Clark

MIT/LCS/TR-239

ARTWORK ANALYSIS TOOLS FOR VLSI CIRCUITS

Clark Marshall Baker

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661

This blank page was inserted to preserve pagination.

Artwork Analysis Tools for VLSI Circuits

by

Clark Marshall Baker

Copyright (c) Massachusetts Institute of Technology 1980

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

Artwork Analysis Tools for VLSI Circuits

by

Clark Marshall Baker

Submitted to the Department of Electrical Engineering and Computer Science

on May 18, 1980 in partial fulfillment of the requirements for

the Degrees of Master of Science and Electrical Engineer

Abstract

Current methods of designing VLSI chips do not insure that the chips will perform correctly when manufactured. Because the turnaround time on chip fabrication varies from a few weeks to a few months, a scheme other than "try it and see if it works" is needed. Checking of chips by hand simulation and visual inspection of checkplots will not catch all of the errors. In addition, the number of transistors per chip is likely to increase from ten thousand to over a million in the next few years. This increase in complexity precludes any manual verification methods; some better method is needed.

A series of programs that use the actual mask descriptions for input are described. These programs perform various levels of checks on the masks, yielding files suitable for simulation. Some of the checks are the usual "design rule" checks of looking for minimum line widths and adequate spacing between wires. However, there are many more constraints in VLSI circuits than are expressed by the usual design rules. The programs check these constraints using the mask descriptions as input. All of the errors mentioned so far can be classified as syntactic errors; in addition, certain semantic errors are detected. The detection of semantic errors requires various levels of simulation. The input to the simulators is derived from the artwork.

Name and Title of Thesis Supervisor:

Stephen A. Ward,
Associate Professor of Electrical Engineering and Computer Science

Key Words and Phrases:

VLSI, artwork analysis, circuit extraction, design rule checking

ACKNOWLEDGMENTS

The original inspiration for this thesis came from the Scheme79 chip designed by Jack Holloway, Guy Steele, Gerry Sussman, and Alan Bell in the summer of 1979. Except for examination of the checkplot for errors and a small amount of hand simulation, there was no way of making sure that it was going to work. For the fall term offering of the VLSI Design course, Randy Bryant had created a switch-level simulator for students to test out their designs before implementing them. It occurred to me that it might be possible to extract the logic circuit of the Scheme chip from the actual mask description. This could then be simulated, with the hope of discovering any remaining bugs. While I went to work on the node extractor, Chris Terman implemented a simulator that could handle such a large circuit. In the end, bugs were found in the Scheme chip and fixed, and after the chip was manufactured, it was found to work.

Many good ideas were generated by Chris Terman, Jon Sieber, and Dave Goddeau. In addition, some of the speed improvements came from discussions with Steve Ward and Bert Halstead.

This thesis would not be as readable as it is without the hard work of Debbie Cohn who turned my draft into grammatically correct, smoothly flowing English. Miriam Alexander and Ronni Rosenberg also provided helpful suggestions about grammer and style.

The comments and corrections made by Chris Terman, Carl Hewitt, Steve Ward, and Larry Seiler have improved the content and presentation of this thesis.

No acknowledgments would be complete without a mention of the great working environment provided by the Real Time Systems group of the Laboratory for Computer Science. Both in terms of people and computers, the RTS group provides a place where one can work on interesting problems, from small "hacks" to large programs, with a minimum amount of trouble.

Many designers have provided me with circuit descriptions to be debugged. Without them, I would not have been able to test out my ideas and generate a useful series of programs. It is hard to know if an idea is any good without actually trying it out.

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-75-C-0661.

CONTENTS

1. Introduction 7

2. Why VLSI Circuits Might Not Work 10

2.1 Design rules 10
2.2 Pullup ratios 13
2.3 Two threshold drops 15
2.4 Races 15
2.5 High level design errors 16
2.6 Editing errors 16

3. What Artwork Anaylsis Can Do For Us 18

3.1 Design rule checker 18
3.2 Node extractor 19
3.3 Static evaluator 20
3.4 Dynamic evaluator 21

4. Design Rule Checkers 22

4.1 Raster scan method 23
4.2 Rectangle method 33

5. Node Extractor 35

5.1 Basic algorithm 35
5.2 Ponds and islands 38
5.3 Oh where, oh where, can my transistor be? 41
5.4 Further processing 43
5.5 Extensions 44
5.6 The output format 47

6. Static Evaluator 48

6.1 Reading in the network 49
6.2 Depletion mode transistor checks 50
6.3 "Stuck at" checks 51
6.4 Threshold checks 52
6.5 Ratio checks 53

7. Simulators 55

7.1 Different types of simulators 55
7.2 A possible design of a switch-level simulator 56
7.3 Possible speed improvements 61
7.4 User interface 63

8. How Does All Of This Relate To The Real World? 65

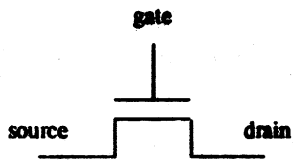
8.1 Design rule checking 66
8.2 Node extraction 67
8.3 Simulation 68
8.4 What happens as chips get even larger 68

9. Conclusions And Directions For Future Research 70

9.1 The Scheme79 chip 70
9.2 Design errors that are not checked 71
9.3 Better design tools 72

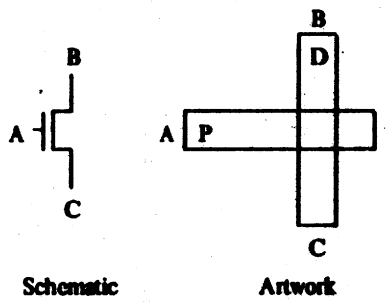
1. Introduction

Since this thesis deals with *Very Large Scale Integrated* (VLSI) circuits, a description of the basics is in order. A more detailed discussion can be found in Mead and Conway.[1] The circuits to be analyzed are composed of interconnected *transistors*.[1] Each transistor can be viewed as a *switch* with three components: *gate*, *source*, and *drain*. When the signal on the gate is high, the source and drain are connected together. When the signal on the gate is low, there is no connection between the source and the drain. For the purposes of this thesis, the terms source and drain are interchangeable.



Enhancement Mode nMOS Field Effect Transistor

The actual circuit is specified by a series of *masks* (or layers). The six masks we will be concerned with are metal, polysilicon, diffusion, contact cut, ion implant, and overglass. The first three are *conductors* and are used for general wiring. However, whenever a polysilicon wire crosses a diffusion wire, a transistor is formed.



Transistor with gate A, source B, and drain C

[1] n-channel, metal-oxide-semiconductor (MOS), field-effect transistor (FET)

The polysilicon wire is the gate and the diffusion on either side forms the source and drain. Metal can run over either polysilicon or diffusion without any connection being made. However, it is sometimes useful to connect metal to polysilicon, or metal to diffusion: the *contact cut* is used in a specific way to accomplish this. In addition, it is possible to connect one end of a polysilicon wire to the end of a diffusion wire through the use of a *butting contact.* The ion implant layer is used to alter the characteristics of transistors. An implanted transistor, also called a *depletion mode* transistor, acts like a resistor. Finally, the overglass layer covers the whole chip with a protective oxide, except where connections must be made to the input and output pads.

One language for specifying integrated circuit masks is the *Caltech Intermediate Form* (CIF). While this language can be read and written by humans, it is expected that in most cases it will be processed solely by computers. CIF supports commands that specify circles, rectangles, and polygons in the various layers. In addition, there is a symbol definition facility in which a collection of geometrical objects to be used repeatedly is placed in a named symbol that can be instantiated many times. Each instance can be reflected, rotated and translated.

The *artwork* for a chip can be created in a variety of ways. Some designers use a *graphics-based* system. Such a system lets a designer manipulate shapes, define symbols, and call symbols, showing what the chip looks like at each step on a graphics display. Another approach to chip design involves writing a *program* that creates the artwork for the chip. This program is often written in a language embedded in a standard programming language, for example LISP. In addition to all the usual language commands, there are commands to manipulate geometrical objects, connect certain points together with wires of a certain type, and so on. A third approach is to design the chip on paper and digitize it into the computer. This method has been and is still very much used in industry. It is hoped that the computer can assist in the design of VLSI chips, but so far

there is no system that comes close to people's expectations or dreams.

The overall idea the reader should have about VLSI design is as follows. Currently, chips are designed by hand, with computer assistance in keeping track of some of the detail. The designer specifies his design in terms of geometrical objects and a certain number of masks. At a higher level of abstraction, the designer is working with transistors and their interconnections. At an even higher level, he may be thinking about logic gates, shift registers, memory cells, programmed logic arrays, and so on. However, he still specifies everything in terms of masks. The design is converted to a standard format such as CIF and sent out to be manufactured. At some later point, a chip is returned to be tested. It has some small number of inputs and outputs through which the designer must interface with the chip. It is not possible for him to look at arbitrary signals within his design, unless he has provided for this beforehand. Typical chips being manufactured today have 10,000 to 100,000 transistors. Future chips will have 10 to 100 times the current number of transistors.

There are many chances for errors to occur in the design of such a large chip. In addition, there are many different types of errors that can occur, any one of which may cause the whole chip to fail, possibly without the designer having any idea why. Some tools are needed that will help designers debug their chips before they are manufactured, so that the chips have a better chance of working when they are actually implemented. After a brief discussion of some types of errors that can occur in the design of VLSI circuits, the rest of this thesis will describe some tools that have been created and used at MIT to aid in chip design.

2. Why VLSI Circuits Might Not Work

There are many reasons why a particular design may not work. These range from very low level problems, such as two signals shorted together because they were too close to each other, to high level "bugs" in algorithms. In another dimension, VLSI circuits may fail due to production problems or bonding errors. However, the latter class of errors is beyond our concern here. We will concern ourselves with errors that can be discovered from the mask descriptions that will be sent for fabrication. Any errors that are introduced after that are someone else's responsibility!

The following list of design errors does not include all possible errors, nor all possible error categories. Mistakes are discovered by studying the design process, observing errors on actual chips, and thinking about various *consistency checks* that might be violated. Some of these errors are specific to a particular process or computer-aided-design system, while others are universal errors that can occur in all designs.

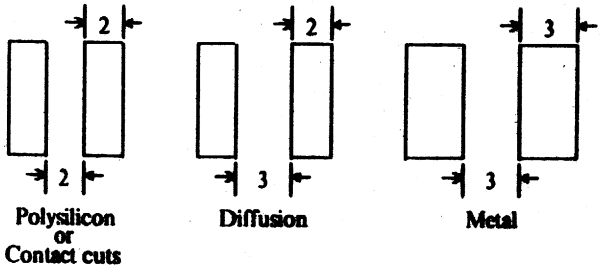
2.1 Design rules

There are many low level rules that define various relationships within and between masks. These rules differ from one *process* to another. Mead and Conway have defined a set of *design rules* that are scalable (within limits) and are based on a unit of length called *lambda*.[1] All their design rules are expressed in terms of lambda and are not tied to a particular process. However, the design rules are very conservatave, and there can be layouts that violate the design rules but still work.

The following is a description of the Mead and Conway design rules: the first set deal with *width* and *spacing* within a specific layer. The width of a diffusion wire cannot be less than two

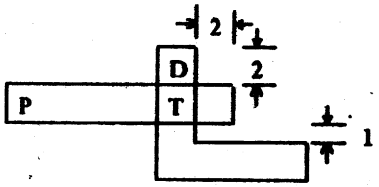
[1] In 1980, lambda was 200-250 centimicrons.

lambda and the distance between two diffusion wires cannot be less than three lambda. The minimum width and spacing for polysilicon and contact cuts is two lambda while the minimum width and spacing for metal is three lambda.



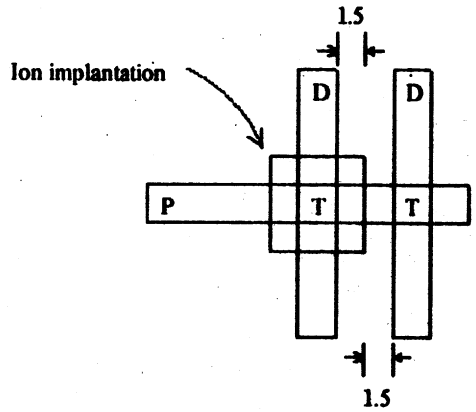
Width and Spacing Design Rules

Next, there are some rules for making *transistors*. Remember that a transistor is formed by the crossing of a diffusion wire (minimum width two lambda) with a polysilicon wire (minimum width two lambda). The minimum distance between a polysilicon wire and a diffusion wire is one lambda. When forming a transistor, both the polysilicon wire and the diffusion wire must overhang the gate area by two lambda.



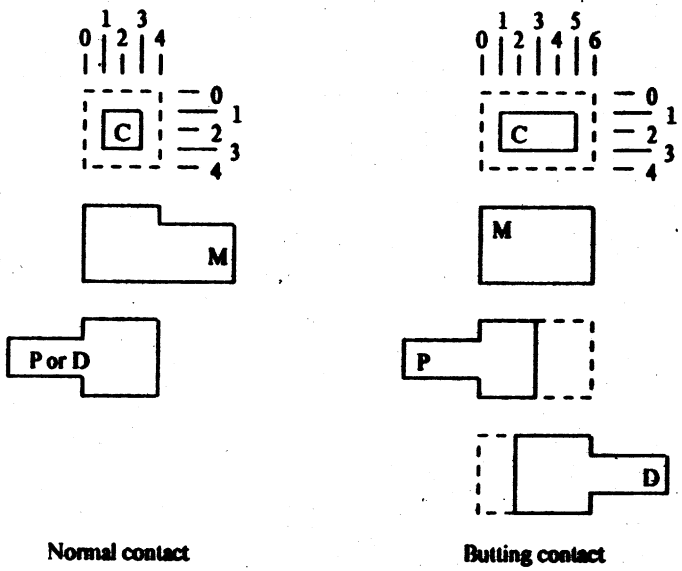
Transistor Design Rules

When making an ion implanted transistor, the ion implant must overhang the gate area by one and a half lambda in all directions. In addition, the ion implant must come no closer than one and a half lambda to a non-implanted transistor.



Ion Implantation Design Rules

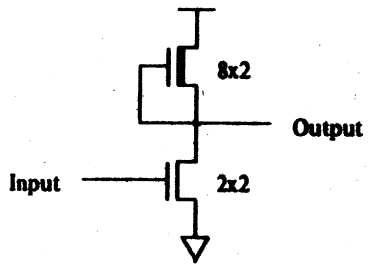
The final set of design rules is for *contact cuts.* There must be a minimum of one lambda overhang of polysilicon, diffusion, or metal around a contact cut. Also, a polysilicon wire must be two lambda away from any contact cut in diffusion. Finally, there is a special method of connecting polysilicon to diffusion called a *butting contact.* The end of the polysilicon wire overlaps the diffusion wire by one lambda. A rectangle of metal (four by six) is placed over the whole constriction, and a two lambda by four lambda contact cut completes the butting contact.



Contact Cut Design Rules

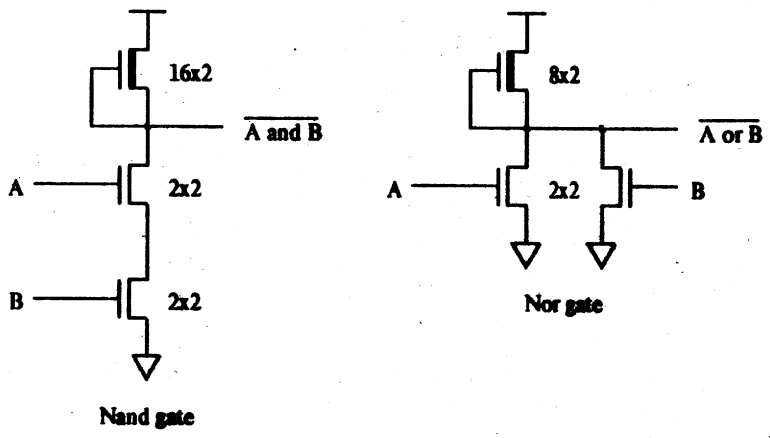
2.2 Pullup ratios

One of the basic building blocks of integrated circuit design is the *inverter.* More complicated versions of the inverter include *nand* and *nor* gates. The basic inverter is composed of two transistors: a *depletion mode pullup* transistor and an *enhancement mode pulldown* transistor. The gate area of each of these transistors has a certain channel length and width. A design rule specifies that under certain conditions (see below) the *ratio* of the length to the width be four. While small deviations from four are allowed, numbers as far off as two or eight represent errors.



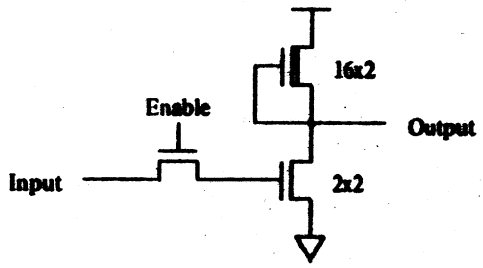
Basic Inverter (Ratio = 4)

This pullup/pulldown ratio rule can be extended to nand gates. In this case, the effective channel length is the sum of the two individual pulldown channel lengths. In reality, the length/width ratio is the same as resistance, and a nand gate contains two resistors in series. However, in computing the correct ratio for a nor gate, it is as if only one pulldown is there; it is not the same as two resistors in parallel.



Simple Gates

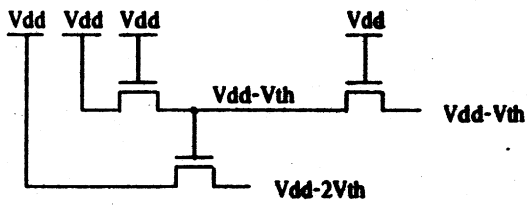
On a simple inverter, the pullup/pulldown ratio should be eight if the pulldown is driven through a *pass transistor*. There is a voltage drop in the signal going through the pass transistor, and so the signal will not turn on the pulldown transistor as much as when it was directly driven. This is compensated for by making the pullup weaker. This can also be generalized to nand and nor gates that have inputs that are driven through pass transistors.



Inverter Driven through a Pass Transistor (Ratio = 8)

2.3 Two threshold drops

The following equation relates the source voltage V_s to the gate voltage V_g and the drain voltage V_d: $V_s = \max(0,\min(V_d,V_s-V_{th}))$. If we put V_{dd} through a pass transistor with a gate of V_{dd}, we will get $V_{dd}-V_{th}$ out. However, if we put $V_{dd}-V_{th}$ into a pass transistor with a gate of V_{dd}, we get $V_{dd}-V_{th}$ out, not $V_{dd}-2V_{th}$. Consider what happens when there is a pass transistor driving another pass transistor. The output of the first pass transistor is $V_{dd}-V_{th}$. The output of the second transistor is $V_{dd}-2V_{th}$. This voltage is too small to be safely used as the input to anything, and represents a design error.



Examples of Threshold Drops

2.4 Races

A *race condition* occurs when the output of a particular piece of logic depends on one of two signals reaching a certain place before the other. A typical example is the output of a carry chain being gated to some further piece of logic by a clock. There is a race between the carry output and the clock transition. The carry output should get there before the clock transition occurs, but the speed of both signals might depend on the number of gate delays involved. Usually, races are avoided through the use of clocks with periods long enough to assure that all signals have been propagated as far as possible.

2.5 High level design errors

The large class of errors that do not relate to the layout, but represent the wrong algorithm implemented correctly is referred to as *high level design errors*. An example of this is a PLA automatically programmed from microcode when there is an error in the microcode. It is felt that these errors should be caught by high level simulation, but sometimes they sneak through and end up in the actual layout.

2.6 Editing errors

Each chip editor seems to have its own peculiar types of errors that it introduces into the design. The following are some common errors that have been discovered on chips generated at MIT and Xerox PARC.

A graphics editor that makes it easy to lay a rectangle in the currently selected layer, and which allows displays at arbitrary scales, can place an unwanted rectangle on the chip. If editing is done at a scale which is large in relation to the size of the rectangle, this rectangle may go unnoticed.

With a graphics editor implemented on a display that does not have a good method for displaying the different layers so that they can be distinguished, it is possible for a rectangle to be drawn in the wrong layer.

Once all the subsections of a chip have been created, they must be wired together. When this interconnect wiring is done using a graphics editor, one is forced either to use a small scale, thereby not getting an overall view, or to use a large scale, causing the current wire to shrink drastically. Either way can lead to errors. Another typical mistake is to wire a trunk of bits from one place to another and get the bits reversed.

In layouts generated by computer programs, there have been various roundoff errors that

have generated gaps in wire runs. Once these programs are debugged, the problem goes away, but it still may happen the first time. Most of these errors show up as design rule violations.

3. What Artwork Anaylsis Can Do For Us

The artwork is a very low level representation of the design. It contains no indication of how the particular design was created, nor the function of the chip. However, it does contain the information necessary to manufacture the chip. In theory, there is enough information contained in the artwork to extract the electrical circuit along with its associated parameters such as resistances and capacitances. This information will help us check for all the errors listed in the previous chapter.

It should be noted that the low level of the information can cause problems. It is hard to relate errors discovered in the artwork to the higher level entity that generated the particular piece of artwork. An error in a replicated section of artwork will be reported many times by the analysis program. The computation time necessary to analyze a whole chip may be much larger than that necessary to analyze each of its components.

In its favor, an analysis of the artwork is an analysis of what is to be manufactured. There are design errors that can show up here that will not exist at higher levels of the design process. If such a whole chip check can be performed in an acceptable amount of time, it will be worthwhile.

The tools listed below perform artwork analysis. Each is described along with the types of errors it can discover.

3.1 Design rule checker

A design rule checker checks most of the geometrical constraints that are imposed by the particular process. The ways in which this might be accomplished will be discussed in the next chapter. Often, the design rule checker is implemented as a *geometry engine* driven by commands that implement the necessary constraints. It will be seen that the checking of design rules is not as straightforward as it might first appear.

There is no formal language for specifying design rules. While English and pictorial descriptions of design rules intuitively make sense, there are many cases that a computer would consider errors but the creator of the rules would consider correct. Once the rules have been specified, the checking can be very time-consuming if performed on the whole chip. Even though it may be very time-consuming, a check should be done on the whole artwork just before it is sent off to be manufactured, if time permits. This check may reveal errors that will not show up when design rule checking is performed on a module at a time.

There are some design systems that make design rule violations much harder to construct. In DAEDALUS [2], the user can specify constraints between pairs of objects. If one object is moved, the other object may possibly have to be moved or adjusted so that all of the constraints are still obeyed. If the user specifies enough constraints, it will be difficult to create designs with design rule violations.

In the CABBAGE system [3] the individual cells are specified in a symbolic description language ("stick diagram"). The CABBAGE system will convert a stick representation to an actual layout, compacting as it goes. The layouts produced by CABBAGE are free from design rule violations.

3.2 Node extractor

Other verification programs need higher level information extracted from the artwork by the *node extractor.* The first piece of information to be extracted is a list of transistors. Each transistor contains the names of three nodes: the gate, the source, and the drain. In addition, there is an indication of the mode of transistor: enhancement or depletion. While extracting this information, there are some *syntactic checks* that can be performed.

A contact cut that contains no metal represents an error. It may be argued that this type of error should be specified in the design rules; nevertheless, the node extractor will catch it too. At some point, the designer may give *symbolic names* to some of the nodes. Names must be given to VDD and GND,[1] are usually given to all of the input and output pads, and are sometimes given to the more important internal nodes (*e.g.* the outputs of a PLA). Given these names, a syntactic check can be made to be sure that no two nodes with different names are shorted together and that all nodes with the same name are really connected together. Sometimes, it is possible to have the design system provide a list of signal names, layers, and coordinates, along with the artwork.

Further information can be extracted from the artwork. The *circuit parameters*, including node capacitances, resistances, and transistor geometries, would be useful. It should be kept in mind that the resulting output is likely to be very large. In addition, some of these parameters are difficult to compute.

3.3 Static evaluator

It might seem that the next logical verification step is simulation of the extracted circuit. However, simulation is very time-consuming and any errors that can be detected before simulation can save a lot of time later. Continuing with the compiler analogy of syntactic and semantic errors, the *static evaluator* will look for *semantic errors*.

Typical errors detected by this evaluator include transistors with gates that are VDD or GND, malformed superbuffers, incorrectly used depletion mode transistors, and transistors which if turned on would short VDD to GND. In addition, a check is made to ensure that every node can

[1]The node in the extracted circuit which will be connected to V_{dd} in the actual chip will be referred to as VDD. The node that will be grounded in the actual chip will be referred to as GND.

potentially be pulled up and pulled down. A check is also made to detect two threshold drops.

3.4 Dynamic evaluator

At some point, there are certain errors that can be detected only through *simulation.* Using circuits derived from the artwork, there are different levels of simulation possible. For checking the actual function computed by the chip, a *switch-level* simulator is needed. For checking the performance of small sections of the design, a *SPICE* [4] type circuit simulator would be best. Such a simulator accepts circuit descriptions that include resistors, capacitors, and transistors, and performs numerical integration to find a solution to the circuit. A third simulator may be necessary for detection of race conditions and for performing gross timing estimates.

There are many simulators in existence.[5,6,7, 8,9] Few of them expect to have input derived from the actual artwork, and most are based on gates instead of transistors. Often they offer facilities for defining large objects such as registers, memory, and PLAs. Most of these simulators are unsuitable, since we need a simulator that can handle the bi-directional nature of pass transistors. Fortunately, it is not too hard to create a simulator that can use the output of the node extractor.

The design of such a simulator is simplified because of the uniform low level input: transistors and nodes -- initially there are no gates, no PLAs, and no registers transistors. Also, there is no hierarchical description of the input. This means that the simulator must simulate each bit of a memory array, each bit of a shift register, and each term in a PLA. This makes it hard to write a simulator that runs fast. Some of the speed problems can be overcome through the use of clever algorithms, and some, through the use of fast computers. If switch level simulations of whole chips are considered important enough, *special purpose hardware* can be created.

4. Design Rule Checkers

One of the first verification tools that a chip designer uses is a design rule checker. Having designed and laid out a chip according to certain geometrical constraints, a designer wants a tool that will check the work. At first, this might seem like a simple though possibly time-consuming task. All one must do is feed the rules into the computer and ask it to look for violations. We shall soon see that it is not that easy.

The reasoning behind the design rules should be kept in mind when thinking about programs that check for violations. One underlying premise is that the various layers, when manufactured, may be misaligned by as much as a lambda. This explains the one lambda overlap required around contact cuts and the one lambda spacing required between polysilicon and diffusion. Diffusion must be spaced greater than polysilicon because the diffusion process is harder to control, possibly resulting in wider diffusion lines than desired. Metal is patterned last, and runs on top of all the other layers. Since they have such a rough terrain to follow, metal wires must be wide and spaced far from other metal wires.

One construction that poses problems for the design rule checkers is the butting contact. It violates many of the design rules but is still considered legal. A butting contact can be viewed as two normal contacts (one from polysilicon to metal, and one from metal to diffusion) placed closer together than is otherwise allowed by the design rules. This is a space saving design "trick" that is known to work. The design rule checker must make sure that butting contacts obey the butting contact design rule and that the rest of the artwork obeys the other design rules.

There seem to be two basic approaches to design rule checking. The first, which was researched for this thesis, is called the *raster scan method* and takes as input a *bitmap* representation of the artwork. The second approach, referred to as the *rectangle method*, deals with the artwork as a

series of rectangles and performs operations on these rectangles. The latter method is the most commonly used one in performing design rule checks. A discussion of the raster scan algorithm for performing design rules checks comes next, followed by a brief discussion of the way a typical rectangle method works.

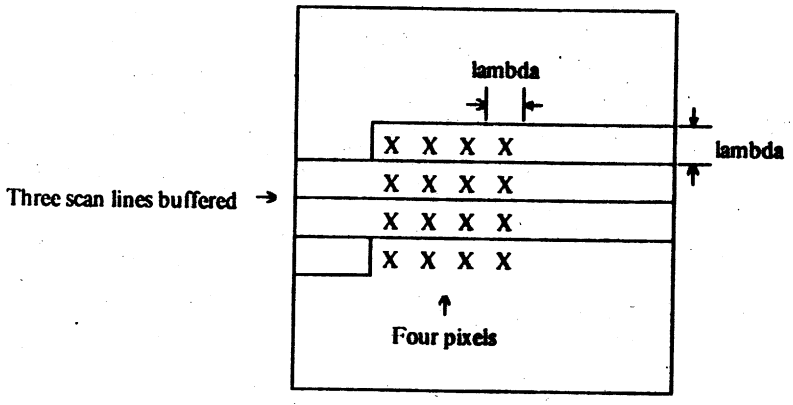
4.1 Raster scan method

The raster scan algorithm is based on the assumption that design rules can be checked locally, and that an examination of a small area of the chip is sufficient to check the design in that small area. A small *window* is passed over the chip, and if all design rules are obeyed in the small window, then the overall chip also obeys all the design rules. The problem of checking design rules overall is therefore reduced to the problem of checking design rules in a small area.

Assume that the artwork can be represented on a lambda grid with each *pixel*[1] containing a bit for each layer. The small window is four lambda square, the smallest size it can be to check that metal lines are at least three lambda wide. The window is moved over the bitmap, such that every pixel appears in every position of the window. This can be accomplished by buffering three scan lines[2] plus four pixels in memory, and reading the bitmap in raster scan order. At each position, the four-by-four square is checked for legality.

[1] A term meaning "picture element", borrowed from computer graphics. We will use this to represent the smallest square unit of resolution.

[2] Really, portions of four scan lines are buffered. The space used is equivalent to three whole scan lines.



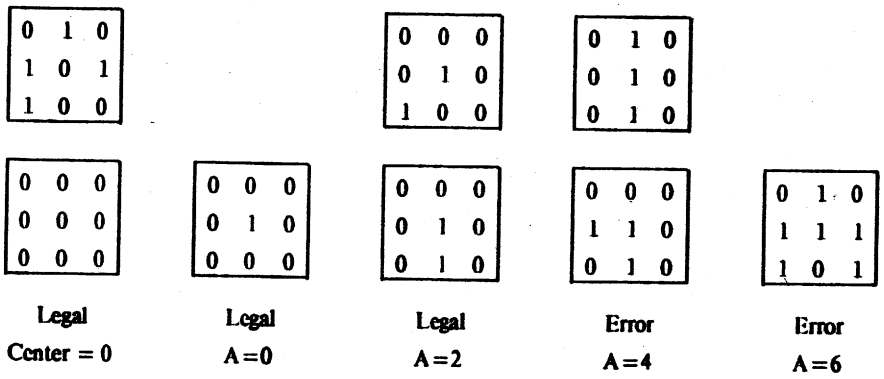
Buffer Three Scan Lines and Four Pixels

The problem has now been reduced to design rule checking of four-by-four squares. The reader should pause and consider the problem; ariving at an acceptable solution required a great deal of effort.

The design rules break down naturally into three types of checks: single-layer width and spacing checks, transistor checks, and contact cut checks. The width and spacing checks will be considered first. The sub-problem to be solved is finding an algorithm that, given a *three-by-three* box consisting of zeros and ones, can check to determine whether any possible larger view containing that box is legal; *i.e.*, whether the ones meet the constraint of being at least *two wide*. Since the total number of three-by-three boxes is only $2^9 = 512$ (including rotations and reflections), they could be enumerated by hand. This would not help us solve the four-by-four case, however. The algorithm finally used will not find *all* the design rule violations, just those that are critical; *i.e.* those wires through which current might actually flow.

For an error to occur, at least a single "1" must be alone. In some three-by-three box, this "1" will be in the center. Scanning around this center "1", an *alternation* (Λ) of zeros and ones will be found. If the perimeter contained "10010101", then there would be three "1" to "0" alternations and three "0" to "1" alternations, or six alternations total. If $\Lambda = 0$, then the perimeter must be either

all zeros or all ones, either of which is acceptable. If A=2, the box will look like a group of ones poking in from the outside to the center, which is fine. Four or more alternations will look like a *fuse*, where a wire enters from outside the box, goes through the center (which is one wide) and exits out another side. This last case is an error.



Examples of the Width of Two Checking Algorithm

With this algorithm, a bitmap that is 512 bits long can be created that indicates which three-by-three squares are legal and which are not. This will allow width checks on polysilicon and diffusion. Since a spacing check on polysilicon is the same as a width check on white space, a spacing check can also be performed on polysilicon.

The above algorithm uses three-by-three boxes to perform a *minimum width of two* test. Using four-by-four boxes, the same method can perform a minimum width of three test, given that the width of two test has already been passed. At some point after the minimum width of two test has been passed four ones will appear in the center of a four-by-four box. When such a box is found, the alternation rules explained previously are used to check whether the width of three test has been passed. A four-by-four box that does not have four ones in the center automatically passes, because it does not give any additional information for the width of three test.

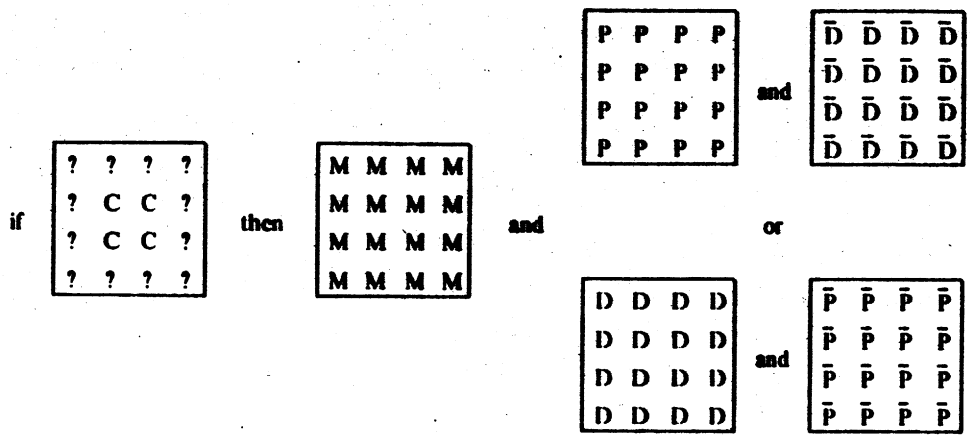
Given the minimum width of two test and minimum width of three test, we can perform the

following design rule checks:

1)	width2(polysilicon)	/* poly width */
2)	width2(not polysilicon)	/* poly spacing */
3)	width2(diffusion)	/* diffusion width */
4)	width3(diffusion)	
5)	width2(not diffusion)	/* diffusion spacing */
6)	width2(metal)	/* metal width */
7)	width3(metal)	
8)	width2(not metal)	/* metal spacing */
9)	width3(not metal)	
10)	width2(contact cuts)	/* contact cut width */
11)	width2(not contact cuts)	/* contact cut spacing */

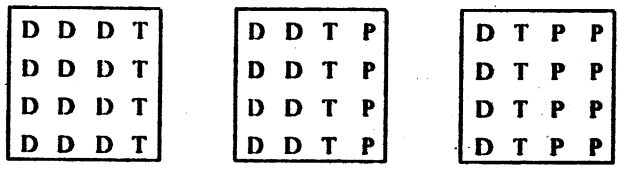
The next set of checks to be performed are those involving contact cuts. Contact cuts are to be used in very constrained ways. Ignoring the butting contact for a moment, the only contact cuts the author has ever seen have been either 2x2 or 2x4 in size. Using a four-by-four window, there is no way to constrain the contact cuts to be either 2x2 or 2x4. They can only be constrained to a size of 2xn where n>1.[1] The easiest way to enforce this 2xn constraint is to create a bitmap for contact cut width checking and change step # 10 above to use this new bitmap. If the center of the four-by-four window contains contact cut, then the whole window must contain metal. Also, in that case the whole window must contain polysilicon and no diffusion, or diffusion and no polysilicon.

[1]While there is no design rule for the maximum size of contact cuts, large ones are considered bad.



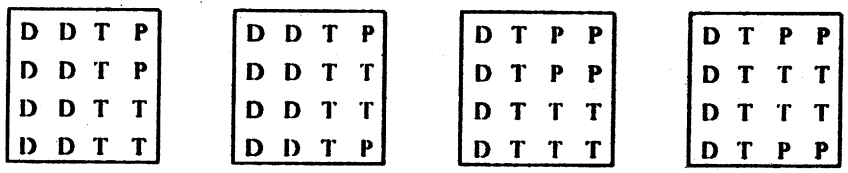
Simple Contact Cut Rule

When butting contacts are considered, there are a few more cases to handle. There are three views of a butting contact that have a contact cut in the center. A check is made to be sure that the current view is one of the three legal views.



Butting Contact Extenstions to the Rule (T = P and D)

Another factor that must be taken into account is a two-wide diffusion that designers often extend under the polysilicon in pullup resistors.



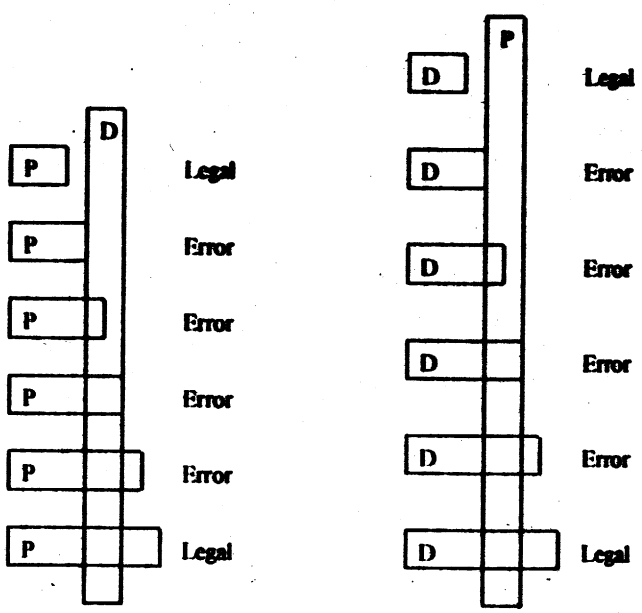
More Butting Contact Extensions to the Rule

Another contact cut design rule indicates that a contact cut to diffusion must be at least two

lambda from a transistor. This is checked by looking for both polysilicon and contact cut present in the window. When that is found, a new spacing check of width two is performed on the new layer created from the union of the polysilicon and contact cut layers. All of these contact cut design rules are summarized here:

```
10')  special_width2(contact cuts)                    /* contact cut width (2xn) */
12)   if (center is contact cut) contact()            /* check special contact cut cases */
13)   if (contact cuts and polysilicon) width2(not (contact cut or polysilicon))
                                                      /* check distance from transistor */
```

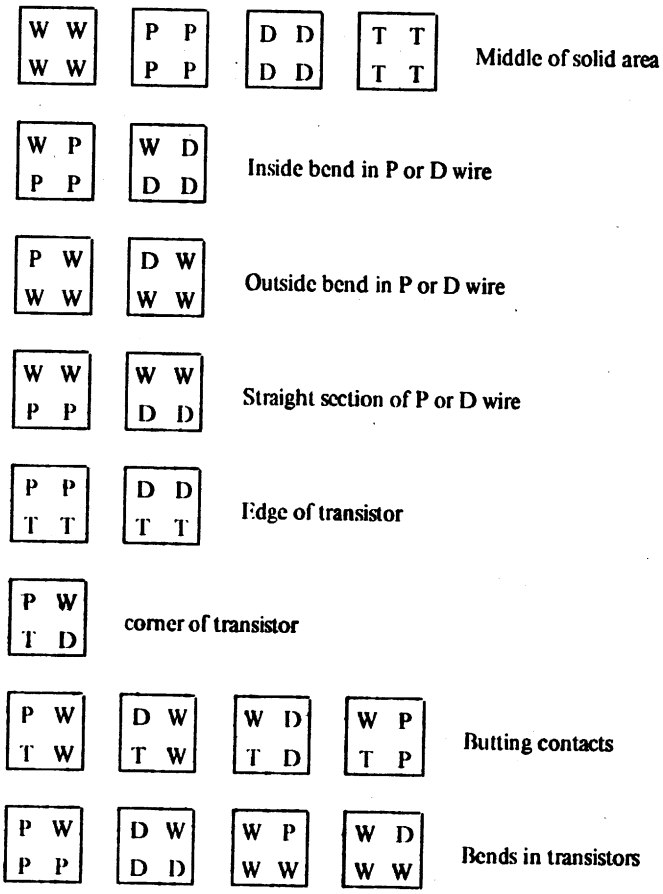
The last set of checks is for transistors. Polysilicon must overlap diffusion by two lambda. In addition, diffusion must overlap polysilicon by two lambda. Catching all of the cases in one check is difficult. Consider the following case. A vertical, two wide diffusion wire is present. To its left is a horizontal, two wide polysilicon wire whose right end is one lambda away from the diffusion wire. There is no design rule error here because the minimum spacing between polysilicon and diffusion is one lambda. If the polysilicon wire is moved one lambda to the right, a spacing error occurs. If it is moved another lambda to the right, a transistor error occurs. Another one lambda move to the right results in another transistor error. Yet another one lambda move to the right causes the wire to poke out the other side by only one lambda and this is also an error. One final move one lambda to the right gives a legal transistor. All these cases must be detected.



Examples of Transistor Errors

The one lambda overlap can be detected by subtracting the diffusion layer from the polysilicon layer and looking for an object which is one wide. The same check can be used for diffusion extending past polysilicon.

The polysilicon to diffusion spacing can be checked by looking at a two-by-two box. Since the number of legal two-by-two boxes is small, the list of all possible boxes was generated by hand. This check handles the case where polysilicon and diffusion touch but do not cross.



Legal Two-by-Two Views of Transistors (W = White)

The other two cases can be checked by looking at a two-by-three box in which the lower left and lower center cells contain both polysilicon and diffusion. The legal combinations of the rest of the elements have been determined experimentally and entered by hand. To summarize the transistor checks:

14) width1(polysilicon-diffusion)

15) width1(diffusion-polysilicon)

16) pdspace(polysilicon,diffusion)

17) tcheck(polysilicon,diffusion)

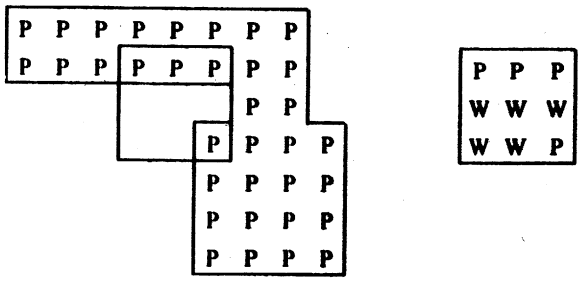
Scaning at one lambda per pixel will not allow the checker to verify that ion implantation extends one and a half lambda beyond transistors. However, it will allow the checker to look for at least a one lambda overlap. If the center of a three-by-three square contains T and I, then a check is made to be sure that the whole three-by-three square contains T. A design rule states that ion implantation must be one and a half lambda away from a non-implanted transistor. Using the same scheme, if the center of a three-by-three square contains T and no I, then the whole three-by-three square should contain no I.

18) checki(polysilicon,diffusion,ion)

All the above checks performed at once will determine if a four-by-four window obeys the design rules. Even if the function that performs these checks is slow, a caching scheme can be used to speed up the program.

Though the raster scan design rule checking algorithm works, there are some problems with it. No checks with the ion implantation mask are currently made. Since all rectangles are rounded to the nearest lambda coordinate and since the ion implantation mask is usually on a half lambda boundary, some careful thought is necessary to fit it into this scheme. Rounding everything to lambda boundaries can cause other problems. Some designers make use of the half lambda grid to avoid spacing errors. The design rule checker might report spacing errors when these designs are rounded to a lambda grid. Minimum width diagonal rectangles will contain width errors when placed on a lambda grid. Trying to avoid these problems by moving to a half lambda grid does not work, for two reasons. First, the design rule check would take four times as long. Second, the window would be seven-by-seven and the current algorithms for looking at four-by-four windows and detecting errors do not scale up. A new algorithm would be needed.

Spacing errors are often reported when in fact none exist. This situation can occur when a wire goes right, up a lambda, and left. At the bend, there is a one lambda square of white space. It looks like there are two wires too close together, though they are really both part of the same wire.



Spacing Error that Should Not be Reported.

These situations are examined more closely by another part of the program, so that only the real spacing errors are reported. This part of the program needs knowledge of the connectivity (*i.e.*, contact cuts). Currently it does a ponds and islands check on the current layer using a six-by-six window. This removes most of the spurious error reports. Perhaps the node finder described in the next chapter should be run before the design rule checker.

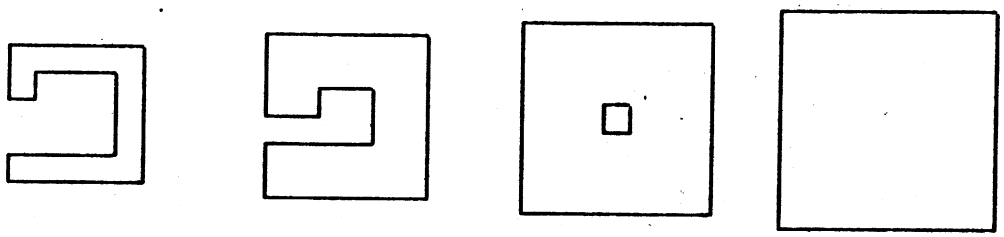
It should be noted that all knowledge of the design rules is embedded in the actual code of the program and in pre-generated bit tables. The design rules themselves are not input directly into the checker. The rules were interpreted by the author, not by a program, to produce the necessary checks. When the design rules change, the programs will require modification. It is even possible that some future design rules cannot be checked with a four-by-four window.

The advantages of this method are its speed, its ability to check entire chips, and its ability to report only legitimate errors. In addition, this checker could possibly be implemented as a VLSI chip itself, allowing the checking to be very fast.

4.2 Rectangle method

The rectangle method can best be described as a geometry engine that works on a set of rectangles and accepts commands like union, expand,[1] width, and so on. The design rules are expressed in terms of these commands. To avoid spurious error reports, all the intersecting and abutting rectangles of a single layer must be combined together into a polygon. A width check is then performed on this polygon, and spacing checks are performed between polygons. A typical way to perform spacing checks is to enlarge each polygon by half of the minimum spacing and then look for intersecting polygons.

Care must be taken in implementing the various operations of the geometry engine. The simple approach for finding intersecting rectangles is to compare each rectangle to all the others. This results in $O(n^2)$ performance. Speed improvements can be realized by either sorting or partitioning the input. The expand operation can be tricky to implement, since it may cause a simple polygon to acquire an interior area that did not exist previously.



The Expanding Polygon Problem

The research performed for this thesis did not include the development of a rectangle-based

[1]Expand will enlarge a rectangle (or polygon) by a specified amount in all directions. This is useful in checking minimum spacing.

design rule checker. Most of the existing design rule checkers use the rectangle approach, and the reader is referred to them for more information: McCaw [10], Wilcox [11], Rosenberg and Benbassat [12], Lindsay and Preas [13], and Seiler [14].

5. Node Extractor

This chapter discusses the program that has come to be known as the *node extractor.* It extracts information about all the transistors along with their connectivity from the mask descriptions. In the process, certain types of errors are detected. First, a basic description of how the node extractor works is presented. Following that is a discussion of some of the extensions that have been implemented.

5.1 Basic algorithm

The node extractor has two tasks to perform. First, it must follow the *connectivity* of the wires. Second, it must find *transistors.* The original input format consists of a hierarchical set of *symbols.* Each symbol may contain both basic rectangles and calls to other symbols. A clever program might be able to extract the circuit description from the lowest level symbol (*i.e.* a symbol that contains only boxes), and using that, build up the whole circuit, following the symbol-calling hierarchy and extracting each symbol only once. Since CIF places no restrictions on the combination of symbols and boxes, a program of this type would have many strange cases to consider. The chip designer may run wires over a symbol. These wires might cause new transistors to be created or certain nodes to be connected together (*e.g.* as in PLA programming). This approach seemed too hard to implement, even though it would potentially run very fast.

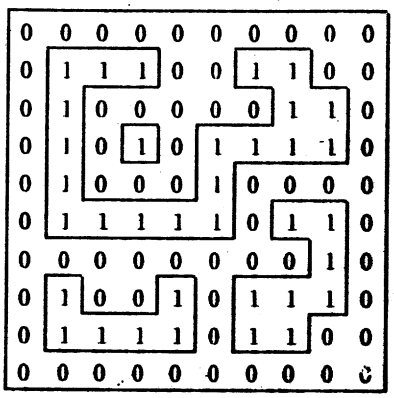
If the CIF symbol hierarchy is not used, it seems worthwhile to *fully instantiate* the chip, creating a file of *rectangles.* The connectivity can be followed by finding all the rectangles of a given layer that either abut or intersect. A transistor is formed whenever a polysilicon rectangle intersects a diffusion rectangle. Though it sounds like this method should work, there are many problems. Finding intersecting rectangles is a time-consuming task. Finding all the diffusion rectangles that

intersect and calling them a node is not really correct, since a diffusion node becomes two nodes whenever the original is crossed by polysilicon. The polysilicon rectangle could have been made up of many smaller polysilicon rectangles. In general, the whole chip could have been made up of one lambda square rectangles. This means that the rectangle method could not rely on the designer having specified everything with "nice" rectangles, but must assume the worst possible case. The general solution seems to require the merging into polygons of all rectangles that intersect or abut. Therefore, algorithms now deal with the unions and intersections of polygons. This method seemed too complicated, so a simpler, but possibly slower method was sought.

Since the design rules were expressed in lambda, and since the designers with whom I worked designed in terms of lambda, a *bitmap*-based approach seemed feasible. In such a scheme, the whole chip would be represented as a big bitmap with each element representing one square lambda of the chip. The term bitmap is a little misleading since each "bit" really contains one bit for each layer; pixel-map might be a better term. While it might be impractical to store the whole bitmap anywhere at one time, there might be algorithms that can process it in raster scan order, buffering only a few scan lines in memory at any one time. Raster scan order is left to right, top to bottom.

Before moving on to some algorithms that deal with bitmap images, there are a few words to be said in their favor. Once the rectangle format has been converted into a bitmap, information about which rectangle created what bit has been lost. This is good because it has the effect of merging intersecting and abutting rectangles together with little effort. All subsequent algorithms are insulated from geometrical "features" such as arbitrary polygons, round flashes, and so on. These have to be dealt with only in one place. The disadvantage of moving to a bitmap version of the chip is that the run time of any algorithms will probably be proportional to the area of the chip.

The basic algorithm for following connected regions in a bitmap image comes from the classic *ponds and islands* problem which is defined as follows. Given a two dimensional array of zeros and ones, where the zeros represent water and the ones land, write a program that counts the number of land masses and prints out the area of each one. Assume that land must connect horizontally and vertically but not diagonally.



Answer for this example:

4 land masses

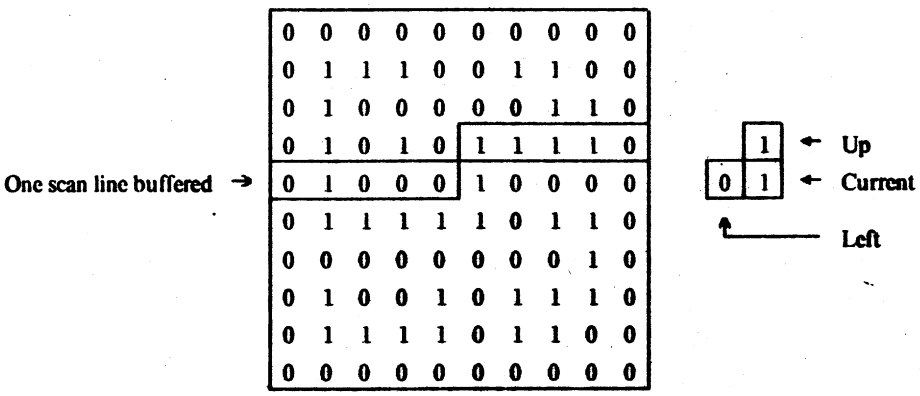
Areas are 1, 6, 8, and 20

Classic Ponds and Islands Problem

A common solution uses a procedure that places a *footprint* on the current piece of land and calls itself *recursively* for each of the four surrounding squares, returning when the current square is water or contains a footprint. The main driving program scans the whole array, calling the footprint procedure for each piece of land that has not yet been walked on. This solution is simple to program and easy to understand. For our purposes it is not suitable, since it randomly accesses the bitmap array. This would result in many page faults when following a node like the metal layer that makes up V_{dd}. It would also require the whole bitmap to be part of the program's virtual address space ($>2^{24}$ pixels for large chips).

5.2 Ponds and islands

While I was thinking about the ponds and islands problem, the following algorithm came to mind. The data can be processed in *raster scan* order, with one scan line buffered in memory.[1] At each point, access is needed to three bits of information: the current bit, the bit to the left, and the bit above.

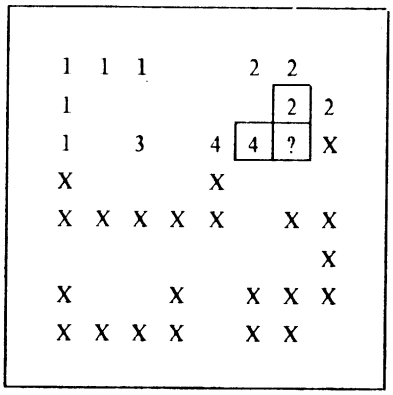


Buffer One Scan Line; Look Up and Left

Since there are three bits of information, there are eight cases to consider. Four of them can be handled at once. (1-4) If the current bit is water, there is nothing to do. (5) If the current bit is land and the other bits are water, then the upper left corner of a new piece of land has been found. This information is remembered in an array as big as the scan line, and the new piece of land is assigned a unique number. (6) If the current bit is land and the bit to the left is land, and the bit above is water, then this is the top of a horizontal strip and the number is the same as the number to the left. (7) When the top is land and the left is water, then this is the left edge of a vertical strip and the number is the same as the number above. (8) The interesting situations occur when all three bits are

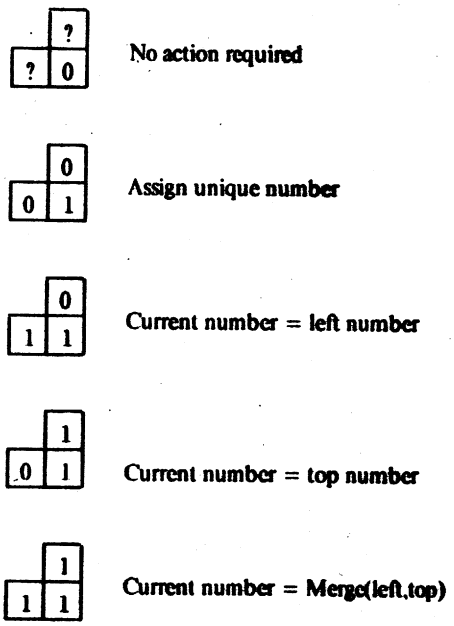
[1] Really, portions of two scan lines are buffered. The total space used is equivalent to that of a single scan line.

land. (8a) If the number above is the same as the number to the left, then that must be the number for the current bit. (8b) If the numbers are different, the lower right inside corner of some shape has been found, and two pieces of land which previously seemed distinct are found to be part of the same mass.



An Example of Case (8b), 2 and 4 Must be Merged Together

Some bookkeeping is needed to update the counts of one land mass number with those of the other land mass.



Basic Raster-Scan Ponds and Islands Algorithm

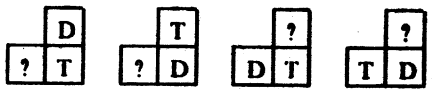
The same algorithm that works with land and water will also work with polysilicon, diffusion, and metal. For the purposes of following wires and finding transistors, four *derived* layers will be used. Metal (M) and polysilicon (P) correspond to the mask layers by the same name. In the node extractor, diffusion (D) will be the diffusion mask *minus* the polysilicon mask and transistors (T) will be the diffusion mask *intersected* with the polysilicon mask.

The basic algorithm for following connectivity given a raster scan version of the masks can be thought of as a ponds and islands search on four layers at once. The electrical properties of contact cuts can be added by checking to see if the current cell contains P, M, and C. If it does, merge the number of the polysilicon layer with the number of the metal layer. If not, check the current cell for D, M, and C, and merge the number of the metal layer with the number of the diffusion layer if this is the case. A simple design rule check can be performed at this time if desired. If the current cell contains M and C but no P or D then it represents an unnecessary use of the contact cut and should probably be flagged as an error.

5.3 Oh where, oh where, can my transistor be?

While all the above has been necessary, it does not find transistors. However, it does give us a good data base upon which to base a transistor detection algorithm. The following information is available in the interior of a transistor: the node number of the T layer, the node number of the P layer, and the ion implantation bit. However, edges of the transistor layer actually contain the useful information, namely the node numbers of the diffusions. At the center of the transistor, there is no diffusion, and hence no diffusion node numbers.

The node extractor finds *pieces* of transistors and writes them into a file. These pieces will be processed further by another program. A piece of a transistor consists of the node numbers for T, P, and D, along with a bit for ion implantation. During the ponds and islands processing, the transistor finder looks for one of four cases: (1) current cell is transistor and left is diffusion, (2) current cell is diffusion and left is transistor, (3) current cell is transistor and up is diffusion, or (4) current cell is diffusion and up is transistor. For each match (and there may be more than one), a transistor record is generated.

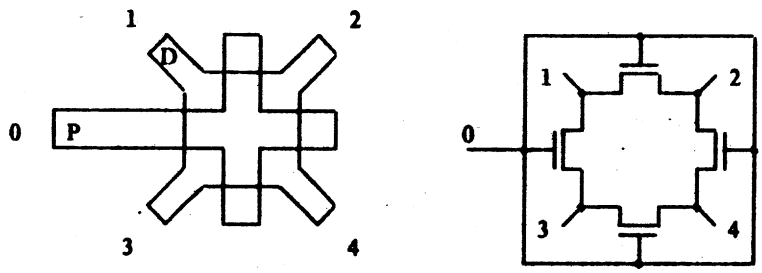


D = diffusion minus polysilicon
T = diffusion and polysilicon

Basic Transistor Finding Algorithm

After the node finder has finished, some further processing is needed on the transistor pieces to turn them into transistors. Since some of the node numbers may have changed from the time the transistor piece was written out until the time the node finder finishes running, all the node numbers in the transistor pieces must be updated to reflect the final node numbers. The pieces are then

sorted by their transistor node numbers, bringing all the pieces of a particular transistor together. Reading through the sorted file, all the information on one transistor is gathered, making sure that the polysilicon numbers are the same for each record, and a list of diffusion numbers is created. If there is only one diffusion number, then a degenerate transistor (*i.e.* a MOS capacitor) has been found; these are currently ignored. Most of these are caused by the one lambda overlap of polysilicon and diffusion found in butting contacts. If there are two diffusion numbers, then a normal transistor has been found, and it is written out to the circuit file. If there are three or more different diffusion numbers associated with one transistor, then an *unusual transistor* has been found. While these transistors are theoretically possible, I have not yet found one on an actual chip. These unusual transistors can either be flagged as errors or converted to some number of normal transistors.



Example of an Unusual Transistor

While this scheme for finding transistors might seem too simple at first, it has worked on many designs, including some constructed with the intent of confusing it. Even butting contacts and butting contacts in the middle of depletion mode pullups do not cause confusion.

5.4 Further processing

There is an opportunity here for some further checking. Assuming that the designer has *symbolic names* for some of his nodes, the program can make sure that two different symbols do not have the same node number. If there is a provision for the designer to pass symbolic names through the CIF language and if he gives symbolic names to many signals (including the same name at many different locations), then this check will catch shorts. A similar check for the same name having two different node numbers will catch open connections.

The program that writes the final transistor file keeps track of some information for each node number. It knows whether that node number has been used as the gate of a transistor, and whether the node number has been used as the source or drain of a transistor. After all the transistors have been processed, a check can be made for material that is not connected to any transistors, nodes that are defined but not referenced, and nodes that are referenced but not defined.

A node that is not connected to any transistors indicates a piece of material on the chip that is not connected to anything. These pieces occur for many reasons, the most common of which is caused by the designer's name or logo.[1] Extraneous pieces of material on the chip can also come from the use of library cells that have unused busses. Some graphics editors make it easy to accidentally drop small pieces of material in the chip. Some designers use certain layers for alignment marks. The node finder will detect all these cases.

A node that is referenced but not defined is a node that has only the gates of transistors connected to it. This must represent an error, since such a node will be *floating* in the manufactured

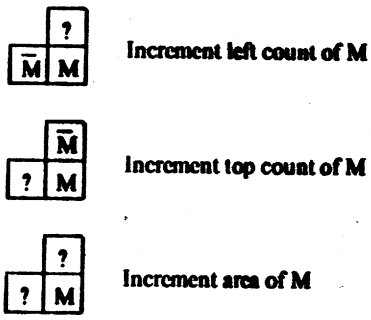
[1]These logos cause problems in all phases of artwork analysis. However, it is probably best to include them, just to check for the case when a misplaced logo interferes with the rest of the circuit.

chip. Input nodes will not be reported as "referenced but not defined" because the standard input pad has a transistor with gate and source connected to ground[1] and drain as the input. This transistor protects the rest of the chip from static induced overvoltage.

When the transistor file is created, there are some single transistor checks that are performed. Any transistor with a gate of VDD or GND is flagged as a possible error. In addition, any transistor which, if turned on, would connect VDD and GND together, is flagged.

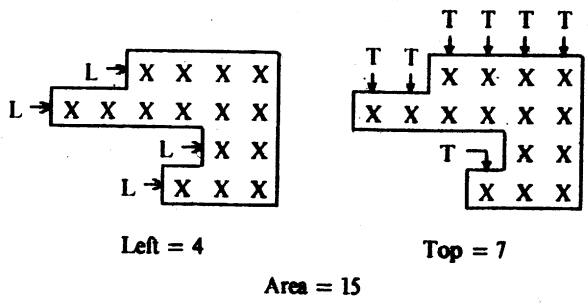
5.5 Extensions

For further checking and simulation, it will be useful to gather some additional information during node extraction. The extraction of any complicated information may significantly degrade the performance of the node extractor. The approach taken was to extract some simple parameters about each node, and see what could be derived that information. For each of the layers on which the ponds and islands search is performed, three numbers are extracted: the length of the left edge, the length of the top edge, and the area.



Simple Parameter Extraction Algorithm

[1]By this point in the processing, the designer has received checkplots of his chip that have each node labeled. He has told the node extractor the numbers of VDD and GND.



Example Using the Simple Parameter Extraction Algorithm

From this information, we can derive the *capacitance* of each node, and the *length/width ratio* of each transistor. The capacitance of a node is one of the factors that influences the overall speed of the chip. The length/width ratio of a transistor will allow certain ratio checks to be performed by the static evaluator.

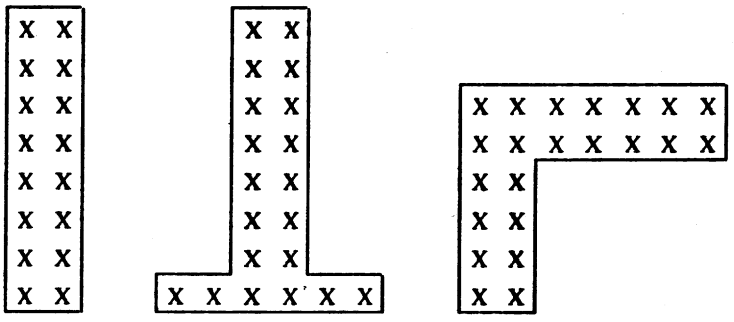
The capacitance of a node depends on its *area* and, in the case of diffusion, its *perimeter* (side-wall capacitance). Most of the capacitance will be between a node and the *substrate*. Although when metal crosses over polysilicon there is a small capacitor created between the metal and the polysilicon, those cases will be ignored. All capacitance will be assumed to exist from the node to the substrate. This assumption breaks down when the designer specifically constructs a *bootstrap capacitor* by placing polysilicon over diffusion.

While there is insufficient information extracted to compute the length/width ratio of an arbitrarily shaped transistor, most transistors fall into one of three classes which can be computed from our simple numbers. The gate area of most transistors is rectangular and therefore can be calculated exactly and easily. If the top (T) times the left (L) is equal to the area (A), then the gate area must be rectangular, since this formula only holds for rectangles.

If the gate area is not rectangular, the resistance can be estimated. To know which way the current flows through the transistor, two *orientation bits* (OB's) must be added to the transistor record. One bit indicates a *vertical transistor* (*i.e.*, diffusion was found above or below the gate area)

and the other bit a *horizontal transistor* (*i.e.*, diffusion was found to the left or right of the gate area). If both bits are set, the transistor has bends in it (as in the output pads). Usually, these transistors are two lambda wide. The length can be guessed as the area divided by two, minus one for each bend. Since the number of bends is unknown, only one is assumed. The equation is Length = A/2-1, Width = 2. There is really not enough information for obtaining exact length/width ratios, so some other method would be required if exact numbers were needed. For now, the above approach seems to work.

The last case to consider is that of a non-rectangular transistor with only one of the two orientation bits set. Experience indicates that these usually occur in depletion mode pullups with butting contacts. Here, the gate area is two wide at the top, changing to six wide at the bottom. If the gate area is two wide at one end and the general shape has one change of width over its length, we can calculate the resistance from our top, left, and area information. The equation is Length = (L*W-A)/(W-2), Width = 2.



8x2	7x2	10x2
L=8	L=8	L=6
T=2	T=6	T=7
A=16	A=20	A=22
OB=10	OB=10	OB=11

Transistors whose Resistances are Calculated Correctly

Currently, no attempt is made to extract the resistances of the various nodes. This information would be useful but is hard to obtain, as mentioned before. The length/width ratios of the transistors will be used by the static evaluator described in the following chapter.

5.6 The output format

No attempt has been made to create a *network definition language* suitable for all the different levels of description and simulation. Instead, a simple format with four different types of records is used. There is a separate record type for each of the following: enhancement mode transistors, depletion mode transistors, input nodes, and node dimension records. Both types of transistor records contain the names of the gate, source, drain, channel length, channel width, and coordinates on the checkplot of the transistor. The input records contain the names of the nodes that are inputs, for later use by the static evaluator. The node dimension records contain the area and perimeter information for each layer of each node, for later use by the simulator. There is no declaration entry in the output file for each node. Instead, the names of all the nodes can be derived from the transistor information.

The node extractor should be able to produce output suitable for input to the SPICE simulator, if needed, since all necessary information is extracted. If bootstrapping is used, the node extractor would have to be changed to detect capacitance between nodes (as opposed to capacitance from a node to the substrate).

6. Static Evaluator

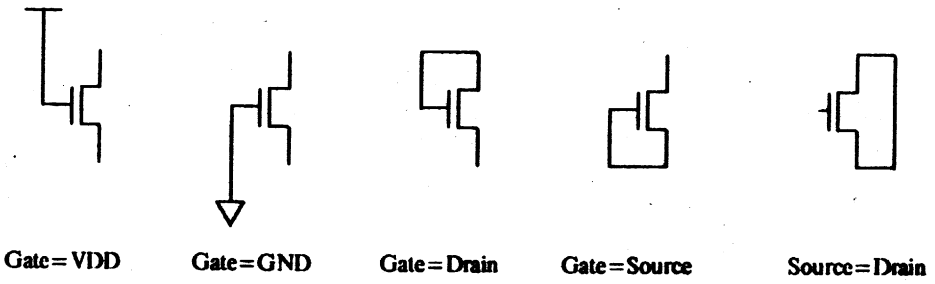
The development of the node extractor allowed easy conversion from artwork to a circuit description. The obvious next step seemed to be *simulation*. In the process of simulating various designs, it became clear that there were some errors detected by simulation, which could have been detected earlier by a program that analyzed the circuit. Such a program would perform a *static analysis* of the circuit, looking for anomalous configurations of transistors. In addition, there are various errors which switch level simulation does not detect, but for which the static evaluator could check.

An analogy to compilers can be drawn here. The errors that the design rule checker finds are like the errors detected by the syntax phase of the compiler. Errors detected by the static evaluator are similar to errors detected by the semantic phase of a good compiler. Finally, errors discovered during simulation correspond to errors discovered during execution or interpretation of a program. A compiler might warn the user that his program contains variables which are set but not used, or used before given a value. It might also warn the user that there are statements in the program that can never be reached. Similarly, the static evaluator will find parts of the circuit which depend on nodes that can never be given a value, and it will locate nodes which can never be turned on (or off).

The static evaluator takes, as its input, the list of transistors and input nodes output by the node extractor. Each transistor is identified by the node numbers of the gate, source, and drain, along with its length/width ratio. In addition, there are assumed to be two distinguished nodes: VDD and GND. No assumption is made about clocking. The identification of input nodes is necessary to distinguish them from undefined nodes. An input node is assumed to be potentially pulled up or pulled down (grounded).

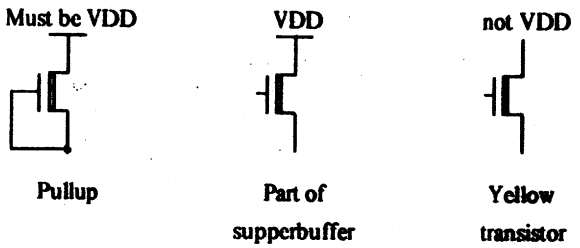
6.1 Reading in the network

As each enhancement mode transistor is read, it is added to the program's data base of transistors. Certain checks are made immediately, based solely on the information contained in that transistor. A diagnostic is generated if the gate of the transistor is either VDD or GND. A diagnostic is also generated if any of the gate, source, or drain nodes are the same.



Illegal Enhancement Mode Transistors

Depletion mode transistors are handled in a slightly more complicated way. A depletion mode transistor is typically used as a *pullup resistor*, in which case the drain is connected to VDD and the source and gate are connected together. Sometimes depletion mode transistors are used as *superbuffers*, in which case the drain is still connected to VDD, but the source and gate are not connected together. The final use of a depletion mode transistor is as a "*yellow transistor*", one in which the designer wants the polysilicon and diffusion wires to cross, but without creating a transistor. If it is implanted (usually indicated as a yellow layer), a depletion mode transistor is created. That transistor is like a *resistor*, in which the two wires cross each other at the expense of some speed. The classic use of yellow transistors occurs in multiplexors. A yellow transistor can be detected because neither the source nor the drain are VDD, something that is never true for a depletion mode transistor used as a pullup resistor.



Types of Depletion Mode Transistors

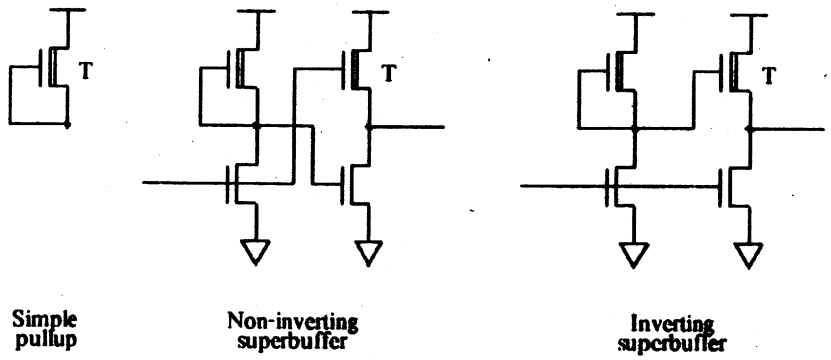
When a depletion mode transistor is read, a check is made to see if the gate is the same as either the source or the drain. If so, the other node must be VDD; otherwise a diagnostic is generated. This detects unpowered pullup resistors. If this transistor is a yellow transistor, it is converted into an enhancement mode transistor with a gate of VDD. At this point, a check is made to detect whether both the source and drain are VDD. All non-yellow transistors are entered on a list of pullup resistors.

When an input node is read, it is marked as being possibly pulled up, and possibly pulled down. This completes the initial processing of the circuit.

6.2 Depletion mode transistor checks

Once yellow transistors have been converted, there are only three ways in which depletion mode transistors are used: normal pullup, inverting superbuffer, and non-inverting superbuffer. While there is no rule that says these are the only uses of depletion mode transistors, these three are the only ones which have been encountered in practice by the author.[1] As other uses are found, they can be added to the list of legal ones.

[1]One designer has depletion mode pulldowns to ground on some of his input pads. Currently these would be flagged as an error.



Legal Uses of Depletion Mode Transistor (T)

To check for proper use of depletion mode transistors, the list of these transistors is scanned. For each transistor, a pattern match is performed against the set of legal uses. Any that do not match are flagged as potential errors. In addition, the total number of each pattern found is reported as information for the designer.

6.3 "Stuck at" checks

For a particular node to be useful, there must be some way to give it the value 1 and some way to give it the value 0. In general, it is impossible to determine if a given node can ever take on a particular value without simulating the circuit. However, a simple check can be made by assuming certain transistors can be turned on and verifying that for each node, a path exists to a pulled up node and to ground.

A series of passes is made over the enhancement transistor data base until no more *propagations* occur. At each transistor, a check is made to see if one side of the transistor is pulled up and the other unmarked. If so, the unmarked side is marked as pulled up (this counts as a propagation). The same test is made for ground. When this process settles, a scan is made of all the nodes. A node that does not have both the pulled up and pulled down bits set is flagged as an error.

In an actual chip, especially ones designed either by a computer or with libraries of cells,

these "stuck at" conditions often occur and are not considered errors. Typically, certain parts of a PLA come in pairs, with two pullups in each pair. If one of the *minterm* lines is not used, there will be no way to set it to zero; it will be stuck at one. If a designer uses only part of the function of a predefined cell, the unused part may contain some "stuck at" errors. Even though a spurious error message might occur, the checks seem worth performing. It does not take long to review all the warnings output by the program.

6.4 Threshold checks

Earlier, the problems of driving a *pass transistor* with a signal that is *one or more threshold drops* below V_{dd} were discussed. The database that has been built so far can be used to check for that type of error. Two items must be detected: (1) a signal one threshold drop below V_{dd}, and (2) what a pass transistor looks like.

In the previous section, multiple passes were made over the network during which each node was marked as potentially pulled up. To determine threshold drops, more care in our marking is necessary. Any node that has a depletion mode pullup resistor attached to it will be marked "pulled up", as will any node that is an input node. The nodes that can be reached from pulled up nodes will be marked "indirectly pulled up". This indicates that these nodes are at least one threshold below V_{dd}. A pass transistor with a gate driven by an "indirectly pulled up" node is an error.

Currently, the method for detecting pass transistors is not foolproof; some of them might be missed. Pass transistors must be distinguished from transistors in pulldown chains. One distinguishing characteristic of transistors in pulldown chains is that there are no transistor gates connected to the intermediate nodes in the pulldown chain. For a pass transistor to be useful, the

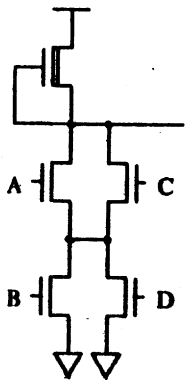
gate of some other transistor must eventually use the value passed through it, which means that there must be a node with gates connected to it somewhere down the line. The current scheme does not implement the "down the line" check. If it finds a transistor with a gate connected to an "indirectly pulled up" node, a source connected to a node that is "indirectly pulled up", and gates connected to the drain, the transistor is flagged as a possible error.

Though depletion mode pullups used in superbuffers should have their gates connected to nodes that are "pulled up", this check is not currently made.

6.5 Ratio checks

The ratio of the size of the pullup resistor to the size of the pulldown resistor should be four.[1] The effect of driving a pulldown transistor with a V_{dd}-V_{th} signal is to double its resistance. Since the node extractor makes an attempt to calculate the length and width of each transistor, an attempt can also be made to check the pullup/pulldown ratio. The first problem is locating the appropriate transistors on which to perform the check. Finding the pullup transistor is easy; from there it is necessary to look for possible *pulldown paths*.

The following scheme for finding pulldown paths was devised after much experimentation. A pulldown path is a path from a pulled up node to ground that passes through pulldown transistors. Each pulldown transistor is visited no more than once. There are no gates connected to an intermediate node, and none of the intermediate nodes are pulled up. No pulldown path is longer than seven transistors (or some other arbitrarily chosen small number).



Possible pulldown paths are:

A,B
A,D
C,B
C,D

Examples of Possible Pulldown Paths

After finding a pulldown path, the ratio calculation is fairly straightforward, though it should include the fact that transistors with gates that are "indirectly pulled up" are twice the resistance of those with gates that are "pulled up". One problem pertains to ratios that are not exactly four. Most chips will work if the ratio is off by a small amount. The ratio affects both the switching speed, and the thresholds at which certain voltages are said to be zero or one. One point of view allows for accepting a range of legal ratios instead of an absolute ratio of four. On the other hand, a chip designer who very carefully made sure that each ratio was exactly four might be interested to know where a mistake occurred. Currently, the program checks for the exact value. Experience indicates that chips either contain almost no violations or very many. The output can be sorted on the ratios so that the extremes can be examined first, and multiple occurrences of the same errors are listed together.

A nor gate contains several pulldown paths. The ratio check is performed for each path independently. If both pulldown transistors were turned on at once, the node would be pulled down faster than if only one were turned on. This is not bad, and corresponds to a ratio of eight to one. Each of the pulldown paths in a nor gate could have a different resistance. These will be considered one at a time and any that are in error will be reported.

7. Simulators

After all the errors that can be detected through static analysis have been removed, it is time for dynamic analysis, or *simulation*. Though the research performed for this thesis did not result in the creation of a simulator by the author, it did result in some further simulation research by other people. An in-depth discussion of simulation does not belong in this thesis, but an overview of simulation is appropriate, along with a description of two possible algorithms for performing switch-level simulation. For deeper coverage of simulation issues, the reader is referred to Bryant [15] and Terman [16].

7.1 Different types of simulators

There are many different levels at which simulation of VLSI circuits can be performed. Usually, simulation at a low level implies *circuit simulation*. The input is a set of circuit elements: resistors, capacitors, transistors (with length/width ratios), voltage sources, and input signals (*e.g.*, square waves of a specified frequency). The output is a series of graphs, showing the waveforms of each signal. The algorithm performs many separate integration steps for each unit of simulated time. This type of simulator is exemplified by SPICE which is usually run on small circuits (on the order of an output pad) and is expensive to run.

The next level up from circuit simulation is *switch-level simulation*. In this type, transistors are modeled as switches that are either on or off. Fixed delays are associated with the transmission of signals and with the changing of state of transistors. This will be the level of simulation emphasized in the rest of this chapter.

One level up from switch-level simulation is *gate-level simulation*. (A gate is composed of two or more transistors.) The input to a gate-level simulator consists of a list of objects with their

inputs, outputs, and information describing their interconnections. Typical objects include nand gates, inverters, and possibly registers and memories. This level of simulation is used for debugging TTL circuits. It is not a good method for modeling components of VLSI circuits, because there are certain circuit configurations which occur in VLSI circuits that cannot be modeled as objects with inputs and outputs. Examples include pass transistors and circuits with charge sharing.

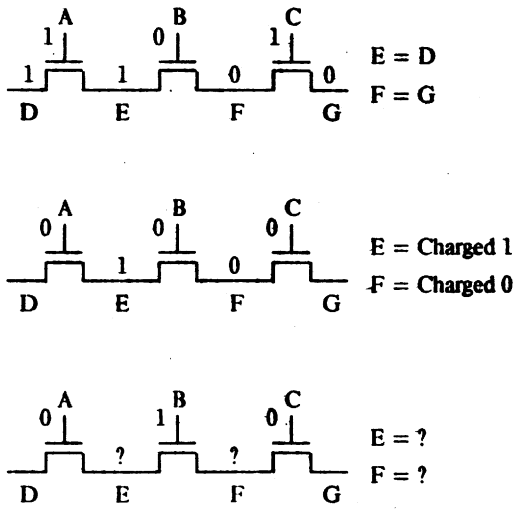
The highest level of simulation is usually called *functional simulation.* This level of simulation does not have any information about the underlying circuit (since the circuit might not have been designed yet). Instead, it tries to model the input/output behavior of the component modules. For example, if the chip is going to have a finite state machine as the main controller, then that will be simulated as a single module. Program variables are used to represent the chip's registers. Subroutines are used to model particular pieces of the chip. This type of simulation usually runs very fast, allowing the designer to simulate many clock cycles. The clever designer will develop test data with a functional simulator and use it to test the final chip. Ideally, the output from the functional simulator should agree with the output from a switch-level simulator.

7.2 A possible design of a switch-level simulator

For simulating the information derived from the artwork, switch-level simulation is the appropriate level of simulation to use. Circuit level simulation is too low a level, since the user is usually not interested in the actual waveforms that occur at each node in the chip. If all of the conservative design rules have been obeyed, the chip should work (though it might logically compute a result different from what is desired). In addition, today's designs are too large to be simulated as a whole at the circuit level. On the other hand, gate-level simulation is too high a level, since this type of simulator does not model all of the possible nMOS circuits well. The output of the

node extractor would require additional processing to group transistors into gate information, which could be used as input for a gate-level simulator. This would be difficult, however, since it is not always possible to form gates from all transistors. Often a gate-level simulator contains an extensive user interface, including macros and editing capabilities, which allow an easy and concise method for entering a circuit. When the input is computer-generated, such capabilities are not needed. The programmer of a simulator might spend so much time providing an easy method for entering input that he becomes distracted from the real problem of simulation. Some attempts have been made to modify a TTL simulator to work with nMOS, but with little success. Starting with a switch-level simulator would probably have given better results.

Before a discussion of some possible implementations of a switch-level simulator, it is first necessary to consider some issues which arise in typical circuits. Transistors are used not only as switches, but also as pullup resistors. This means that a pulled up node has a value of 1 unless it is also connected to ground. *Charge* can be stored on the gates of transistors and on nodes with enough *capacitance* (such as long wires). This charge will retain the state of that node, even when the driving force is removed. In the simulator, the assumption is made that the charge lasts forever, though in reality, the charge leaks away slowly and must be refreshed (dynamic logic). *Charge sharing* can also occur, *i.e.*, two isolated pools of charge can be merged if the pass transistor between them is turned on.



Example of Charge Sharing

The difficult question to answer is what happens when a zero is merged with a one: is the result zero or one? Often it is neither. If charge sharing is not handled, or if it comes from two equally sized pools of 0 and 1 merging, the resulting value will be undefined (X). Charge sharing can be either ignored, since most designs do not make use of it, or handled by the simulator using the capacitances reported by the node finder. Initial values for the internal nodes must also be considered. Three possibilities are (1) initialize all internal nodes to zero or one, (2) set each node to zero or one randomly, or (3) introduce another value (I) that indicates an initial value. The simulator should treat X's and I's in the same manner (both as "undefined") when computing new values, but the distinction can be useful in notifying the designer if any problems occur.

Two possible methods of switch-level simulation will be presented. The first one, the *equivalence class method*, is easy to explain and easy to implement. The second one, the *event driven method*, is slightly more complicated but runs much faster.

7.2.1 The equivalence class method

In the equivalence class method of simulation, three pieces of information are associated with each node: an old state, a new state, and a bit which is set if there is a pullup transistor on the node. The user sets all the input variables to the desired values and instructs the simulator to simulate the circuit until it *settles*, at which point the values of the nodes (either internal or output pads) can be displayed.

The simulator makes repeated passes, called *microsteps*, over the circuit. When a pass is made and no more changes take place, the simulation has settled. A microstep consists of the following steps.

1) Place each node in its own equivalence class.

2) For each transistor that is turned on (*i.e.* its gate has a value of 1), merge the equivalence classes of its source and drain.

3) For each equivalence class, determine the value of the equivalence class by looking at the old value of each node in the equivalence class. The value is determined from a collection of nodes that may be connected to VDD, connected to GND, pulled up, charged 1, charged 0, initial, or undefined. Once this value is determined, the new value of each node in the equivalence class is set to this value.

4) For each node, copy the new values to the old values, noting if any changes occurred.

5) If any changes occurred, repeat from step 1. Otherwise, the circuit has settled.

Some issues are ignored in this simple statement of the algorithm. The two major points to consider are the *merging* of equivalence classes and the determination of the new value for the elements in an equivalence class from the collection of old values. What should happen when the user connects VDD and GND together? What does it mean when a pass transistor has a gate of X?

The real problem with this algorithm is its speed. At each microstep every node and every transistor must be examined, whether a change has occurred or not. If the simulator only took action when a change occurred, it could compute its work much faster. A good test example is an inverter chain 1000 inverters long. To simulate a signal propagating through the inverter chain, the equivalence class algorithm requires 1000*1000 operations. However, an algorithm that recomputes only when something changes should take only 1000 operations.

Though an inverter chain 1000 long does not usually occur on a real chip, and though many chips contain many signals moving at the same time, an improvement in speed can be realized by an algorithm that does not recompute the whole chip at each microstep. The event based simulator incorporates such an algorithm.

7.2.2 The event based method

In an event based simulator, each node has a bit that is set if it is pulled up, and a variable that contains its current state. In addition, an *event list* is used to store a list of actions to be taken. Initially, the circuit is in a consistent state (possibly all 1's) and the event list is empty. The user changes the value of some node (usually an input) and instructs the simulator to perform computations until the circuit settles. Once it has settled, the values of any nodes can be examined.

When the user changes the value of a node, the simulator enqueues on the event list an event that contains the name of the node and its new value (*i.e.*, forced 0 or forced 1). Then, the following algorithm is executed until the event list is empty.

1) The first event is removed from the event list. A new potential is calculated for the node, based on the potentials of all the other nodes connected by turned-on transistors to this node.

2) If this new potential is the same as the old, then this event has no effect. No further processing is performed on this event, and the execution continues from step 1.

3) A scan is made of all transistors with sources or drains that are connected to this node. If the change to the new value could possibly change some other node (that might be on the other side of a transistor with a gate of "X"), that node is enqueued on the event list.

4) If the change in value would affect any transistors with gates that are connected to the node, then all such transistors are enqueued on the event list.

5) The value of this node is updated to reflect the new value and processing continues from step 1.

Computations of new values based on old ones are performed with *table lookups*. The network is stored so that it is easy to find all transistors with sources and drains (or gates) that are connected to a particular node. The net result of this algorithm is that simulation is performed only on the pieces of the circuit that change.

Many simulation schemes are possible, but the above two should give the reader some ideas about selecting an implementation of his own. The specific details of the simulation step have been found to be particularly sensitive; a slight change in a functioning simulator often causes it to stop working. It is important to have a set of test examples to verify that the simulator still works after a modification has been made.

7.3 Possible speed improvements

There are many possible ways to speed up the simulator. A few ideas which have been partially implemented [16] are discussed here.

Though it has been indicated here that a circuit should be viewed as a collection of transistors by the simulator, often an actual designer thinks in terms of gates. An improvement in

speed could be made if it is possible to find all the gates in a circuit, and if it is faster to simulate the gates rather than the individual transistors which make them up. Any transistors that are not part of a gate would still be simulated in the usual way.

A gate can be recognized by finding a node that is pulled up and then is *simply pulled down*, *i.e.* there are paths from the pulled up node to ground, through transistors with intermediate nodes that are not used anywhere else. The output of the pulled up node may go through pass transistors, but eventually it can only be connected to a transistor's gates. Output from such a node is a strict function of its inputs (the gates of the transistors in the pulldown chain), and it does not require simulation of its component transistors to determine its value.

A common method of implementing combinational circuits and read-only memory uses *Programmed Logic Arrays* (PLAs). If a PLA can be recognized, it can be replaced by a table lookup. The general form of a PLA is some number of inputs connected to a number of minterms, which are in turn connected to outputs. The minterms are pulled up and potentially pulled down by various inputs. The outputs are pulled up and potentially pulled down by various minterms. Such a structure can be recognized by one of two methods.

A check can be made for structures that fit the specific shape of a PLA. This will work, but additional optimization is possible. Most PLAs have superbuffered outputs. An algorithm that looks for blocks of logic with outputs that can be computed from their inputs would not only find the PLA, but would detect that the real output is on the far side of the superbuffers. In addition, such a program may find pieces of logic that were not implemented as PLAs, but nevertheless can be converted into tables. For example, it might be possible to convert an inverting superbuffer into a simple inverter.

7.4 User interface

An area that is often overlooked in simulator design is the *user interface.* A poor user interface can make the fastest algorithm useless. Unfortunately, there is no set of rules to follow to create a good user interface. A few of the interface-related issues to keep in mind when designing a simulator are mentioned below.

The circuit input format is not really a problem when a simulator is driven from extracted circuits, though it should be readable by the user. Many errors that are discovered during simulation can be patched in the circuit file, using a text editor. This means that it is not necessary to re-extract the entire circuit before simulating it again.

It should be possible to specify different step sizes during the simulation. When debugging a circuit for the first time, the user might discover an unclocked feedback loop resulting in a circuit that never settles. This user needs a command to execute a single microstep or event at a time, letting him examine values in between. After completing that process, he might want to step through the various clock phases of the circuit himself, examining variables at each step. When the user is sure that everything is working correctly, he needs a command to execute a full clock cycle. When individual cycles work, he will want to set up certain input vectors and run the chip through many cycles. Some chips designed to interface with memories may require an interface more complicated than vectors of inputs, since they require simulation of a piece of the outside world. This leads to the next capability.

Some chips use a straightforward two-phase clocking scheme, while others use a more complicated one. It seems that the only way to handle all the possible clocking and input/output requirements is by providing the user with a facility for writing macros or programs that can call the simulator. Such a language could be embedded in LISP. This would allow the user to tailor the

simulator to his own needs, writing routines for reset, clocking, input, and so on.

Designers often create families of custom chips. In the future, there will be a need for a simulator to accept the circuits from a family of chips along with an interconnection list, and to simulate the entire family at once, making sure the interface between them is correct. This technique may also be useful to the designer of a chip who has not yet completed the final wiring, but who has completed and simulated all the major pieces. He may want to simulate the entire chip by hooking up the individual pieces. This facility would allow final simulation to be performed at the same time as final wiring.

8. How Does All Of This Relate To The Real World?

Even if the work in this thesis were applicable only to designs following the Mead and Conway approach, it would still be useful to a group of university researchers. The hope is that it can be applied to a much larger selection of designs than those used so far. In this chapter, we will touch on some of the problems that might occur with this design verification when other types of designs are used.

Some overall observations can be made before looking at each program in detail. The underlying assumption that was used in this thesis is that designs are expressed in terms of lambda (or fractions of lambda) and that the designs are expressed in terms of rectangles. While most designers use some basic unit of resolution, it is often much smaller than lambda and certainly too small to use as a scale for rasterization. Most design systems support additional primitive shapes in addition to the rectangle. Examples include round flashes (*i.e.*, circles), wires (a locus of points a specified distance from a multi-segment line), and polygons (both convex ones and others). In industry, the assumption of orthogonal geometry does not hold.

The assumption that there are only six layers is not a valid one in industry. Some common additional layers that must be handled include *buried contacts*,[1] a second layer of metal, a second layer of polysilicon, and two or three more layers of ion implantation.[2] None of these extra mask layers add any new concepts. Though the node finder will contain more layers to follow and the circuit file will have more types of transistors, the basic algorithms will remain the same.

When a change is made in technology (*i.e.*, from nMOS to cMOS or bipolar), larger changes

[1] A method of connecting polysilicon to diffusion that does not use metal. This allows metal to be run over poly-diffusion contacts without any interaction.

[2] The different ion implantation masks are used to control the thresholds of the transistors.

will have to be made. While the author has never seen a design in either of these other technologies, the assumption is that the basic transistor finding algorithm will have to be changed. Chang [17] presents an algorithm for performing higher level checks on bipolar devices. He claims that it is possible to design many devices that pass all the design rule checks, but are still incorrect. Perhaps a scheme similar to his can be used to recognize and extract the transistors in bipolar circuits.

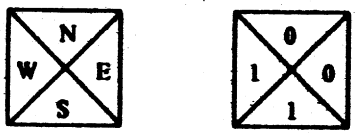
The different design verification tools will be considered one at a time. Any limitations that can be foreseen will be discussed, along with possible solutions to these problems.

8.1 Design rule checking

This algorithm suffers the most when brought to bear on real world problems. The one lambda grid no longer applies. The complexity of the current design rule checking algorithm scales up very poorly when a window larger than four-by-four is used. Also, the design rules used in industry are much more complicated than those stated in this thesis. This complication comes from the need to save space in order to increase yield and, in the long run, decrease costs. Typical rules specify one particular spacing unless some other condition occurs, in which case the spacing can be reduced a little. While industry has whole-chip design rule checkers, they are based on the geometry engine approach, with the Cray 1 as the engine.

Even using the Mead and Conway design rules, the raster scan design rule checker cannot handle diagonal lines. When these are converted to a raster image, an error occurs in either width, spacing, or both. Losleben and Thompson [18] also use a raster scan algorithm for performing topological analysis. Though they restrict their discussion to orthogonal geometries, they present a clever method by which 45^{o} rectangles can be handled. Each "bit" is now represented by four bits, with 0000 representing white and 1111 representing black. The bit pattern 0101 represents a square

in which the lower left is black and the upper right is white.



NSEW 0101

Losleben and Thompson 45° algorithm

The codes have been constructed such that logical operations (such as "and" and "or") have the same results on the codes as on the original single bit. It may be possible to use this algorithm to implement a raster scan design rule checker that uses a small window.

8.2 Node extraction

The granularity of the rasterization should not cause the node extractor any problems. Wires must have some minimum width. If the basic pixel is made somewhat smaller than half the minimum width of a wire, then each wire is sure to be detected by the node extractor.

The effects of differing technologies has already been discussed. Currently, the changes necessary to process cMOS are not considered to be much of a problem, though bipolar might require more work. As more exotic technologies are invented, the hardest part of node extraction will be transistor recognition. As long as transistor information is directly related to the mask layers, this should not be a problem.

8.3 Simulation

The simulator is the program least affected by "real world" designs. The input is still just a file of transistors, and the only difference might be in total size of the file. As designs become larger, the simulator will run slower. Currently, its speed depends on the number of transistors that are changing and the depth of the circuit. Faster simulation algorithms will no doubt be developed, and since simulation presents an opportunity for multiprocessing, there may be some very fast simulators in the future.

Differing technologies may require small changes to the simulator, but no major changes are anticipated. The current simulator [16] is table driven and could be made to read in a set of technology-dependent simulation rules.

8.4 What happens as chips get even larger

The speed of the node extractor is dependent on the size of the chip. As designs get larger, the node extractor slows down. This effect will be offset by faster computers and better algorithms. In addition, the node extracting algorithm is simple enough to be implemented with a small amount of special purpose hardware. At some point, chips may become too large for such a brute force approach. Some of the other methods considered for this thesis, but rejected as too complicated, may be necessary.

The static analyzer will also be slowed down by larger circuits. Since it never took very long to run, however, its speed should not be a factor when compared to the speed of the node extractor and simulator.

The scaling problems of the simulator have already been mentioned. In addition to multiprocessing and better algorithms, extraction of additional higher level functions from the

circuit (memories, registers, alu's) should improve the speed of simulation. Other ideas being considered include the generation of complied code that will perform the simulation and the creation of special purpose hardware that takes advantage of the parallel aspects of simulation.

Designs are going to get larger, but the approach taken in this thesis should be able to keep up with the larger designs for at least the next few years.

9. Conclusions And Directions For Future Research

Most of the ideas contained in this thesis have been implemented, and used to verify over 30 designs. The general reaction of designers is that the time spent running the programs, checking each of the error reports and performing simulation, was time well spent. Few designs emerged unscathed. Many of the errors detected would have prevented the chip from working after manufacture.

9.1 The Scheme79 chip

The chip for which these tools were originally written and for which the most experience has been accumulated is the Scheme79 chip.[19] This chip implements a 32-bit LISP microprocessor, complete with an on-chip evaluator and garbage collector. The chip is 3000 lambda by 2375 lambda and contains 7811 enhancement mode transistors, 1637 depletion mode transistors and 2411 electrically distinct nodes. The circuit for the Scheme chip can be extracted from the mask information in about 5 hours of CPU time on a PDP-11/70. This includes the time necessary to produce a plot that shows all the node numbers which the the simulator will use.

When the designers considered the chip to be finished and ready for fabrication, 11 errors were detected through artwork analysis and simulation. The initial errors were discovered by symbolic naming of the input and output pads and subsequent detection that some pads really belonged to the same node. Next, some named internal nodes were discovered to have the same node numbers. After this, most of the errors were discovered during simulation. By the time the deadline for fabrication came, the simulated Scheme chip had both performed a garbage collection and interpreted a simple LISP program. Though this did not constitute an exhaustive test, it was all that time allowed.

After the Scheme chip was manufactured and returned, its tests were successful; the chip worked. During further testing of the actual chip, two more errors have been discovered. One, a "bug" in the garbage collection algorithm, could have been detected if the particular case had been simulated. The other, a race condition in some of the logic added to an output pad in an attempt to make it latch a signal, was not detected, because the simulator was not designed to detect race conditions. Fortunately, the correct value of this pad can be determined by other means, so that the chip is usable.

9.2 Design errors that are not checked

Other designs have been run through all the checks and have been fabricated, but none have been tested enough to determine if they work. It would be nice if we could be sure that any design that passes all of the tests described in this thesis would work when fabricated. Such a statement cannot yet be made; therefore some mention should be made about the kinds of errors that might slip by all the checks and cause a chip to fail.

The largest area in which no checks are performed is timing. The simulator does not have a good idea of *timing*, nor is there any analysis of *critical paths*, nor checking for *race conditions* or *hazards*. In dynamic circuits, no checking is done for *stale bits*, bits that were not refreshed often enough. More work is needed in these areas.

Another unchecked problem area is power and ground *bus sizing*. When large DC currents flow through small aluminum wires, the aluminum atoms *migrate*. This causes the wire to become even thinner, which increases the migration. Eventually, the wire breaks, causing an open circuit. A scheme has been developed by the author for checking power/ground bus sizing, but it has not yet been implemented.

Further thought is needed in the generation of *test cases* for the simulator. It is hard for the designer to create exhaustive test cases. The simulator can aid in this task by keeping track of nodes which have never changed state and reporting them to the user. He can then try to devise better test cases that would cause those nodes to change state. If the user identifies all state vectors (*i.e.*, PLA feedback terms) to the simulator, an additional piece of useful bookkeeping can be performed. The simulator could keep track of the states that have been visited and report on those which have not.

This concludes the list of currently undetected problems that could lead to non-functional chips. As the relationship between the Mead and Conway design style and the actual analog functions of the chip is better understood, more areas can be checked. At some point, it will not be worth investing more computer time in the extraction or simulation of difficult cases, and the best approach will be actual fabrication and testing.

9.3 Better design tools

If the tools used to design the chip were better, then none of these design verification programs would be necessary. This thesis has attacked the easy problem of *design verification.* The computer-aided creation of the designs is a harder problem. Currently there is little agreement in the field about the right way to do design automation in VLSI. The computer should be able to help the designer control some of the complexity, but the amount of help it can provide in the actual design process remains to be seen.

Although predictions can be made about the operation of design tools of the future, it seems more profitable to look instead at how some of the circuit extraction software can be integrated into an overall design system, both in the present and in the future.

Integrating the current software with a design automation system will have the effect of

"closing many loops". The output of simulation based on circuits extracted from the artwork can be compared with that of functional simulation. Individual cells can be design rule checked, circuit extracted, and simulated as they are created. All these different representations can be made available to the designer. There will still be a final check performed on the whole chip, but the number of errors detected should be very small if all these other checks have been performed along the way.

REFERENCES

1. Mead, Carver and Conway, Lynn, *Introduction to VLSI Systems*, (Reading, Massachusetts: Addison-Wesley Publishing Co., 1980).

2. Shrobe, Howard, Private communication.

3. Hsueh, Min-Yu, "Symbolic Layout and Compaction of Integrated Circuits," Electronics Research Laboratory Memorandum No. UCB/ERL M79/80, December 10, 1979, University of California, Berkeley.

4. Nagel, L. W. and Pederson, D. O., "SPICE (Simulation Program with Integrated Circuit Emphasis)," Electronics Research Laboratory Memorandum No. ERL-M382, April 12, 1973, University of Calafornia, Berkeley.

5. Wilcox, P., "Digital Logic Simulation at the Gate and Functional Level," *Proceedings 16th Design Automation Conference*, June, 1979, San Diego, California, pp. 242-248.

6. Sherwood, W., "A Hybrid Scheduling Technique for Hierarchical Logic Simulators," *Proceedings 16th Design Automation Conference*, June, 1979, San Diego, California, pp. 249-254.

7. Navabi, Z. and Hill, F. J., "Efficient Simulation of AHPL," *Proceedings 16th Design Automation Conference*, June, 1979, San Diego, California, pp. 255-262.

8. Giambiasi, N., Miara, A., and Muriach, D., "SILOG: A Practical Tool for Large Digital Network Simulation," *Proceedings 16th Design Automation Conference*, June, 1979, San Diego, California, pp. 263-271.

9. Hill, D. and vanCleemput, W., "SABLE: A Tool for Generating Structured, Multi-Level Simulations," *Proceedings 16th Design Automation Conference*, June, 1979, San Diego, California, pp. 272-279.

10. McCaw, C. R., "Unified Shapes Checker -- A Checking Tool for LSI," *Proceedings 16th Design Automation Conference*, San Diego, California,