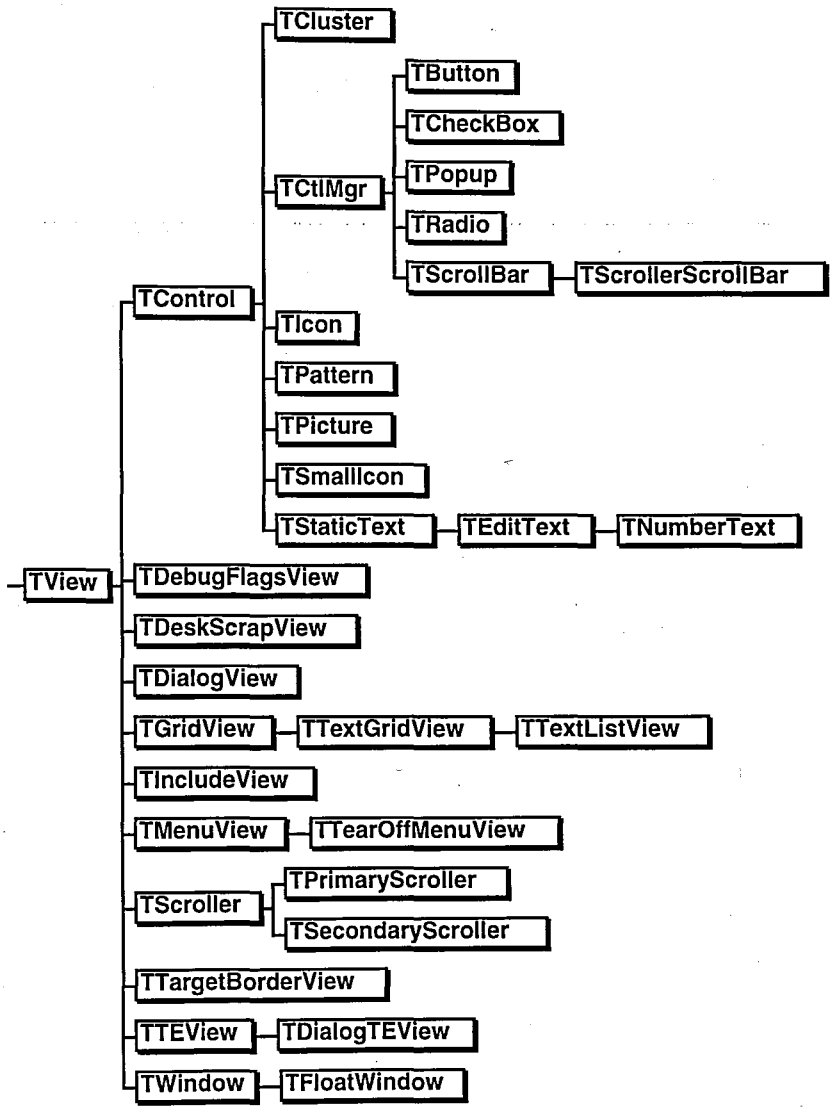


FIGURE 2.3.3 MacApp class hierarchy for descendants of TView.

MacApp's code draws the window and responds if the user manipulates the size box, title bar, or zoom box. The programmer draws the contents of the window and responds if the user manipulates those contents. Also some code must be added if the window components will be working unconventionally.

In MacApp, the window itself and each part of it is represented by a view object. These view objects are organized hierarchically, with the window at the top of the hierarchy. One view object represents the window frame, another represents each scroll bar, and yet another represents the main content of the window. Each view object can draw

the part of the window it represents and can respond to events relating specifically to its part.

For example, in a typical window that displays a document on the screen, the view object that represents the window itself is responsible for drawing the window frame and the blank white background of the window. It also responds to mouse clicks in the window frame, which may signify dragging, resizing, zooming, or closing of the window. The view object that represents a scroll bar is responsible for drawing the scroll bar, as well as for responding to mouse clicks in the scroll bar, which typically cause the scroll bar to be redrawn and tell the main content view of the window to scroll itself.

The behavior of the view object that represents the main content of the window varies widely from application to application.

Document Class

MacApp also helps the programmer to handle the notion of a document with its document class. Documents have windows to let users manipulate the document data the relation between the document and its window (or windows) is well organized requiring both MacApp and the programmers code.

The programmer creates document objects to represent the data while contained in the document. MacApp provides a guide through its messages, in when to read and write the data. MacApp responds to user actions associated with menu items related to documents, but of course the programmer is responsible to get the document data from the disk and put it back.

Failure Handling

MacApp includes a failure-handling mechanism, that allows applications to clean up in a suitable way after a code failure and to continue execution at a specified point. Thus conditions that might cause system halts or crashes might be detected and the user might be warned for those conditions without any data loss, using the failure handling mechanism. Of course, the programmer must implement this mechanism in a suitable way to detect all possible failures. This mechanism is also one of the most important benefits of a MacApp program.

3.3.3. Basic Correlations

Basic Program Structure Comparison

Magic, the UNIX implementation was designed and implemented to run as a complete system to run on UNIX operating systems. The system does not depend on native operating system mechanisms and the event loop is fully implemented using simple system level input/output mechanisms. Even though Magic user interface is somewhat event driven[10]. The whole event mechanism was implemented in a way specific to Magic. Magic's event mechanism and the program structure is schematically given in Figure 2.3.4.

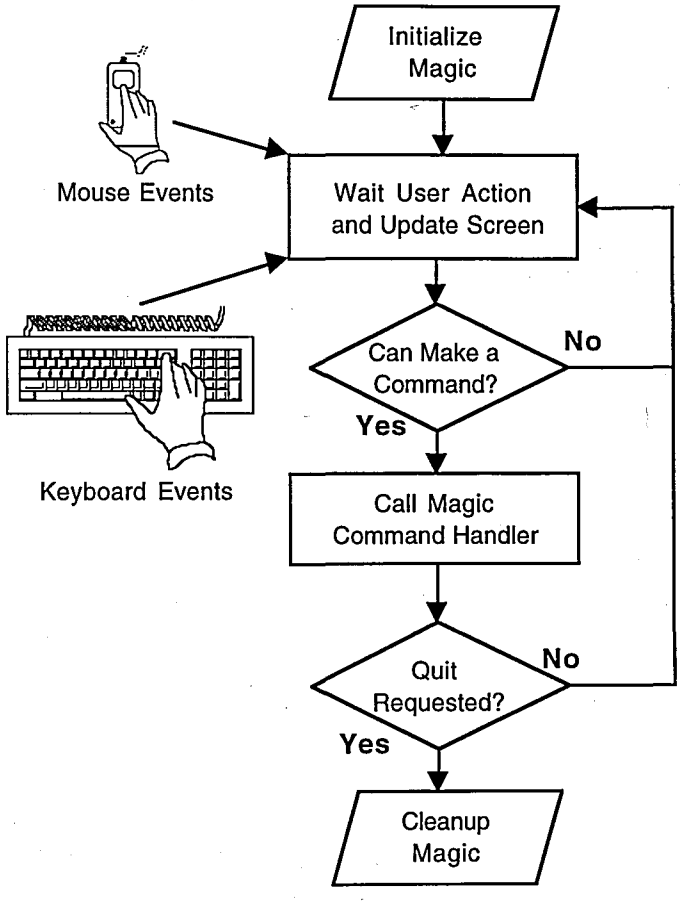


FIGURE 2.3.4 Block diagram of original Magic event structure.

---

[10] Event driven programming style is used in graphical user interface environments and is based on waiting for and responding to user actions as long as the program runs. Thus, generally speaking the program loops polling for any user or system event and calls the appropriate event handler once an event is retrieved. The older style of programming forced the user to respond on specific points and did not consider user interaction on other parts of the program.

The main loop of the program waits for specific happenings as user inputs or any UNIX system interrupt and calls the update routines if any of the windows need update. Once such a happening is received, it will be packed into a Magic event package and put into the event queue. The event package is further examined in the rest of the discussions. The loop also checks for the event queue and sees if the available events are sufficient to make up a Magic command. One or more events might be required to complete a single command, such as two mouse clicks to complete a rectangle or a few key hits to complete a long text command. Once a command is ready, a command package is created packing more information that might be needed by the command handler. Later the command handler for the command is called to handle the real command.

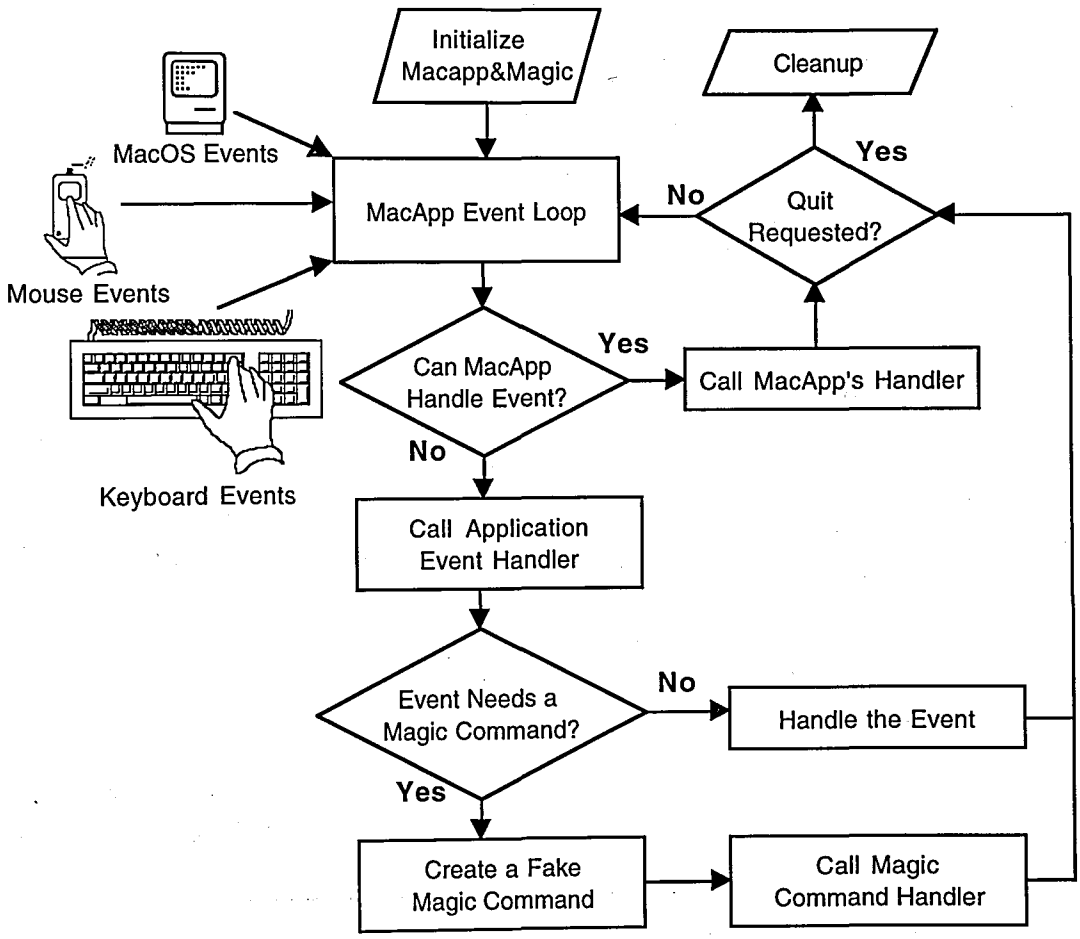


FIGURE 2.3.5 Block diagram of new implementation main loop.

With the notion of keeping the changes transparent to the Magic system in the light of the discussions above, this event loop was designed and implemented using MacApp and dispatching the commands to Magic as pictured in Figure 2.3.5. In the new implementation, MacApp's event handlers were overridden in particular places to handle the event, if the event requires a Magic interaction, the MacApp event (or events) is

transformed into a Magic style command and dispatched to the Magic command handler. Of course if the command results in a change in the layout, appropriate updates are made to provide feedback to the user.

Commands

Magic command structure is a package that can hold information related to both keyboard based events and mouse based events. Once an event sequence to complete a command is ready, the sequence is used to fill in the command structure given in Listing 2.3.5

LISTING 2.3.5 The command data structure.

```
typedef struct {
/* A command -- either a button push or a textual command. */
      Point tx_p;       /* The location of the pointing device
                        when this command was invoked. */
      int tx_button;    /* The event type (button number). */
      int tx_buttonAction;    /* The action of the button
                              (if any), such as BUTTON_UP, DOWN */
      int tx_argc;      /* The number of textual command words,
                        including the command itself. 0 means no
                        textual command. */
      char *tx_argv[TX_MAXARGS];    /* An array of pointers to the
                        words (if any) that make the command. */
      int tx_wid;       /* The ID of the window that this command
                        is destined for. The point 'tx_p' is in
                        this window's coordinate system.
                        (See windows.h for window IDs.) */
      char tx_argstring[TX_MAX_CMDLEN];
                        /* The storage used for the command line.
                        Tx_argv[] points into this. */
} TxCommand;
```

Document vs. Cell

MacApp document class was used to correspond to the cell structure of Magic. The document class is overridden to implement a cell layout with multiple windows. Each instantiaton of the cell i.e. each time the user wants to see the same cell in a new window, a new window is created and appended to the document's open window list. The document (or cell) may be edited using any if its open windows. Any change in any of the windows is reflected directly in other windows belonging to same cell. This gives the user the ability to start editing a piece of the layout in a new window in detail and observe the results in the whole layout. Closing the windows does not close the document (i.e. the cell file) unless the last window belonging to the document is closed. Closing the last window closes the cell file asking the user to discard or save the changes made. In fact the document structure holds a pointer to the Magic's cell definition structure as discussed below, and maintains a list of open windows.

MacApp Window vs. Magic Window

MacApp has its own window class to manage the Macintosh windows and this class was subclassed to maintain a Magic layout window in the new implementation. On the other hand, Magic window structure was also used to some extent to keep consistency. Magic stores the layout in the *cell definition* structure and displays the same structure in different windows through *cell uses* as discussed in the previous sections. The cell definition in fact does not only contain the information to display the layout on screen. Layout structure is kept in much higher accuracy compared with screen accuracy (In fact 1/4096th of a screen pixel). Besides as each window may be viewing a different part of the layout and in a different scale, a screen transformation is essential for each window. Thus, each window contains a 'surface', whose contents are maintained depending on the location and scale of the window with respect to the layout. The window structure maintains the transform from surface coordinates to screen coordinates. That transform structure is kept in place in the new implementation and used from within the subclassed layout window. As one might guess, pointers to each other were added to both structures to make access from one structure to its counterpart possible.

3.4. IMPLEMENTATION DETAILS

This chapter goes further on implementation introducing the basic problems in the implementation, depending on the differences in operating system.

3.4.1. File Systems

Macintosh file system differs in most file systems in many aspects. This difference was one of the main drawbacks during the implementation as the implementation was based on the idea of letting the original system without major changes.

UNIX relies on the path name for reference to a file. In the UNIX implementation some specific file paths are known to exist, once Magic is installed. Magic specifies a few pre-defined path names and does a search in those paths once a file is requested.

Macintosh file system specifies the file through volume and directory reference numbers. The UNIX approach can not be used in the MacOS for two reasons. First, there might be more than one volume, (or file server) with the same name, and using the path name might result in a confusion, and second, a Macintosh user is accustomed to specify a file in any volume, server, or other media, so the search in pre-defined paths approach will not be consistent with the Macintosh interface.

The new implementation used the Macintosh based file system references for specifying files. The user is given the chance to select and open, or save into any file in volume or server. Of course the Magic sources required a different file specification to be running and to keep the compatibility.

As discussed above, Magic makes a search in a pre-defined range of paths to access a file and this search begins from the current directory, i.e. the path of the running software. The solution to the file access problem was to change the current directory before passing the file to Magic for access. Once the user specifies a file in some other directory and this file must be accessed from within the existing Magic sources, the current directory was changed to show the directory of the file selected by the user and only the file name was passed as the file reference. This way when Magic starts the search of the file specified by the name from the current directory, the file will be accessed right there. So using only names for file reference and changing the current directory, access to any file on any volume became possible, which would be an obvious requirement for any Macintosh program.

3.4.2. Coordinate Systems

Every drawing on the computer screen is based on a coordinate system. Generally windowing systems provide a coordinate system with respect to the window that the drawing is concerned. In this way the drawing performed on the window will be independent of the location of the window. So the programmer won't be concerned with the location of the window when he needs to draw something on the screen.

The coordinate system used in MacOS and the coordinate system used in Magic do not use the same origin, even though both take the approach discussed above. MacOS takes the top left point of the window as the origin for drawing into the window. Magic, takes the lower left point of the window as the origin. So, a point in Magic is denoted by its two coordinates, x and y for instance, which define the distance of the point from the lower left point of the window in horizontal and vertical coordinates respectively. The MacOS takes the same approach but this time using the top left corner.

This difference of the window origin between the two environments arouse the need for a coordinate transformation between the two coordinate systems. A coordinate transformation was implemented with its appropriate structures, that transform Magic objects to MacOS objects and vice versa. So as the program runs once a structure should be used in Magic, the structure is transformed into its Magic form and used. If the structure

should be used to display data on the screen, as the display parts fully take advantage of the MacOS, the structure is transformed to its MacOS counterpart with an inverse transform.

Use of this coordinate transform approach made it possible to use the existing structures and files of the Magic system on UNIX and run the implementation on the Macintosh in exactly the same way with the same data as its UNIX counterpart.

4. RESULTS

This chapter explains the basic user interface improvements in the new implementation with the performance discussions of the resulting software.

4.1. Magic; the Macintosh Implementation

4.1.1. User Interface

The design for the new implementation was, since the start aimed a better user interface than its UNIX counterpart. Use of palettes and menu based interface on the Macintosh made it possible to come up with a simple drawing program, though further study will still lead to a better user interface. Obviously further program use and designer feedback is essential for such a user-interface intense subject.

Color Plate 3 gives a screen shot during a simple layout design which pictures the new user interface. As seen in the plates once the application launches, the user is presented with a group of palettes and a *terminal window.*

The terminal window is basically used for long text commands. Currently long text commands are still essential for the use of Magic because current implementation focuses on drawing interface and further usage, like design rule checking, and additional operations are supplied through text commands. The terminal window also displays error messages related to miss-typed or invalid commands, and presents information to the user. Also when the user uses other interface elements like palettes or menus, the Magic command counterparts (previously discussed fake commands) of the interface elements, will optionally be presented to the user. This approach gives the user additional information when using Magic. For instance once the user clicks the colors palette to select some layer and drags the mouse to paint a layer, its corresponding "*paint <layer>*" Magic command will be displayed in the terminal window.

Palettes are the basic improvement of the user interface in the new implementation, we have three of them;

Tools Palette

This palette is used to select the behavior of the mouse operations on the layout, i.e. is similar to the *tool* command in Magic. The first tool is the selection tool which lets the user manipulate existing paint and layers. The selection notion is a bit different in the Macintosh implementation when compared with the existing version. The user selects paint

with a single click which will select all of the paint connected to the click point. Thus the implementation takes an object based approach to the design. The user can move, resize and delete existing paint layers using this tool. The second tool is used for painting only. Once the user drags the mouse on the design window, the drag area will be painted with the layer selected in the technology palette. Rest of the tools are designed for netlist and wiring operations and will be used for advanced Magic user interface.

Technology Palette

This palette offers the user a palette of layers available for painting. This palette is dependent on the technology and will change if the technology changes. This palette gives the user a chance to see all layers available in current technology. The selection on the palette shows what the new material painted on the layout will be.

Geometry Palette

This palette is used to supply the user with a ruler information so that he can adjust the sizes of the paint as the drawing is concerned and minimize the errors to deal with. The size of every selected paint object will be displayed in this palette during the design. As the user drags the mouse to create a new paint, current size of the paint will be displayed in this palette as the dragging continues.

4.1.2. Performance

The new implementation of Magic has proven to run at acceptable speeds on the 68K family and was as fast as its UNIX counterpart on the PowerPC. Though accurate performance measurements were not carried out for they are out of the scope of this study, the program, running at a 25Mhz 68030 machine is well fast enough to carry out a complex layout. Color Plates 3 and 4 show screen shots of the program in layouts of different complexity.

Performance improvements might be important, for reduction of the application launch time, which might take up to five to ten minutes on 68K based machines. This delay originates from the design rule parts of the technology parsing routines where the technology files are, in a way, compiled and read into memory for the design rule checker. This process, requiring too much of parsing and small memory allocations, were found to be rather inefficient on 68K processors. PowerPC processors do not cause any similar delays, which helps reveal the fact that the delay is associated with the memory allocation, and fragmentation in memory of 68K based machines. One solution to this problem might be to compile the technology file once, and save the compiled output to a file, as technology

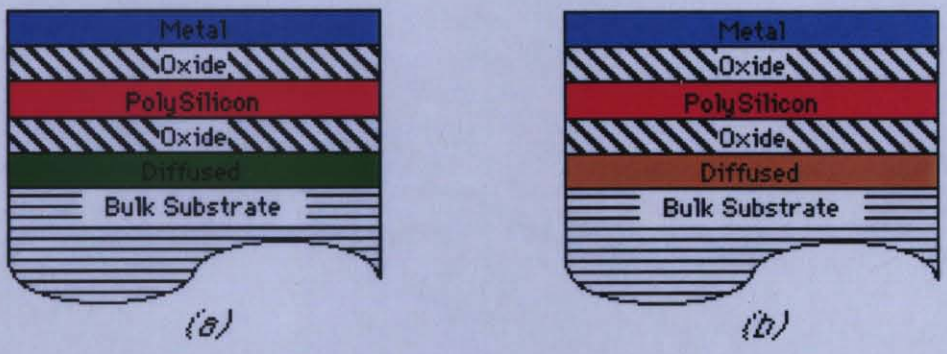
files will not be changing very often and this delay is not problematic for the rare technology changes. Another solution, might be to implement a memory allocator, specific to the parsing process, which allocates one large block of memory for the use of the technology parser. This would result in a much improved technology parser, for this will reduce the memory allocation delay to a minimum.

5. CONCLUSIONS

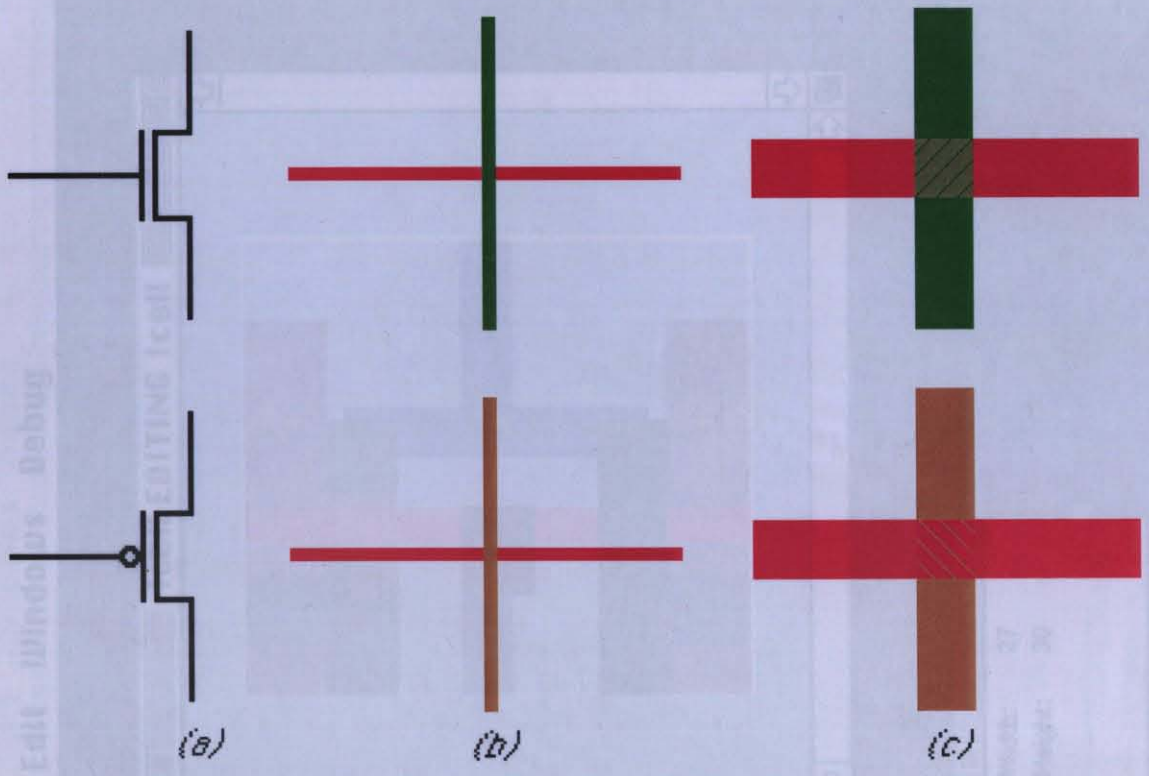
The study achieved its initial goal of making the Magic system run on a desktop computer with moderate hardware requirements and a simpler user interface though as in any software project has not attained complete maturity in the initial release. There is still work to be done both in software development effort and product development effort as further testing and documentation.

This study has formed a basis of porting the Magic system to a new environment and renew the user interface and a framework has been created to implement any Magic command with a new user interface. Thus in the current version of the implementation, the interface is not completely graphical. User still has to use the terminal window with a command line interface for advanced use of the Magic system i.e. some of the Magic commands. All of the commands might be supplied with a new graphical interface through the use of the implemented command framework. Besides, existing interface should be improved with feedback from the users.

Performance improvement for the launch time might also be considered taking the facts mentioned in the previous chapter into account.

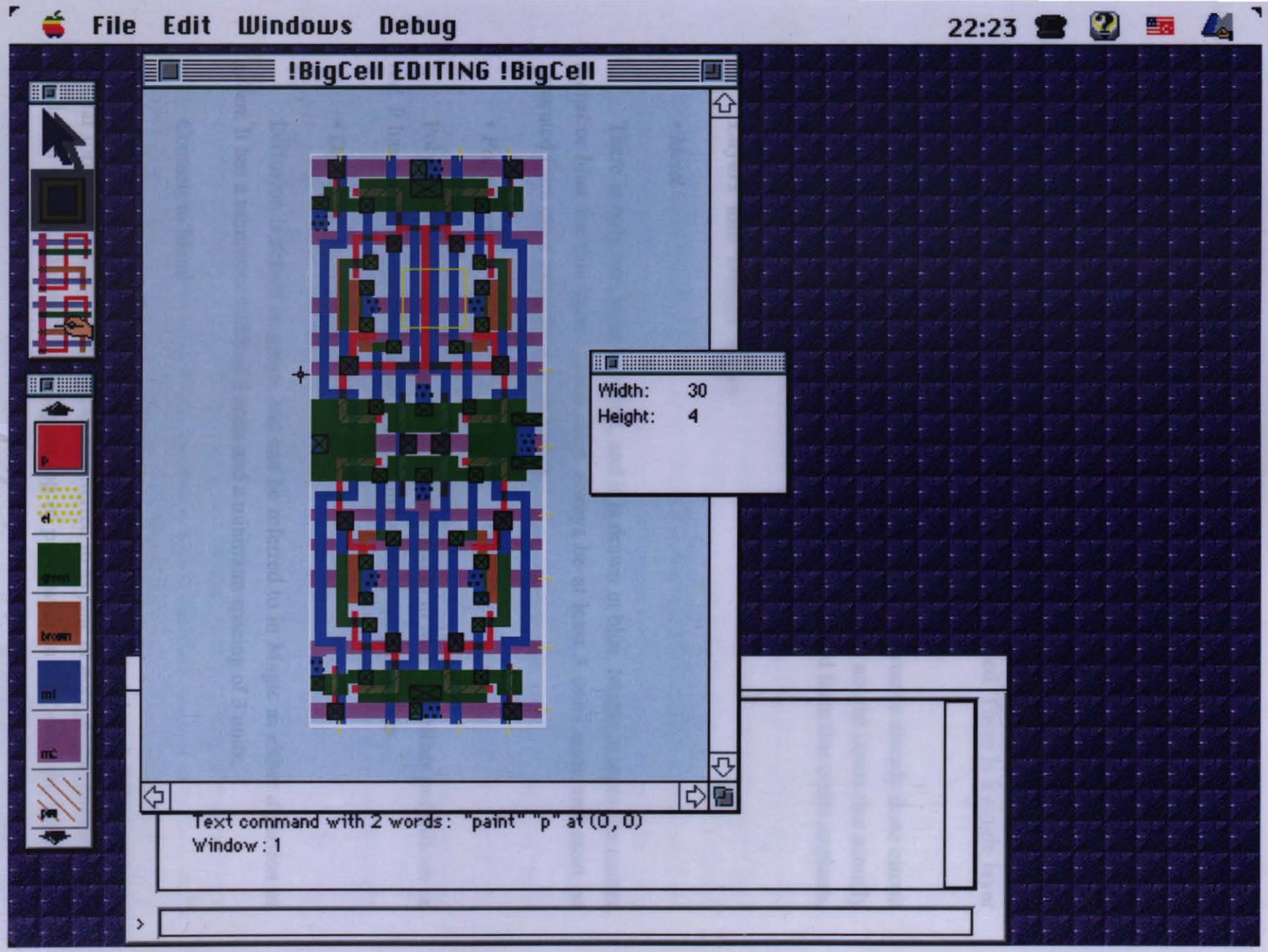


Color Plate n-channel (a) & p-channel (b) transistor structures.



Color Plate Circuit symbol (a), stick representation (b), and layout repres
for an NMOS and PMOS FET.

Color Plate The new interface elements of Magic, the palettes.



Color Plate 4 Editing of a circuit layout in Magic, on the Macintosh.

APPENDIX A

This appendix describes Magic's NMOS technology. It includes information about the layers, design rules, routing, CIF generation, and extraction. This technology is available by the name NMOS.

The design rules described here are for the standard Mead and Conway NMOS process with butting contacts omitted and buried contacts added. There is a single layer each of metal and polysilicon.

If you've been reading the Mead and Conway text, or if you've already done circuit layout with a different editing system, don't forget that these are not the layers that actually end up on masks. Contacts and transistors are drawn in a stylized form that omits implants, vias, and buried windows.

Layers and Design Rules

• *Metal*

There is only one layer of metal, and it is drawn in blue. Magic accepts the names *metal* or *blue* for this layer. Metal must always be at least 3 units wide and must be separated from other metal by at least 3 units.

• *Polysilicon*

Polysilicon is drawn in red, and can be referred to in Magic as either *polysilicon* or *red*. It has a minimum width of 2 units and a minimum spacing of 2 units.

• *Diffusion*

Diffusion is drawn in green, and can be referred to in Magic as either *diffusion* or *green*. It has a minimum width of 2 units and a minimum spacing of 3 units.

Contacts to Metal

Contacts between metal and polysilicon, and between metal and diffusion, have similar forms. Poly-metal contacts can be referred to as *pmc* or *poly_metal_contact*; they are drawn to look like metal running on top of poly, with an ``X'' over the area of the contact. Diffusion-metal contacts are similar, except that they look like metal running on top of diffusion, and have names *dmc* and *diff_metal_contact*. Contacts are drawn differently in Magic than they will appear in the CIF: you do *not* draw the via hole. Instead, you draw

the outer area of the metal pad around the contact, which must be at least 4 units on each side. Magic will fill in the appropriate via when CIF is generated. If you draw contacts larger than 4 units on a side, Magic will fill in as many 2-by-2 CIF via holes (with 2-unit spacings) as it can. Contacts areas must be rectangular in shape: contacts of the same type may not abut.

An additional kind of contact, called *glass_contact*, is used to generate holes in the overglass layer for use in bonding to pads. This layer is drawn as gray stripes over blue, and includes both metal and the overglass hole.

Transistors

There are three transistor structures in the NMOS technology. Enhancement transistors are known by the names *efet* and *enhancement_fet*, and are drawn to look like red over green, with green stripes. You get efet automatically when you paint poly over diffusion or vice versa. Depletion transistors are known by the names *dfet* and depletion_fet, and are drawn the same way, except with yellow stripes. A third type of material is called *depletion_capacitor* or *dcap*. It is displayed with yellow crosses over the transistor area, and is identical to dfet except that there are no overhang design rules for it since it is assumed to be used only as a capacitor. You do not drawn any implants in Magic, but just use a different material for the transistor. Magic will generate the implants automatically. All transistors must be at least 2 units on each side, and there must be a poly or diffusion overhang for 2 units on each

side of efet or dfet (this is not required for dcap). Poly must be separated from diffusion by at least one unit except where it is forming a transistor. Dfet and dcap must be at least 3 units from efet in order to keep the implant from contaminating the enhancement transistor.

• *Buried Contacts*

Buried contacts go by the names *bc* and *buried_contact*. They are drawn in a brownish color (the same as transistors), except with solid black squares over their area. As with other contacts, you draw just the area where the two connecting materials (poly and diffusion) overlap; Magic will generate the CIF buried window, which is actually larger than the overlap area. Buried contacts come in two forms. The normal form is 2 units on a side, and no poly or diffusion overhang is required. The second form is used only next to depletion transistors, and is a 3-by-2 structure abutting the depletion transistor. This form is a little controversial, since it results in larger-than-normal variations in the size of the depletion transistor. As a consequence, Magic reports design-rule violations wherever

buried contacts abut depletion transistors less than 4 units long. In butting bc-dfet structure, you should measure the transistor length from the bc-dfet boundary.

WARNING: There is one additional rule for buried contacts that is NOT enforced by Magic. Where diffusion enters a buried contact, there must be no unrelated polysilicon for 3 units on that side of the buried contact. This rule is necessary because the buried window extends outward from the buried contact by one unit on the diffusion side, and polysilicon must be far enough away to avoid shorting to the diffusion through the buried window. Unfortunately, there is no way to check this rule in Magic without being extremely conservative (the rule would have to require no poly whatsoever on the diffusion side, even if the poly was connected to the buried contact). So, for now, this rule is not checked.

• *Transistor Spacings*

Transistors must be spaced at least 1 unit from any contact to metal, in order to keep the contact from shorting the transistor. In addition, buried contacts must be at least 4 units from enhancement transistors in the diffusion direction. This rule applies only to the side of buried contact where diffusion leaves the contact.

• *Hierarchical Constraints*

The design-rule checker enforces several constraints on how subcells may overlap. The general rule is that overlaps may be used to connect portions of cells, but the overlaps must not change the structure of the circuit. Thus, for example, it is acceptable for poly in one cell to overlap poly-metal contact in another cell, but it is not acceptable for poly in one cell to overlap diffusion in another (thereby forming a transistor).

For contacts, there are additional restrictions. A contact in one cell may not overlap a contact in any other cell unless the two contacts have same type and they occupy exactly the same area. Partial overlaps are not permitted, nor are abutting contacts of the same type (contacts of different types may abut, as long as the abutment doesn't violate any other design rules). The contact restrictions are necessary to guarantee that CIF can be generated correctly in a hierarchical fashion.

• *Routing*

If you use Magic's automatic routing tools on an NMOS design, the routing will be run in metal and polysilicon, with metal as the primary layer. The routing will be placed on a 7-unit grid.

• *Reading and Writing CIF*

There is only one CIF output style available in the NMOS technology: *lambda=2*. The CIF layers in this style, and their meanings, are:

Name	Meaning
–	
NP	polysilicon
ND	diffusion
NM	metal
NI	depletion implant: generated around depletion
\^	transistors and depletion contacts
NC	contact via: generated as small squares inside
\^	poly-metal contacts and diffusion-metal contacts
NB	buried window: generated around buried contacts
NG	overglass via: generated for overglass contacts

To see exactly where each CIF layer is generated for a particular design, use the *:cif see* command.

There is also just one CIF input style. It is called *lambda=2* and can be used to read files written by Magic in the *lambda=2* style, or files written by Caesar using the standard NMOS technology with a scale factor of 200.

• *Extraction*

Transistors of type *efet* or *dfet* in the NMOS technology must have at least two diffusion terminals. A diffusion terminal is a contiguous region along the perimeter of the transistor channel that connects to diffusion.

A transistor may have more than two diffusion terminals, in which case it is modeled as a collection of two-terminal transistors. If only one diffusion terminal is present, the the extractor flags this as an error and outputs a transistor with the source and drain shorted together.

Transistors of the special type *dcap* may have as few as one diffusion terminal. Although their normal use is as capacitors, the extractor will output them as though they were a *dfet*. It is up to simulation programs to compute the capacitance of a dcap from the area and perimeter of its channel.

The NMOS technology file currently contains little information on

parasitic coupling capacitances. As a result, overlap capacitance, and sidewall overlap capacitance will always be zero.

APPENDIX B

This is the technology manual for MOSIS' Scalable CMOS provided by MOSIS on September 5, 1990. This manual corresponds to Revision 6 of the MOSIS scalable rules with low noise analog layers.

- ***Layers and Design Rules***

Again, layers you draw in Magic are ``abstract layers" or ``logs", and do not correspond exactly to the mask patterns be used to fabricate your circuit. As a result, following rules may have different numbers when compared with MOSIS CMOS Scalable Rules which is specified for CIF layers.

- *Second-level Metal*

The top level of metal is drawn in a purple color, and has the names metal2 or m2 or purple. It must always be at least 3 units wide, and metal2 areas must be separated from each other by at least 4 units.

- *First-level Metal*

The lower level of metal is drawn in blue and has the names metal1 or m1 or blue. It has a minimum width of 3 units and a minimum spacing of 3 units.

- *Polysilicon*

Polysilicon is drawn in red, and can be referred to in Magic as either polysilicon or red or poly or p. It has a minimum width of 2 units and a minimum spacing of 2 units.

- *Electrode*

Electrode (or Second Poly) is drawn in yellow cross patterns, and can be referred to in Magic as either poly2 or electrode or el or p2. It has a minimum width of 2 units and a minimum spacing of 3 units.

- *Diffusion*

In the SCMOS technology, it is unnecessary (however it is recommended) for you to specify wells and implant selection layers explicitly. Instead, there are four different layers that correspond to the two kinds of diffusion in the two kinds of wells. Based on

these four layers, Magic automatically generates the masks for active, wells, and implant select.

The most common kinds of diffusion are p-diffusion in an n-well (n-substrate) and n-diffusion in a p-well (p-substrate); they are used for creating p-type and n-type transistors, respectively. P-diffusion in an n-well is drawn with a light brown color; Magic accepts the names pdiffusion, pdiff, and brown for this layer. N-diffusion in a p-well is drawn in green, and can be referred to as ndiffusion, ndiff, or green.

The other two kinds of diffusion are used for generating well (substrate) contacts and guard rings; they consist of a strongly implanted diffusion area in a well of the same type. P-diffusion in a p-well is drawn with a light brown color and has stippled holes in it. It goes by the names psubstratepdiff, psd, ppdiff, ppd, or pohmic. N-diffusion in an n-well is drawn in a light green color with stippled holes in it, and goes by the names nsubstratendiff, nsd, nndiff, nnd or nohmic.

The basic design rules for the first two kinds of diffusion are the same: they must be at least 3 units wide and have a spacing (to the same kind of diffusion) of at least 3 units. The second kinds of diffusion used as substrate (well) contacts must be at least 4 units wide and have a spacing (to the same kind of diffusion) of at least 3 units. Spacing rules between diffusions of different types are discussed below.

• *Metal 2 Contacts*

All contacts involve the metal1 layer. In Magic, contacts aren't drawn as two areas of overlapping material with a via hole in the middle. Instead, you just draw a single large area of a special contact type, m2contact in the case of metal2 contacts. This corresponds to the area where the two wiring layers overlap (metal and metal2 in the case of metal2 contacts). Magic will automatically output metal1, metal2 when it generates CIF or Calma output and will also generate the small via hole in the center of the contact area. For large contact areas Magic will automatically generate many small via holes in CIF or Calma output. All contacts must be rectangular: two contacts of the same type may not abut. Contacts from metal1 to metal2 are called m2contact, m2cut, m2c, via, or v. They appear on the screen as an area of metal1 overlapping an area of metal2, with a black waffle pattern over the contact area. Metal2 contacts must be at least 4 units wide. It is recommended that metal2 contacts be sized in the following quantized values : 4, 9, 14, 19, 24 ... (or 1+1+2+5k where k=0, 1, 2, ,3 4, ...). This is to assure that metal2 contacts are always on the lambda grid.

There is an additional special rule for metal2 contacts: there must not be any polysilicon or diffusion edges underneath the area of the contact or within 1 unit of the

contact. This rule is present because it is hard to fabricate a metal2 contact over the sharp rise of a poly or diffusion edge. It is acceptable for poly or diffusion to lie under a m2contact area, as long as it completely covers the area of the m2contact with an additional 1-unit surround.

• *Polysilicon, Electrode and Diffusion Contacts*

Contacts between metal1 and polysilicon go by the names polycontact, pcontact, polycut, and pc. As with all contacts, you only draw the outer boundary of the overlapping area of poly and metal1; Magic fills in the contact cut(s) at mask generation time. Pcontact areas must be at least 4 units wide and must be separated from unrelated polysilicon and pcontact by at least 3 units. This is one unit more than the normal poly-poly separation, and is required because MOSIS bloats the polysilicon around pcontacts. Pcontacts must be 1 away from diffusions.

Contacts between metal1 and electrode (or poly2) go by the names electrodecontact, econtact, ec, poly2contact and p2c. As with all contacts, you only draw the outer boundary of the overlapping area of electrode and metal1; Magic fills in the contact cut(s) at mask generation time. Econtact areas must be at least 4 units wide and must be overlapped by 1 lambda with the layer electrode. This guarantees the total overlap of electrode over the real cut to be at least 2 lambda. Therefore there is no spacing rule governs the econtact layers. Econtacts must be at least 3 units away from polycontacts even if they are connected electrically by metal1. There is another type of electrode contact to metal1 layer, capcontact. This layer has different design rules. It is used in making electrode contacts for poly/electrode capacitors. Its properties will be describe in more detail in section 2.9.

In digital design there are four kinds of contacts between metal1 and diffusion, one for each of the kinds of diffusion. There are other diffusion contacts used only to build analog (NPN) devices. These layers will be detailed in section 2.13. Contacts between metal1 and ndiffusion are called ndcontact, ndiffcut or ndc. Contacts betwen metal1 and pdiffusion are called pdcontact, pdiffcut or pdc. Contacts between metal1 and nsubstratendiff are called nsubstratencontact or nncontact or nsc or nnc or nohmic. Lastly, contacts between metal1 and psubstratepdiff are called psubstratepcontact or ppcontact or psc or ppc or pohmic. All diffusion contacts must be at least 4 units wide, and must be separated from unrelated diffusion by 4 units, one unit more than the normal diffusion-diffusion separation. It is recommended that contacts to poly or diffusion be sized by multiple of 4. It is to warrant the cuts to be on the lambda grid.

Both poly, electrode and diffusion contacts appear on the screen as an overlap between the two constituent layers, with a cross over the contact area. All contacts must be rectangular in shape. Pdc may abut nsc and and ndc may abut psc; all other contact abutments are illegal. Pcontact must be at least 2 units from any diffusion contact, even if the two contacts are electrically connected as on the top/right of the figure. Electrodes must be at least 2 units away from any diffusion contact. Econtacts, with the 1 lambda overlap of electrode, will be at least 3 lambda away from and diffusion contacts, as a result. See Section 2.10 for more rules on substrate contacts.

• *Transistors*

P-type transistors are drawn as an area of poly over-lapping pdiffusion, with brown stripes in the transistor area. Magic accepts the names pfet or ptransistor for this layer. N-type transistors are drawn as an area of poly overlapping ndiffusion, with green stripes in the transistor area. The names nfet, and ntransistor may be used. Transistors of each type can be generated by painting polysilicon and diffusion on top of each other, or by painting the transistor layer explicitly. The design rules are the same for both types of transistor: transistors must be at least 2 units long and 3 units wide. Polysilicon used as gate must overhung transistor by at least 2 units. Diffusion (ndiff or pdiff) must overhung transistor by at least 3 units. They must be separated from nearby poly contacts by at least 1 unit. Polysilicon must be at least 1 unit away from diffusion, except where it is forming a transistor. Transistors must be at least 2 units away from each other.

Electrode layer can also be used to create transistors by over-lapping electrode over pdiffusion or ndiffusion. Magic accepts the names epfet and eptransistors or enfet and entransistor respectively. Transistors of each type can be generated by painting polysilicon and diffusion on top of each other, or by painting the transistor layer explicitly. The design rules are the same for both types of transistor: transistors must be at least 2 units long and 3 units wide. Electrode used as gate must overhung transistor by at least 2 units. Diffusion (ndiff or pdiff) must overhung transistor by at least 3 units. They must be separated from nearby electrodes/econtacts by at least 1 unit. Eelctrode must be at least 1 unit away from diffusion, except where it is forming a transistor. Transistors must be at least 3 units away from each other.

With two layers of polysilicon, you may overlap them with diffusion to get floating gate devices and buried channeled CCD's. You may also use the two polysilicon to make capacitors. Rules on buried CCD and capacitors will be discussed in the following sections. In making floating gate devices, Magic accepts the names doubleptransistor, pfloating-gate, pfloatg, pffet, or pfg for P-type floating gate devices. It accepts the names

doublentransistor, nfloating-gate, nfloatg, nffet, or nfg for N-type floating gate devices. Floating gate devices must be at least 2 unit in length and 3 unit wide.Since Magic has no way of knowing the overlapping two polysilicon layers are used for floating gates, it will flag rule violations. It basically treat the overlap of two polysilicon as capacitors and insists that poly must overlap electrode by 2. In general, if you are doing floating gate devices you should ignore these design rule violations.

• *Polysilicon/Electrode Capacitors*

Over-lapping polysilicon with electrode will give you precision capacitors for analog designs. Since regular electrode layers must be 2 units away from polysilicon a special magic layer is created for capacitor. It accepts the names capacitor, poly cap, pcap or just cap. In making a capacitor, the electrode layer must be overlapped by 2 units of poly. The electrode layers must be 2 units away from wells and polycontacts. When making a electrode contact on a capacitor the electrode must overlap the composite contact layer, capcaontact, by 2. This capcontact layer can also be called ccontact or capc or cc. Magic will try to extract capacitor as a two terminal fet. (At MOSIS we have made some modification to ext2sim and sim2spice which allows the automatic extraction of capacitors. You may obtain these by sending an ATTN message to MOSIS.) It is recommended that you make the capacitor layer larger enough to compensate fabrication variations.

• *Substrate Contacts*

There are several additional rules besides those in Section 2.6 that apply to substrate contacts. Substrate contacts are used to maintain proper substrate voltages and prevent latchup. Nncontact is used to supply a Vdd voltage level to the n-wells (or n-substrate) around p-transistors, and ppcontact is used to supply a GND voltage level to the p-wells (or p-substrate) around n-transistors. Nncontact must be separated from p-transistors by at least 3 units to ensure that the n+ implant doesn't affect the transistor. Nncontact may be placed next to pdcontact in order to tie a transistor terminal to Vdd at the same time. Because of diode formation, nncontact will not connect to adjoining pdiffusion unless there are contacts to metal1 to strap them together. Similar rules apply to ppcontact.

There is a special technology file written by prof. Fred Rosenberger of Washington University at Saint Louis which will check if you have placed enough substrate contacts in a well. This technology file and accompany program is distributed by MOSIS User Group (MUG). Please contact prof. Don Bouldin at University of Tennesse to obtain a copy of the program and technology file. It is recommended that you place plenty of well and substrate contacts in your layout, however. If you place too few, you risk latch-up in your circuit, so

a good rule of thumb is to place one nncontact for each ptransistor that has its source tied to Vdd, and one pwcontact for each ntransistor that has its drain tied to GND. To avoid having wells floating (especially auto-generated wells), it is recommended that you put a well (substrate) contacts for every cluster of transistors.

• *Spacings between P and N*

Ndiffusion, ntransistor, ndcontact, and ppcontact must be kept far away from pdiffusion, ptransistor, pdcontact, and nncontact, in order to leave room for wells. Ndiffusion and pdiffusion must be 10 units apart. Substrate contacts can be 2 units closer to material of the opposite type (the wells needn't surround them by as many units), so nncontact need only be 8 units from ndiffusion, ppcontact need only be8 units from pdiffusion, and nncontact and ppcontact need only be 6 units apart.

• *Wells and Rings*

For the most part, you should not need to draw explicit wells. However it is recommended that you use painting explicit well as a mean to plan for your layout topology. When writing out CIF or Calma files, Magic generates them automatically. Nwell is generated around pdiffusion, ptransistor, pdcontact, and nncontact. Pwell is generated around ndiffusion, ntransistor, ndcontact, and ppcontact. Magic merges nearby well areas into single large wells when possible. For example, two ndiffusion or pdiffusion areas will share a single well if they are within 10 units of each other. If you're curious about exactly what Magic will use as the well areas, you can use the Magic command :cif see along with the CIF layer names described in Section 5. There may be a few cases where you'd like to guarantee that certain areas are covered with wells, e.g. pads. For these cases you may paint the explicit layers nwell and pwell. Nwell appears on the screen as diagonal green stripes, and pwell appears as diagonal brown stripes. The explicit well layers that you paint will supplement the automatically-generated wells. Pwell must be at least five units from pdiffusion, ptransistor, or pdcontact, and three units from nncontact. Nwell must be at least five units from ndiffusion, ntransistor, or ndcontact, and three units from ppcontact. Nwell and pwell may abut but not overlap (if you paint one well on top of the other, the new one replaces the old one). All nwell and pwell areas that you paint must be at least 10 units wide and are separated from other wells of the same type by at least nine units.

Guard rings may be created using the nndiffusion and ppdiffusion layers. Nncontacts and ppcontacts should be used to strap the rings to Vdd and GND, respectively. Nndiffusion must be at least 3 units from pwell, 6 units from ppdifffusion or ppcontact,

and 8 units from ndiffusion or ndcontact (these are the same rules as for nncontact). Similar rules apply to ppdiffusion. Guard rings are probably the only things that nndiffusion and ppdiffusion will be used for.

• *NPN Transistors*

An extra p- base diffusion layer is allowed on selected MOSIS runs. This layers allows the design of (1) in well isolated NPN bipolar transistor; (2) higher voltage P-channel MOSFETs; (3) in well P and N-channel junction FETs; (4) in well higher threshold nMOS devices; (5) in well isolated photoreceptors; (6) in well metal semiconductor Schottky diodes (assuming nwell process).

There are two different styles in designing the NPN bipolar transistors - SCEA and SCGA. Magic accepts the SCEA style of designs. In SCEA style, the drawn PBASE (CBA) layer (eg. P- in NWELL) is bloated by 2 lambda per side to give the PBASE IMPLANT mask by MOSIS. Moreover, the PBASE (CBA) layer is orred with the drawn ACTIVE layer (CAA) and then biased by an appropriate amount to become the ACTIVE mask. Please remember that the PBASE (CBA), therefore, is a composite layer.

The base layer in Magic used to build NPN bipolar transistors is named pbase or pb. Pbase contact to metal1 is named pbasecontact, pbcontact or pbc. The emitter layer is named emitter, emit or em. Its contact to metal1 is emittercontact, emitcontact or emc. The collector layer is named collector, col,co, cl. Its contact layer to metal1 is named collectorcontact, colcontact, colc, coc or clc. Pbc and emc must be at least 7 units away from each other. This is because Magic bloats emc by 2 to get the N+ select layer, and it bloats pbc by 1 to obtain the P+ select layer. Moreover the N+ select and P+ select must be at least 4 units away. The total distance, thus, is 4+2+1=7. Pbc must be overlapped by pbase by 1 unit. Clc must be 5 units away from pbase. Again this is because Magic bloats clc by 1 to get the active layer layer and collector active bloated must be 4 away from pbase. Well (NWELL) do not have to be painted explicitly. However, we strongly recommend you paint your own well for NPN bipolar transistors.

At the present time we have no way to extract NPN bipolar transistor automatically.

• *Buried CCD Layers*

Three extra layers are used in Magic to accept designs for BCCD's. The first layer is named bccdiffusion or bd for short. The second layer is named nbccdusion or nbd for short. The last layer is named nbccdiffcontact or nbdc for short. The nbd and nbdc layers are used to create input/output nodes. The overlapping rules of Nselect and BCCD implant

are guarantee by the CIF generation. However the user has the responsibility to assure that the first and the last gates overlap the input/output nodes (nbd) by 2. There is no Magic rule checking on bd, nbd and nbdc, since they are created on a different plane.

• *Overglass and Pads*

Normally, everything in the layout is covered by overglass in order to protect the circuitry. If you do not wish to have overglass in certain areas of the layout, there are two Magic layers you can use for this. The Magic layer pad should be used for drawing pads. It generates a hole in the overglass covering and also automatically includes metal1, metal2, and via. Pad is displayed as metal2 over metal1, with additional diagonal stripes. The rules for pads are in absolute microns, not lambda units: the pad layer must always be at least 100 microns on a side. Since this rule is in absolute units, it is not checked by the Magic design-rule checker. Pads will generally need to be modified in order to fabricate at different scale factors or for different flavors of CMOS.

An additional layer glass is provided to allow designers to make unusual glass cuts anywhere on the chip for probing or other purposes. This layer is drawn in dark diagonal stripes. Probe areas should generally be at least 75 microns wide, but Magic does not check this rule.

• *Hierarchical Constraints*

The design-rule checker enforces several constraints on how subcells may overlap. The general rule is that overlaps may be used to connect portions of cells, but the overlaps must not change the structure of the circuit. Thus, for example, it is acceptable for poly in one cell to overlap pcontact in another cell, but it is not acceptable for poly in one cell to overlap ndiffusion in another, since that would form an ntransistor.

For contacts, there are additional restrictions. A contact in one cell may not overlap a contact in any other cell unless the two contacts have same type and they occupy exactly the same area. Partial overlaps are not permitted, nor are abutting contacts of the same type (contacts of different types may abut, as long as the abutment doesn't violate any other design rules). The contact restrictions are necessary to guarantee that the CIF via holes can be generated correctly in a hierarchical fashion.

• ***Flavor Changes***

The painting tables have been set up in the SCMOS technology to make it easy for you to turn n-flavored things into p-flavor, and vice versa. If you paint nwell over an area,

any ndiffusion in the area will be turned into pdiffusion, nfet into pfet, ndc into pdc, psd into nsd, and psc into nsc. Similarly, if you paint pwell over an area, any pdiff in the area will be turned into ndiff, pfet into nfet, pdc into ndc, nsd into psd, and nsc into psc. Thus, if you'd like to make a symmetrical copy of something, except for the opposite well, you can copy it, paint the opposite well over it, then erase the well to leave just the basic layers.

- ***Routing in CMOS***

If you use Magic's automatic routing tools on an SCMOS design, the routing will be run in metal1 and metal2. Metal1 is the primary routing layer and will be used wherever possible. In order for Magic to route to terminals, they will have to be on layers that connect to either metal1 or metal2. For example, terminals may be on the pcontact layer (since it connects to metal1) but not on the polysilicon layer. In this technology, the router will use an 8-unit grid.

- ***Reading and Writing CIF and Calma***

The SCMOS technology provides several styles of CIF and Calma output, corresponding to different flavors of CMOS and different scale factors. All possible layers that can be generaetd from any of the output styles are:

CMS (Calma layer number 51) Corresponds to the metal2 and m2c Magic layers.

CMF (Calma layer number 49) Corresponds to the metal1 Magic layer, plus all contacts.

CPG (Calma layer number 46) Corresponds to the polysilicon and pcontact layers.

CAA (Calma layer number 43) This is the active mask. It is generated over the areas of Magic's four diffusion layers, plus transistors and diffusion contacts.

CVA (Calma layer number 50) This layer is generated as one or more small squares in the center of each m2contact.

CCP (Calma layer number 47) Generated as one or more small squares in the center of each pcontact.

CCA (Calma layer number 48) Generated as one or more small squares in the center of each contact to diffusion.

CWP (Calma layer number 41) P-well. This layer is generated automatically by bloating the ndiff, nfet, ndc, psd, and psc layers, and OR-ing in the pwell layer.

CWN (Calma layer number 42) N-well. This layer is generated by bloating the pdiff, pfet, pdc, nsd, and nsc layers, and OR-ing in the nwell layer.

CSP (Calma layer number 44) P-plus implant mask. This layer is generated by bloating the pdiff, pfet, pdc, psd, and psc layers.

CSN (Calma layer number 45) N-plus implant mask. This layer is generated by bloating the ndiff, nfet, ndc, nsd, and nsc layers.

COG (Calma layer number 52) Overglass holes: generated from the pad and overglass layers.

CCE (Calma layer number 55) Electrode (Poly 2) layer contact to first level metal.

CEL (Calma layer number 56) Electrode layer (Second poly layer).

CBA (Calma layer number 57) P-base layer for the NPN transistors.

CCD (Calma layer number 58) Buried CCD implant.

If you're curious to see exactly where these layers get generated for a particular design, read about the :cif see command in the Magic tutorials or man page.

There are currently 20 output styles supported for CIF and Calma. They are:

lambda=1.5(pwell) Generates CIF (or Calma) for the MOSIS SCP technology, which uses p-wells and 3.0-micron feature sizes.

lambda=1.0(pwell) Generates CIF for the MOSIS SCP technology, which uses p-wells and 2.0-micron feature sizes.

lambda=0.8(pwell) Generates CIF for the MOSIS SCP technology, using p-wells and 1.2-micron feature sizes.

lambda=0.6(pwell) Generates CIF for the MOSIS SCP technology, using p-wells and 1.0-micron feature sizes.

lambda=1.5(nwell) Generates CIF for the MOSIS SCN technology, using n-wells and 3.0-micron feature sizes. Both selects layers are presented.

lambda=1.0(nwell) Generates CIF for the MOSIS SCN technology, using n-wells and 2.0-micron feature sizes. Both selects layers are presented.

lambda=0.8(nwell) Generates CIF for the MOSIS SCN technology, using n-wells and 1.2-micron feature sizes. Both selects layers are presented.

lambda=0.6(nwell) Generates CIF for the MOSIS SCN technology, using n-wells and 1.0-micron feature sizes. Both selects layers are presented.

lambda=1.5(oldnwell)

lambda=1.0(oldnwell)

lambda=0.8(oldnwell)

lambda=0.6(oldnwell) These styles are the same as the 4 styles above except they generate only N-select layer not p_select.

lambda=1.5(gen) Generates CIF for the MOSIS SCE technology, which includes both p-wells and n-wells and has 3.0-micron minimum feature sizes.

lambda=1.0(gen) Generates CIF for the MOSIS SCE technology, which includes both p-wells and n-wells and has 2.0-micron minimum feature sizes.

lambda=0.8(gen) Generates CIF for the MOSIS SCE technology, which includes both p-wells and n-wells and has 1.2-micron minimum feature sizes.

lambda=0.6(gen) Generates CIF for the MOSIS SCE technology, which includes both p-wells and n-wells and has 1.0-micron minimum feature sizes.

lambda=1.5(error) Generates only the error layers with lambda=1.5

lambda=1.0(error) Generates only the error layers with lambda=1.0

lambda=0.8(error) Generates only the error layers with lambda=0.8

lambda=0.6(error) Generates only the error layers with lambda=0.6

The default style is ``lambda=1.5(pwell)". Other styles may be selected with the Magic command :cif ostyle.

For reading CIF or Calma (stream) file, there are twenty-two styles. Twenty of them correspond exactly to the styles listed above for output. The remaining two are ``cbpm3u"

and "oldcbpm3u"; they can be used to read in cells designed under the old MOSIS 3-micron rules. Designs converted using style ``cbpm3u" or "oldcbpm3u" will probably have numerous design-rule violations. Whenever CIF is read, explicit wells are generated and left in the Magic files. You'll have to go in by hand and delete them if you don't want them, or modify the technology file locally so it doesn't generate them on read-in.

Be careful to select the correct style when reading CIF: if you use the wrong style you're likely to get many errors in the resulting Magic file, with very little warning from the CIF reader.

- **Extraction**

The SCMOS technology extracts four types of transistor: pfet, nfet, epfet and enet. All of them must have at least two diffusion terminals. A diffusion terminal is a contiguous region along the perimeter of the transistor channel that connects to diffusion, as shown below: At the present time capacitors (poly/poly2) are also extracted as a special type of fet. With modified ext2sim and sim2spice, the special fet will be converted to capacitance. Floating gates can not be extract at this time.

A transistor may have more than two diffusion terminals, in which case it is modeled as a collection of two-terminal transistors. If only one diffusion terminal is present, the the extractor flags this as an error and outputs a transistor with the source and drain shorted together.

At the present time bipolar transistors are extracted into 2 diodes in Magic version 6.0. We need to modify the programs ext2sim and sim2spice in order to convert these diodes into transistors. These modification have not been finish and will be supplied in later time through MOSIS. Magic 4.10 cannot extract bipolar transistors.

REFERENCES

1. Geiger, R.L., Allen P.E., Strader N.R., *VLSI design techniques for analog and digital circuits*, McGraw-Hill Publishing Company, Singapore, 1990

2. Mead C.A., Conway L.A., *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, Reading, Mass., 1980

3. Ousterhout J., Mayo R.N., *Magic Tutorial*, University of California, Berkeley, 1990

REFERENCES NOT CITED

Pucknell D.A., Esraghian K., *Basic VLSI Design*, Prentice-Hall of Australia Pty Ltd. Sydney, 1988

Fabricus Eugene D., *Introduction to VLSI Design*, McGraw-Hill Book Co., Singapore, 1990

Fabricus Eugene D., *Introduction to VLSI Design*, McGraw-Hill Book Co., Singapore, 1990

Programmer's Guide to MacApp, Developer Technical Publications, Apple Computer, Inc. 1992

Ousterhout J., Scott W., *Magic Maintainer's Manual*, University of California, Berkeley, 1990