2. IC Design Tools

The key to fast-turnaround integrated circuit design is the emergence of powerful tools which make it possible for the designer to be assisted by computers in an effective manner. To enter his ideas into the design system, suitable interactive graphics terminals have become available. As another option, layout languages are being developed which one day will provide a means to enter designs in a symbolic manner at a rather high level of abstraction. Displays and plots of the various mask levels of an IC are important to close the interactive loop between man and machine, and are indispensable in the final debugging phase. There are additional ways in which the computer can assist the designer. A "mechanical" check for design rule violations helps eliminate potential problem spots in the fabrication of IC's. Circuit and logic simulation can be used to predict the performance of critical parts of the circuit and to test the correctness and the timing on a larger scale, respectively. Further, the computer can be a tremendous help in the management of the large amount of information associated with an IC design.

This chapter provides an overview of the types of tools available and under development. It also tries to bring out the point that, with only little expense and effort, a minimal set of tools can be acquired that makes IC design possible.

2.1 Automated Design Aids

[section contributed by Stephen Trimberger, UC Irvine]

The role of a design aid is to reduce errors and design time. Therefore, a design aid should first attack tedious and error-prone aspects of design. In addition, the design aid should present the design to the designer so that he can catch errors. Rather than have a computer take over the entire design task, the design effort should be a cooperative one between the designer and the design aid, in which the design aid relieves the tedious and exacting chores, giving the designer more time to do what he does best: design.

2.1.1 Plotting

A necessary design aid in any design environment is hardcopy output. *Checkplots* are absolutely essential to reduce the number of errors in IC layouts. They enable the designer to graphically check alignment and positioning to catch design rule errors as well as typographical and logical errors in the design. Such checks cannot be made from examination of CIF code (Caltech Intermediate Form -- see chapter 4 of [Mead 1978]). See section 2.3 for more on checking.

Good checkplots must distinguish between mask layers and show their overlaps. Checkplots with filled-in rectangles clearly show the overlap of layers in the circuit. Filled-in color checkplots are ideal because of their high information density, but plotters of this type are new and fairly expensive. Color line-drawing plotters are nearly as good, especially if the layers are "hatched" in the appropriate color to show the interior of boxes. Icarus's *stipples* [Fairbairn 1978], gray-pattern shading, or differently hatched rectangles can be used in a black-and-white environment to distinguish between layers.

Usable checkplots of low resolution can be obtained from an ordinary lineprinter, using different characters to represent different layers and separate characters or overstruck characters to show overlapping layers [Gibson 1976, Larsen 1978]. Storage-tube displays (sometimes with hardcopy units) are often used to display IC designs, but in a complicated design it is difficult to visualize overlapped areas from outlines. Their advantages are fast response, relatively small expense and short development time.

2.1.2 Implementation of a Basic Design Aid System

A minimum IC design system might consist of a text editor to enter a design in CIF code, and checkplot-generation routines to view the design on the lineprinter. Although this may sound tedious to those who have used high-powered CAD systems, this method of IC layout has been used with acceptable results. However, for just a small investment of time, a much nicer layout system can be had.

2.1.3 Implementation of a Better Design Aid System

CIF code was not intended to be used as an IC design language, and so it is not oriented to a human operator. A more efficient, human-oriented method is needed to enter designs into the CAD system. There are essentially two ways to build a more efficient system, each of which addresses a different class of problems in IC design. A more powerful *design language* reduces problems of relative positioning of objects and an *interactive graphic system* reduces problems of coordinate entry and editing. The implementation of a design language may initially appear to be a monumental task. However, if approached in the correct manner, for example by embedding it in an existing language, it can be developed in a few months. This effort is well spent, since such a language can serve as an extremely powerful and satisfying design tool. An interactive graphic system requires more hardware and more complex software, but such a system can reduce the circuit design time greatly, since even the initial sketches are made directly within the CAD system.

2.1.4 Layout Language

Many errors in IC design stem from mis-positioning of objects due to the movement of adjacent objects. The mis-positioning problem could be avoided if there were facilities in the language for *parameterization* of objects, for example, basing the position or size of one object on the position or size of another. This is similar to passing parameters to a procedure in a programming language. In addition, it would be nice to specify a shift register by the number of bits, or a PLA by its program. This requires loops and conditional statements in the layout language; the addition of such features makes the layout language look a lot like a general programming language.

An easy way to get a powerful layout language is to implement the CIF commands of drawing a box, wire, symbol and so forth as procedure calls in your favorite programming language. Then,

you can use the entire power of FORTRAN, PASCAL, SIMULA or WHATEVER to do the conditionals, the loops, the parameterization of the geometry and any other special constructs that might be necessary or convenient. Several systems of this type have been built, and their advantages include short lead time to a very powerful system as well as infinite expandability and an incredible richness of language [Locanthi 1978]. This type of language is recommended as an initial design system. More specifics on layout languages are found in section 2.2.

2.1.5 Graphic Input

There are two ways to get graphic designs like IC layouts into a computer without typing numbers, by *digitizing* hand drawings and by drawing the design directly into the computer with an interactive graphic system. Digitizing starts with a clean scale drawing of the layout, which is entered by an operator using a digitizing table or camera; it is less prone to errors than typing numbers, but leaves the tedious task of editing the design to the layout person with pencil and eraser. These tasks are performed more easily with an interactive graphic system.

A special purpose interactive graphic editor, engineered for the special needs of IC layout, is a most effective design tool. Ideally it should be as easy to use as paper and pencil, yet directly produce precise layouts on a specified grid. An interactive graphic editor enables the designer to experiment with a number of possible designs quickly, shortening the design time immensely [Fairbairn 1978].

Unfortunately, the ideal system is not yet commercially available, and to develop such a system on your own is a substantial task. The necessary hardware includes a *pointing device* with which to draw the layout, and a *graphic display* to show the design as it is being entered, allowing the designer to instantly correct any errors that occur. A discussion of pointing devices and graphic displays can be found in [Newman 1973]. Even with a good display and pointing device, it is difficult, for example, to route long wires around a complicated design.

2.1.6 Considerations for an Advanced IC Design System

An unaided graphic system cannot easily handle parameterization and conditional placements. On the other hand, layout languages have the problem of tedious cell layout. Clearly, the ultimate IC design system should allow both interactive graphics and programmatical positioning of objects. An ideal system would have a near instantaneous response from a change in the layout program to a change on the layout graphics and vice versa, and have both representations interactively

displayed on a high-resolution video display.

An important idea in design systems is the concept of *hierarchical* or *structured design* -- dividing the given problem into more easily handled subproblems. This cuts down the amount of information the human designer must handle at one time in order to solve the design problem. In IC design, the elements of the hierarchy are called *cells* or *symbols*. An *instance*, or "use" of a cell can be embedded within another cell, meaning that the contents of the cell are supposed to be inserted into the design at that point (see section 2.2). Instances of cells are embedded in other cells much the same way procedure calls are embedded within other procedures in a programming language. This hierarchical design guarantees that all instances of a cell are correct, provided the original cell is correct.

Multiple *representations* allow the designer to include in the description of a cell all the information relevant to that cell, enabling him to describe cells graphically or programmatically or any one of a number of different ways. Thus an integrated circuit can be viewed as a mask layout, a stick diagram, a functional description, an electronic circuit, text documentation, and so forth. It is important to keep this information together because no one piece adequately describes the cell.

Representations can be generated from one another, thereby ensuring that the circuit that was sent to the circuit simulation program, for example, was indeed the same circuit that appeared on the mask. It would aid the designer greatly if he could visualize the cell with all its representations as a unit, but it adds tremendously to the expandability of the system if each representation data piece is implemented independently of the other representations because we cannot now foresee what design concepts will be available in the future. For this reason, the design database should be flexible enough to accommodate new, as yet unspecified, representations.

2.1.7 Conclusions

With the advent of VLSI, our circuits will have the capability of being orders of magnitude more complex than they are now. The use of cells and instances will help ease the complexity with which the individual must cope. Representations will enable us to keep all the relevant information together. The computer will ease the task of dealing with this complexity, handling the enormous housekeeping chores and organizing the overall design effort.

2.2 IC Layout Languages

[section contributed by Maureen Stone, Xerox ASD]

The most desirable format of a language that describes IC mask layouts depends strongly on the point in the IC design process where a description of the IC is needed. At the end, in the mask generation process, the format of the description is entirely determined by the needs of the pattern generator employed. At an intermediate level, it is most desirable to have a description in a form that is most suitable for easy conversion into the many different formats needed by different output devices. An example of such a language is Caltech Intermediate Form (CIF), described in [Mead 1978]. For the original creation of an IC design, the human engineering aspect is most important. That is, the language description should reflect the design process. Repetitive and redundant information should be compressed, and the syntax should seem straightforward to the designer.

This section will describe how languages can be used in IC design, what constitutes a basic set of functions for a simple description language and how such systems should be organized. The last section will discuss more advanced layout languages, especially in reference to using the power of programming languages as a design tool. Appendix D contains a description of ICLIC, a layout language that was originally developed at Caltech this spring (1978). This language and the design examples presented in the Appendix will be referenced throughout this section.

2.2.1 Designing with a Layout Language

The issue of describing a mask can be examined in from two different points of view. At the lowest level is a geometric description of each layer. The shape and position of each element is described with respect to some coordinate system. The design process is then just a matter of digitizing the layout or typing in the coordinates for each shape. However, a layout language should not be just a digitizer in text form. It should, ideally, approximate the way the designer thinks about the layout, in terms of its function and its contraints. Even very simple languages can be organized in a manner that is more descriptive of a design than just its geometry.

The process of using a language in a CAD system involves the following steps:

> First, the design is partitioned into cells and subcells. The more the layout is partitioned into repetitive modules, the less language it will take to describe it.

Second, each cell is sketched, or drawn to scale as much as is necessary. At this point the coordinate system for the layout will have to be defined. Critical points and distances are defined and labeled. In general, the more primitive the language, the more will have to be drawn to scale.

Third, for each section the description is entered into the system. Some sort of graphical output must be produced to verify the correctness of the description. Corrections are made to the source file until the cell is completed. Lower level cells are then combined to describe upper level cells until the whole chip is defined.

The response or turnaround of the CAD system will determine how the designer is going to use it. For example, if the facilities are batch mode processing with a four hour turnaround, the designer will be likely to draw most of the circuit to scale and painstakingly check the source code. The other extreme could be a color interactive graphics terminal with very fast translation from input to display.

2.2.2 Basic Features of a Layout Language

This section describes the features of a simple layout language and how they might be organized. An example of such a language is in the first 3 chapters of the ICLIC description given in Appendix D.

Most simple layouts use only a few basic shapes, predominantly rectangles and wires. These can be described on a unit square coordinate system or grid and constrained to right and 45 degree angles. Therefore, a layout language can begin with a description of these shapes, and a way to designate their layers.

A simple rectangle or *box* is a rectangular area with the sides aligned parallel to the axes. It can be specified by two opposing corner points, or by one point (e.g. center or corner) with a width and a height.

A *wire* is a track of uniform width defined by its center line and width. The path of the wire is defined as a list of coordinates. In general, a shorthand notation is used for paths that only make right angle turns. That is, only one coordinate changes at once so only that change is specified.

For example: X=x1, Y=y1 $ Y=y2 $ X=x2 $ X=x3,Y=y3 $

The "$" means make new wire section. A more powerful syntax uses an infix notation for points and has a symbol for the current X and Y.

For example: `x1#y1 ; .#y2 ; x2#. ; x3#y3`

The "." means current coordinate, and ";" is a delimiter. Having a notation for the current coordinate means that it is possible to compute with it, thus leading to relative coordinates.

For example: `x1#y1 ; .#.+d1 ; .-d2#. ; x3#y3.`

The notation ".+d1" means use the current coordinate plus the distance d1 for the new coordinate.

The mask layer to which a particular shape belongs is specified by some mnemonic. ICLIC uses color codes, such as red for polysilicon, green for diffusion, as suggested in [Mead 1978]. Some languages associate a layer specification with each item. Others make it a global switch that affects all subsequent objects, as does CIF.

Attempting to define a layout with just these simple primitives will result in an explosion of data. Therefore, it is essential to have some mechanism for grouping and reusing sets of shapes. Such a collection of shapes is often called a *symbol* or *cell*. A layout language needs some way of defining and using symbols. It should be possible to nest symbol *calls*, and to use symbol calls in definitions of other symbols. However, it is not necessary to be able to nest symbol *definitions*.

Using symbols implies some way to position each instance of the symbol. In general, the call will map the *symbol origin* to the current X,Y. A rectangular array of symbols is such a basic layout feature that some method for easily generating it should be a part of any layout language. The syntax of such a construct needs only the starting position, the number of symbols in the X and Y directions, and the spacing in the X and Y directions. Such a construct is described in section 3.5 of the ICLIC manual in Appendix D.

Besides being positioned, symbols may also be *transformed*. Standard transformations are: *scaling*, *mirroring*, *translation*, and *rotation* (see [Newman 1973] for a full discussion of transformations). For a simple layout language, translation, rotations by increments of 90 degrees, and mirroring about the axes are sufficient. Scaling, which is changing the size of a symbol, and rotation by arbitrary angles are rarely used.

The use of transformations brings up the issue of nesting transformations. The order in which transformations are performed is important, and even the simple case of combining a rotation with a translation can produce very different results depending on the order in which the operations are executed. Therefore, the syntax for calling transformation routines should be very clear about the order in which the functions are performed.

Besides the issue of the user interface, there are many difficulties involved with the definition of general transformations for integrated circuits. One example is the problem of how to adjust the results of a transformation back to a grid (rounding of coordinates) without causing unintendend breaks or overlaps. See the description of CIF in [Mead 1978] for a more complete discussion of the problems of transformations in integrated circuit design.

2.2.3 Variables, Parameters, and Relative Positioning

A *variable* is simply a name which can be assigned a value. The benefits resulting from the use of variables within an IC design language are equivalent to those obtained in a general purpose language. That is, once a name is used for a value, all instances of that value can be changed simply by changing the assignment to that variable. Furthermore, a name can be descriptive, such as "InverterCenterX". It is much easier to make changes and find errors in a list of descriptive names than among a mass of anonymous numbers.

A symbol whose definition is controlled by a set of named values is said to be *parameterized*, and the names are said to be its *parameters*. A simple example of this is a symbol which uses variables for size and positioning information. Such a symbol can be changed by assigning different values to the variables used in its description. In a language which supports more advanced programming constructs, such as subroutines, loops and conditionals, parameters can be used more extensively to control the properties of the symbol. For example, the parameters of a PLA subroutine could describe its size and function.

The use of *relative coordinates* involves defining symbol parameters with respect to other elements in the cell, instead of with respect to the absolute coordinate system for the layout. Changes made in a symbol which is defined in this manner can ripple through the layout, keeping design rules intact.

For example: RW({OUT1 ; .+RWFRMG#. ; .#IN2.Y ; IN2})

This example draws a red (poly) wire from the point OUT1 to the point IN2. The path followed is: horizontal from OUT1 to the minimum distance a red wire can be from a green area; vertical to the Y coordinate of IN2; horizontal to IN2. This wire will remain connected and running at the minimum design rule distance from the first device whenever OUT1 and IN2 are changed.

2.2.4 Program Organization

Methods of organizing a language description for a design are similar to techniques used in general purpose programming languages. They also relate strongly to methods discussed in reference to the design process as a whole. The following points will be summarized here: documentation, modular design, parameterization, defaults and conventions.

Documentation includes not only comments on the source code, but the use of descriptive names for variables. All programs should be thoroughly documented, of course.

Modular design is the process of coding and debugging each cell separately, then combining the working cells to make larger ones, as has been emphasized throughout this document. By making large programs out of a collection of small ones, not only is the design cleaner, but the process of coding, executing and debugging the programs is simplified. Uniformity reduces the complexity, and hence the errors, in a layout. Therefore, using standard symbols for circuit elements is highly recommended. Some symbols, such as contacts, are common enough to be defined in the language. Other symbols could be kept in library files, see Appendix E.

The advantages of parameterization can be broken into two main areas. First, changes to a cell can be more easily made; for example, if the parameters are defined as relative coordinates, the cell can be somewhat self-adjusting with respect to small changes. Second, the same code can be used to produce similar, as opposed to identical, cells. Therefore, there is less code to debug, reducing errors.

One of the initial steps in writing any program is defining *defaults* and conventions for variable names and parameter values. An example of this is the convention in ICLIC of specifying all design rule distances as cFRMc, where the c is replaced by a layer mnemonic. For example, RFRMG or GFRMR is the minimum design rule distance between red and green areas. The default value is 1 lambda (3 microns in 1978). Defaults and conventions should be documented at the start of the program, and adhered to throughout. It is useful for the language to contain a set of default parameters and naming conventions for wire widths, wire spacings, minimum design rule

distances, etc.

2.2.5 *Advanced Layout Languages*

There are two ways to make extensions to the basic language concepts outlined in the previous sections. One is to continue to treat the design process as a set of calls to pre-defined functions, and to extend the language by increasing the complexity of the functions. For example, ICLIC has a function for defining wires which run on more than one layer which automatically inserts the contacts between layers. The second, more powerful method involves treating IC design as a programming process. Therefore, the design language needs to be extended to include subroutines, loops, and conditionals. The issues then become those of conventional language design.

A good way to implement a layout language is to define a set of procedure calls within an existing language. The elegance of the finished set will depend somewhat on the base language used, but it seems safe to say that something akin to CIF with variables could be implemented in any language that allows dynamic storage allocation. A language like ICLIC depends heavily on the concepts of concatenating symbols together, and the ability to compute relative coordinates symbolically from the current coordinate. ICLIC was implemented as a set of procedures in ICL [Ayres 1978] which already supported these concepts. A similar system has been implemented at Xerox PARC in SMALLTALK as a programming example for a forthcoming book. A different system, which contained a basic layout language plus some automatic wire routing routines, has been implemented in SIMULA at Caltech [Locanthi 1978].

There are a number of advantages to basing a layout language on an existing language. Development time, compared to writing a compiler from scratch, is much reduced, and the programming environment is one that is already familiar to the user community. In addition, the resulting layout language has the full power of the underlying language.

The concept of a language description provides a means for a functional description of the circuit being designed. At the layout level, one could imagine the process of developing cells that are passed a set of input specifications and return the layout plus a set of output specifications. Such modules could then be connected by running wires from an output definition to an input definition, either by hand or by an automatic wire routing system. The point being made here is that a program which describes a design element can be extended to provide a much richer description than just the data for the mask. Therefore this description can be used for a variety of

purposes, such as wire routing, design rule checkers, and simulation.

2.2.6 Conclusions

While the use of a set of basic functions to describe a layout is straightforward, there are important issues with regard to the design and use of such languages. These issues have been discussed in sections 2.2.1-2.2.4. Even such simple systems can be very useful for design. However, the real power of a language description comes not from the expansion to fancier functions, but from invoking the power of a general purpose programming language as a design tool.

2.3 Checking Your Design

[section contributed by Wayne Wilner, Xerox PARC]

Integrated circuit design requires near-perfection. Certain flaws, such as a short between two clock lines, can render the whole chip useless. Checking your design adds a week of tedium to your project, but without that week, four months can go down the drain (to say nothing of cost).

Checking can be done by eye and by computer. Since sight-checking is available to all, we'll discuss it first. Actually, sight-checking is superior to automatic checking in several ways.

2.3.1 Checking by the designer

Many fatal errors in a design do not exhibit themselves as violations of design rules. Consider logic errors. Consider state machines which are initialized to terminal states. Consider transistors which are wired incorrectly, but within the design rules. Consider an array of cells which are supposed to abut and do not; if the space between them is larger than the minimum spacing for all layers, design rules may be observed while the array is grossly in error. Consider the placement and continuity of busses. These errors are representative of flaws for which the designer is singularly responsible.

Many of these can be spotted on checkplots of relevant subsets of layers. A plot of metal and contacts can reveal errors in continuity which would otherwise be lost in the details of a full plot. A plot of poly, diffusion, contacts and implants enables one to check their all-important overlap. The effectiveness of this technique is very high when the plots are large, clean, and of high contrast.

A plot of each individual layer should be sight-checked by someone besides the designer. It's amazing how many design errors look odd on such plots, even to those who don't know the circuit.

2.3.2 Checking by computer

Some flaws are violations of design rules. Two basic tactics can be employed to avoid them: (1) the use of programs which check the final mask descriptions for certain violations, and (2) the use of programs which generate layouts in a manner that guarantees that there are no violations. At present, there are no programs which generate layouts completely. An intermediate step in that direction is the "Sticks" approach [Williams 1977] which starts with a grossly-spaced circuit which is then trimmed mechanically to minimum spacing by the CAD system.

2.3.3 Error-checking programs

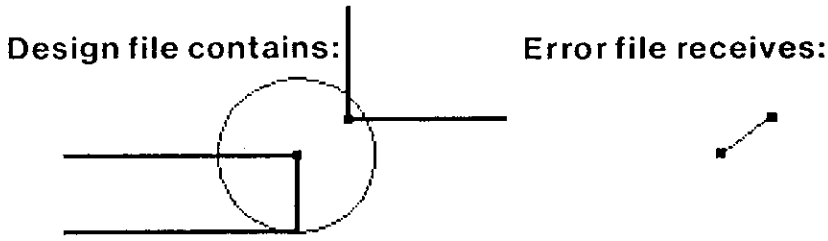
Error-checking programs embody the rules for a particular process and examine pattern generation tapes for violations, reporting each instance in terms of coordinates or patterns, along with the nature of the violation. Design rules typically assign minimum distances to:

dimensions of features, such as breadth of runs or size of contact cuts;

spacing between features in the same layer, such as distance between runs;

spacing between features in different layers, such as overlap of metal and contact windows.

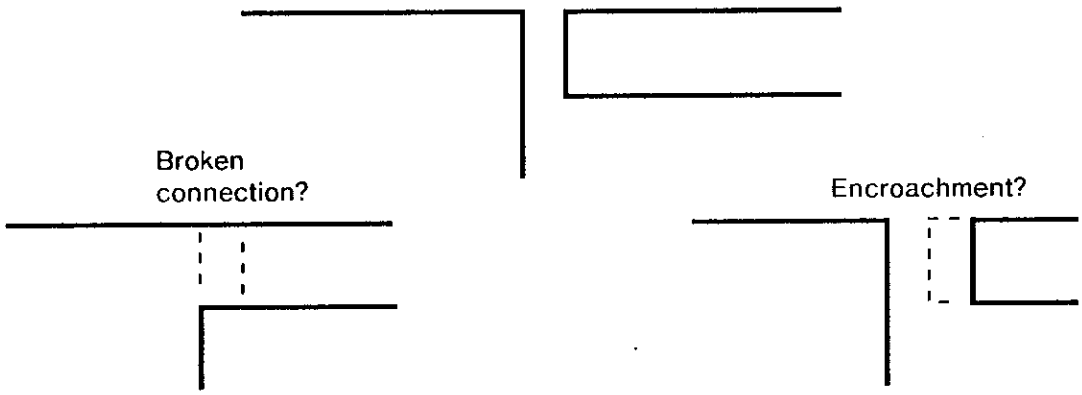
For example, suppose unconnected areas of polysilicon must be 2λ apart. In the diagram below, a circle of radius 2λ centered at the upper right corner of the left-hand area reveals that the right-hand area is too close.



This design rule violation may be reported in terms of a line segment, that is, two points, one at the periphery of each area, and their (insufficient) separation. It is a non-trivial problem to present violations to the designer in the most convenient way.

2.3.4 Inherent limitations

An inherent limitation of design rules comes from their pertaining solely to the lowest level of detail. Consider the following diagram. Two areas are separated by less than their minimum spacing.



It is clearly a design rule violation, but is it a broken connection or is it an encroachment? The designer will have to decide and fix it appropriately.

Checking for design-rule violations appears to be a problem of geometry, of a well-defined, computationally-tractable nature. This is an illusion. Checking design rules requires so many computations that the problem is really one of managing main and secondary storage, in other words, an operating system problem, not well-defined, and different for each individual computer.

2.3.5 Design rule checking in the context of structured design

Another point of view on design rule checking is that of structured design. Using structured design, most projects are constructed from a few basic cells which are very simple. Their simplicity makes sight-checking design rules adequate.

Automatic checking is quite useful, still, for several situations. Assembling systems from cells introduces errors in cell placement. Cells which interconnect must abut, not overlap or be separated. Cells which overlap may introduce design rule violations, even though the cells themselves are correct. Cells which provide alternate interconnections for a given signal must dispose of all unused connections. Wide-ranging interconnections between cells are often hard to scan thoroughly and show up small and indistinct on checkplots which span their extent.

2.3.6 Parameterizable design rule checking

In laboratories which experiment with different processes, the critical distances may vary from month to month. In experiments with custom circuits, the objective may be to find how exceptions to the design rules can be exploited. Therefore, while the types of rules may be rigidly bound into a checking program, specific distances should not be.

Is it feasible to make a design rule checker in which the design rules are parameters? A fundamental problem is that the nature of design rules varies. Some involve unconditional distances: "metal runs must be at least seven microns wide." (Of course, verifying this can be extremely difficult.) Others involve conditions: "polysilicon must extend at least four microns beyond diffusion, unless the gate dimensions are small, where six microns are required". It is an unsolved problem to mechanically create a program which can efficiently verify geometrical constraints of varying nature. It would seem harder to mechanically generate checkers than to mechanically generate layouts. The latter is clearly preferable because of the greater range of errors which are detected or eliminated.

2.3.7 Summary

Unless circuits are generated mechanically, they must be checked thoroughly. Sight-checking of critical distances is astonishingly effective when done on a large, clean, high-contrast plot, due to the pattern-recognition power of the brain. Mechanical checking can serve as a further check; it is limited by machine resources and imprecise or insufficient descriptions of the rules.

2.4 Simulation as an IC Design Tool

[section contributed by Richard Lyon, Xerox PARC]

Simulation is a design technique widely used in a variety of engineering disciplines. When it is too difficult to verify the correctness of a design by inspection, by proof, or by test, simulation may help. Simulation allows the designer to test a design before building it, by modelling in detail the components from which the design is built, and by computing their interactions under various conditions. Simulation is useful at many levels in integrated circuit and system design; system-level, register-transfer-level, logic-level, and circuit-level simulators are useful at various stages of the IC design process. A related activity is the design of IC fabrication processes, which can benefit from process simulation; the simulation of process variations may become more important as VLSI approaches the physical limits of device sizes, where the set of devices used by the system designer must be carefully matched to the technology.

Unfortunately, not many generally useful simulators are readily available. Even when such a program is available to run on your computer, the problem of preparing data in a form suitable to the simulator can be formidable. It is easy to write a register-transfer-level simulator, for example, but the hard part that makes it useful is to provide an automatic link from the design language to the simulator input language. There is not yet enough commonality of design methods in the digital system design field to result in wide availability of such a program. In the circuit design field, on the other hand, the method of design has traditionally been standardized to drawing by hand on paper the interconnection of standard types of lumped circuit elements. From here it is logical to assume hand translation to the language of a circuit simulator. For this reason, circuit simulators have been developed in standard languages (Fortran IV) and are widely available. Two such simulators, somewhat tailored for IC simulation, are SPICE from U. C. Berkeley, and MSINC from Stanford; their input languages are similar, and one example should serve to illustrate both.

Circuit simulation can, be very useful to the integrated circuit/system designer if it is applied to those problems that require it, but should not be relied on to verify the correctness of a complicated system design. In digital system design with a consistent design philosophy, it is usually possible to identify the critical parts of the design (for example the longest chain of pass transistors, the new RAM cell, or the node with the highest fanout); in this way, critical parts can be identified for simulation (see [Mead 1978] chapters 1 and 7 for information on critical timing; see chapter 4 for more on simulation and testing). Of course, even simulation will not verify that the design will run fast enough if the simulation parameters and models do not realistically reflect the process used to make the circuit.

As an example, we have simulated the output pad driver called *PadOut*, which was designed in *Icarus* (Integrated Circuit ARtwork Utility System, an interactive layout design system) according to the Mead and Conway design rules, with lambda equal to 3 microns. This is a driver intended to interface NMOS chips to other popular logic families, at speeds and voltages comparable to TTL. It uses push-pull enhancement-mode output drivers, driven in turn by *super-buffers* (see [Mead 1978] chapter 1). The fanouts are generally somewhat higher than the theoretical optimum of *e*, to reduce space and power at the expense of speed. Figure 2.4.1 is the Icarus layout picture of PadOut; notice that the output transistors are both wrapped around the pad. The schematic diagram is shown in Figure 2.4.2; it includes node numbers and element names which are needed for translation to the simulator input language.

The simulator SPICE was used at Xerox PARC, on the MAXC2 computer, which has no floating-point hardware; therefore, the execution of the Fortran program was blindingly slow. Figure 2.4.3 shows the *input deck*, an ASCII text file. The SPICE program, like most widely available programs, was written for the card-reader/line-printer/batch-computing environment which is found at the typical university computing center. Therefore, be careful of input formats; only 72 columns of 80-column cards are used -- long lines use continuation marks in column 1, as in Fortran. The documentation is sparse, but keep in mind that you should not do anything you could not do on a keypunch, such as lower case letters. See *User's Guide to SPICE* by E. Cohen and D. O. Pederson, from U. C. Berkeley Dept. of Electrical Engineering and Computer Science.

In the input listing, each line is called a *card*. The first line is the *title card*, and lines starting with * are *comment cards*. Each *element card* names a component (the first letter of the name determines the element type, such as M for MOSFET), tells what nodes it is connected to (in order, such as drain, gate, source, substrate), and gives a few parameters (such as width and length in centimeters). There are also *model cards* and *control cards*, which will not be described here, but can be seen in the listing.

We have described in the element cards the circuit of Figure 2.4.2 (some of the parameters are estimates, such as AS and AD, areas of source and drain). The first inverter is not part of PadOut, but represents a typical signal source, which is in turn driven by a 3.5 volt, 20 Mhz square wave generator with 2 nsec rise and fall times.

The output file produced by SPICE from the input shown was too long to include here. The most interesting part of it is shown in Figure 2.4.4, the graph of the time response of the various nodes, which is plotted line-printer style by typing the node numbers in appropriate columns. To make it

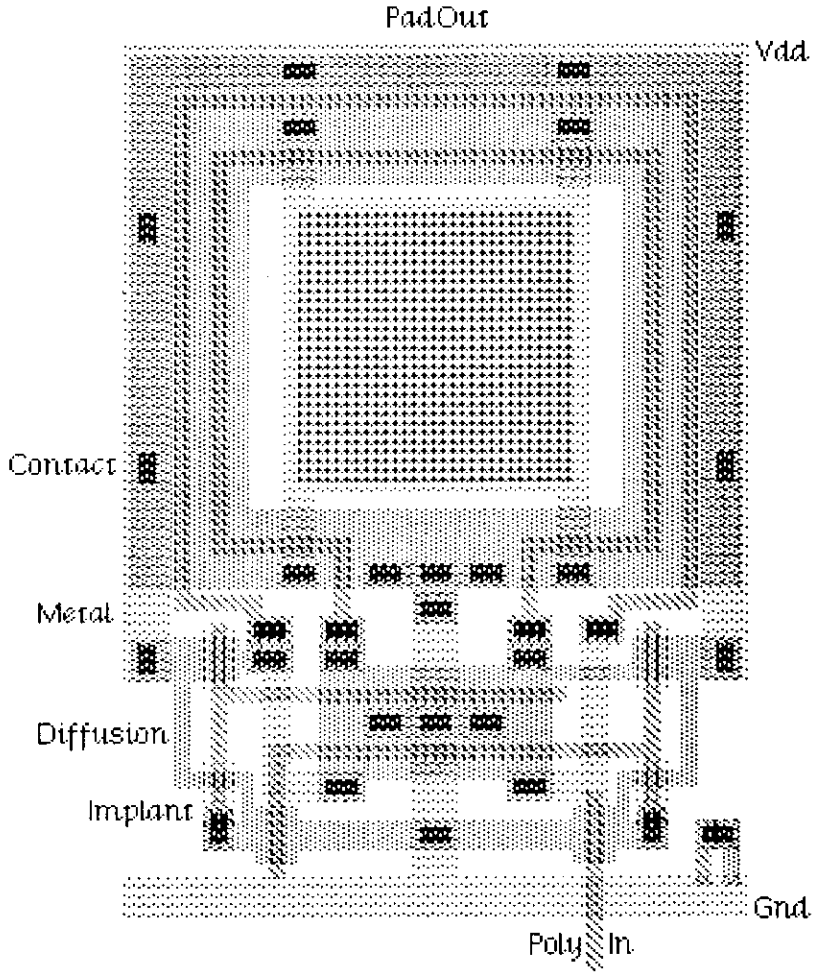


Figure 2.4.1. Icarus Layout of "PadOut"

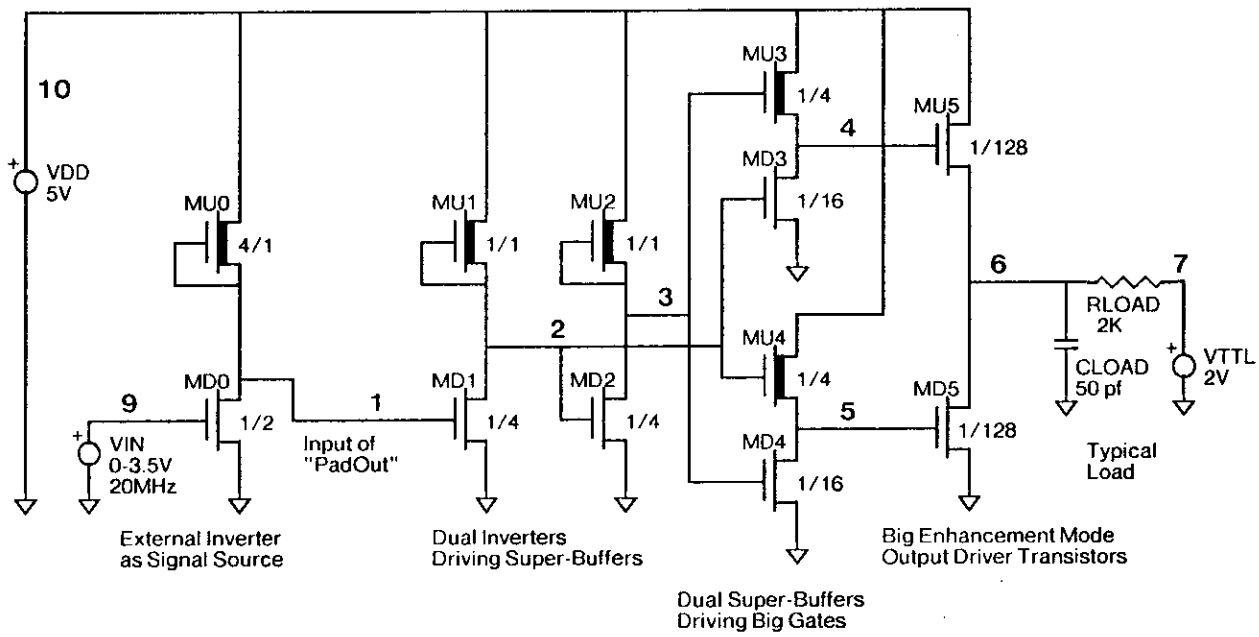


Figure 2.4.2.

Diagram of "PadOut" Output Pad Driver with element names and node numbers for simulation with SPICE.

```
PAD DRIVER SIMULATION
* RF LYON -- JULY 13, 1978
*
VDD 10 0 DC 5VOLTS
VTTL 7 0 DC 2VOLTS
VIN 9 0 PULSE 3.5VOLTS 0VOLTS 2NS 2NS 2NS 23NS 50NS
*
MD0  1 9 0 0 ENH W=12E-4 L=06E-4 AS=144E-8 AD=144E-8
MU0 10 1 1 0 DEP W=06E-4 L=24E-4 AS=144E-8 AD=144E-8
MD1  2 1 0 0 ENH W=24E-4 L=06E-4 AS=144E-8 AD=144E-8
MU1 10 2 2 0 DEP W=06E-4 L=06E-4 AS=144E-8 AD=144E-8
MD2  3 2 0 0 ENH W=24E-4 L=06E-4 AS=144E-8 AD=144E-8
MU2 10 3 3 0 DEP W=06E-4 L=06E-4 AS=144E-8 AD=144E-8
MD3  4 2 0 0 ENH W=96E-4 L=06E-4 AS=600E-8 AD=600E-8
MU3 10 3 4 0 DEP W=24E-4 L=06E-4 AS=144E-8 AD=144E-8
MD4  5 3 0 0 ENH W=96E-4 L=06E-4 AS=600E-8 AD=600E-8
MU4 10 2 5 0 DEP W=24E-4 L=06E-4 AS=144E-8 AD=144E-8
MD5  6 5 0 0 ENH W=768E-4 L=6E-4 AS=4000E-8 AD=4000E-8
MU5 10 4 6 0 ENH W=768E-4 L=6E-4 AS=4000E-8 AD=4000E-8
CLOAD  6 0   50P
RLOAD  6 7   2K
*
.MODEL ENH NMOS (NGATE=1E20 TPS=1 XJ=1E-4
+ CGD=4E-12 CGS=4E-12 CGB=2E-12 TOX=95E-7
+ NSS=-22E10 NSUB=8E14 )
.MODEL DEP NMOS (NGATE=1E20 TPS=1 XJ=1E-4
+ CGD=4E-12 CGS=4E-12 CGB=2E-12 TOX=95E-7
+ NSS=80E10 NSUB=8E14 )
*
.TRAN 1.0NS 80NS
.PLOT TRAN V(1) V(2) V(3) V(4) V(5) V(6) (0,8)
.WIDTH OUT=72
.END
```

Figure 2.4.3. SPICE Input Deck for *PadOut*

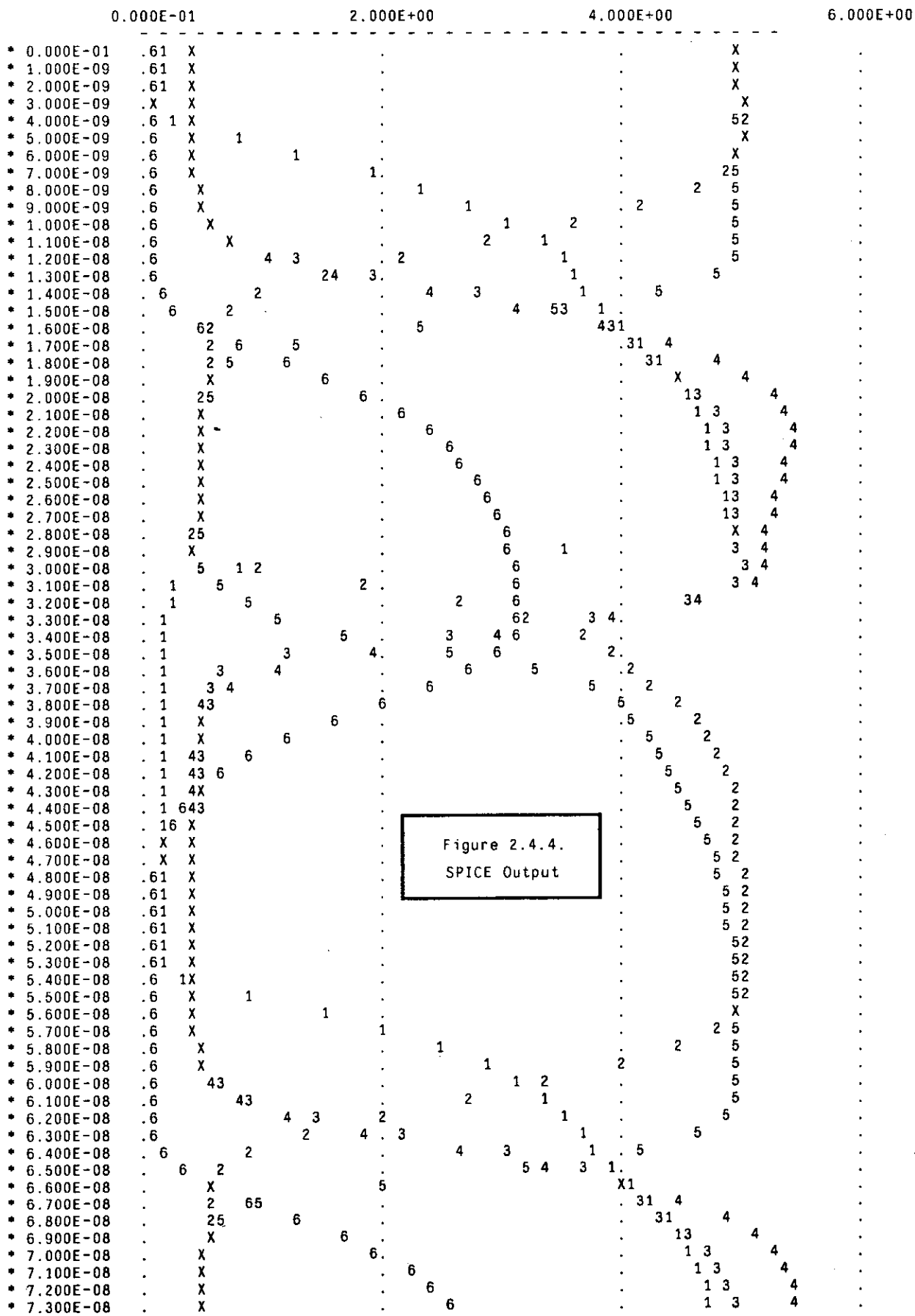


Figure 2.4.4.
SPICE Output

readable, take a bunch of colored markers and draw in the curves for the nodes of interest. You will see that the response from node 1 to node 6 is noninverting, with t_{PLH} of 13 nsec and t_{PHL} of 9 nsec, measured at a 2 volt threshold (or more nearly symmetrical at 11 nsec if measured somewhere below 1 volt).

Is PadOut really this fast? Probably not on most processes; the model cards used here have estimates of the Spice model parameters which were felt to be realistic, but which gave results that are probably too optimistic for most typical 1978 processes. The inverter-pair delay from node 1 to node 3 is seen to be 6 nsec, where the inverter ratios are $k=4$ and the fanouts are $f=5$ (actually 6 for the first inverter). The delay estimate according to [Mead 1978] is then $(k+1)f\tau = 25\tau = 6$ nsec, so we may conclude that we have simulated a process with $\tau = 0.24$ nsec (transit time), which certainly is optimistic. The actual performance of PadOut will have to be determined by test, and will depend on where it is fabricated; some lines would be three times slower than this simulation.

IC designers have relied on simulation as a design tool for years. When the performance of a part being designed is critical (as is typical in manufacturing for sale), and the production/test turnaround is slow (also typical in the IC manufacturing business), circuit simulation is a necessity. However, in the preliminary topological design phase, circuit simulation is not needed; and if turnaround is fast, measurement may be a better way to determine performance than simulation is. Thus, we are now at the point of being able to design complicated *digital* systems without the aid of circuit simulation, by following strict design conventions; however, circuit simulation is still useful for the analog and interface aspects of system design, and is a valuable aid in understanding circuit behavior. We encourage the use of circuit simulation where it is appropriate. We also encourage the development and use of higher-level simulation tools to aid the design of digital systems.