VINCENT AND THE SPLIT CELL

Bart J. Geraci

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-17 May 1988

Vincent and the Split Cell

Bart J. Geraci
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

Abstract

Current VLSI layout editors and other design tools have used a model where each logic circuit is represented by a single object called a cell. This paper proposes that the cell be divided into two parts: the input and output wires of the cell, the *interface*, and the body of the cell, the *implementation*. Some of the advantages of the split cell model are major: each cell has many versions which are easily interchangeable, hierarchical design rule checks are bounded by the scope of the current cell, a design methodology is provided to reduce errors, matrices of cells are easy to generate and correct, and large teams can build large designs more easily. The restrictions on cells built with the split cell model are listed, and it is shown that many of these restrictions are already used for unsplit cells. The practicality of the split cell model is demonstrated by Vincent, a VLSI editor that manipulates split cells. Like other modern VLSI editors, Vincent has a design rule checker, keeps track of electrical connectivity, and is technology independent.

"Exaggerate the essential and leave the obvious plain."
VINCENT VANGOGH

1 Introduction

A VLSI design is a hardware representation of a logic circuit. Circuits, in general, have become larger and more complex as fabrication sizes have become smaller. To deal with this inherent complexity, large circuits are composed of many smaller circuits, and these of even smaller circuits, and so on, resulting in a hierarchy of circuit composition. Each primitive circuit in a hierarchy is a *cell*. Cells contain *calls*, which are references to smaller cells, and wires joining these calls. For instance, a four-bit adder cell is built from wires joining four calls to a one-bit adder cell. This approach to circuit description, which we will henceforth refer to as the *unsplit cell paradigm*, is well-known as an integral basis for major VLSI layout editors, such as Caesar [Ous82a], KIC2 [Kel82], and Magic [Ous84].

One effect of the increasing size of circuits is that teams of designers are now needed to construct these large circuits. In a team environment, there may be some designers who are changing the implementation of a cell while other designers change the environment that calls the cell. Experience in software engineering has shown that a clean definition of interfaces in software allows different designers to work effectively on different parts of a common project. The same situation should hold true for hardware design. So just as procedures can have interfaces and multiple implementations in the Mesa [Mit79] and Modula-2 [Wir82] programming languages, the same can be said for circuits. Applying these ideas to circuits lead us to the *split cell paradigm*.

The split cell paradigm divides the cell into an *interface* region and an *implementation* region. The interface contains the input and output wires of the cell and the implementation contains the computational

circuitry. Figure 1 shows how a cell interface surrounds its implementation in the split cell paradigm. In this figure, the two boxes that cross into the implementation region belong to the interface. There is only one box in the implementation region.

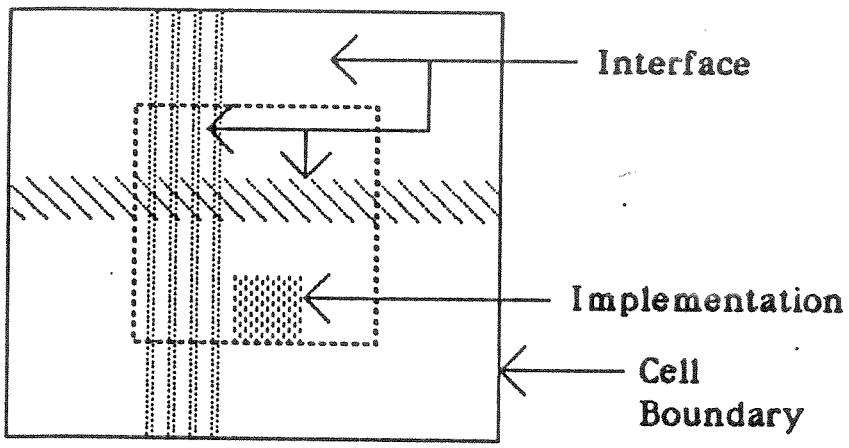

Figure 1: The Split Cell

The idea of splitting a cell was motivated by a database data model for VLSI CAD objects [Bat85]. The paper provides a system based on *molecular objects,* which are objects that have an interface and an implementation. The goal is to have a system of VLSI design tools accessing a common design database. The first tool in this system is Vincent, a VLSI editor that operates on split cells. Vincent provides an opportunity for designers to test the effectiveness of the split cell model.

In the next section, the concepts and effects of the split cell model are described. Section 3 describes the functions of Vincent from the perspective of the designer. Section 4 details the internal workings of Vincent, including several of the algorithms used and the file formats. Section 5 is how the knowledge of two previous VLSI editors we have implemented, COAX and CCOAX, affected the construction of Vincent. Finally, possible extensions to Vincent are listed.

2 The Concepts of Vincent

This section begins with some terms of the split cell paradigm being defined. After these definitions, the limitations, tradeoffs, and advantages of this paradigm are shown.

2.1 Split Cell Definitions

Given that a cell is one interface and many implementations, a few more definitions are needed. A cell *version* is the union of the interface of a cell with one of its implementations. For each implementation, there is a version. A *cell part* (or *cpart*) refers to either an implementation or an interface.

Splitting the cell affects how calls are defined. Previously, a call referred to an entire unsplit cell. In our approach, a call refers only to the interface of a cell. A call is *bound* if it is filled with an implementation, otherwise it is *unbound.* A call is *completely bound* if the call is bound and all of the calls in its implementation are bound, and so on throughout the entire hierarchy. A *parameter list* of an implementation is the list of

unbound calls of the implementation. These are the calls that the user is free to fill in, just like a procedure parameter list.

There are two variations allowed in the split cell paradigm. One is that a split cell can have implementations that compute different logical functions. This variation allows easy construction of repeating cells; see Section 2.5 for an example involving a PLA. The other variation is that an interface can contain wires in addition to the input and output wires. It is acceptable to design an interface with a totally disjoint wire running through the middle of the cell. One reason to do this is to reduce the resistance of a wire.

2.2 Split Cell Restrictions

Since implementations are functionally equivalent, replacing one implementation with another preserves the logical operation of a split cell. But implementations are subject to design rule checks against neighboring wires and calls. Therefore, some rules were developed to restrict the wires and calls surrounding implementations. These rules enable one implementation to be replaced by another without triggering design rule checks. The rules below affect interfaces, implementations, and overlapping calls.

Design Rule Checks

- The interface has to pass design rule checks by itself.
- Every implementation has to pass design rule checks in the presence of its interface since the implementation is always within the context of the interface.

Interface Width

- The interface has to be wide enough to prevent material outside its boundary from generating a minimum separation error with any material in the implementation. For example, the largest minimum separation of any two layers in nMOS is 3 lambda, so any nMOS interface has to be at least 3 lambda wide on all four sides.

Interface Calls

- The only calls allowed in the interface are vias (contact cuts); transistors and other computational cells are prohibited. This enforces the idea that the interface contains only the input and output connections of the cell.

Overlapping Calls

- An implementation cannot overlap another implementation.
- An implementation cannot overlap the interface of another call.
- Interfaces can overlap only where they share the same material. Figure 2(a) shows two interfaces with the same material in the overlap. Figure 2(b) shows two interfaces that do not have the same material in the overlapping region; this is prohibited.
- A wire can cross into a call only where there is a similar piece of material in the interface. Figure 2(c) shows the wire crossing into the interface.
- A wire cannot cross into an implementation region.

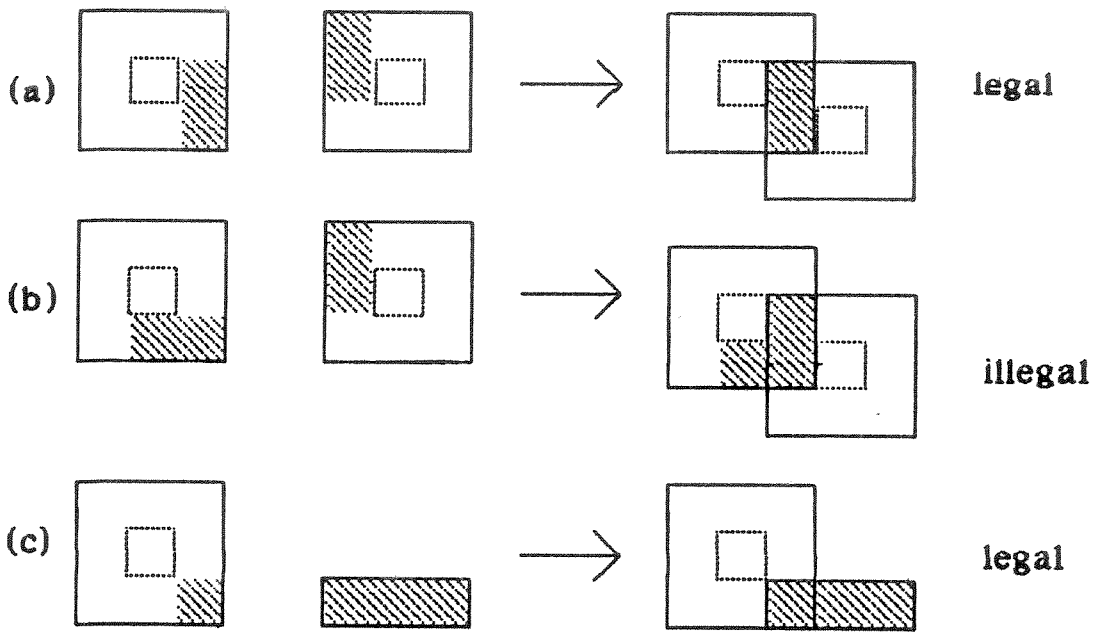

Figure 2: Overlapping Interfaces

2.3 States of Cell Parts

States are used to distinguish working interfaces and implementations from those that need further editing. Attaching states to cell parts also reduces the need to do design rule checks. These states are called *unreleased, released,* and *frozen.* Table 1 summarizes these states and their restrictions on editing interfaces and implementations.

	Unreleased	Released	Frozen
Interface	no access full edit	access no edit	
Implementation	no access full edit	access limited edit	access no edit

Table 1: States of Interfaces and Implementations

An unreleased state implies that the cpart is not finished. All editing operations are allowed, but the cpart cannot be referenced in other cells. An unreleased interface cannot be called by another cell, and an unreleased implementation cannot be used to bind a call.

A released state implies the cpart passes design rule checks and it can be referenced by other cells. A released interface can be called, but further editing is prohibited. A released implementation can be used to bind a call, but editing is restricted to preserve the parameter list of the implementation. That is, no new unbound calls can be introduced, nor can existing unbound calls can be deleted. Wires, bound calls, transistors, etc. can be moved or added as needed.

A frozen state implies the cpart can be referenced by other calls and that editing the cpart is disallowed.

A frozen interface can be called, and a frozen implementation can be used to bind a call. Note that a released interface is the same as a frozen interface and it is referred to as the latter.

As a prerequisite to changing states, the cpart must pass design rule checks. Converting an implementation from released to frozen also requires that the parameter list remain unchanged. In addition, any cpart that changes its state gets written to its file.

Frozen or released cparts are never deleted automatically. Attempts to edit a frozen cpart generates a new unreleased cpart with the same data. Attempting to save a released implementation that either fails design rule checks or modifies the parameter list demotes the implementation to the unreleased state and this implementation is renamed. The original implementation is restored to the state of the last save.

2.4 Split Cell Advantages

There are three advantages of split cells over unsplit cells. First, replacing one implementation of a cell with another is easy and it is done without a design rule check. Second, design rule checking the entire cell hierarchy is no longer needed. Third, the split cell discourages bad design practices, such as splitting a transistor across two cells.

One problem in the unsplit cell paradigm is that replacing one version of a call with another may be a complex operation. The old call is removed and the replacement call is added. Even if the replacement call is the same size or smaller, the old connections may need to be changed to accommodate the new call. And if the replacement call is larger than the original, the entire cell may require redesigning. This problem is likely to occur when the design is functionally correct, but the design fails additional specifications such as timing or power consumption.

In the split cell model, replacing one version of a call with another means replacing the implementation bound to the call with the new implementation. This operation does not trigger a design rule check due to the restrictions for overlapping calls in Section 2.2.

A second problem is that doing a hierarchical design rule check on a cell requires a great deal of computation. For each cell, every call is checked against its neighbors and each different subcall is checked only once. Then the procedure goes down the hierarchy for each subcall. For instance, in a four-bit adder cell, the one-bit adder by itself is checked only once and the four references to the one-bit adder are checked within the context of their neighbors.

In the split cell model, the only checks that are needed are those involving interfaces and wires. Cell implementations are not rechecked because they already have been tested against their interface and cannot be affected by any material outside of the interface. Furthermore, only vias are allowed in the interface; this means that all of the hierarchy in a call is found in the implementation. Therefore, hierarchical design rule checking of split cells is unnecessary.

A third problem is that systems without cell overlap restrictions may not discourage bad design practices. For example, a pair of cells can be constructed such that each cell has one half of a transistor. When the cells abut, a transistor is created straddling the cells. What looked like pieces of wires has turned into a transistor, as in Figure 3. Such practices could lead to errors in the design.

Figure 3: Transistor Construction

The cell overlap restrictions in Section 2.2 discourage this and other bad design practices. The restrictions emphasize that the division of a cell should not be arbitrary; special meaning is attached to the two parts of the cell. This will encourage the novice designer to create well-defined interfaces. Another advantage of these restrictions is that design rule checking and circuit extraction are easier to perform. Converting wires into a transistor involves replacing the wires by a transistor call and recomputing the electrical connectivity information since a transistor is different from normal wires.

2.5 Split Cell PLAs

Yet another advantage of the split cell model is that matrices of cells are easy to generate and correct. A Programmable Logic Array (PLA) can be built from an interface with two implementations as shown in Figure 4. The interface is used as an underlying grid and one of the implementations joins the crossing wires whereas the other one leaves them separated. A matrix of calls can be constructed, each call bound with the latter implementation. At the squares where the wires should connect, the individual call is rebound to the former implementation. The function of the matrix is changed by rebinding a call with its opposite implementation—and this rebinding operation does not require a design rule check.

Recognizing that the calls of a matrix in the traditional cell paradigm often overlap each other, one concern was that the strict rules in the split cell model would increase the size of the matrix. However, popular repeating constructs such as the PLA and the Weinberger array, both described in [Muk86], have overlaps that are allowed under the split cell overlap restrictions. In these cases, the matrix size remains constant. If the overlap of a repeating cell is not permitted in the split cell model, that cell can be recast in an interface that allows overlap. This recasting would only increase each call by the minimum width of the interface.

2.6 Split Cell Tradeoffs

The split cell model requires a designer to consider tradeoffs between the size of the implementation region and the possible number of implementations. A smaller implementation region means a smaller total cell size, but it also means there is less room to explore alternate implementations.

At one extreme, the implementation region is too small to allow more than one implementation to be built; the advantage of a cell having multiple implementations is lost. At the other extreme, an implementation region may be very large with respect to the size of an implementation; binding the interface with this implementation wastes space. Somewhere in the middle, a new implementation for an existing cell can be constructed that does not fit the interface. The designer must either create a new interface just for this implementation or create a new interface for all of the implementations.

3 The Operations of Vincent

Since basic operations of Vincent are similar to other VLSI editors, this section points out their differences. Most of these differences are a consequence of the split cell paradigm.

3.1 Wires

Like other VLSI editors, wires are created in Vincent using the fill command. The designer draws a box, selects a layer, and uses the command to fill the box. Similarly, the empty command removes the layer from the box. All of these wires can be labeled with node markers so the designer can keep track of which wire carries which signal.

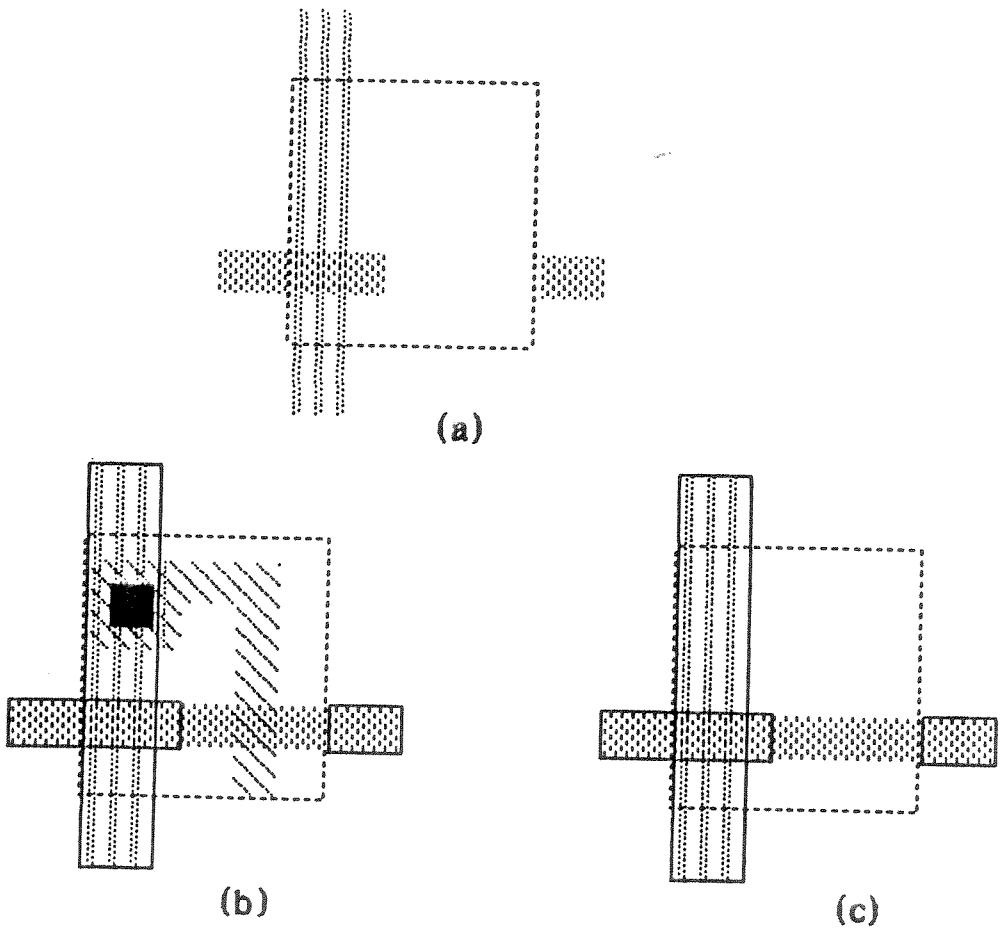

Figure 4: The PLA interface is (a), the implementation connecting the crossing wires is (b), and the implementation that leaves the wires split is (c). The outlined boxes in the implementations are those that belong to the interface.

Vincent differs from a few editors by providing a painting metaphor[1]. Wires are drawn in the current layer by dragging a paintbrush across the screen; erasing the current layer is done in the same way. The problem of painting a straight line led to using the shift key to constrain the line horizontally or vertically, a method borrowed from MacPaint[2]. As wires are added and deleted, Vincent maintains electrical connectivity by adding and merging nets.

By adjusting two parameters in Vincent, which is done simply by pointing at the numeric field and typing in the new values, multiple parallel lines of paint can be drawn. A small example is shown in Figure 5. The first parameter is the pick resolution, which limits the points picked on the screen. A pick resolution of N lambda rounds the pick location to the nearest intersection in a grid with units N lambda square. The other parameter is the width and height of the paintbrush, which both default to the minimum width of the layer. To draw wires 2 lambda wide separated by 2 lambda, the pick resolution is set to 4 lambda, the brush is set to 2 lambda by 4 lambda, then the brush is dragged across the screen.

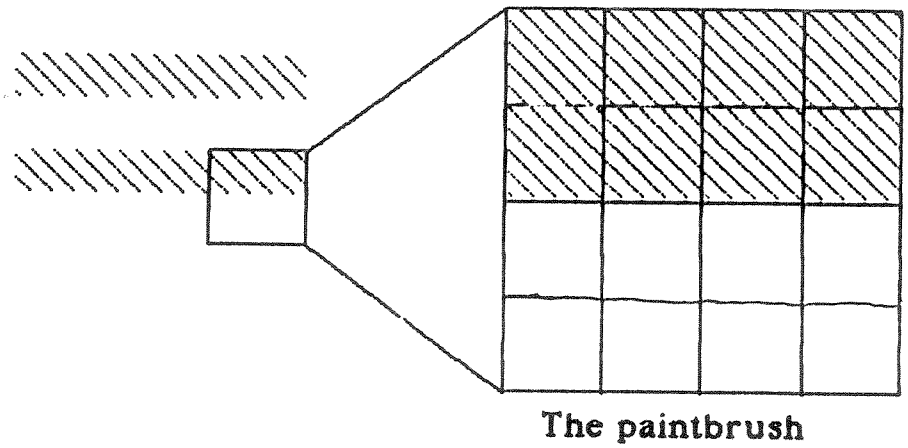

Figure 5: Multiple Parallel Lines

3.2 Calls

Despite the split cell model, Vincent provides the same call operations as other editors, treating the call as a single entity. These operations include adding a call, deleting a call, adding an array of calls, moving a call, rotating a call, and so on. Naturally, one difference is that to add a bound call, the designer fills in two data fields in the function window: the interface and the implementation.

The split cell model needed a call mode in which the user can select a call by pointing to it on the screen. The call mode is used in functions common to other VLSI editors such as deleting a call or copying a call. But this mode is also used to bind and unbind calls with implementations, two common operations. Notice that not only is the call containing the implementation bindable, but so are the unbound calls in the implementation, and the unbound calls in their implementations, and so on.

Just selecting a call does not indicate the name of the interface nor the name of the implementation. Hence, Vincent has a function that retrieves the names of the interface and implementation of the current

[1] "Vincent" stands for "Visual INteractive Chip Editor for Numerous Technologies", but the name is also a reference to Van Gogh, the artist. It should not be surprising to see a *paintbrush* object, a *paint* mode, a *draw* command, and so on.

[2] MacPaint is a trademark of Apple Computer, Inc.

call and copies them to their respective data fields. The advantage is that both fields are used to add a call and the implementation data field is used to bind a call. So to bind call A with the same implementation as call B, the designer selects call B, retrieves its name, then selects call A and binds it.

3.3 Transistors and Vias

Like other VLSI editors, Vincent treats transistors and vias as special objects because they do not follow the normal rules of electrical connectivity. A via joins wires that would normally be on separate nets. A transistor has separate wires that would normally be on the same net. Therefore, a designer who creates a transistor from scratch will get the connectivity data wrong. One solution is to let the designer draw the wires and require the editor to convert wires to transistors and vias. If a transistor is recognized, then the nets are changed automatically to reflect the connectivity of the circuit. The solution used by Magic is to define transistors and vias as separate layers and use the fill operation to add these objects.

The approach used in Vincent is to define transistors and vias and then provide a fast and easy method of adding them to the cell. Vincent presents a list of all the transistors and vias and keyboard equivalents for adding them. The designer selects a transistor or via from the list. To add the smallest sized transistor, the designer points to the screen and presses a Vincent-designated key. To add a larger transistor, the designer defines the transistor size with a box and presses a key. This is easier than drawing all of the boxes for each layer to build a transistor or via. Also, unlike Magic, which has its abstract layer as a completely different stipple pattern (or color), transistors and vias in Vincent are shown in their normal layers: metal, diffusion, poly, and so on.

Dealing with transistors and vias in Vincent led to a series of design decisions. First, transistors and vias would become predefined cells stored in the technology file and loaded when Vincent begins. They are predefined as cells in this file because they are a part of the technology definition and because they fit within the regular cell definition as a collection of wires. Therefore, transistors and vias can be handled like a call to any other cell. Each box in each wire in these cells is assigned its net number. The boxes in a via get the same net number and the boxes in a transistor get different net numbers. Other than this difference, transistors and vias are treated the same way.

The next decision was to define transistors and vias in terms of a variable box size since they can come in too many different sizes to list in the technology file. In addition, each transistor and via has a minimum size. Since this is the size most often used, it is used as a default size. If a designer draws a box on the screen and selects a transistor, then the transistor is added, expanding to fit the box. If the designer just points to a location without defining a box, then the default size transistor appears at that location.

Since transistors and vias are the property of Vincent, these cells are saved by Vincent in its own subdirectory. When a new transistor is requested, Vincent looks in its subdirectory to see if it has been created. If not, it creates a new transistor and stores it in the subdirectory. Furthermore, most of the transistors and vias that are used are of the minimum size. So on average, there are only a few, say less than 10, different sizes of each transistor or via.

In Vincent, drawing a transistor or via from scratch will not get the electrical nets correct. In fact, it may generate an error, depending on the rules specified in the technology data file. For instance, the nMOS technology file has a rule that says if a diffusion wire crosses a polysilicon wire, the two wires are flagged as a minimum separation error. This rule does not get applied to the cells defined as transistors because these cells are assumed to be correct. Thus, these cell definitions are used to define constructs that do not follow normal design rules. Furthermore, these cells eliminate some of the more complex rules from the technology file and reduce the number of remaining rules that have to be applied to normal cells.

3.4 Splitting a Cell

Constructing a split cell is simple: the designer creates an unsplit cell then draws a box to divide it. The portion of the cell strictly inside the box goes to an unreleased implementation and the rest goes to an unreleased interface. The external boundary of the interface is the external boundary of the cell before the split. A cell is shown in Figure 6 both before and after the split operation. Prerequisite to the split, three conditions must be met: the interface has to be wide enough to insulate the implementation from minimum separation errors, the only calls in the interface must be vias, and both interface and implementation has to pass design rule checks.

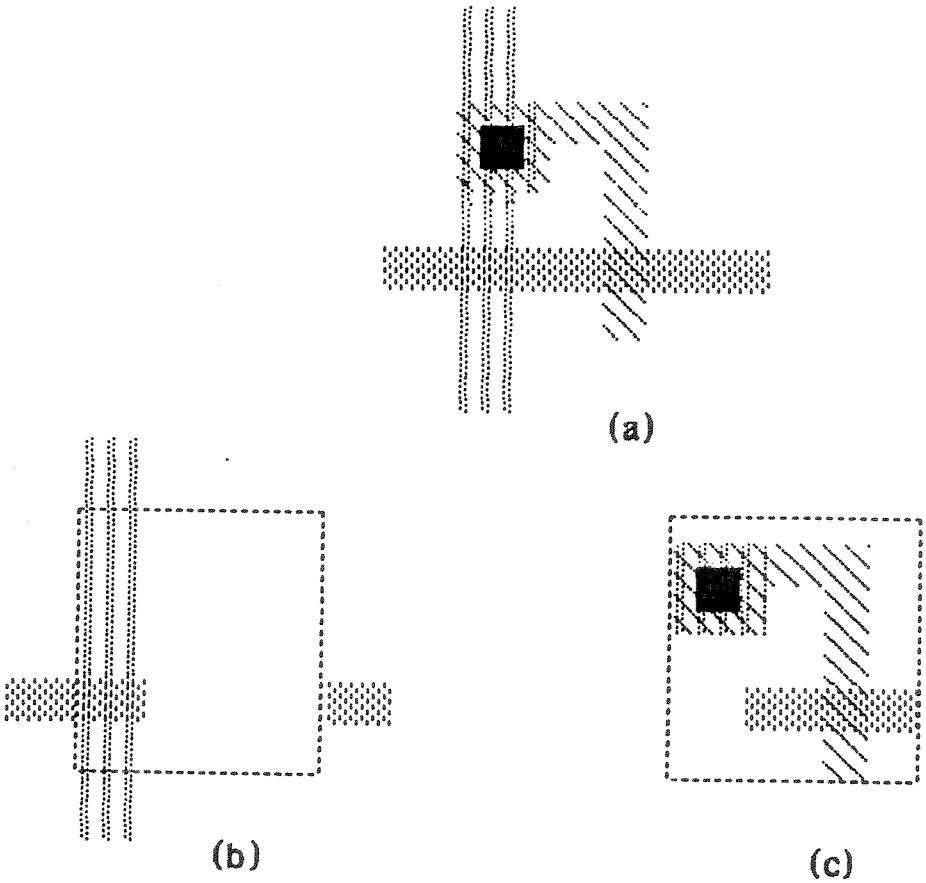

Figure 6: The cell before the split is (a). After the split, the cell is divided into an interface (b) and an implementation (c).

There is a corresponding unsplit operation. This function takes an interface and an implementation and merges their data to form an unsplit cell. So the unsplit operation acts as an "Undo" for the split operation.

3.5 Design Rule Checking

Like other VLSI editors, a design rule checker is built into Vincent. These checks can be done automatically after each editing operation or manually at the request of the user. The design rules checks and the specification syntax are based on [Chu85].

There are four design rule checks, the first three dealing with the individual layers. The first test is the minimum width of a wire in a layer. The second test is the minimum separation between two wires, whether or not they are the same layer. In addition, there are two separation values, depending on whether or not the wires are on the same net. Two wires of the same layer can be much closer if they are on the same net than if they were on different nets; this is why electrical connectivity is maintained. The third test is the minimum overlap of one wire against another for two different layers.

The last test is a result of the split cell paradigm. It verifies that the material in the region intersected by two interfaces or by an interface and a wire is present in both parties. If this rule was not enforced, then one implementation of a call could pass design rule checks, yet another implementation would violate a minimum separation rule with material that is overlapping the interface.

There are factors that simplify design rule checks in Vincent. First, vias and transistors are defined as cells. Therefore, the checker can treat them like normal cells as opposed to having special rules for transistors and vias. Second, boxes are merged to form the largest boxes possible. This means that checking for minimum width errors is simply checking the minimum width of each box[3].

4 The Internals of Vincent

The internals of Vincent provide an opportunity to look at the tradeoffs made. Most of the design decisions came from the split cell model. A few more decisions arose from the references. And some decisions were influenced by reactions from users to two previous VLSI editors, COAX and CCOAX, which we have built.

4.1 User Interface

Vincent is built using the SunView system, which facilitates the creation and use of interactive programs. Vincent consists of one function window and many cell windows. The function window is divided into two parts: the upper portion is where the command buttons and data fields are located, the lower portion is where errors and other messages are posted. A cell window operates exactly like any other SunView window; it can be stretched, hidden, opened, closed, or deleted. As a convenience, the icon of a closed cell is labeled with its name. These features are shown in Figure 7.

4.2 File Format

EDIF Version 2 0 0 [EDI87] is the file format used for expressing both technology data and cell data in Vincent. Not only can EDIF express mask layout data, such as wires and calls, but it can also describe technology data, such as design rule constraints, and program data, such as the color of a layer as it appears on the screen. These provisions have proven to be very useful for our purposes. Also, each EDIF statement is in a LISP-like form: (*function arg1 arg2* ...). This allows future extensions to EDIF to be added smoothly. Furthermore, Chu and Lien [Chu85] demonstrate how a VLSI editor can become technology independent by using the EDIF system.

Despite these advantages, Vincent uses a modified EDIF syntax. A problem with true EDIF is its verbosity. In many cases, there is too much overhead for a piece of data. Figure 8 shows an example of

[3] Almost. There is a case where a box is less than the minimum width in one dimension, but the total wire does not violate the minimum width rule.

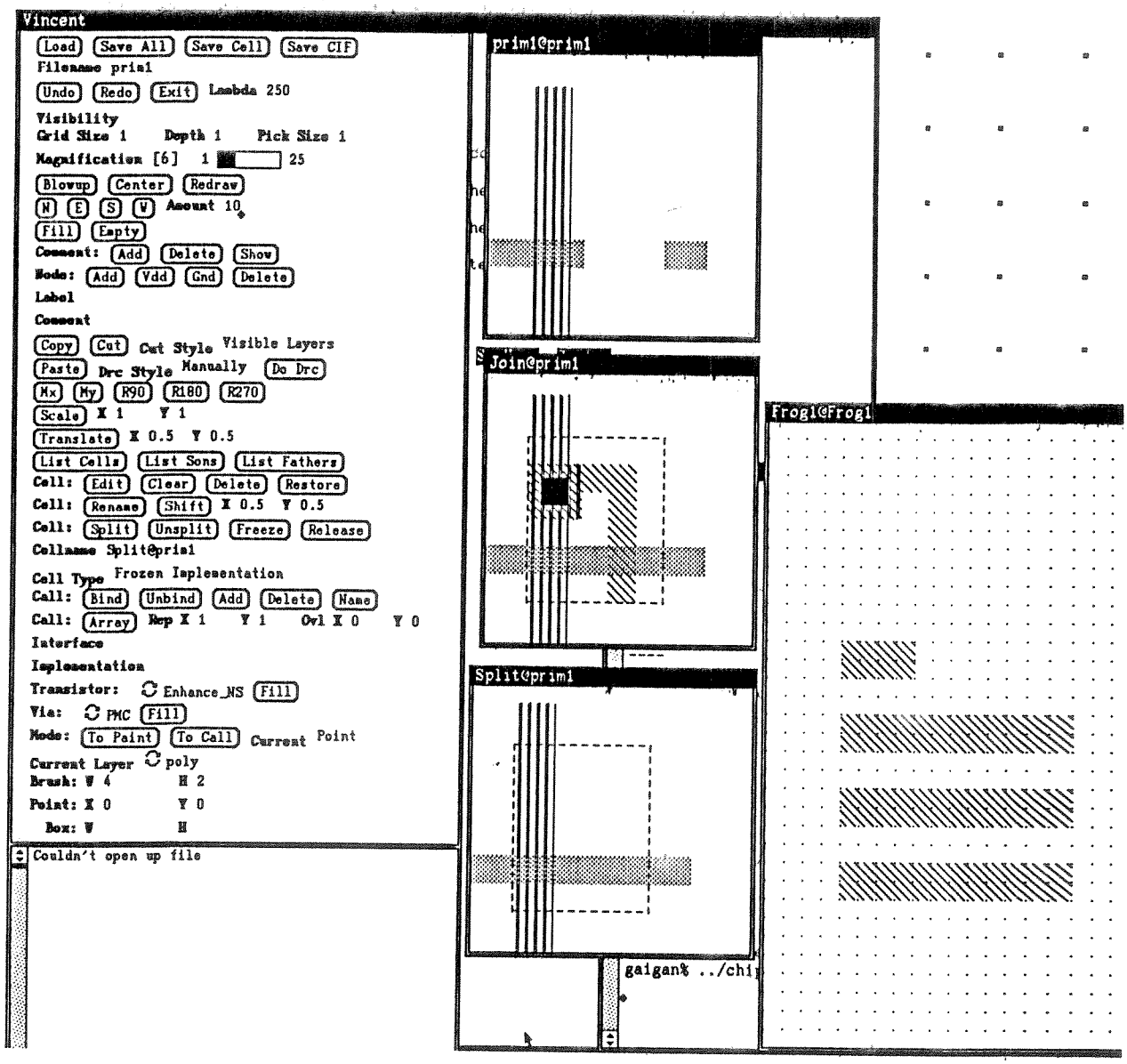

Figure 7: User Interface of Vincent

how Vincent shortens the data represented in EDIF. Vincent also restricts its parser to accept a subset of EDIF based on the mask layout view of the cell. EDIF can represent cells in other views such as schematic, behavioral, symbolic, and so on.

EDIF version:
```
(rectangle (pt 1 5) (pt 3 4) (property net_number (integer 2)))
```
Vincent version:
```
(ebox 2 5 1 4 3)
```

Figure 8: `ebox` **is:** *net_number top left bottom right*

The parser used in Vincent is based on [Pai85], but several modifications were made. First, normal EDIF comments, (`comment` { *astring* }), can now appear at any location in the file. Second, a scoping comment was invented, (`skip` { *EDIF_expression* }), so it could enclose EDIF expressions, unlike the normal comments. A scoping comment can too appear at any location in the file. Third, identifiers and symbolic constants can have underlines in their names. Fourth, a new type, *lambda numbers,* was invented because the resolution of a lambda parameter is one-half lambda. For example, the minimum overlap of a transistor region by an implant layer in nMOS technology is 1.5 lambda. Fifth, Vincent can read and write numbers like `2.5`. The previous version of EDIF [EDI85] could also do this, but the current version of EDIF uses the form (`e 25 -1`), which is interpreted as 25 times 10^{-1}.

4.2.1 The Technology File

The technology file specifies the parameters for a VLSI technology. It begins with the names of the source and power lines. This is followed by the technology layers: their names, stipple patterns (the equivalent of color on monochrome displays), minimum widths, and so on. Then the design rules for minimum separation and minimum overlap are listed. The last part defines the vias and transistors.

Our file format for the technology data is an extension of the format presented in [Chu85]. The first extension is that the minimum size for each transistor and via is now defined. Second, rectangles in the transistor and via definitions now have connectivity data. This made it unnecessary to explicitly define a rectangle as either a switch or a terminal. Third, the stipple patterns may be specified using either binary or hex strings. The hex strings are in the same format as the output of IconEditor, which was used to generate the stipple patterns. This format is much better than EDIF as Figure 9 shows.

4.2.2 The Cell File

A cell file contains an EDIF-based specification of a cell. A split cell has its interface and all of the implementations in one file. Each cell part contains the cell type, boxes that violate the design rule checks, labels that mark wires, comments the designer added to explain features of the cell, wires, and calls.

The obstacle that prevented Vincent from using the proper EDIF syntax for the cell was that EDIF failed to provide an accurate representation of the split cell model. Specifically, the idea that each split cell is composed of one interface and many implementations. EDIF has the idea that each `view` has an `interface` and a `contents` component, the latter being equivalent to an implementation. But at most one `contents` section is allowed in each `view`, hence, at most one implementation for each cell is allowed under the EDIF definition.

EDIF version:

```
(boolean
    (boolean
        (true) (false) (false) (false)
        (false) (false) (false) (false)
        (true) (false) (false) (false)
        (false) (false) (false) (false))
    (boolean ...)  ...)
```

Vincent version:

```
(hex "0x8080,...")
```

Figure 9: Syntax for Stipple Patterns

4.2.3 CIF Compatibility

Vincent can also write out a file in CIF, a widely used file format described in [Mea80]. This is important because many design tools (e.g. Magic and Caesar) output this representation. All mask layout components of the split cell have a corresponding representation in CIF. The bound call in Vincent is written in CIF as two separate calls, one for the interface and one for the implementation. Some data is lost when a cell is written in CIF, including connectivity information and the relationship between interfaces and implementations.

4.3 Internal Features

There are three main features of the internal workings of Vincent. First, a package of generic linked list routines was written since most of the data structures are organized in simple lists. This package eased the effort of coding the rest of Vincent because it allowed algorithms to be expressed more abstractly. In addition, the linked list package can be used in other programs.

Each technology layer uses an R-tree [Gut84] to add, delete, and retrieve wires. An R-tree is a height-balanced B-tree where each node has a boundary describing the region covered by its sons. When a R-tree node is filled, it is split into two nodes dividing the data so that the overlap of the two nodes is kept to a minimum. Hence, R-trees are efficient in finding all boxes in a given region. Vincent implements R-trees using rectangular boundaries and the linear splitting method described in [Gut84]. Each leaf of an R-tree points to a unit, which contains a box boundary and is linked to other units that touch it; this is the way wires are built and connectivity is maintained.

Another algorithm in Vincent merges and expands boxes. Merging is used to reduce the amount of boxes written to the output file by combining many small boxes into a few large boxes. Expanding a box makes a box as large as possible. Expanding and merging boxes eliminate small boxes that would violate the minimum width rule in design rule checking. For instance, one box may be expanded by being adjacent to another box. If the expansion is not done first, the first box may be flagged with a minimum width error. The results of two boxes overlapping is shown in Figure 10.

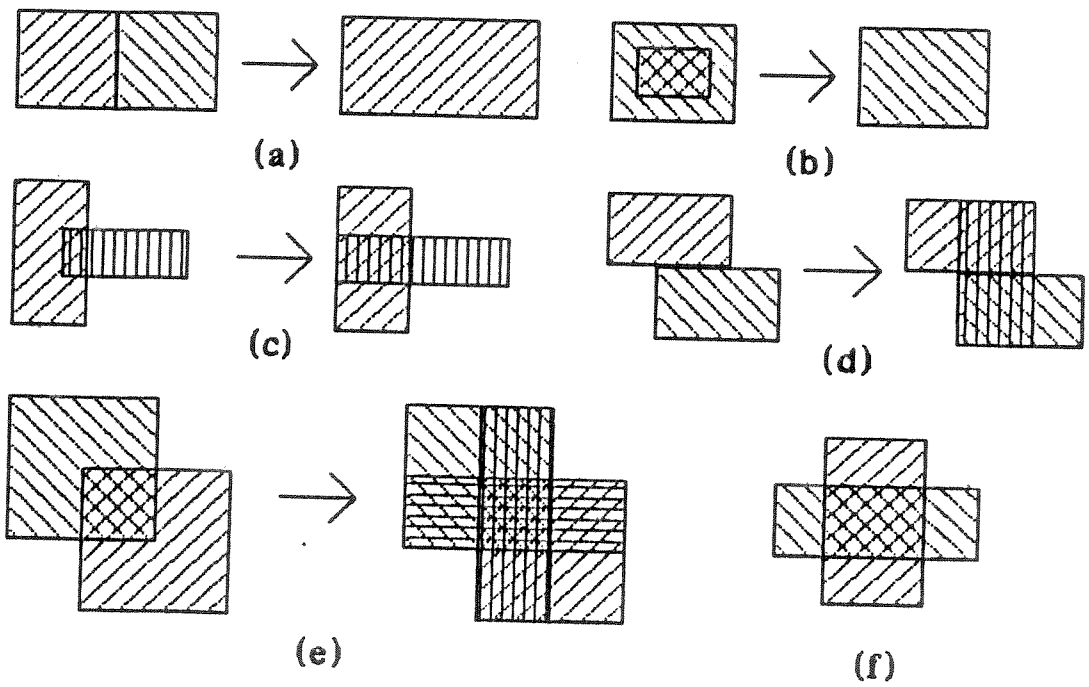

Figure 10: The intersection of two boxes can either merge into one box (a,b), expand one box (c), generate one new box (d), generate two new boxes (e), or do nothing (f). This process is repeated until boxes are neither generated nor merged.

4.4 Program Data

Coding started on Vincent in October 1986, and in October 1987 the first version was finished and tested. Vincent is written in C [Ker78] under Berkeley 4.2 Unix[4] using the SunView Interface system. Vincent runs on a Sun-3[5] workstation with a monochrome monitor. Vincent is approximately 22,000 lines (including blank lines and comments); Table 2 divides the line count by category.

Component	Size (lines)
Database	2650
Graphics and User Interface	3450
Design Rule Checker	1550
Library	3900
File Input/Output	3950
Cell Operations	3800
Circuit Extraction	800
Undo	1100
Initialization	650
Global Declarations	900
TOTAL	22750

Table 2: Vincent line count by category

Emphasis was placed on the possibility that Vincent would get ported to other machines and systems. So code dependent upon the SunView system is in only in 17% (6 out of 35) of the modules and in 13% (3000 out of 22750) of the lines.

5 Previous VLSI Editors

Vincent is not our first VLSI editor, it is our third. Aside from the listed references, some design decisions were made from exposure to previous VLSI editors.

The first editor we have used was Cifvi, an nMOS technology editor running on a Sun 2. Cifvi was studied and its flaws were examined before writing our first editor, COAX, an nMOS technology editor running on the Xerox Dandelions. One flaw was that Cifvi did not merge rectangles together to form larger rectangles. So COAX was written with an algorithm that merged each new rectangle drawn with its neighbors. Fortunately, this algorithm was fast enough not to slow down the user. Also, designers used the keyboard shortcuts rapidly after learning them, especially for commands that were used often like fill and empty. Eventually, the designers would have one hand on the mouse and the other hand on the keyboard. These observations assisted in refining COAX, followed by building CCOAX, a CMOS technology version of COAX.

Going from COAX to Vincent revealed more subtleties of VLSI design. First, redrawing the current screen should be as fast as possible. This coincides with [Ous82b], which found redrawing to be the most used command for Caesar. To make the redraw command fast, the data structures for the wires had to be able to retrieve any rectangular portion of the design. This led to using R-trees to organize the wires. Also, a few designers wanted an easier way to join wires; this led to the painting metaphor in Vincent. Other designers wanted to attach a label to a wire to keep track of the connections; this too was added to Vincent.

[4] Unix is a registered trademark of AT&T Laboratories.

[5] Sun-3 is a trademark of Sun Microsystems, Inc.

6 Conclusions

The split cell model is proposed as an improved way to describe the representation of a logic circuit. The model defines a cell as having one interface and many implementations. The interface corresponds to the input and output paths of a cell, where the implementation is the calculation of the cell. Although all implementations compute the same function, the availability of many different implementations allows the designer to choose one based on factors such as speed and power consumption. The idea of splitting an object into an interface and an implementation is not new. Programming languages such as Mesa [Mit79] and Modula-2 [Wir82] have been doing this for several years. Vincent can be seen as an experiment in the application of the well-known software engineering advantages of modularity, information hiding, and the enforcement of clear interfaces among modules in a design to the area of VLSI CAD.

In the split cell model, one implementation can be replaced by another without triggering a design rule check outside the environment of the cell. The design rule check condition strengthens a few of the old advantages and adds some new advantages of the split cell model. This operation required a few restrictions, such as the minimum size of an interface, or the way calls can overlap, but some of these restrictions are already being used.

There are several goals that we wish to accomplish through the split cell model. We will find out if these goals are reached once Vincent becomes widely used among designers. First, a team of designers should be able to construct large designs faster. Second, matrices of cells should be constructed and corrected more easily. Third, the interface and implementation division should prevent design errors by discouraging bad design practices. These advantages would lead to designs that are much easier to modify.

Although Vincent is in use, there is room for improvement. Simulation tools, such as a circuit extractor, could be integrated with Vincent. New functions like a river router could also be added. Vincent could be ported to other machines; this would likely be preceded by rewriting the graphic routines in X or NeWS, depending on which one becomes the standard. In addition, designer feedback from using Vincent may suggest some other changes.

I was first exposed to VLSI design from Al Mok's class "Introduction to VLSI Design". It was the paucity of VLSI design editors available for his course that led to events culminating in the writing of Vincent. I also wish to thank Ed Lien of MCC for the helpful papers on EDIF. Thanks to Shinichiro "Nick" Haruyama, Kim Man Lau, and Mosfeq Rashid for taking the time to test Vincent and giving me their comments. And not least, but last, special thanks go out to my advisors Don Batory and Don Fussell.

References

[Bat85] Batory, Don S. and Won Kim. "Modeling Concepts for VLSI CAD Objects." *Supplement to Proc. ACM-SIGMOD,* 1985, pp. 18–32.

[Chu85] Chu, Kung-Chao and Y. Edmund Lien. "Technology Tracking for VLSI Layout Design Tools." *Proc. 22nd Design Automation Conference,* 1985, pp. 279–285.

[EDI85] EDIF Steering Committee. *EDIF Electronic Design Interchange Format Version 1 0 0.* Washington D.C.: Electronic Industries Association, Jan 1985.

[EDI87] EDIF Steering Committee. *EDIF Electronic Design Interchange Format Version 2 0 0.* Washington D.C.: Electronic Industries Association, May 1987.

[Gut84] Guttman, Antonin. "R-Trees: A Dynamic Index Structure for Spatial Searching." *Proc. ACM SIGMOD,* 1984, pp. 47–57.

[Kel82] Keller, K. H. and A. R. Newton. "KIC2: A Low-Cost, Interactive Editor for Integrated Circuit Design." *Proc. Spring COMPCON,* 1982, pp. 305–306.

[Ker78] Kernighan, Brian and Dennis Ritchie. *The C Programming Language.* Englewood Cliffs, New Jersey: Prentice-Hall, 1978.

[Mea80] Mead, Carver and Lynn Conway (Eds.). *Introduction to VLSI Systems.* Reading, Massachusetts: Addison-Wesley, 1980.

[Mit79] Mitchell, J. G., W. Maybury, and R. E. Sweet. *Mesa Language Manual.* Technical Report CSL-79-3, Xerox Corporation, Palo Alto Research Center, Palo Alto, California, 1979.

[Muk86] Mukherjee, Amar. *Introduction to nMOS and CMOS VLSI Systems Design.* Englewood Cliffs, New Jersey: Prentice-Hall, 1986.

[Ous82a] Ousterhout, John K. "Caesar: An Interactive Editor for VLSI Layout." *Proc. Spring COMPCON,* 1982, pp. 300–301.

[Ous82b] Ousterhout, John K. and David M. Ungar. "Measurements of a VLSI Design." *Proc. 19th Design Automation Conference,* 1982, pp. 903–908.

[Ous84] Ousterhout, John K., Gordon T. Hamachi, Robert N. Mayo, Walter S. Scott, and George S. Taylor. "Magic: A VLSI Layout System." *Proc. 21st Design Automation Conference,* 1984, pp. 152–159.

[Pai85] Painter, Paul B. "A Table Driven, Recursive Descent EDIF Parser." MCC VLSI CAD Program Technical Document, Austin, Texas, May 1985.

[Wir82] Wirth, Niklaus. *Programming In Modula-2.* New York, New York: Springer-Verlag, 1982.