Magic: A VLSI Layout System

John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo,
Walter S. Scott, and George S. Taylor

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Magic is a "smart" layout system for integrated circuits. It incorporates expertise about design rules and connectivity directly into the layout system in order to implement powerful new operations, including: a continuous design-rule checker that operates in background to maintain an up-to-date picture of violations; an operation called *plowing* that permits interactive stretching and compaction; and routing tools that can work under and around existing connections in the channels. Magic uses a new data structure called *corner stitching* to achieve an efficient implementation of these operations.

Keywords and Phrases: interactive layout editor, corner stitching, design-rule checking, routing, stretching, compaction.

Magic: A VLSI Layout System

John K. Ousterhout, Gordon T. Hamachi, Robert N. Mayo,
Walter S. Scott, and George S. Taylor

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

Magic is a "smart" layout system for integrated circuits. It incorporates expertise about design rules and connectivity directly into the layout system in order to implement powerful new operations, including: a continuous design-rule checker that operates in background to maintain an up-to-date picture of violations; an operation called *plowing* that permits interactive stretching and compaction; and routing tools that can work under and around existing connections in the channels. Magic uses a new data structure called *corner stitching* to achieve an efficient implementation of these operations.

Keywords and Phrases: interactive layout editor, corner stitching, design-rule checking, routing, stretching, compaction.

1. Introduction

Magic is a new interactive layout editing system for large-scale MOS custom integrated circuits. The system contains knowledge about geometrical design rules, transistors, connectivity, and routing. Magic uses its knowledge to provide powerful interactive operations that simplify the task of creating layouts. Moreover, once a layout has been entered, Magic makes it easy to modify it; this permits designers to fix bugs easily, experiment with alternative designs, and make performance enhancements.

Magic provides several new operations for its users. Design rules are checked continuously and incrementally during editing sessions to keep up-to-date information about violations. When the layout is finished, then so is the design-rule check. A new operation called *plowing* allows layouts to be compacted and stretched while observing all the design rules and maintaining circuit structure. Routing tools are provided that can work under and around existing wires in the channels (such as power and ground routing) while still providing the traditional efficiency of a channel router.

Two aspects of Magic's implementation make the new operations possible. First, the system is based on a data structure called *corner stitching* which is both simple and efficient for a variety of geometrical operations [6]. Without corner stitching, most of Magic's new operations would be too slow for interactive use. Second, designs in Magic are specified using *abstract layers*, rather

than actual mask layers. The abstract layers represent circuit structures such as contacts and transistors in a form that appears somewhat like sticks [14] except that objects are seen in their actual sizes and positions. The abstract layers incur no density penalty, but they simplify the designer's view of the system and provide more explicit information about the circuit structure.

This paper gives an overview of the Magic system. Section 2 describes the specific problems Magic attempts to solve, and the overall approach of the system. Sections 3 and 4 describe the data structure and abstract layers used in the Magic implementation. Sections 5-11 discuss Magic's new operations, and Section 12 presents the implementation status of the system. Three additional papers in this technical report discuss design-rule checking, plowing, and routing in detail [2,11,12].

2. Background and Goals

Our previous layout editing systems, Caesar [5,7] and KIC2 [3], have been used since 1980 for a variety of large and small designs in several MOS technologies. They are similar to systems currently in use in industry. Although our systems have proven quite useful, we uncovered a few areas where they (and most other existing layout systems) are inadequate. The most severe inadequacy is in the area of routing, where most systems provide little support. We estimate that between 25% and 50% of all layout time for our circuits is used

for hand-routing the global interconnections, even though the circuits are highly regular to begin with. The task of routing is tedious and error-prone.

A more general problem is one of flexibility. Once a design has been entered into the layout system, it is hard to change. This makes it difficult to fix bugs found late in the layout process, and almost impossible to experiment with alternative designs. If designers cannot experiment with and evaluate alternatives, it is hard for them to develop intuition about what is good and bad. Routing is the most extreme example of the flexibility problem. It takes so long to route a circuit that it is out of the question to re-route a chip to try a new floor-plan. Even small cells are difficult to change: modest changes to the topology of a cell often require the entire cell to be re-entered. In many industrial settings, layouts are so difficult to enter and modify that designs are completely frozen before layout begins.

Our overall goal for Magic is to increase the power and flexibility of the layout editor so that designs can be entered quickly and modified easily. When the system is complete, we hope it will provide order-of-magnitude speedups for three different parts of the design process:

1) Once a large circuit has been routed, it should be possible to remove the routing and re-route in a few hours. Even the initial routing should not require more than a few days for a large custom circuit. With our current systems, routing requires a few weeks to a few months.

2) The turnaround time for small bug fixes should be less than 15 minutes. For example, if a bug is found while simulating the circuit extracted from a layout, it should be possible to fix the layout, verify that the new layout meets the design rules, and re-extract the circuit, all in 15 minutes. This process currently requires several hours of CPU time and at least a half-day of elapsed time.

3) It should not take more than 30 seconds to 1 minute to re-arrange a cell to try out a different topology. With our current systems this requires anywhere from tens of minutes to several hours.

Magic meets these goals by combining circuit expertise with an interactive editor. It understands layout rules; it knows what transistors and contacts are (and that they must be treated differently than wires); and it knows how to route wires efficiently. Magic uses the circuit knowledge to provide interactive operations that re-arrange a circuit *as a circuit*, rather than as a collection of geometrical objects. It also performs analysis operations, like design-rule checking, *incrementally*, as the circuit is created and modified. This means that only a small amount of work must be done each time the circuit is modified.

3. Corner Stitching

In Magic, as in most other layout editors, a layout consists of cells. Each cell contains two sorts of things: geometrical shapes and subcells. Magic represents the contents of cells using a technique called *corner stitching*. Corner stitching is a geometrical data structure for representing Manhattan shapes. It provides the underlying mechanisms that make possible most of Magic's advanced features. Corner stitching is simple, provides a variety of efficient searching operations, and allows the database to be modified quickly. What follows is a brief introduction to corner stitching. See [6] for a more complete description.

The basic elements in corner stitching are *planes* and *tiles*. Each cell contains a number of corner-stitched planes to represent the cell's geometries

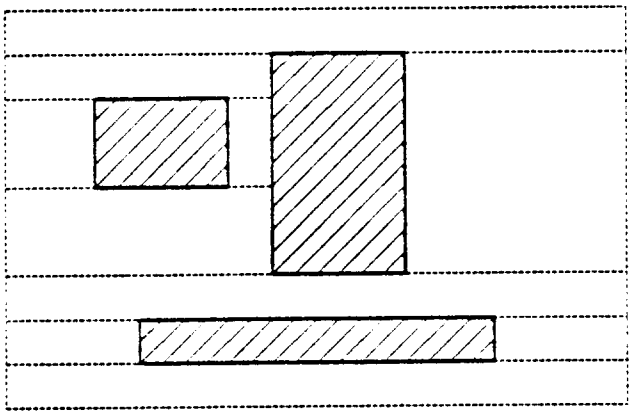

Figure 1. Every point in a corner-stitched plane is contained in exactly one tile. In this case there are three solid tiles, and the rest of the plane is covered by space tiles (dotted lines). The space tiles on the sides extend to infinity. In general, a plane may contain many different types of tiles.

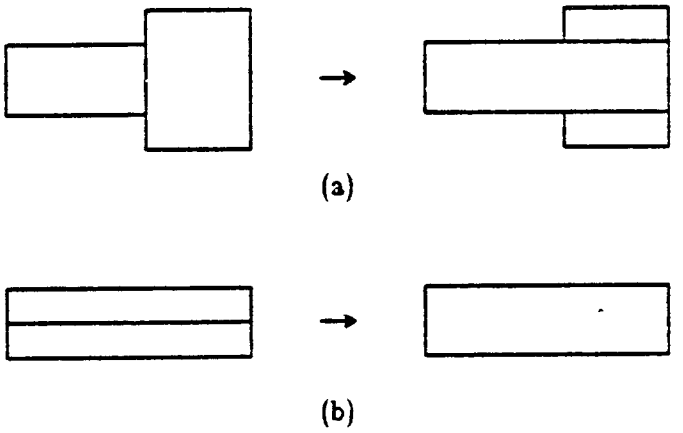

Figure 2. Areas of the same type of material are represented with horizontal strips that are as wide as possible, then as tall as possible. In each of the figures the tile structure on the left is illegal and is converted into the tile structure on the right. In (a) it is illegal for two tiles of the same type to share a vertical edge. In (b) the two tiles must be merged together since they have exactly the same horizontal span.

and subcells; each plane consists of a number of rectangular tiles of different types. There are three important properties of a corner-stitched plane, illustrated in Figures 1, 2, and 3:

Coverage: Each point in the x-y plane is contained in exactly one tile (Figure 1). Empty space is represented as well as the area covered with material.

Strips: Material of the same type is represented with horizontal strips (Figure 2). The strip structure provides a canonical form for the database and prevents it from fracturing into a large number of small tiles.

Stitches: Tiles are linked together at their corners. Each tile contains four of these links, called *stitches* (Figure 3).

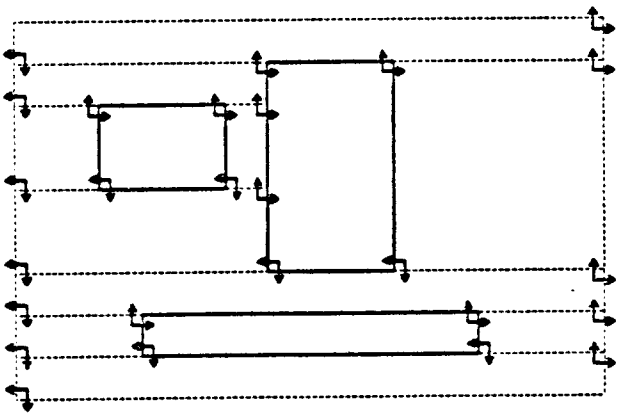

Figure 3. Each tile is linked to its neighbors with four pointers, called *corner stitches*. The corner stitches provide a form of two-dimensional sorting. They permit a variety of geometrical operations to be performed efficiently, such as searching an area or finding all the neighboring tiles on one side of a given tile.

The stitches permit a variety of search operations to be performed efficiently, including: finding the tile containing a given point; finding all the tiles in an area; finding all the tiles that are neighbors of a given tile; and traversing a connected region of tiles. The coverage property makes it easy to update the database in response to edits, and the strip property keeps the database representation small. To the best of our knowledge, corner stitching is unique in its ability to provide these efficient two-dimensional searches and yet permit fast updates of the kind needed in an interactive tool. The only disadvantage of corner stitching in comparison to less powerful data structures is that it requires more storage space (about three times as much space as structures based on linked lists of rectangles). Even so, the storage requirements do not appear to be a problem for chips likely to be designed in the next several years.

4. Abstract Layers

There are several ways in which corner-stitched planes might be used to represent the mask geometries in a cell. One alternative is to use a separate plane for each mask layer; each plane contains space tiles and tiles of one particular mask type. The disadvantage of this approach is that many operations, such as design-rule checking and circuit extraction, require information about layer interactions (such as polysilicon crossing diffusion to form a transistor, or implants changing the type of a transistor). With a separate plane per mask layer, these operations will spend a substantial amount of time cross-registering the information on different planes.

Another alternative is to place all the mask layers into a single corner-stitched plane. Since there can be only one tile at a given point in a given plane, different tile types must be used for each possible overlap of mask layers. This eliminates the registration problem, but results in a large number of small tiles where several mask layers overlap. Even though many of the layer overlaps are not significant (such as metal and implant), separate tile types have to be used to represent them. As a result, the database fragments into a large number of tiles, and the overheads for all operations increase.

The solution we chose for Magic lies between these two extremes. We decided to use a small number of planes, where each plane contains a set of layers that have design-rule interactions. If layers do not have direct design-

Plane	Tile Types
Poly-Diff	Polysilicon
	Diffusion
	Enhancement Transistor
	Depletion Transistor
	Buried Contact
	Poly-Metal Contact
	Diffusion-Metal Contact
	Space
Metal	Metal
	Poly-Metal Contact
	Diffusion-Metal Contact
	Overglass Via to Metal
	Space

Table I. The corner-stitched planes and tile types used to represent the mask information for an nMOS process with buried contacts and single-level metal. Since polysilicon and diffusion have design-rule interactions, they are placed in the same plane. Metal interacts with polysilicon and diffusion only at contacts, so it is placed in a separate plane. Contacts between metal and diffusion or polysilicon are duplicated in both planes.

rule interactions (such as poly and metal), they may be placed in different planes. Some layers, such as contacts, may appear in two or more planes. In our single-metal nMOS process there are two planes: one for polysilicon, diffusion, transistors, and buried contacts; and one for metal (see Table I).

We also decided not to represent every mask layer explicitly. Instead of dealing with actual mask layers, Magic is based around *abstract layers*. The abstract layers do not include implants, wells, buried contact windows, or contact vias. Instead, the abstract layers include separate tile types for each possible kind of transistor and contact. Magic generates the missing mask layers when it creates CIF files for fabrication. Table I gives the planes and abstract

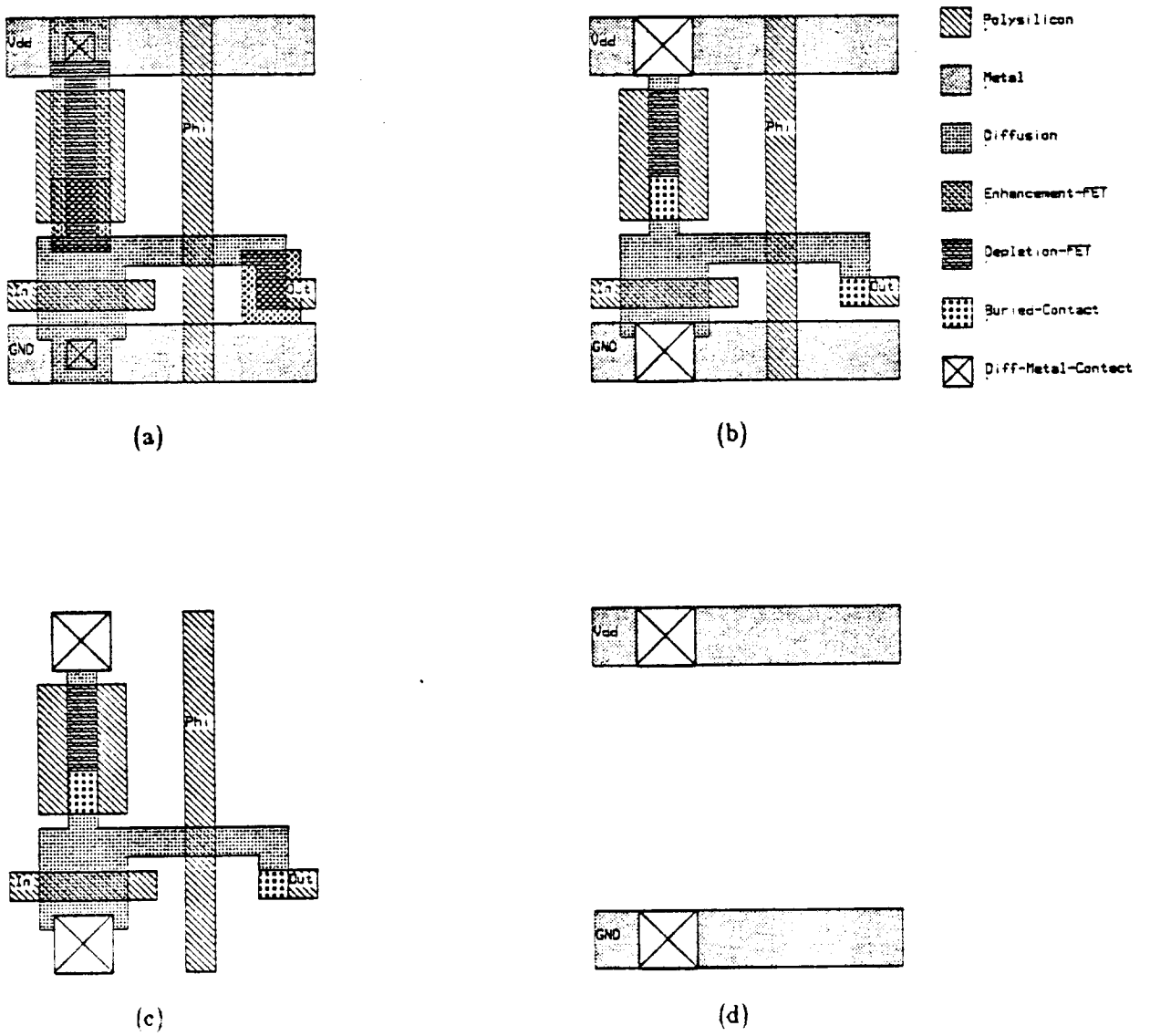

Figure 4. In Magic, transistors and contacts are drawn in an abstract form: (a) a three-transistor shift-register cell, showing actual mask layers; (b) the same cell as it is seen in Magic; (c) the information in Magic's poly-diff plane; (d) the information in Magic's metal plane. Contacts are duplicated in each plane.

layers used in Magic, and Figure 4 illustrates how the abstract layers are used in a sample cell. Abstract layers change the way a circuit looks on the screen, but they do not incur any density penalty.

The Magic design style is similar to sticks and symbolic systems such as Mulga [13] and VTVID [10], except that the geometries are fully fleshed. Designers draw the primary interconnection layers and simplified forms of contacts and transistors. Magic fills in the structural details. As in sticks, there are simple operations for stretching and compacting cells. The advantage of Magic's abstract-layer approach is that designers can see the exact size and shape of a cell while it is being edited, and they only work with a single representation of the circuit. When using sticks, designers go back and forth between the sticks and mask representation; the final size of the cell is hard to determine until it has been compacted and fleshed out. The following sections will show how the abstract layers simplify design-rule checking, plowing, and circuit extraction.

In addition to the planes used to hold mask geometry, each cell contains another plane to hold information about its subcells. Subcells are allowed to overlap in Magic; each distinct subcell area or overlap between subcells is represented with a different tile in the subcell plane. Each tile contains pointers to all of the subcells that cover the tile's area. By using corner-stitching in this way, it is easy to find subcell interactions and to determine which (if any) subcells cover a particular area.

5. Basic Commands

The basic set of commands in Magic is similar to the commands in Caesar [5,7]. Mask geometry is edited in a style like painting: a rectangle is placed over an area of the layout, and mask layers may be painted or erased over the area of the rectangle. Additional operations are provided to make a copy of all the "paint" in a rectangular area and copy it back at a different place in the layout. The corner-stitched representation is invisible to users.

Magic also provides commands for manipulating subcells. Subcells may be placed in a parent, moved, mirrored in x or y, rotated (by multiples of 90 degrees only), arrayed, and deleted. Subcells are handled by reference, not by copying: if a subcell is modified, the modifications will be reflected everywhere that the subcell is used.

6. Incremental Design-Rule Checking

Design-rule checking is an integral part of the Magic system. Our main goal was to make the checker very fast, particularly for small changes: the cost of reverifying a layout should be proportional to the amount of the layout that has been changed, not to the total size of the layout. To achieve this, Magic's design-rule checker runs continuously in the background during editing sessions. When the layout is changed, Magic records the areas that must be reverified. The design-rule checker then rechecks these areas during the

time when the user is thinking. For small changes, error information appears on the screen instantly (and also disappears instantly when the problem has been fixed). For large changes (such as moving one large subcell on top of another), it may take seconds or minutes for the design-rule checker to complete its job. In the meantime, the designer can continue editing. If reverification hasn't been completed when an editing session ends, the areas still to be reverified are stored with the cell so that reverification can be completed the next time the cell is edited. Error information is also stored with cells until the errors are fixed. With this mechanism, there is never a need to check a layout "from scratch."

Magic's basic rule-checker works from the edges in a design. Based on the type of material on either side of an edge, it verifies constraints that require certain layers to be present or absent in areas around the edge. There are several reasons why corner stitching and the abstract layers allow edge rules to be checked quickly. Each corner-stitched plane can be checked independently. All the "interesting" edges are already present in the tile structure, so there is no need to register different mask layers. The abstract layers make it unnecessary to check formation rules associated with implants and vias. Lastly, corner stitching provides efficient algorithms for locating all the edges in an area and for searching the constraint areas.

In addition to a fast basic checker, the incremental rule checker contains algorithms for handling hierarchy. When a cell in the middle of a hierarchical layout is changed, Magic checks interactions between this cell and its subcells, and also interactions between this cell and other cells in its parents and grandparents. More details on the basic DRC mechanism and on Magic's hierarchical approach can be found in [12].

7. Plowing

Plowing is a simple operation that can be used to rearrange a layout without changing the electrical circuit that it represents. To invoke the plow operation, the user specifies a vertical or horizontal line segment (the *plow*) and a distance perpendicular to it (the plow distance). See Figure 5. Magic

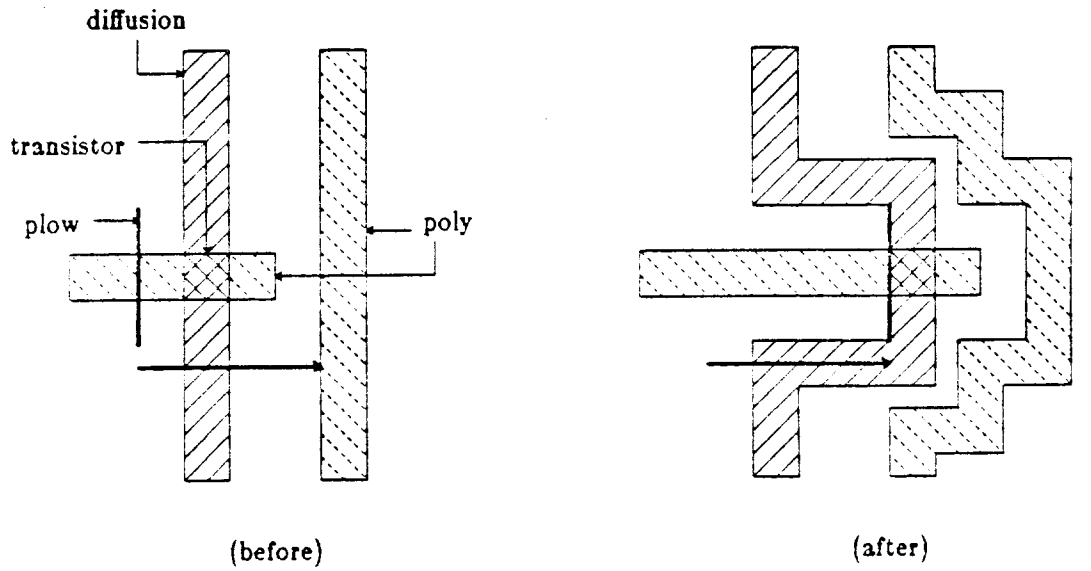

Figure 5. In plowing, a horizontal or vertical line is moved across the circuit, pushing material out of its way. Design rules and connectivity are maintained.

sweeps the plow for the specified distance, and moves and moves all material out of the area swept out by the plow. The edges of this material are likewise treated as plows, pushing other material in front of them. Mask geometry in front of the plow is compacted as it is moved, and mask geometry that crosses the initial position of the plow is stretched behind the plow. Jogs are inserted at the ends of the plow. The plow operation maintains design rules and connectivity so that it doesn't change the electrical structure of the circuit. Most material, such as polysilicon, diffusion, and metal, may be stretched or compacted by plowing; transistors and contacts may be moved, but their shape will not change.

Plowing provides all the operations of a sticks-based system, while still working with fully-fleshed geometry. If a large plow is placed to one side of a cell and then moved across the cell, the cell will be compacted. If a large plow is placed across the middle of the cell and moved, the cell will be stretched at that point. A small plow placed in the middle of a cell can be used to open up empty space for new transistors or wiring. Plowing may be used both on low-level cells containing only geometry, and on high-level cells containing subcells and routing. Plowing moves each subcell as a unit, without affecting the contents of the subcell.

The implementation of plowing is dependent on corner stitching, abstract layers, and the edge-based design rules. Corner stitching provides the fast

geometric operations used to search out plow areas. The abstract layers tell Magic about materials that cannot be stretched or compacted (such as transistors). The edge-based design rules indicate what must be moved out of the way when a particular edge of material is moved. By working from the same data structure used for editing and design-rule checking, the plowing operation avoids the overhead of converting between representations. See [11] for a detailed presentation of the plowing operation and its implementation.

8. Circuit Extraction and Cell Overlaps

The Magic database makes circuit extraction almost trivial for individual cells. Because of the abstract layers and corner stitching, the circuit is almost completely extracted to begin with. All that is needed is to traverse the tile structure and record information about what connects to what. There is no need to register layers or infer the structure and type of transistors and contacts: all this information is represented explicitly.

For hierarchical designs, the situation is complicated when cells overlap. Each cell uses a separate set of corner-stitched planes, so information from the separate planes must be combined in order to find out what connects to what. If arbitrary overlaps are allowed, then transistors may be split between cells, or may be formed or broken by cell overlaps. In this case, circuits cannot be extracted hierarchically, since the structure of a cell may be changed by the

way it is used in its parents.

One approach to the overlap problem is to prohibit cell overlaps. This has two drawbacks, however. First, it makes for clumsy designs, since overlap areas must be placed in separate cells. This makes it harder to understand designs and harder to re-use cells. Second, it doesn't eliminate the problems in circuit extraction, since information will still have to be registered along the boundaries of abutting cells. For example, a cell abutment can cause two separate transistors to join together.

Instead of prohibiting overlaps, we decided to restrict them. In Magic, cells may abut or overlap as long as this only connects portions of the cells without changing their transistor structure. Overlaps and abuttments may not change the type or number of transistors from what it would be without the overlap (e.g. polysilicon from one cell may not overlap diffusion from another cell, since this would create a new transistor). These restrictions can be verified by using a special set of design rules in the part of the design-rule checker that deals with cell overlaps.

Our solution still requires information to be registered between subcells, but it allows the extracted circuit to be represented (and extracted) hierarchically. The extracted circuit for any cell consists of the circuits of its subcells, plus the circuit of the cell itself, plus a few connections between the subcells.

9. Routing

Routing is the single most important area where we hope Magic will speed up the design process. Most of the Magic routing effort has been spent in two areas: a) creating a channel router that can work around obstacles in the channel (such as previously-placed interconnections and power and ground routing); and b) developing an interface between grid-based routers and non-gridded custom designs.

Magic uses a standard three-phase approach to routing. In the first phase, called *channel decomposition*, the empty space of the layout is divided up into rectangular channels. In the second phase, called *global routing*, nets are processed sequentially to decide which channels will be crossed by each. In the third phase, called *channel routing*, each channel is considered separately and wires are placed to achieve the necessary connections within the channel. Magic's channel decomposer (which is not yet implemented) will be based on the bottleneck approach of the BBL system [1]. Global routing (also not yet implemented) will use a standard wavefront approach [4]. Both of these will use corner-stitching to keep track of the channel space. The channel router has been implemented, and is an extended version of Rivest's greedy router [9]. Magic does not provide placement tools: in our design style, placement is an important architectural decision and must be handled by designers.

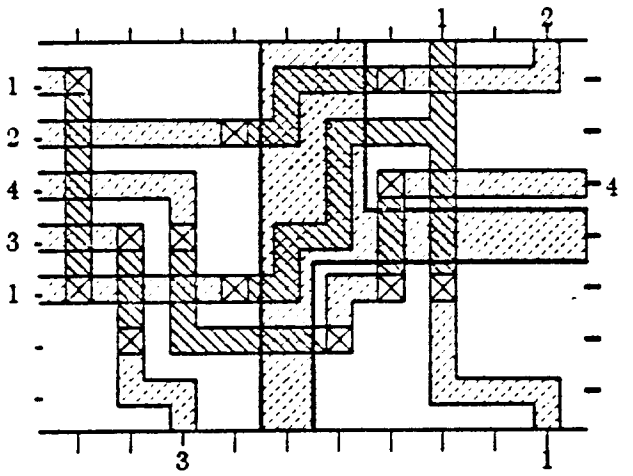

Figure 6. An example of routing with a single-layer obstacle in the channel. The router tries to avoid the thickest part of the obstacle if possible.

In order to make the routing tools useable in a custom design environment, we have developed a channel router that can work around obstacles in the channels. It is important for designers to be able to wire critical nets by hand, and to have the automatic routing tools route the less critical nets without affecting the hand-routed ones. It is also convenient to run power and ground routing tools as a separate step before signal routing, and have the signal router work around the power and ground wires. Where there are obstacles in the channels, Magic will route under them if possible, and will route around those that block both routing layers. For very large obstacles in one layer, such as a wide metal ground bus, Magic can make interconnections under the obstacles using river-routing. See [2] for details on how Rivest's greedy router has been extended to handle obstacles. Figure 6 shows an example of results produced by the Magic router.

The evasive router, combined with plowing and the other editing features, provides designers with considerable flexibility. Critical signals and power and ground can be routed by hand. Then the router can be invoked to complete the rest of the interconnections. If the router is unable to make all connections, the final ones can be placed by hand. Or, plowing can be used to rearrange the placement and the router can be re-run. The plowing operation will maintain the existing connections.

We have also extended the standard routing approach to handle designs that are not based on a uniform routing grid. Most channel routers assume a uniform grid based on the minimum wire spacing: channel dimensions must be an integral number of grid units, and all wires must enter and leave channels on grid points. Unfortunately, custom cells are not usually designed with the router's grid in mind, so the cell boundaries and terminals do not line up on a

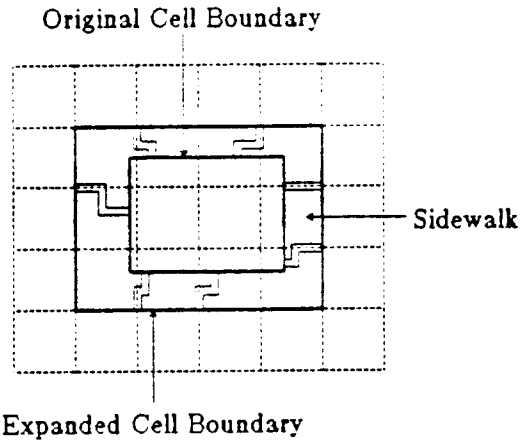

Figure 7. In the sidewalk approach, each cell is enlarged so that its boundary is grid-aligned. Then connections on the edge of the original cell are routed to grid points on the outside of the sidewalk.

master grid. We are experimenting with two approaches to this problem, called *sidewalks* and *flexible grid*.

The sidewalk approach is illustrated in Figure 7, and involves a pre-routing step where all cells are expanded so that their dimensions are integral grid units. This additional cell area is called its *sidewalk*. In addition, wires are added to connect the terminals of the cell to grid points on the outer edge of the sidewalk. After the sidewalk generation stage, everything is grid aligned so standard routing tools can be used. Magic currently implements the sidewalk approach. Sidewalks are inefficient because the sidewalk areas cannot be used for channel routing, even though they usually contain little material. Sidewalks typically cause the channels to be reduced in size by 2-3 tracks and 2-3 columns.

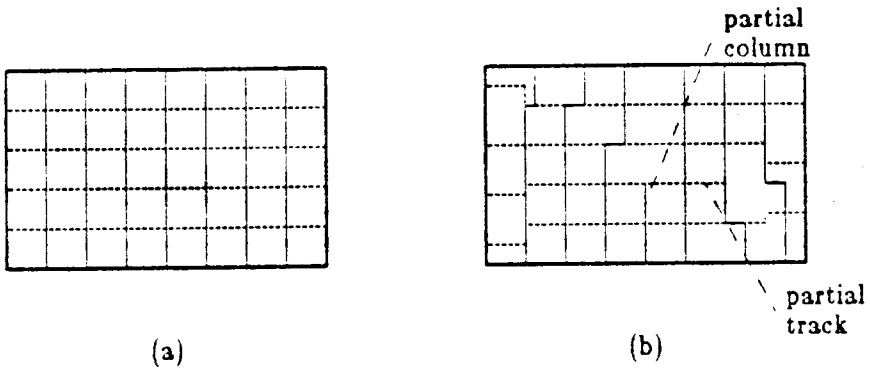

Figure 8. Rather than expand cells to grid points as in the sidewalk approach, the flexible grid approach modifies the track and column structure of the channel. The channel is grid-based in the center, but the grid lines jog at the edges to meet up with non-gridded connections. (a) shows the standard orthogonal channel structure, and (b) shows a channel whose grid structure has been flexed. The flexible grid approach can result in tracks or columns that don't extend all the way across the channel.

The flexible grid approach distributes the sidewalks among the channels by jogging the track and column structure at the ends to match up with connection points that don't fall on grid lines. This is illustrated in Figure 8. In the flexible grid approach, wasted space occurs within the channel because some columns and channels cannot extend all the way across the channel. However, there appears to be less wasted space in this approach than in the sidewalk approach. In the worst case, the wasted space is equivalent to two tracks and two columns per channel. If connection points are sparse, however (and this is usually the case), the flexible grid approach has almost zero wasted space. We are still in the early stages of exploring this alternative.

10. User Interface

Magic displays the layout on a color display, and users invoke commands by pointing on the display with a mouse and then pushing mouse buttons or typing keyboard commands. Magic provides multiple overlapping windows on the color display. Each window is a separate rectangular view on a layout. Different windows may refer to different portions of a single cell, or to totally different cells. Windows allow designers to see an overall view of the chip while zooming in on one or more pieces of the chip; this permits precise alignments of large objects. Information can be copied from one window to another.

11. Technology Independence

Although Magic contains a considerable amount of knowledge about integrated circuits, the information is not embedded directly in code. All the circuit information is contained in a technology file that Magic reads. This file defines the abstract layers for a particular technology, the corner-stitched planes used to represent them, and the assignment of abstract layers to planes. It tells how to display the various layers and defines the semantics of the paint and erase operations from Section 5 (for example "if poly-metal-contact is painted over diffusion, erase the diffusion and place poly-metal-contact tiles on both the poly-diff and metal planes"). The technology file contains the design rules used in design-rule checking and in plowing. Lastly, it tells how to fill in the structural details of transistors and contacts when generating CIF for circuit fabrication. The technology file format is general enough to handle a variety of nMOS and CMOS processes. Our technology file for an nMOS process with buried contacts and single-level metal contains about 130 lines.

12. Implementation

The implementation of Magic was begun in February of 1983. By early April 1983, a primitive version of the system was operational. Although the first system was based on corner stitching and abstract layers, it provided user features only equivalent to Caesar. During the summer of 1983 implementa-

Subsystem	Implementation Status
Edge-based DRC	Operational 9/1/83
Hierarchical and Continuous DRC	Operational 11/1/83
Circuit Extraction	Not begun
Plowing	Simplified version operational 10/1/83 Full version expected 1/1/84
Net List Editing	Operational 5/1/83
Channel Decomposition	Expected 1/1/84
Global Router	Expected 2/1/84
Channel Router with Obstacle Avoidance	Operational 10/1/83
Multiple Windows	Operational 11/1/83

Table II. The implementation status of Magic.

tion was begun on the subsystems for routing, multiple windows, plowing, and design-rule checking. As of this writing, most of the advanced features are either operational or expected to be operational in the near future. See Table II. The system has been in use since April 1983 by the designers of a 32-bit microprocessor [8], and since September 1983 by several dozen students in an introductory VLSI design class.

Operation	Speed
Painting tiles into corner-stitched database	200 tiles/sec.
Design-rule checking	200 tiles/sec.
Simplified Plowing	100 tiles/sec.
Channel routing ("Deutsch's difficult example," 60 nets)	3 sec.

Table III. Some sample measurements of the speed of the Magic system. All measurements were made on a VAX-11/780.

Magic is written in C under the Berkeley 4.2 Unix operating system for VAX processors. The current implementation works only with AED color displays with special Berkeley microcode extensions. Altogether, Magic contains approximately 45000 lines of code. Table III gives a few sample performance measurements of pieces of the system.

13. Conclusions

We have not yet had enough designer experience with Magic to evaluate the system thoroughly, but the initial response has been favorable. The only major problem encountered so far has been one of education: if designers are accustomed to working with actual mask layers, then the abstract layers in Magic are confusing at first. This problem was exacerbated in the early versions of the system because the design-rule checker wasn't implemented. With continuous feedback from the checker, we hope that it will be much easier for designers to learn the abstract layers. We expect that the abstract layers will be easier for designers to work with than the actual mask layers, since they hide many irrelevant details.

The pieces of the Magic system work well together. Corner stitching appears to be a complete success: it provides all the operations needed to implement Magic's advanced features, and results in simple and fast algorithms. The design-rule checker's edge-based rule set meshes well with the

corner-stitched data, and is used also for plowing. The abstract layers simplify the design rules, provide information needed for plowing and circuit extraction, and simplify the designer's view of the layout.

We hope that Magic's flexibility will change the VLSI layout process in two ways. First, we hope that it will enable designers to experiment much more than previously. At the cell level, they can use plowing to rearrange cells quickly and easily. Cells can be designed loosely, then compacted. At the chip level, plowing and the routing tools can be used together to rearrange the floorplan, route the connections, compact or stretch, and try again. The ability to experiment means that students will be able to develop better intuitions about how to design chips; it also means that designers will be able to fix bugs and enhance performance more easily.

Second, we hope that Magic will make it easier to reuse pieces of designs. To design a new chip, a designer will select cells from a large library, use plowing and painting to make slight modifications in their shape or function to suit the new application, and perhaps design a few new cells. Then the routing tools will be used to interconnect the cells. We hope that this approach will result in a substantial reduction in design time for large circuits.

14. Acknowledgements

As tool builders, we depend on the Berkeley design community to try out our new programs, tell us what's wrong with them, and be patient while we fix the problems. Without their suggestions, it would be extremely difficult to develop useful programs. The SOAR design team, and Joan Pendleton in particular, have been invaluable in helping us to tune Magic. Randy Katz, Dave Patterson, and Carlo Séquin all provided helpful comments on this paper. The Magic work was supported in part by the Defense Advanced Research Projects Agency (DoD) under contract N00034-K-0251, and in part by the Semiconductor Research Cooperative under grant number SRC-82-11-008.

15. References

[1] Chen, N.P., Hsu, C.P., and Kuh, E.S. "The Berkeley Building-Block Layout System for VLSI Design." Memorandum No. UCB/ERL M83/10, Electronics Research Laboratory, University of California, Berkeley, February, 1983.

[2] Hamachi, G.T. and Ousterhout, J.K. "A Switchbox Router with Obstacle Avoidance." In this technical report.

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

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

[5] Ousterhout, J.K. "Caesar: An Interactive Editor for VLSI Layouts." *VLSI Design*, Vol. II, No. 4, Fourth Quarter 1981, pp. 34-38.

[6] Ousterhout, J.K. "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools." Technical Report UCB/CSD 82/114, Computer Science Division, University of California, Berkeley, December 1982. To appear in *IEEE Transactions on CAD/ICAS*, January 1984.

[7] Ousterhout, J.K. "The User Interface and Implementation of Caesar." Technical Report UCB/CSD 83/131, Computer Science Division, University of California, Berkeley, August 1983.

[8] Patterson, D.A. ed. "Smalltalk on a RISC." Final reports from CS292R, Computer Science Division, University of California, Berkeley, April 1983.

[9] Rivest, R.L. and Fiduccia, C.M. "A Greedy Channel Router." *Proc. 19th Design Automation Conference," 1982, pp. 418-424.*

[10] Rosenberg, J. et al. "A Vertically Integrated VLSI Design Environment." *Proc. 20th Design Automation Conference, 1983, pp. 31-36.*

[11] Scott, W.S. and Ousterhout, J.K. "Plowing: Interactive Stretching and Compaction in Magic." In this technical report.

[12] Taylor, G.S. and Ousterhout, J.K. "Magic's Incremental Design Rule Checker." In this technical report.

[13] Weste, N. "Virtual Grid Symbolic Layout." *Proc. 18th Design Automation Conference*, 1981, pp. 225-233.

[14] Williams, J. "STICKS - A Graphical Compiler for High Level LSI Design." *Proc. 1978 National Computer Conference*, pp. 289-295.

Magic's Incremental Design-Rule Checker

George S. Taylor and John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences Department
University of California
Berkeley, California 94720

ABSTRACT

The Magic VLSI layout editor contains an incremental design-rule checker. When the circuit is changed, only the modified areas are rechecked. The checker runs continuously in background to keep information about design-rule violations up-to-date. This paper describes the basic rule checker, which operates on edges in the layout, and the techniques used to perform incremental checking on hierarchical designs.

Keywords and Phrases: design-rule checking, interactive layout editor

1. Introduction

Almost all existing design-rule checking (DRC) programs are batch oriented [1] [2]. They read in a complete circuit layout and check the entire design. If the circuit is changed, the only way to find out whether design rules have been violated is to recheck the entire design, no matter how small the change or how large the design. For chips with tens of thousands of transistors, batch DRC run may require hours of computer time.

This paper describes a different approach to design-rule checking. As part of the Magic VLSI layout editor [3], we have built a checker that operates *incrementally*. When the layout is modified, Magic records which areas have changed and rechecks only those areas. While the user continues editing, the checker runs in background and highlights errors as it finds them. There is no set-up time because it works from the same data structure used to represent the layout. Since most changes made with the interactive editor are small and the checker is fast, it can usually display errors instantly.

The user's view of design-rule checking is a simple one. As he edits the circuit, small white dots appear over areas that contain layout errors. As soon as the errors are fixed, the white dots go away. Error informaton is stored with the design and it will reappear during the next editing session if the violation has not been fixed. This information is always kept up-to-date, so there is never any need to run a batch checker.

In the next section, we describe Magic's internal representation for a layout and explain how particular features contribute to fast incremental checking. Section 3 describes how the basic checker works from edges in the layout and how design rules are specified. Section 4 shows how we use the basic checker for incremental checking of individual cells, and section 5 describes how hierarchical designs are handled. Section 6 gives measurements of the checker's speed.

2. Representation of a Layout

In Magic, a layout is represented as a hierarchical collection of cells. Each cell contains mask information plus pointers to subcells. For now, we will consider only a single cell at a time (Section 5 generalizes the solution to handle hierarchical designs).

Magic represents the mask layers of a cell with rectangular *tiles*, which means that it handles only Manhattan geometries. Each tile indicates the type of mask layer it represents. Tiles are connected to form *planes* by a technique called *corner-stitching* [2] illustrated in Figure 1. The tiles in a plane are non-overlapping and cover it completely. Empty areas are covered with tiles of type "space."

Each cell contains several planes of mask information. Mask types that interact (such as polysilicon and diffusion) are stored together in the same

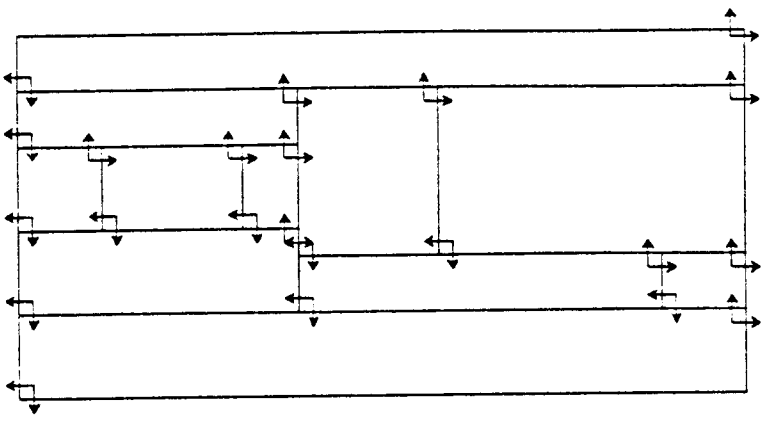

Figure 1. An example of a corner-stitched plane. Each plane contains tiles of different types that cover the entire area of the plane (space tiles are used where there is no mask material). Each tile contains four pointers that link it to neighboring tiles at its corners. The pointers make it easy to find all the material in a given area.

plane, while those that do not interact (such as polysilicon and metal) are stored in different planes. Contacts between mask types on different planes are represented in both of them. Our nMOS process has two planes: one for metal and one for polysilicon, diffusion, and transistors.

Instead of working directly with physical mask layers, Magic uses *abstract layers* to represent structures such as transistors and contacts. The abstract layers appear in the database as tiles with special types. For example, instead of representing an enhancement transistor as a polysilicon tile over a diffusion tile, it is represented with a tile of type "enhancement transistor." A more complete explanation of the abstract layers is given in [3]. What matters here is that all the interesting features are represented explicitly: there is no need to cross-register diffusion and polysilicon to discover the transistors.

The design-rule checker takes advantage of Magic's database in three ways. First, the corner-stitched tiles allow DRC to find material in a given area very quickly. Second, division of mask information into planes allows the checker to work with one plane at a time, ignoring irrelevant geometry on other planes. Third, there is no need to extract features by registering layers: the abstract layers represent the important features explicitly. Because of these features, there is no need for the checker to manage a separate structure of its own: it works directly from the layout database.

3. The Basic Checker

This section describes the basic design-rule checking paradigm used to validate an area of a single corner-stitched plane. Later sections show how this basic checker is used to perform incremental checks on a single cell, and then on a hierarchy of cells.

3.1. Edge-based Rules

Magic's design rules are based on edges between tiles. Each rule can be applied in any of four directions, two for horizontal edges and two for vertical edges. The rule database contains a separate list of rules for each possible combination of materials on the first and second sides of an edge. In its simplest form, a rule specifies a distance and a set of mask types: only the given types are permitted within that distance on the second side of the edge. This

area is referred to as the *constraint region*. See Figure 2.

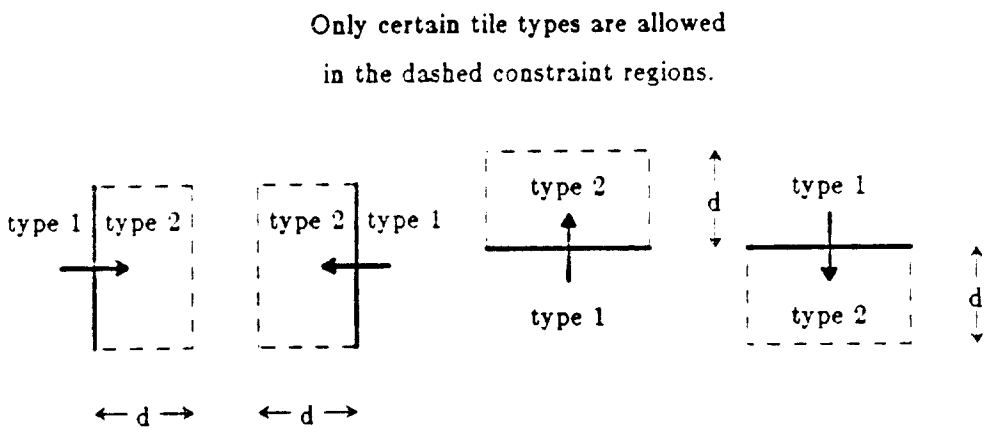

Figure 2. Design rules are applied at the edges between tiles in the same plane. A rule is specified in terms of *type 1* and *type 2*, the materials on either side of the edge. Each rule may be applied in any of four directions, as shown by the arrows. The simplest rules require that only certain mask types can appear within distance d on *type 2*'s side of the edge.

Unfortunately, this simple scheme will miss errors in corner regions, as shown in Figure 3. To eliminate these problems, the full rule format allows the constraint region to be extended past the ends of the edge under some circumstances. See Figure 4 for an illustration of the corner rules and how they work. Table 1 gives a complete summary of the information in each design rule.

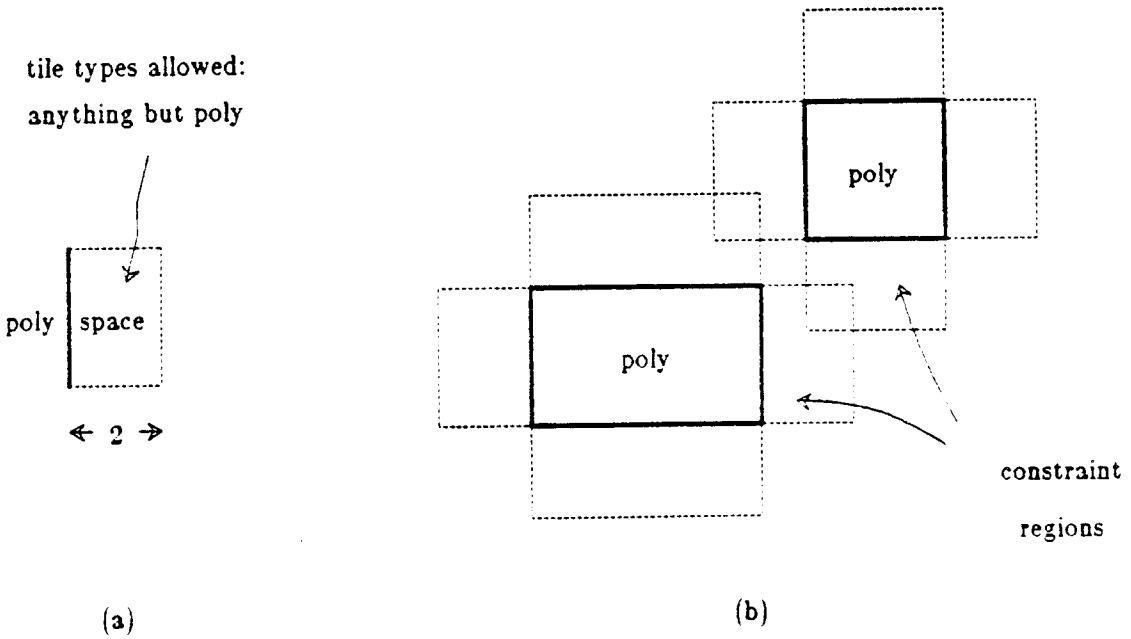

Figure 3. If only the simple rules from Figure 2 are used, errors may go unnoticed in corner regions. For example, the polysilicon spacing rule in (a) will fail to detect the error in (b).

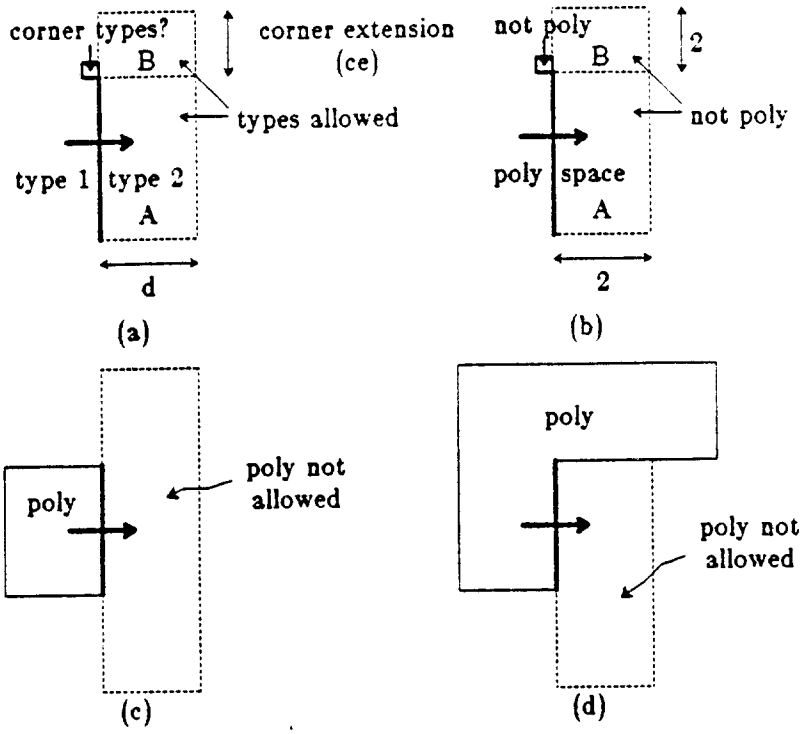

Figure 4. The complete design rule format is illustrated in (a). Whenever an edge has *type 1* on its left side and *type 2* on its right side, the area A is checked to be sure that only *types allowed* are present. If the material just above and to the left of the edge is one of *corner types*, then area B is also checked to be sure that it contains only *types allowed*. A similar corner check is made at the bottom of the edge. Figure (b) shows one of the polysilicon spacing rules, (c) shows a situation where corner extension is performed on both ends of the edge, and (d) shows a situation where corner extension is made only at the bottom of the edge.

Parameter	Meaning
type 1	Material on first side of edge.
type 2	Material on second side of edge.
d	Distance to check on second side of edge.
layers allowed	List of layers that are permitted within d units on second side of edge.
corner types	List of layers that cause corner extension.
ce	Amount to extend constraint area when corner types match.

Table 1. The parts of an edge-based rule.

3.2. Applying the Rules

To check an area of a single plane, Magic must first find all the edges in that area. This is accomplished by searching for all the tiles in the area. The corner-stitched data structure is well suited to searches of this sort: see [4]. For each tile, the checker examines its left and bottom sides (the top and right sides of the tile will be checked by the neighbors on those sides). Since the tile may have neighbors of different types on the same side, the checker searches through all the neighbors to divide the side of the tile into edges with a single material on each side.

To process an edge, the mask types on each side of it are used to index into the rule table to find the list of rules for that kind of edge. Each rule in the list is checked, and white dots are displayed for any areas where the constraints are not satisfied. For each edge there are two rule applications: left-to-right and right-to-left (for vertical edges) or bottom-to-top and top-to-bottom (for horizontal edges). A different list of rules is applied in each direction, since the layers are reversed.

3.3. Specifying Design Rules

Design rules are specified in a technology file that contains the rules and other technology-specific information. When Magic starts executing, it reads this file and builds the rule table. Initially we specified rules in the detailed form of Table I, with one line for each edge rule. This scheme proved to be

unworkable, because there were many rules and it became difficult to convince ourselves that the rule set was complete and correct.

In order to simplify the process of creating rule sets, Magic now permits rules to be specified with high level macros for width and spacing. For example, the macro

spacing ef DP 1

is expanded into several rules to verify that types e and f (enhancement and depletion transistors) are always separated from types D and P (diffusion-metal contacts and poly-metal contacts) by at least one unit. The macro

width pPBef 2

is expanded into the set of edge rules needed to verify that the entire region containing any of the five types P, B, e, f or p (polysilicon) is always at least two units wide.

Most of the rules for our processes are simple width and spacing checks, so these two macros considerably simplify the writing of rule sets. Our nMOS rule set contains 8 width rules, 6 spacing rules, and 9 of the detailed edge rules for situations that cannot be handled by the width and spacing rules (e.g. transistor overhangs). Magic expands these 23 high-level rules into 126 detailed edge rules. The complete high-level rule set for nMOS is given in the Appendix.

The width and spacing macros make Magic's checker more efficient because the width and spacing rules are symmetric. If layers x and y are too close together, the violation can be detected from either an edge of x or an edge of y. This means that it is unnecessary to check the rules from both edges. Magic takes advantage of this symmetry by checking width and spacing rules in only two directions (left-to-right and bottom-to-top). In addition, symmetric rules mean that corner extension is only necessary on one end of each edge. Since most of the detailed edge rules come from the width and spacing macros, this speeds up the checking process by almost a factor of two.

4. Continuous Design-Rule Checking

This section shows how the basic checker is used to provide continuous incremental rule validation. As in the previous section, we consider only single-cell designs here.

In order to perform DRC incrementally, Magic maintains two extra kinds of information with each cell, stored in the same form as mask layers. First, Magic keeps information about rule violations that have been detected but haven't been corrected. The violations are represented by error tiles that cover the areas where rule constraints are not satisfied. The second kind of information consists of tiles describing the areas of the circuit that need to be reverified. The error tiles and the reverify tiles are stored in separate corner-

stitched planes. Each cell contains its own error and reverify planes.

When a designer changes a cell, Magic creates reverify tiles that cover the area modified. The design-rule checker runs in background while Magic is waiting for the designer to enter the next command. DRC first searches for reverify tiles. Then it invokes the basic checker over the area covered by each tile found. The basic checker reverifies the area on each of the cell's planes, updates error tiles, and erases the reverify tile. Changes to the error information are reflected immediately on the graphics screen.

If the designer invokes a command while the checker is running, the checker stops so that the command can be processed without delay. After the command finishes, the checker resumes by starting over on the area that it was working on just before the interruption. Large reverify tiles are broken up into small ones before checking, in order to reduce the amount of work that might have to be repeated. When there are large areas to be reverified, the checker works across the design in a style like "Pac-Man," gobbling up reverify tiles and spitting out error tiles.

If incremental checking is done carelessly, errors may not be detected when new violations are introduced, and error information may be left in the database even after the violations have been corrected. Figure 5 illustrates the problem and Magic's solution. When an area is modified, error information may be affected in both the area that was modified and in the surrounding

area (for example, material in area A may be too close to something in the surrounding area B). We call the surrounding area the *halo*. Its width is equal to the largest distance in any design rule. Error information must be recomputed in the modified area and its halo. However, errors in the halo don't necessarily involve the inner modified area. They may come from interactions between the halo and a second halo outside it. To regenerate errors in the first halo correctly, information in the second halo must be considered.

If area A of Figure 5 were modified, Magic would recheck it by deleting all error information in A and B. The checker would then generate new error information in both areas by invoking the basic checker over areas A, B and C. Any errors found during this process would be clipped to the area of A and B, so that error information outside the region where errors were erased would not be affected.

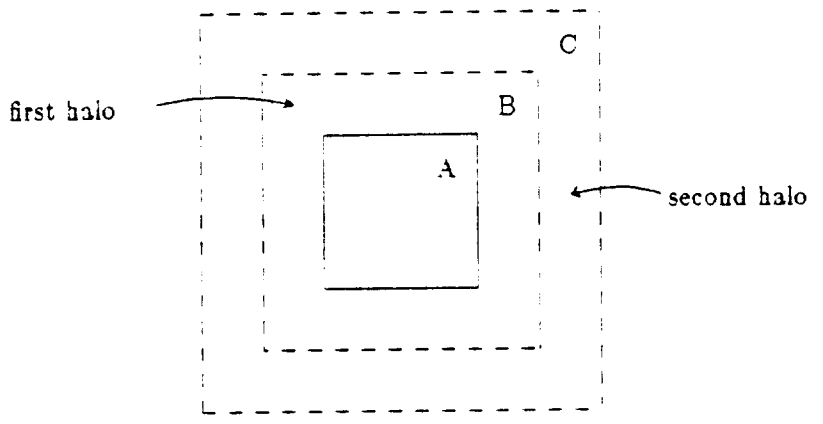

Figure 5. If area A is modified, the design-rule checker erases existing error information in both A and B. Errors in B could have come from information in A, B or C, so all three areas must be checked to regenerate all of the errors. The width of the *halos* B and C is equal to the largest distance in any design rule.

The reverify and error tiles are stored with cells so that they are not lost at the end of an editing session. Normally, there will be no reverify tiles left at the end of a session, but if a large area has been changed recently, it is possible that it won't have been reverified when the session ends. In this case, the reverify tiles are written to disk with the cell. When the cell is read in during the next editing session, the design-rule checker will notice the reverify tiles and continue the reverification process. The reverify and error tiles are identical to the tiles used to represent mask layers, except that they are not manipulated directly by the designer.

5. Hierarchical Checking

Most of the layouts created with Magic consist of hierarchical cell structures rather than single cells (Figure 6). Each cell may contain subcells, and the subcells may overlap other subcells or mask information in the parent. A subcell may be appear any number of times in any number of parents.

In hierarchical designs, errors can arise in any of three ways:

a) the mask information of an individual cell may be incorrect;

b) a subcell may interact incorrectly with another subcell; and

c) a subcell may interact incorrectly with mask information in its parents.

Magic's incremental checker includes facilities to detect all of these errors. Overlapping subcells are no more difficult to handle than subcells that merely

abut, because interaction errors are possible in either case.

5.1. Simple Checks and Interaction Checks

Two overall rules guide the hierarchical checker. First, the mask information in every cell is required to satisfy the design rules by itself, without consideration of subcells. Second, each cell and its subcells must together satisfy all the design rules, without consideration of how that cell is used in its parents. If the layout is viewed as a tree structure, the first rule means that each node of the tree must be consistent, and the second rule means that each subtree must be consistent.

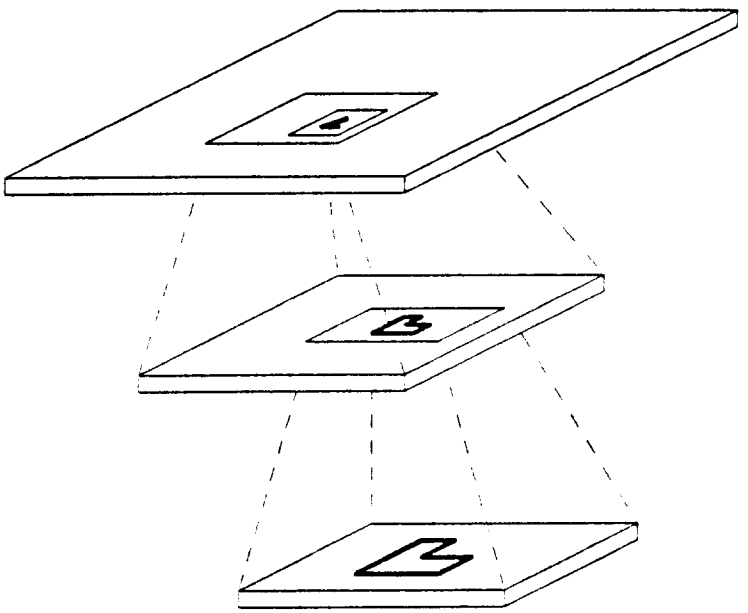

Figure 6. Circuits are defined by cells arranged in a hierarchy. If mask information is changed in a low-level cell, Magic checks to be sure that the cell is consistent by itself and that there are no illegal interactions in parents or grandparents.

The overall rules result in two kinds of design-rule checking. The first rule is verified by running the basic checker over the planes containing mask information for each cell; this is called a *simple check*. The second rule is verified with an *interaction check* which considers interactions involving subcells. Each cell uses separate planes to hold its mask information, so interaction checks must combine information from different planes.

To make an interaction check on an area, the hierarchical structure is "flattened" to produce a new set of corner-stitched planes that combines all the information from all cells in the area to be checked. This includes mask information from the parent cell, plus mask information from subcells and sub-subcells, and so on. Once all the mask information in the area has been collected into a single set of planes, the basic checker is invoked on these planes in the standard fashion (halo expansion is performed as described in Section 4). Errors arising from the interaction check are placed in the parent cell.

Interaction checks are more expensive than basic checks, since they involve flattening a piece of the hierarchy. Fortunately, interaction checks can often be avoided. For example, if an area contains no subcells, then there is no need to perform an interaction check on that area. A simple check will find all errors. The interaction check can also be avoided if there is only a single subcell in an area, with no other subcells or mask information nearby. In

this case any errors must come from within the subcell, and those errors will be found by checks made within that cell. Interaction checks are necessary only in areas where a subcell is within one halo distance of mask information or another subcell. Even then, we only need to check the the area around the interaction.

5.2. Checking Upward in the Hierarchy

When a cell is modified, simple checks and interaction checks have to be performed within that cell, and also within its parents in the hierarchy. For example, suppose mask information has been edited within a cell. Then a simple check must be performed within that cell, as well as an interaction check if there are subcells near the modified area. However, these two checks are not sufficient. If the modified cell is a subcell of other higher-level cells, then the change may have introduced interaction problems within the higher-level cells. For each parent of the modified cell, an interaction check must be performed over the area of the modification. Interaction checks must also be performed in grandparents, and so-on up to the top-level cell in the hierarchy. In the cell that was modified, both simple and interaction checks must be performed, but in the parents and grandparents only interaction checks are necessary.

Magic uses two kinds of verify tiles to handle the two kinds of checks. When a cell is modified, "verify-all" tiles are placed in that cell to signify that both simple and interaction checks must be performed. At the same time,

"verify-interactions" tiles are placed in parents and grandparents to indicate that interaction checks have to be performed. The background checker keeps track of which cells in the database contain verify tiles and performs each kind of check wherever necessary.

In the worst case, the hierarchical algorithm could result in the modified area being rechecked once at each level of the hierarchy above the cell that was changed, with a separate flatten operation required for each check. However, in deep hierarchies most of the interaction checks are avoidable: in cells far above the modified one, the modified area will almost certainly appear in the middle of a single subcell with no mask information or other subcells nearby. Unless there are many large subcell overlaps, any given area of mask information is likely to require an interaction check at only one point in the hierarchy.

5.3. Arrays

One other form of hierarchical check arises because Magic has an array construct. To simplify the creation of cell arrays, Magic contains a special array facility: each subcell may consist of either a single instance or a one- or two-dimensional array of identical instances. Because of the array construct, there is actually a third overall rule that guides the hierarchical checker: each array must satisfy all the design rules, independently of other information in the parent containing the array. Whenever a change is made to an array, the

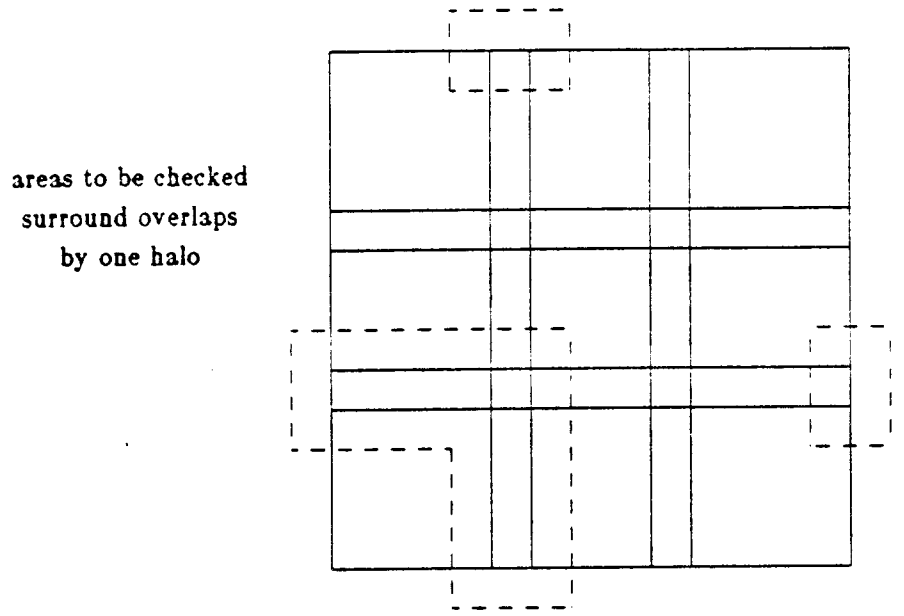

Figure 7. An array is internally consistent if the three dotted areas satisfy the design rules. All possible interactions between elements of the array are identical to the ones that occur these three regions.

array structure is reverified by checking the three areas shown in Figure 7.

6. Implementation and Performance

The design-rule checker is written in C. Its 2000 lines of code are divided into roughly equal thirds for building the internal rule table from the technology file, implementing the basic checker on one plane, and providing for hierarchical checking.

The incremental checking system has just recently become operational. We've made preliminary measurements on single cells with the untuned system. The basic checker processes 200 tiles per second on a VAX 11/780 running Unix. To compare Magic's performance with that of other systems, we

state their speeds in terms of transistors checked per second in Table 2.

A typical change to a circuit involves only a few tiles, so the cost of incremental reverification is dominated by the size of the halos. From this, we estimate that roughly 50 tiles have to be checked per command in an nMOS design. This requires about one-fourth of a second of CPU time.

The average number of edges found per tile is 2.5, but only 1.8 of these have different mask types on the two sides of the edge. An average of 1.7 rules are applied per non-trivial edge.

7. Conclusions

Magic's design-rule checker demonstrates that incremental checking is feasible. We think that circuit designers will find that continuous feedback reduces the time needed to create new designs or modify existing ones. The key to the incremental checker is low overhead: the ability to run from the same database as the interactive editor, the ability to find important edges in the layout quickly, and the ability to find nearby material quickly. The two

System	Transistors / second
Lyra [2]	2
Baker [1]	3
Mart [5]	6-8
Magic	10-15

Table 2. Performance of several design rule checkers. All of the programs were run on a VAX 11/780.

features of Magic's database that reduce overhead are the corner-stitched tile planes and the abstract mask layers. Extending the checker to work in hierarchical designs frees the designer from tedious reverification of interactions when subcells are revised.

8. Acknowledgements

Gordon Hamachi, Bob Mayo, and Walter Scott all participated in discussions that led to the incremental checker and provided many useful comments on drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD), under Contract No. N00034-K-0251.

9. References

[1] C. M. Baker and C. Terman, "Tools for verifying integrated circuit designs," *Lambda* (now *VLSI Design*) Vol. 1, No. 3 (1980), pp. 22-30.

[2] M. H. Arnold and J. K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking," *Proc. 19th Design Automation Conference*, June, 1982, pp. 530-36.

[3] J. K. Ousterhout, G. T. Hamachi, R. N. Mayo, W. S. Scott and G. S. Taylor, "Magic: A VLSI Layout System," included in this technical report.

[4] J. K. Ousterhout, "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools," Technical Report UCB/CSD 82/114, Computer Science Division, University of California, Berkeley, December, 1982. To appear in *IEEE Transactions on CAD/ICAS*, January, 1984.

[5] B. J. Nelson and M. A. Shand, "An Integrated, Technology Independent, High Performance Artwork Analyzer for VLSI Circuit Design," Technical Report VLSI-TR-83-4-1, VLSI Program, Division of Computing Research, CSIRO, Eastwood, SA 5063, Australia, April, 1983.

10. Appendix

To illustrate how Magic is programmed for a particular technology, this section lists the design rules for an nMOS process with buried contacts and a single level of metal. Most rules are specified using width and spacing macros which Magic expands into detailed lower-level rules. Detailed edge and four-way rules may also be specified directly. Table 3 gives the abbreviations that we use for the names of mask types.

Poly/Diffusion plane:	s	space
	d	diffusion
	p	polysilicon
	D	diffusion-metal contact
	P	polysilicon-metal contact
	B	buried contact
	e	enhancement transistor
	f	depletion transistor
Metal plane:	s	space
	m	metal
	X	metal-diffusion contact
	Y	metal-polysilicon contact

Table 3. Single letter abbreviations for the names of mask types.

The rules in Table 4a define minimum line widths and feature sizes. The first three rules are for the line widths of diffusion, metal and polysilicon. The last five rules define the sizes of contacts and transistors. The *types* field may include one or more mask types. Magic creates a detailed edge rule for all combinations of one member of the *types* field, and one of the mask types in the same plane that is not included in the *types* field.

	types	d	*reason*
width	dDBef	2	*diffusion*
width	pPBef	2	*polysilicon*
width	mXY	3	*metal*
width	D	4	*diff/metal contact*
width	P	4	*poly/metal contact*
width	B	2	*buried contact*
width	e	2	*efet*
width	f	2	*dfet*

Table 4a. Width rules.

Table 4b contains spacing rules. We distinguish between spacing rules for types that can never be adjacent and spacing rules that apply only when two pieces of material are separated. In either case, Magic creates a number of detailed edge rules in a manner similar to that for width rules.

The width and spacing macros can be used to specify most symmetrical constraints for a particular technology. The detailed edge rules created from the width and spacing macros are applied only from left-to-right across

	types 1	types 2	d	can be adjacent?	*reason*
spacing	ef	DP	1	no	*transistor – contact*
spacing	e	f	3	no	*efet – dfet*
spacing	B	e	3	no	*buried contact – efet*
spacing	dDBef	dDBef	3	yes	*diff – diff*
spacing	pPBef	pPBef	2	yes	*poly – poly*
spacing	mXY	mXY	3	yes	*metal – metal*

Table 4b. Spacing rules.

vertical edges in the layout, and from bottom-to-top across horizontal edges. These edge rules always check one corner, also.

To specify asymmetrical constraints and constraints that apply alongside edges but not in corners, we use the explicit edge and fourway rules listed in Table 4c. The fourway rules are applied in both directions across all edges in the layout. They also trigger corner checks on both ends of every edge. The edge rules in Table 4c are similar to the ones derived from the width and spacing macros, but could not be written conveniently in either of those forms.

	type 1	type 2	d	layers allowed	corner types	ce	reason
edge	d	spP	1	s	spP	1	*diff – poly spacing*
edge	p	sdD	1	s	sdD	1	*diff – poly spacing*
edge	D	sp	1	s	sp	1	*diff – poly spacing*
edge	P	sd	1	s	sd	1	*diff – poly spacing*
fourway	ef	s	1	0	0	0	*trans can't touch space*
fourway	B	dD	4	sdpDPBf	sdpDPBef	3	*b_c – c/et spacing*
fourway	f	B	3	B	0	0	*b_c next to d/et must be 3x?*
fourway	ef	p	2	pP	p	2	*poly overhang transistor*
fourway	ef	d	2	dD	d	2	*diff overhang transistor*

Table 4c. Edge and fourway rules.

Plowing:
Interactive Stretching and Compaction in Magic

Walter S. Scott and John K. Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

Abstract

The Magic layout editor provides a new operation called *plowing*, for stretching and compacting Manhattan VLSI layouts. Plowing works directly on the mask-level representation of a layout, allowing portions of it to be rearranged while preserving connectivity and layout-rule correctness. The layout and connectivity rules are read from a file, so plowing is technology independent. Plowing is fast enough to be used interactively. This paper presents the plowing operation and the algorithm used to implement it.

Keywords and Phrases: interactive layout editor, stretching, compaction.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00034-K-0251

1. Introduction

Plowing is a new operation provided by the Magic layout editor [OHMST 84] for stretching and compacting Manhattan VLSI layouts. It allows designers to make topological changes to a layout while maintaining connectivity and layout rule correctness. Plowing can be used to rearrange the geometry of a subcell, compact a sparse layout, or open up new space in a dense layout. In a hierarchical environment plowing also allows cell placement to be modified incrementally without the need for rerouting. To avoid dependence on a particular technology, plowing is parameterized by a set of layout and connectivity rules contained in a technology file.

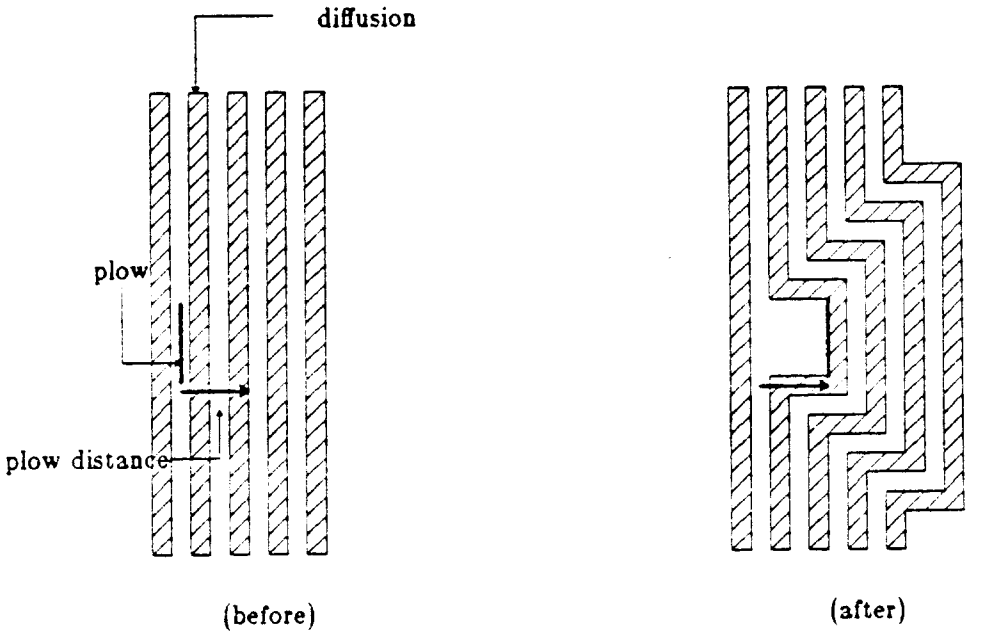

Figure 1. Plowing opens up new space in a dense layout. Geometry is pushed in front of the plow, subject to layout-rule constraints. The connectivity of the original layout is maintained. Jogs are inserted automatically where necessary.

Conceptually the plowing operation is very simple. The user places either a vertical or a horizontal line segment (the *plow*) over some part of a mask-level representation of the layout, and then gives the direction and the distance the plow is to move. Plowing can be done up, down, to the left, or to the right. (The rest of this paper will assume plowing to the right.) The plow is then moved through the layout by the distance specified. It catches vertical edges (boundaries between materials) as it moves and carries them along with it. Since only edges are moved, material behind the plow is stretched and material in front of the plow is compressed. Figure 1 shows how plowing can be used to open up new space. Figure 2 shows how it can be used for stretching. Plowing can be used to compact an entire cell by placing a plow to the left and plowing right, then placing a plow at the top and plowing down.

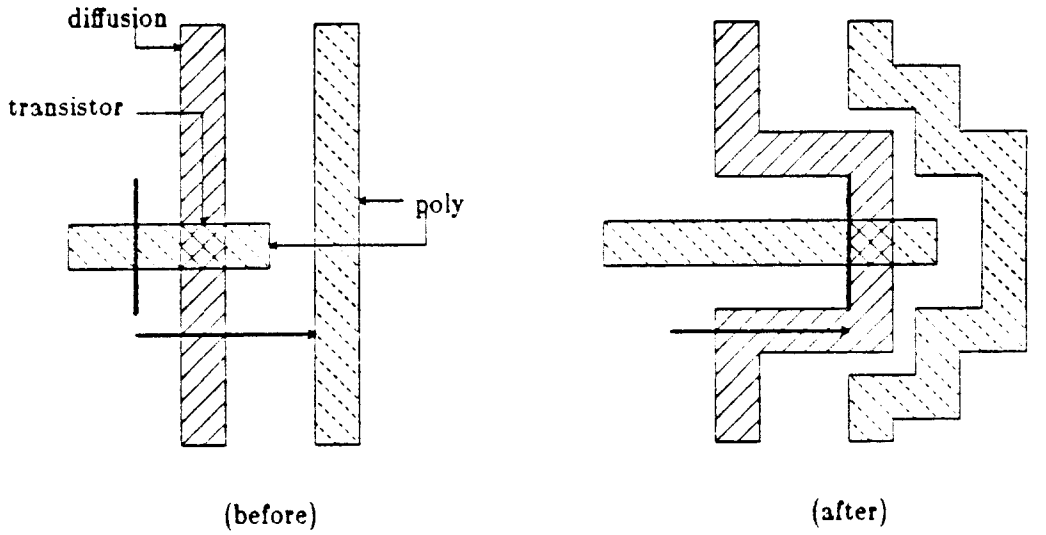

Figure 2. Material to the left of the plow is stretched. Material to the right is compressed. Objects such as transistors do not change in size.

Plowing is so named because each of the edges caught by the plow can cause edges in front of it to move in order to maintain connectivity and layout-rule correctness. These edges can cause still others to be moved out of the way, recursively, until no further edges need be moved. A mound of edges thus builds up in front of the plow in much the same manner as snow builds up on the blade of a snowplow.

Section 2 of this paper discusses plowing in the context of previous work. Sections 3 and 4 introduce the plowing algorithm for a single mask layer. Section 5 extends it to multiple mask layers and hierarchical designs. Finally, Section 6 presents performance measurements and our experience with plowing in the Magic system.

2. Background

VLSI layouts are difficult to modify. Because of this, designers are often committed to the initial choice of implementation, rather than being able to experiment with alternatives. Existing cells often cannot be re-used in subsequent designs because they don't quite fit; it is typically easier to redesign a new cell from scratch than to modify an old one. Bugs in a dense layout are hard to fix, leading to a debugging cycle which can take days.

Many of these difficulties stem from the fact that seemingly small changes to a layout can have disproportionately large effects. Sometimes this is for

electrical reasons. For example, in ratio logic such as nMOS, changes in the size of one transistor may necessitate changes in the sizes of others. However, even purely topological changes—those which preserve the electrical properties of the layout—can require much more work than the size of the change would suggest. As Figure 1 illustrated, merely opening up new space in a layout can cause effects which ripple outward over a much larger area. Rearranging the internal geometry of a cell or modifying the placement of cells in a floor plan can be similarly expensive because of the need to maintain connectivity with the surrounding material.

Previous attempts to cope with the re-arrangement problem have used symbolic design or sticks [RBDD 83, West 81, Will 78]. In the symbolic/sticks approach, designers enter layouts in an abstract form containing zero-width wires, contacts, and transistors. The sticks form is then run through a compactor to generate actual mask information. As part of the compaction, the circuit elements are moved as close together as the layout rules permit. In a sticks design style, cells can be designed loosely without worrying about exact spacings, since the spacings will be determined by the compactor. However, it is not necessarily easy to make major changes to a sticks cell once it has been entered. Virtual grid systems like Mulga and VTVID provide mechanisms for adding new grid lines uniformly across a cell, but it is still difficult to make large topological changes.

The plowing approach has all the advantages of sticks. It allows cells to be designed loosely and then compacted. In addition, plowing can be used to rearrange cells or open up new space, either across the whole cell or in one small portion. Small changes can be made in one area without having to recompact the entire cell (a global recompaction may potentially shift every geometry in the cell). The plowing approach lets the designer see the final sizes and locations of all objects as he is editing; in the sticks approach, it is hard to predict the final structure of a cell from its abstract form, so compaction must be used frequently to see the results of a change to the sticks.

3. Simple plowing algorithm

Plowing works by finding *edges* and moving them. An edge is a boundary, parallel to the plow, between material of two different types. When an edge moves, the material to its left is stretched, and the material to its right is compressed. In this section we will describe how plowing works when only a single mask layer is present. This material will be assumed to have a minimum width of w, and a minimum separation of s. Edges will always be boundaries between this material and "empty" space.

The fundamental step in plowing is to move a single edge. This step involves determining which other edges must move as a consequence of this motion. The following discussion presents plowing as though it moves a given

edge by first recursively sweeping all other edges out of its way, and then sliding the edge into the newly opened space. Section 4 will present a better scheme for ordering edge motions than this depth-first recursion.

3.1. Finding edges

Figure 3 depicts a trivial layout consisting of three unconnected pieces of diffusion. The edge labelled *e* is to be moved to a final position indicated by the arrowhead. This could be either because *e* was caught by the plow, or because it is being moved to make room for some edge to its left. At a very minimum, the rectangular area labelled *A* must be swept clear of any material before the edge can be moved. However, because of the spacing rule, any material inside area *B* would then be too close to the newly moved edge. Con-

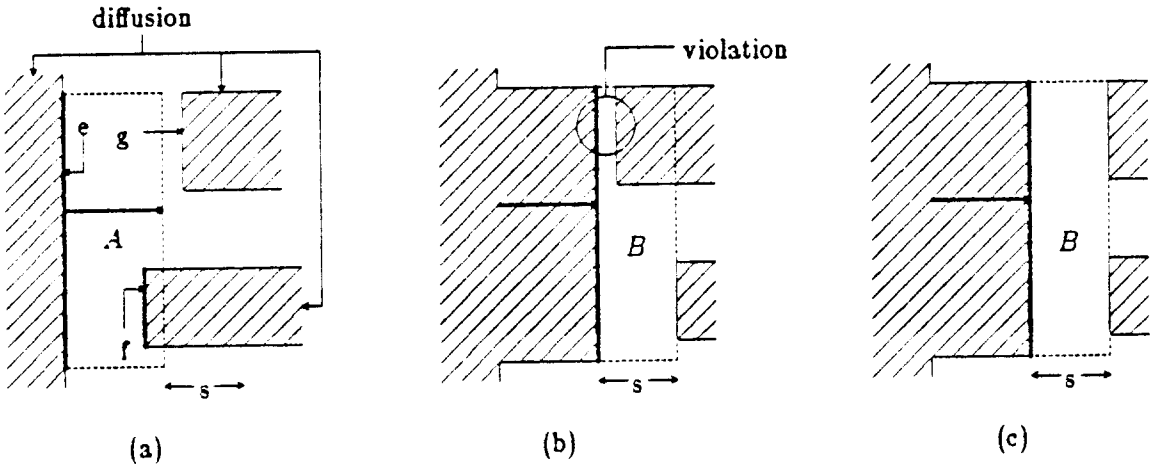

Figure 3. When the edge *e* moves, all edges in area *A* (the area swept out by *e*) must be moved (a). Moving only these edges results in edge *f* moving but not edge *g*. This leaves a layout-rule violation (b) between *e* and *g*. Searching area *B* as well as area *A* avoids this problem. The two areas are referred to collectively as the *umbra* of edge *e*.

sequently, the area to be swept includes both areas A and B. The union of these two areas is referred to as the *umbra* of the edge e*.

Plowing must also search above and below the umbra to prevent the edge from sliding too close to other edges above or below it. Figure 4a shows why this is necessary. If material were moved out of the umbra alone, as in Figure 4b, the result is electrical disconnection. To avoid this, plowing must also move edges out of the areas above and below the umbra. The correct result is

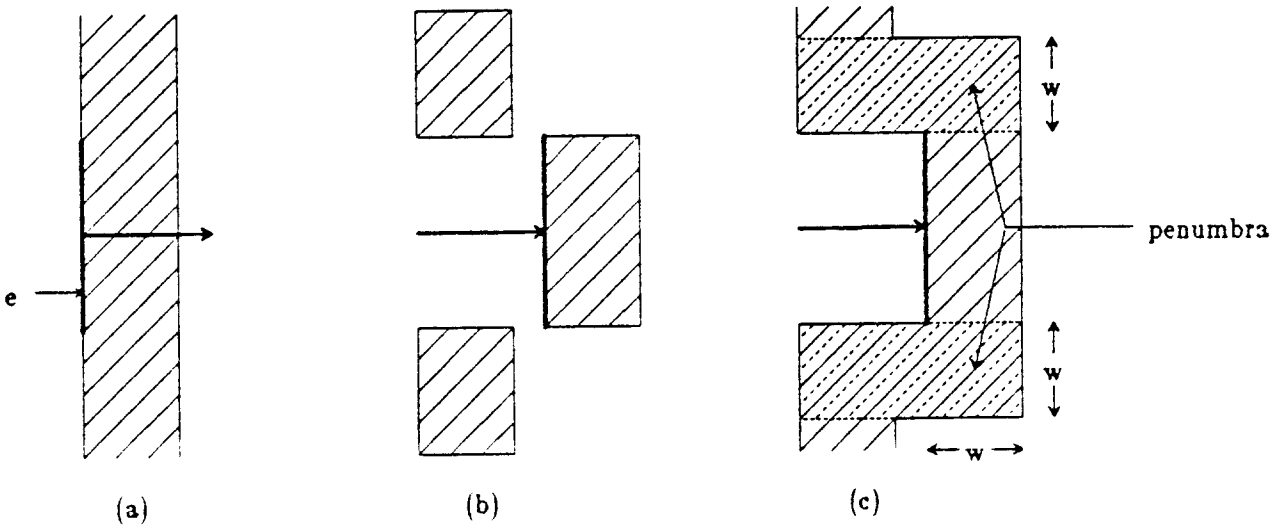

Figure 4. When the edge e moves (a), edges in its umbra must be moved to the right. If only edges in the umbra are moved, however, the result can be electrical disconnection (b). To avoid this, plowing also moves edges in the *penumbra* to the right, giving the correct result shown in (c). This has the effect of inserting jogs automatically. The height of the penumbra is w, the minimum-width for diffusion. If diffusion had been to the left of e instead of to the right, the height of the penumbra would have been s, minimum-separation.

* In a solar eclipse, the *umbra* is that portion of the moon's shadow from which the sun appears to be completely eclipsed. The *penumbra* is the part of the shadow surrounding the umbra from which the sun appears only partially eclipsed. In plowing, the umbra contains edges directly in the path of an edge being moved, while the penumbra contains edges not in the path but nonetheless too close.

shown in Figure 4c. The areas above and below the umbra are referred to collectively as the *penumbra*. Jog insertion is an automatic consequence of searching the penumbra. Moving edges out of the penumbra also prevents electrical shorts, as can be seen by reversing the roles of material and space in Figures 4a-4c.

The left-hand boundary of the penumbra is not always aligned with the edge being moved. Instead, this boundary is formed by following the outline

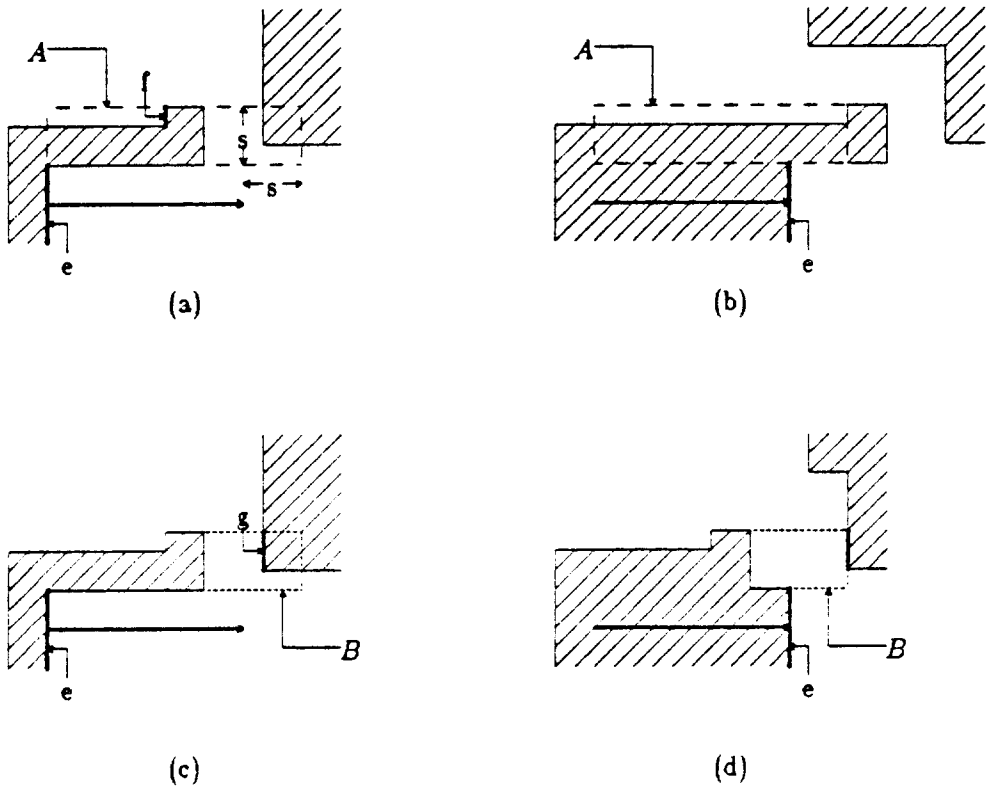

Figure 5. If *e*'s penumbra included all of area *A*, as shown in (a), then edge *f* would be found and moved, resulting in (b). This is undesirable, since *f* need not move in order to preserve layout-rule correctness and connectivity. A better definition of the penumbra would be area *B* only, as shown in (c). Searching this area would result in only the edge *g* being found and moved, as is necessary to preserve layout rule correctness.

of the material forming the edge, as illustrated in Figure 5. This insures that the penumbra contains only those edges which must move in order to preserve layout rule correctness and connectivity. The umbra and penumbra of an edge are collectively referred to as its *shadow*. The shadow of *e* contains all the edges which must move as a direct consequence of moving *e*.

3.2. Sliver prevention

The rules described in Section 3.1 guarantee that plowing never moves one vertical edge too close to another. However, they do allow violations to be introduced between horizontal segments that are formed when material is stretched. These violations take the form of slivers of material or space whose height is less than the minimum allowed. Eliminating such slivers requires that their left-hand edges be moved, as illustrated in Figure 6. The left-hand edge of each sliver lies along the left-hand boundary of the penumbra, so it can be found when tracing the outline of the penumbra.

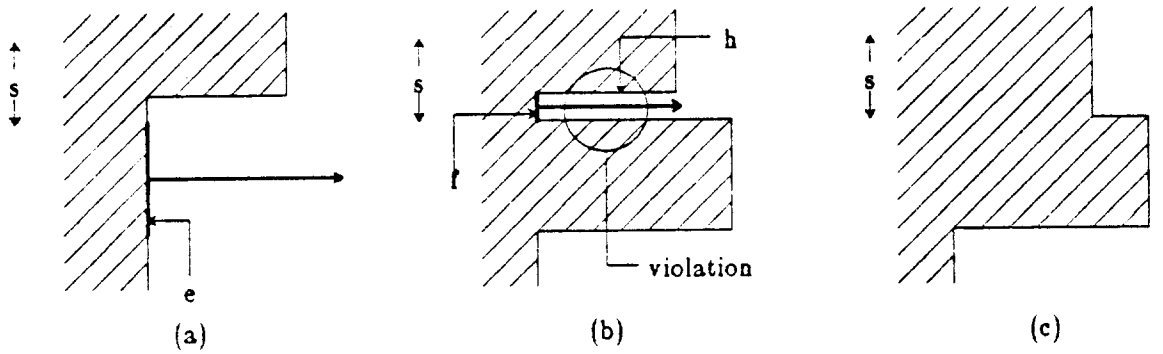

Figure 6. When the edge *e* moves (a), a sliver of space is introduced below the horizontal segment *h*, as shown in (b). To correct this, the left-hand edge of this sliver, *f*, is moved along with *e*, but only as far as the right-hand end of the segment *h* (c).

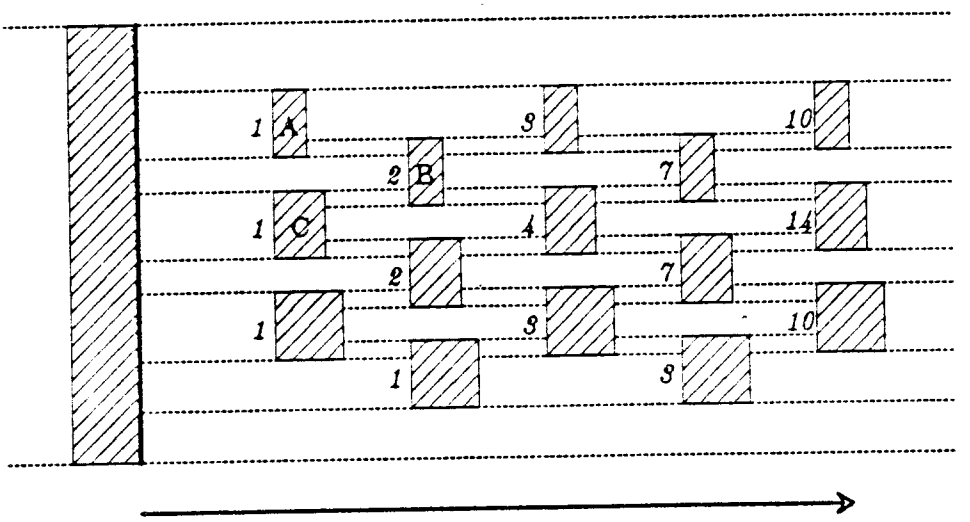

Figure 7. This lattice structure causes exponential worst-case behavior in the depth-first plowing algorithm when edges in the shadow are processed from top to bottom. The objects (*A*, *B*, etc.) must be incompressible to cause this worst-case behavior. Object *B* is moved once when object *A* moves, then slightly farther when object *C* moves. The numbers to the left of each object show how many times each of its edges is moved.

4. Breadth-first vs. Depth-first Search

In the previous section, plowing was described as a depth-first search in which all edges to the right of a given edge were moved before the edge itself. While this approach is conceptually clear, it has poor worst-case behavior. An N-tier lattice structure as illustrated in Figure 7 requires on the order of 2^N edge motions, because plowing performs the recursive search to the right of an edge each time the edge is moved. If, as in the example, each edge must be moved once for each of its two neighbors to the left, the edges at the right-hand side of the lattice are moved a number of times that is exponential in the number of tiers.

Instead, plowing waits until the final position of an edge is known before it performs the search to the right of that edge. This strategy causes the number of edge motions to be linear in the number of edges in the lattice. (A detailed explanation is given in [Oust 84].)

A simple way to insure that edges are moved only once their final positions are known is to use breadth-first search. Magic maintains a list of edges to be moved, sorted in order of increasing x-coordinate. On each iteration, the leftmost edge is removed from the list and the shadow to its right is searched. Any edges discovered by this search are placed in the list along with the amount they must move. Since the final position of an edge can only be affected by edges to its left, the final position of the leftmost edge in the list is always known.

The depth-first algorithm allowed the layout to be modified incrementally as plowing progressed, since an edge was never moved until the area into which it was moving had been cleared. Incremental modification is impossible with breadth-first search, since edges to the right will not be moved as long as there are queued edges to the left of them waiting to be moved. Instead of actually updating the layout as it progresses, the breadth-first version of plowing stores with each vertical edge segment the distance it moves. When the shadows of all edges have been searched, and the distance each edge moves has been determined, plowing invokes a post-pass to update the layout from

the information stored with each edge.

However, if the layout is not modified until all edges have been processed, special care must be taken to avoid the generation of slivers. Figure 8 illustrates the problem. To process each edge correctly, it is important to know what other edges have been already been processed and what their final positions will be. In general, the plowing algorithm must consider edges whose final positions will be in the shadow, rather than those whose initial positions are in the shadow.

The success of the breadth-first algorithm depends on the fact that left-to-right plowing never changes the order of edges along any horizontal line, and never changes any vertical coordinates. Furthermore, edge has stored

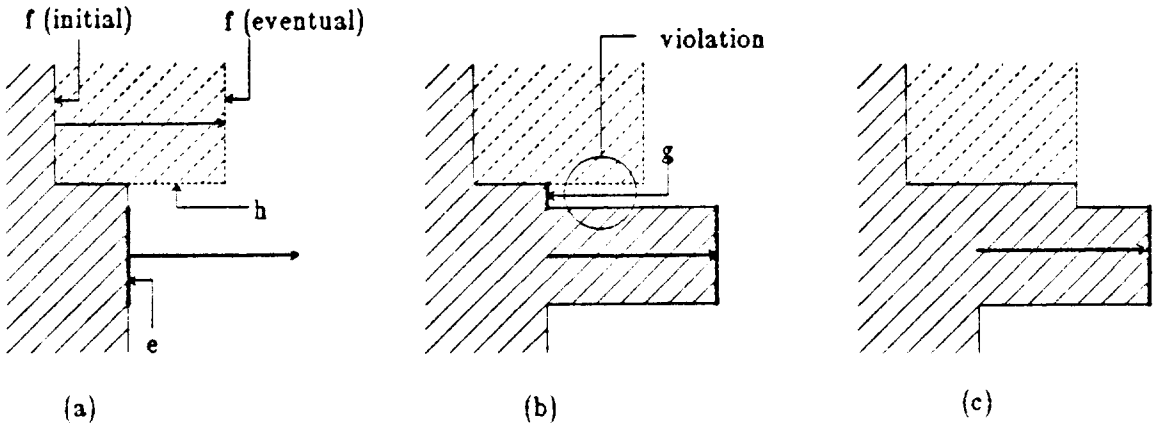

Figure 8. When processing an edge in the breadth-first approach, it is important to use information about the final positions of edges that have already been processed. In (a), it has already been decided to move edge *f*, but the edge will not actually be moved until all other edges have been processed. If edge *e* is processed without considering the new position of *f*, a sliver will result as shown in (b). Instead, the plowing algorithm must consider the eventual positions of edges that have already been processed, to produce the result of (c).

with it the distance it is going to move. As a consequence, plowing can use the initial layout structure for searching, and yet can easily find all objects whose final coordinates fall in a given area.

5. Extensions for real layouts

This section extends the simple plowing algorithm of the previous two sections to handle multiple mask layers. Plowing is also extended to handle features, such as transistors and contacts, whose size should not be changed, and to allow noninteracting mask layers, such as metal and polysilicon, to slide past each other. Finally, since layouts in Magic may be hierarchical, this section closes with a description of how plowing handles hierarchy.

5.1. Multiple mask layers

The simple version of plowing assumed that the shadow extended to the right of the final position of a moving edge by either w (the minimum-width rule) if material lay to the right of the edge, or s (the minimum-separation rule) if material lay to the left of the edge. This insured that the shadow included all edges directly in the path of the edge being moved. Since the same layout rule applied between the edge being moved and any other edge, all edges found during the search of the shadow would have to move.

With more than one mask layer there may be more than one layout rule to apply for a given edge. For example, in our nMOS process, the minimum

separation between diffusion and polysilicon is 2 microns, while that between two pieces of diffusion is 6 microns. Both of these rules apply at an edge between diffusion and empty space.

To insure that the shadow contains all edges which must move, the shadow must extend beyond the area the edge sweeps out by the worst-case layout rule distance applying to that edge. As Figure 9 illustrates, however, not all of the edges found in the shadow search will actually need to move. Each edge found must be checked for its minimum allowable separation from the edge being moved. Fortunately, this can be done very quickly using the same techniques as those used in Magic's incremental layout-rule checker [TaOu 84].

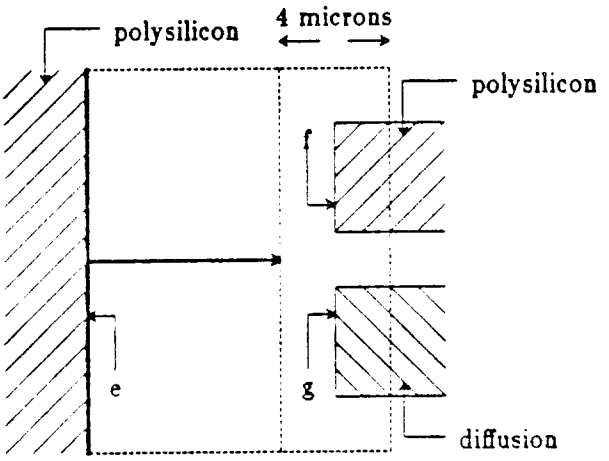

Figure 9. The area of a shadow search is determined by the worst-case layout rule. However, not all edges in that area will have to be moved. Edge *f* must move, because the separation between two polysilicon features must be 4 microns and edge *e* approaches to within 2 microns of *f*. Edge *g* need not move since the minimum separation between polysilicon and diffusion is only 2 microns.

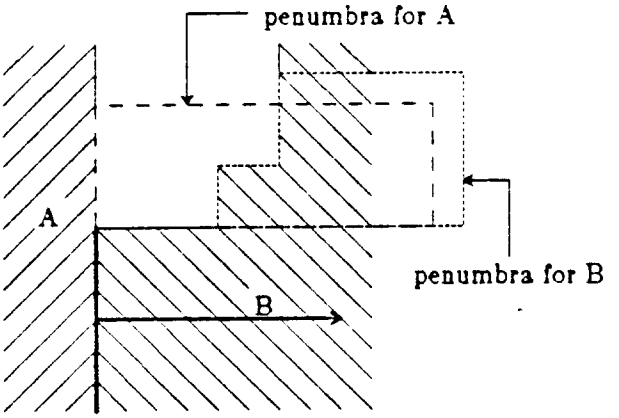

Figure 10. An edge between two different types of material has a penumbra for each. The spacing rules for material of type *A* are applied in *A*'s penumbra. The minimum-width rule for material of type *B* is applied in *B*'s penumbra. The sizes of each penumbra may be different because of the different layout rules applied in each.

If the edge being moved has material on both sides, there is really a penumbra for each type of material. The layout rules applied while searching each penumbra will in general be different. Slivers must be prevented along the boundaries of both penumbra. See Figure 10 for an example.

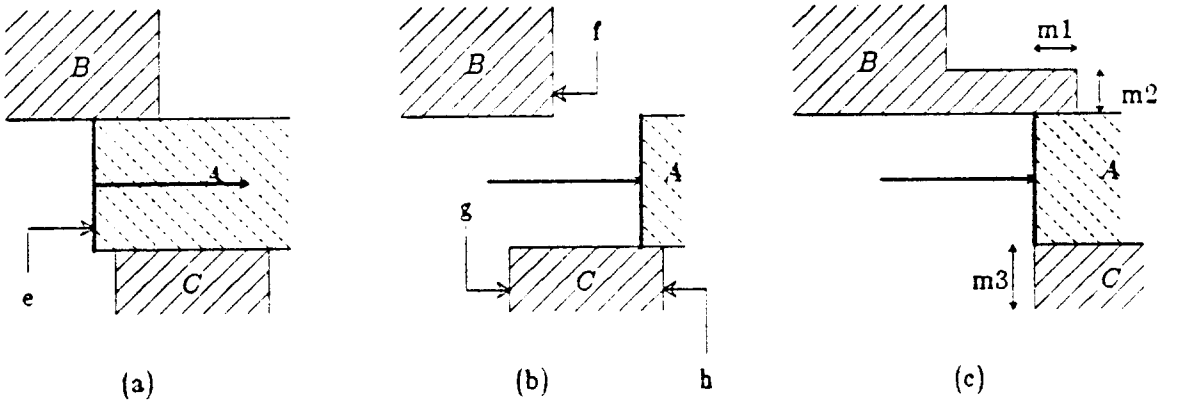

Figure 11. If edge *e* is plowed, material *A* may disconnect from *B* and *C*. To prevent this, a minimum-width segment of edges *f* and *g* is dragged along with *e*. The edge *g* is moved not to maintain connectivity (which would have been achieved by moving *h*), but to prevent *C* from being uncovered. In (c), *m1* is the lesser of the minimum widths for *A* and *B*, *m2* is the minimum width for *B*, and *m3* is the minimum width for *C*.

Multiple mask layers require extra caution to maintain connectivity with material above and below an edge being moved. In the single-layer scheme, the penumbra search guarantees that the material does not become disconnected. However, the penumbra search follows the outline of a single type of material, so it will not by itself guarantee that two adjacent materials of different types will remain connected (see Figure 11).

Special actions must be taken during the penumbra search to handle horizontal edges between different materials. First, if two materials share a horizontal edge, then Magic guarantees that one material does not slide past the end of the other: it maintains a minimum-width connection between the two (this is the case between materials A and B in Figure 11). Second, if one material completely covers the edge with another material (for example, the *A-C* edge in Figure 11), Magic plows the other material as much as is needed to maintain complete coverage. This ensures, for example, that transistors don't get uncovered by plowing polysilicon off one side.

5.2. Inelastic features

Certain features in a layout should not be stretched or compacted. Transistors, for example, have sizes chosen for electrical reasons, as do contacts. Our discussion of edge motion has assumed that the material forming both sides of the edge was stretchable. When material is inelastic, both its left-hand and right-hand edges must be moved in tandem. In particular, if the

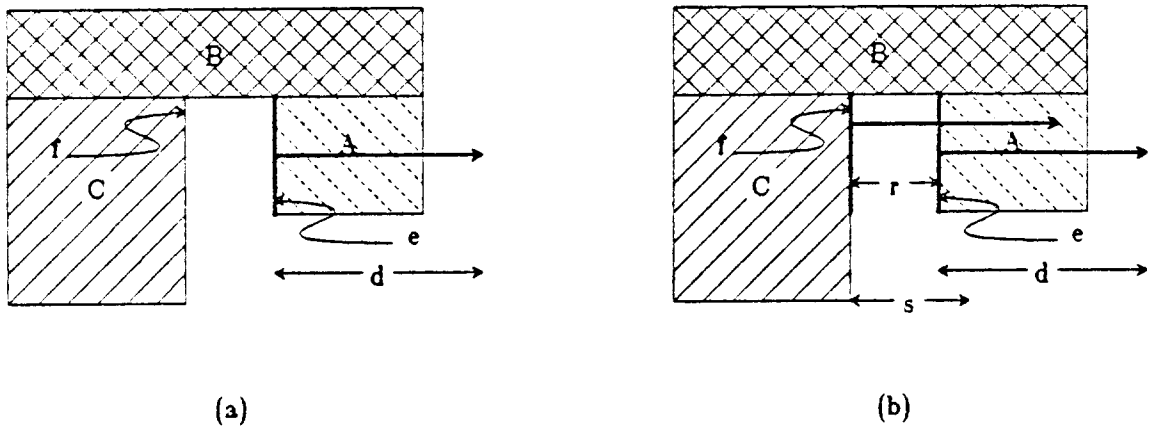

Figure 12. When inelastic objects are present, plowing may have to cope with circular dependencies. Material B is inelastic, and A and C are both minimum-width. When edge e moves by distance d in (a), object B must move by the same distance to prevent A from being uncovered. To prevent C from being uncovered, C's left-hand edge must move, finally causing edge f to move by distance d. Edge e is in f's shadow as a result, but should not be moved a second time.

right-hand edge of a piece of inelastic material moves, its left-hand edge must move also.

A consequence of inelasticity is that moving an edge can cause motion of edges to its left, possibly resulting in a circular dependency. The example in Figure 12 illustrates such a dependency. The depth-first plowing algorithm is completely incapable of resolving such a dependency. The breadth-first algorithm resolves it by comparing the amount an edge is supposed to move with the motion distance already stored with the edge. If the stored motion distance is greater, the edge need not be moved a second time.

If the distance d between edges f and e in Figure 12 is less than s, the minimum separation allowed (ie, there is currently a layout rule violation), looking at the motion distance of e is insufficient. When the shadow of f is

searched, plowing is supposed to move all edges found far enough away so that they cause no rule violations with the newly moved *f*. This would mean that edge *e* would have to move by *d+s-r*, which is more than the motion distance stored with the edge. As a result, the plowing algorithm loops infinitely, each time moving edge *e* by an additional *s-r*. To avoid infinite walks, plowing never moves a shadowed edge (eg, *e*) more than the edge causing the shadow (eg, *f*). This technique prevents infinite looping, but preserves layout rule violations existing in the original layout.

5.3. Noninteracting planes

Section 4 explained that the order of vertical edges along a horizontal line is unchanged by plowing. Thus material being plowed can never slide over other material in its path. There are cases, however, where it is desirable that certain materials in a layout move independently. Metal, for example, does not interact with either polysilicon or diffusion except at contacts, so it should be able to slide over them.

To allow sliding, Magic segregates the mask information in a layout into a collection of non-interacting *planes*. Material in one plane is free to slide past material in any other plane. The nMOS technology, for example, has two planes: one to hold metal wires, and one to hold polysilicon, diffusion, and transistors.

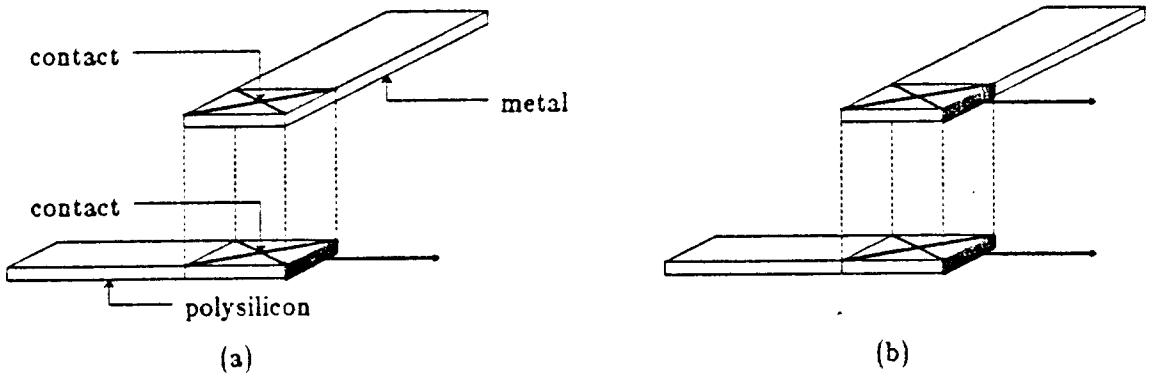

Figure 13. A contact is duplicated on each plane it connects. When an edge of a contact is moved on one plane, it is moved on all other planes as well.

The plowing algorithm operates on each plane independently. The only interaction between planes occurs at contacts, which are duplicated in each plane that they connect. When an edge of a contact is moved in one plane, the corresponding edge of the contact in all other planes is moved by the same amount, as illustrated in Figure 13. This also moves whatever the contact connects to in the other planes, thus preserving connectivity.

5.4. Subcells and hierarchy

One approach for plowing a hierarchical layout, such as that shown in Figure 14a, is to treat it as though it were non-hierarchical and propagate edge motions inside subcells. This might be workable when no subcell is used more than once. However, Magic instantiates subcells by reference, so a change in one instance of a subcell is reflected in all its other uses. Situations in which a subcell is used more than once can produce unsatisfiable sets of constraints, as Figure 14b illustrates.

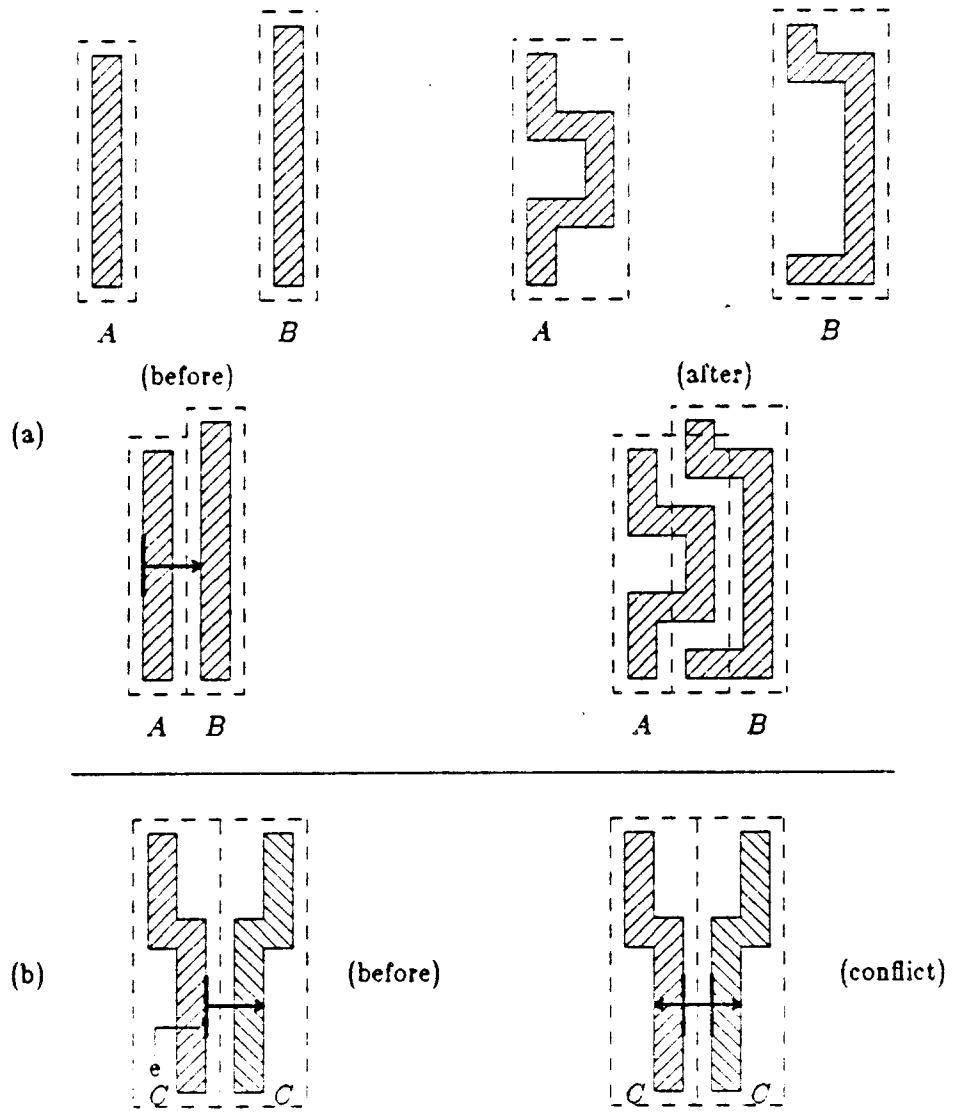

Figure 14. Plowing in the presence of hierarchy. (a) Plowing might treat hierarchy as though it were invisible to the user. Each of cells A and B would be modified. (b) Cell C is used twice, once flipped left-to-right and once in its normal orientation. Both uses refer to the same master definition of C. Moving edge e to the right is impossible, because it requires e to move to the left in order to keep out of its own path. The more edge e is moved to the right in the left-hand use, the worse the violation becomes.

Magic takes a simpler approach, which is to view subcells as black boxes to which connectivity must be maintained by plowing, but whose internal structure should not be modified. A consequence of Magic's approach is that

plowing can be used to modify the placement of cells at the floor plan of a chip, since it only changes the location of subcells, not their contents.

When any mask geometry that abuts or overlaps a cell is moved, the entire cell must move by the same amount. Conversely, whenever a subcell moves, all mask geometry and other subcells that abut or overlap it must also move by the same amount. The net effect is that a cell behaves like flypaper, causing all geometry over its area to "stick" to it and move as a whole when any part of it is required to move.

In addition to preserving connectivity with subcells, when plowing moves other geometry it must avoid introducing any lahout rule violations with the geometry inside a subcell. One approach for dealing with this is to define a *protection frame* [Kell 82] for each cell, an outline around the cell into which no material may be plowed. Magic uses an extremely simple form of protection frame: it assumes that the cell contains all types of material right up to the border of its bounding box.

For example, in our nMOS rule set, the worst-case layout rule involving diffusion is the diffusion-diffusion spacing rule of 6 microns. An edge with diffusion to its left can be plowed to within 6 microns of a subcell before that subcell will itself have to move. The worst-case rule distance involving polysilicon is 8 microns, so polysilicon can only be plowed to within 8 microns of a subcell before the cell must move. Since the contents of subcells are con-

sidered unknown, the closest one subcell can be plowed to another before the other will have to move is the worst-case layout rule in the entire ruleset, which in our ruleset is 8 microns. Of course, if the user wishes to overlap two cells, he can still do that using other editing operations beside plowing.

6. Results and experience

Plowing has been implemented as part of the Magic VLSI layout system. It is written in C under the Berkeley 4.2 Unix operating system for VAXes. A simplified version of plowing (corresponding to that described in Sections 3 and 4) has been operational since October of 1983.

While the full implementation of plowing has not been completed, measurements on the simple version indicate that it is fast enough to be used interactively. An example similar to that presented in Figure 1a, consisting of 48 parallel bars of polysilicon each separated by 4 microns (the minimum separation), took 3.2 seconds of VAX-11/780 CPU time to produce a result similar to that in Figure 1b. Only 1.0 seconds were spent computing the edge motions; the remainder of the time was spent in the post-pass which actually updates the layout.

7. Acknowledgements

Gordon Hamachi, Robert N. Mayo, and George Taylor all contributed to the discussions out of which the plowing algorithm arose. In addition to the above people, Randy Katz, Ken Keller, and Steve and Jean McGrogan all provided helpful comments on early drafts of this paper.

The work described here was supported in part by the Defense Advanced Research Projects Agency (DoD) under Contract No. N00034-K-0251

8. References

[RBDD 83] Rosenberg, J., Boyer, D., Dallen, J., Daniel, S., Poirier, C., Poulton, J., Rogers, D., Weste, N. "A Vertically Integrated VLSI Design Environment." *Proceedings, 20th Design Automation Conference*, 1983, pp. 31-38.

[Kell 82] Keller, K., Newton, A. "A Symbolic Design System for Integrated Circuits." *Proceedings of the 19th Design Automation Conference*, June 1982.

[Oust 81] Ousterhout, J.K. "Caesar: An Interactive Editor for VLSI." *VLSI Design*, Vol. II. No. 4, Fourth Quarter 1981, pp. 34-38.

[Oust 84] Ousterhout, J.K. "Corner Stitching: A Data Structuring Technique for VLSI Layout Tools." To appear in *IEEE Transactions on CAD/ICAS*, Vol 3, No. 1, January 1984.

[OHMST 84] Ousterhout, J.K., Hamachi, G., Mayo, R.N., Scott, W.S., and Taylor, G.S. "The Magic VLSI Layout System." In this technical report.

[TaOu 84] Taylor, G.S., and Ousterhout, J.K. "Magic's Incremental Design Rule Checker." In this technical report.

[West 81] Weste, Neil. "Virtual Grid Symbolic Layout." *Proceedings, 18th Design Automation Conference*, 1981, pp. 225-233.

[Will 78] Williams, J. "STICKS- A Graphical Compiler for High Level LSI Design." *Proceedings of the 1978 NCC*, May 1978, pp. 289-295.

A Switchbox Router with Obstacle Avoidance

Gordon T. Hamachi
John K. Ousterhout

Computer Science Division
Department of Electrical Engineering
and Computer Sciences
University of California
Berkeley, California 94720
(415) 642-9716, 642-0865

ABSTRACT

This paper presents a new switchbox router developed as part of the Magic layout system. Based on Rivest and Fiduccia's "greedy" channel router, the Magic router is capable of routing channels containing obstacles such as preexisting wiring. It jogs nets around large obstacles and multi-layer obstacles such as contacts. Where unable to avoid large single-layer obstacles, it river-routes through them. It combines the effectiveness of traditional channel routers with the flexibility of net-at-a-time routers.

Keywords and Phrases: channel routing, physical design aids, layout, VLSI.

1. Introduction

Previously placed wires such as power and ground routing form obstacles in routing areas. We have developed a new switchbox router as part of the Magic layout system [OHM], capable of routing channels containing such obstacles. The router's novel aspect is its ability to both avoid obstacles and consider interactions between nets as channels are routed. It thus combines good features from net-at-a-time routers and traditional channel routers.

The Magic router is an extension of Rivest and Fiduccia's "greedy" channel router [RiF]. It performs a column by column scan of a rectangular routing region. At each column it applies a series of rules controlling the placement of vertical jogs within the column.

Figure 1 shows the solution to a simple routing problem. Figure 2 shows the same routing problem with an obstacle in the routing area. It illustrates some basic principles of obstacle avoidance. As the router extends nets from left to

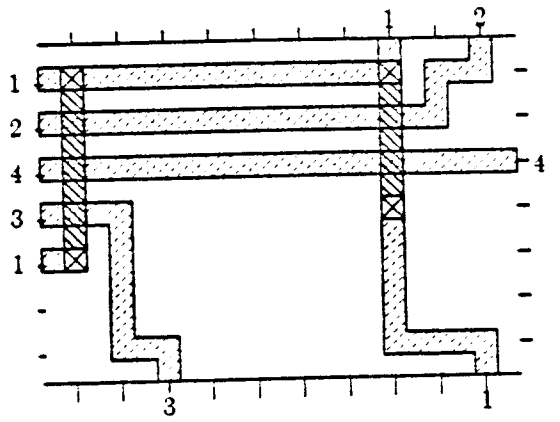

Figure 1. A simple channel routing problem. Numbers around the border of the channel represent pins associated with signal nets. Pins with identical net numbers are connected by the router using a left to right, column by column scan.

right, it tries to avoid large obstacles in the columns ahead by jogging around them. If nets can not jog around large single layer obstacles they *river route* through the obstacles, switching layers if necessary.

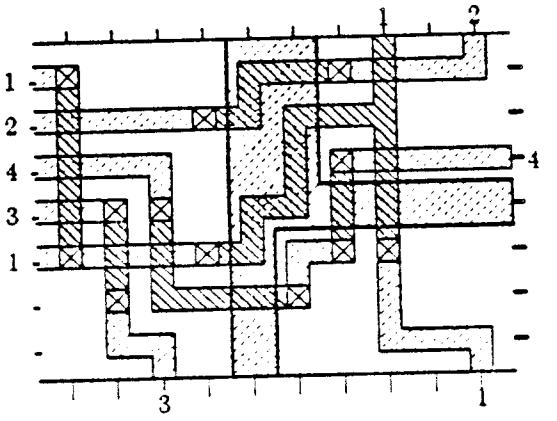

Figure 2. The problem from Figure 1 with obstacles in the channel (drawn with heavy outlines). The router tries to cross obstacles at narrow points. If necessary it river-routes through obstacles.

Section 2 motivates the problem of obstacle avoidance and describes the Magic router's goals. Section 3 summarizes the "Greedy" router, upon which our work is based. Section 4 presents our solutions to a number of problems encountered in adapting the greedy channel router to avoid obstacles. In section 5 we present extensions for routing switchboxes. Section 6 provides a detailed view of the router. Section 7 describes a channel splitting mechanism. The paper

concludes with a discussion of the router's implementation and performance.

2. Motivation

Automated routing systems typically divide the routing of chips into three steps: *channel definition*, *global routing*, and *channel routing*. In the channel definition step, empty areas between cells are divided into non-overlapping rectangular *channels*. The global routing step selects the sequence of channels through which each signal net will be routed to make the desired connections. The channel routing step assigns physical locations to the wires in each channel, realizing the signal routings specified in the global routing step.

A standard model for channel routing assumes a grid of two independent layers of minimum width wiring. Horizontal *tracks* are wired in one of these layers, while vertical *columns* are wired in the other layer. Connections between layers are made with *contacts*; where no contacts appear, layers may cross over each other.

Routers generally assume that channels start off completely free of wiring. Thus, it is impossible to use an automatic router with pre-routed wires. This is a serious limitation since certain signals such as power, ground, and clock lines have special restructions on width, layer, and length which existing channel routers fail to handle.

Because channel routers do not tolerate the presence of obstacles, designers must either accept inferior results generated by automatic routing systems or try to hand patch the router output. Hand patching is difficult because automatic routers leave little room to add wires or to move wires to different layers. Also, if the chip has to be rerouted, the hand patching must be completely redone.

Channel routing systems that do handle obstacles have done so in restricted ways. The PI system [Riv] has the notion of "covered channels" -- areas wired with metal for power and ground routing, through which other signals may be river routed in polysilicon. It fragments large channels containing metal power and ground wiring into many smaller covered and uncovered channels each of which must be routed individually. The problem of routing one large channel is thereby reduced to the problem of routing several smaller, more constrained

channels.

The BBL system [Che], [CHK] handles prewiring from a separate power and ground routing phase. It routes power and ground signals near the edges of channels. BBL then ignores the power and ground routing areas except to bridge other signals across them. It routes all other signals using only the clear parts of the channels. BBL does not allow any hand routing.

Routers using maze [Lee][Hig] routing methods are able to avoid obstacles on multiple layers. The problem with these routers is that they consider only one net at a time. Since they completely route a single net before considering the next net, they cannot consider interactions between nets as a channel is routed. For this reason these routers are inferior to true channel routers for channel routing of general layouts [Sou].

The Magic router provides a general obstacle avoidance capability that combines the advantages of the above approaches. It allows designers to prewire critical nets, putting them at any position and in any layer. The Magic router routes around these prewired nets.

The router considers interactions between nets. Routing decisions are based on an overall strategy rather than on a net-at-a-time basis. Considering tradeoffs between alternatives improves the overall quality of the resulting wiring.

Magic uses single-layer obstructed areas to do useful routing. Since large, loosely constrained areas are easier to route, it avoids fragmenting these obstructed areas into small, highly constrained, hard to route areas.

Particularly in interactive design environments nearly optimal results obtained quickly are more useful than optimal results obtained after long computation. Our router is fast, and produce good results.

3. The Greedy Router

Since the greedy router algorithm is the starting point from which our algorithm was developed, we start with a brief overview of its operation. Three features of the greedy router are of particular importance. First, the greedy router makes a column by column scan of the routing area. It completely wires the current column before extending active tracks into the next column. Second,

it uses a list of rules to control the placement of vertical wiring in a column. Rules are applied in order of importance, to a) avoid "getting stuck"; and b) to make subsequent columns easier to route (Figure 3). The list of rules can easily be modified. Third, unlike constraint graph approaches, the greedy router allows *split nets*, nets that occupy more than one track at a time. Split nets give the router the flexibility to evaluate alternatives and choose the one that is best for the overall routing problem.

Column wiring begins by bringing the nets of a column's top and bottom pins (if any) into the first tracks that are either vacant or already assigned to the nets. Deferring this to a later step might allow vertical wiring to block a net, preventing it from being brought into a vacant track.

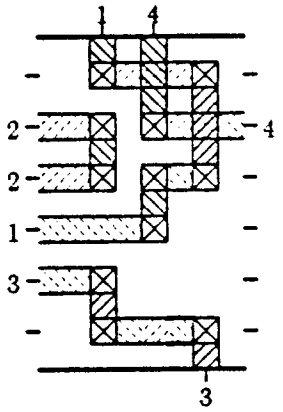

Figure 3. Three columns wired by the greedy router. In the first column net 2 makes a collapsing jog and net 3 makes a falling jog. In the second column net 4 enters the channel, preventing net 1 from making a collapsing jog; however, net 1's lower track makes a jog to reduce the range of tracks assigned to this split net.

Bringing a net into the first available track may leave it *split* on multiple tracks. Split nets can fill up the channel, making it impossible to bring in additional nets. The greedy router thus makes collapsing split nets its next priority. Since conflicting vertical wiring can make it impossible to collapse all split nets in a particular column, the router collapses split nets in the pattern that frees up the most empty tracks for use in the next column.

Vertical wiring conflicts may prevent the router from collapsing all split nets. The router simplifies the routing of these remaining split nets by reducing the range of tracks occupied by these nets. It jogs each split net's highest occupied

track downward and its lowest occupied track upwards. The remaining problem is easier because collapsing can be done with shorter jogs.

Next, unsplit *rising* and *falling* nets are jogged upward or downward toward the edge of the channel with their next pin. This step anticipates the split nets that will be created when upcoming pins' nets are brought into the channel. It attempts to reduce the range of these split nets before they are created. This step prevents split nets if the rising or falling net can be jogged into what would otherwise be the first vacant track seen by a net as it enters the channel from a top or bottom pin.

The handling of split nets and rising and falling nets are examples of decisions based on interactions between nets. Among conflicting alternatives (a jog to raise a rising net may block a jog to lower a falling net) the router chooses the one that does the best job of simplifying the remaining overall problem.

4. Extending the Greedy Router

In modifying the greedy router to avoid obstacles we had to solve a number of problems. The result was an augmented set of rules for placing horizontal and vertical wiring. In the following discussion, an area with a single layer obstacle is called an *obstructed* area. The Magic router river-routes through obstructed areas. An area is *blocked* if it contains a double layer obstacle. No routing may pass through blocked areas.

As it scans a channel from left to right, the greedy router expects that it can always extend a track into the next column if necessary. The router must avoid extending tracks into *blocked* areas (Figure 4).

We solve this problem by anticipating upcoming obstacles and attempting to jog nets out of their way. We do this by identifying areas near obstacles; these areas are called *obstacle thresholds*. A preprocessing step searches the routing area, marking obstacle thresholds. Tracks extending into these marked areas make *vacating* jogs to tracks outside these areas.

Another important issue is the tradeoff between horizontal and vertical wiring. Magic has to decide whether to route horizontal wires or vertical wires over single layer obstacles. It can not do both of these, since an obstacle and a wire

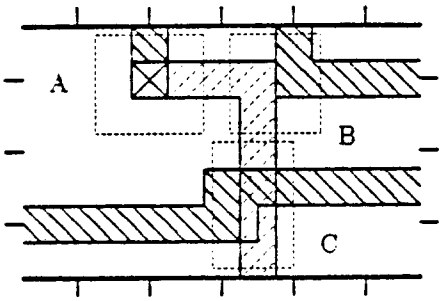

Figure 4. Tracks can not extend into blocked areas (drawn in dotted lines). Note that two adjacent areas of different layers (B) form blocks because there is no place to put contacts to bridge from one area to the other.

crossing it block both routing layers. A thin vertical wire should be bridged horizontally by tracks. Likewise, a thin horizontal wire should be bridged vertically by columns. Intermediate cases are harder to classify (Figure 5).

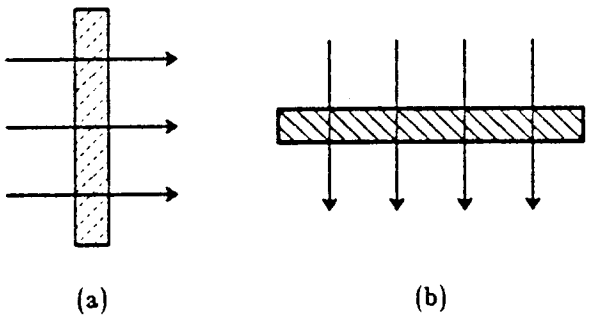

Figure 5. Thin width vertical wires should be bridged horizontally by tracks (a). Thin width horizontal wires should be bridged vertically by columns (b). Intermediate cases are harder to classify.

We solve this problem by always giving priority to horizontal wiring. If vertical wiring is not done in the current column it may be done in some later column. Horizontal wiring is more important: if the router needs to extend tracks but can not, it fails.

Although horizontal wiring gets priority over vertical wiring, we attempt to avoid extending tracks into large single layer obstacles. When tracks do extend into single layer obstacles the Magic router tries to jog them out of these areas, into unobstructed tracks. It is important to do this because a single track running through an obstructed area blocks all columns that might cross the obstructed area (Figure 6).

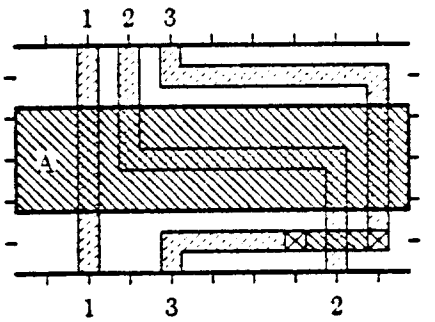

Figure 6. Nets avoid obstructed tracks wherever possible. Failure to do so may create blocked areas. Since net 2 is in an obstructed area, net 3 is forced to make a long detour.

The greedy router assumes that it can make vertical column wiring anywhere the channel is not blocked by vertical wiring it previously placed. The Magic router has to know not only when to place vertical column wiring, but also how to do this. It has to know when areas are blocked, and when to place contacts to switch layers.

Given our wiring model, contact placement is simple. If a contact needs to be placed to allow a layer switch, there is only one place where that contact can go: immediately adjacent to the obstacle. For vertical wiring contacts may be placed immediately above or below the obstacle. For horizontal wiring the locations are immediately to the left and right of the obstacle.

Our wiring model allows horizontal and vertical wiring in either layer; however, only one layer of horizontal wiring and one layer of vertical wiring is allowed at any point. There is a preferred layer in each direction; horizontal tracks and vertical column wires may run in the opposite layer only to bridge an obstacle. Since poly is the preferred vertical layer, a vertical run may bridge a metal obstacle without placing contacts, but contacts need to be placed to bridge a poly obstacle. If the track immediately above the poly obstacle is vacant, then the contact can be placed. If the track is occupied by horizontal wiring, the preferred layer policy says that it must be in metal. The metal/poly boundary blocks the vertical run, since there is no space to bridge the metal track in poly and place a contact before running over the poly obstacle (Figure 7).

The greedy router assumes that channels can be arbitrarily expanded and that terminals on the left and right edges of the channel can "float" up and down

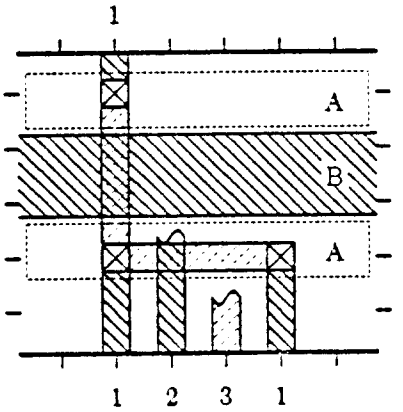

Figure 7. The outlined areas (A) above and below the obstacle (B) are reserved for column contacts necessary if the obstacle is to be bridged vertically. The router tries to keep the areas clear of wiring. Note that the horizontal metal run prevents both the poly (2) and the metal (3) vertical runs from bridging the obstacle, because there is no room to place contacts.

as long as their relative positions remain the same. Tracks may be inserted wherever the router gets "stuck". The Magic router assumes that channels have a fixed number of tracks and that terminals have fixed positions on the edges of the channels. Accordingly, the Magic router omits the greedy router's channel widening step, reporting failure if a net could not be brought into the channel from some top or bottom pin.

5. Routing Switchboxes

The greedy channel router handles pins on at most the top, left, and bottom sides of a channel. To make it a switchbox router, the Magic router contains additional rules to make connections on the right edge of the channel. Furthermore, the Magic router removes the assumption that nets have at most one pin on each end of the channel.

The Magic router deals with switchbox connections by introducing the notion of *reserved* tracks. A track is reserved if it is needed by some net to make a connection on the right edge of the channel. When approaching the end of the channel the router makes vacating jogs to clear reserved tracks and then jogs the appropriate nets into these tracks (Figure 8). Additionally, after nets with only one right edge pin have made their last top and bottom pin connections, their right edge tracks become reserved, other nets vacate these tracks, and the router

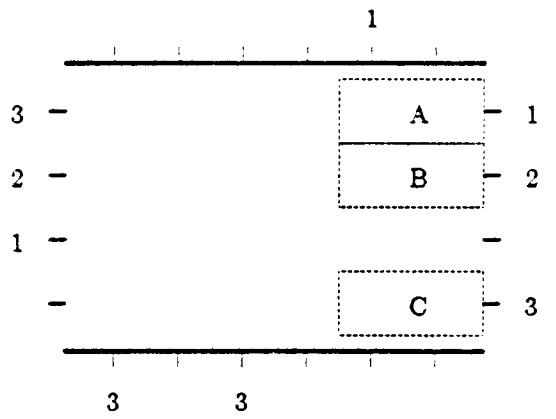

Figure 8. The outlined areas are reserved for nets making connections at the end of the channel. Any other nets entering these areas make vacating jogs, allowing the required nets to occupy the tracks.

tries to jog nets into their final tracks. Vacating reserved tracks uses the same mechanism provided to vacate obstructed tracks.

If a net has more than one pin on the right edge of the channel, the router needs to split the net to connect to them. Split nets occupy tracks that could otherwise be used to help route the channel. Therefore splitting to make multiple end connections is only done when the router gets close to the end of the channel. *Close* is a parameter the user sets to control net splitting. A typical value is two columns.

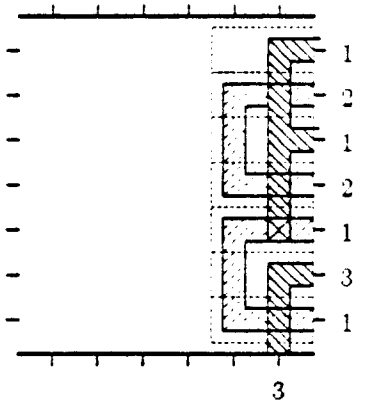

Figure 9. As the router approaches the end of the channel, nets with all of their pins on the right edge of the channel require tracks to be assigned to the nets. This is done if at least two tracks can be allocated and joined with vertical wiring.

Nets with all of their pins on the right edge edge of the channel are another

complication. As the router nears the right edge of the channel it has to decide when to first assign tracks to these *right edge* nets. Since there are no connections to previous pins, a right edge net is introduced into the channel only if it can be assigned to at least two tracks that can be joined by vertical wiring (Figure 9).

We carry this one step further. Groups of two or more tracks for a particular right edge net may be introduced into the channel, even if the groups themselves can not immediately be joined. The task of joining these groups is easier, since the top track of one group need only be connected to the bottom track of a net's higher group.

6. The Magic Routing Algorithm

The Magic router operates in three phases. It begins by making a pre-routing scan of the routing area, identifying obstacle thresholds. After identifying obstacle thresholds, the router extends nets from left edge pins into the routing area and routes it using the column-by-column scan. After routing the channel the Magic router invokes a post processing step to maximize metal and reduce vias.

6.1. Finding Obstacle Thresholds

Obstacle thresholds are generated for all multi-layer obstacles and some single layer obstacles. Multi-layer obstacles such as contacts, crossings, and poly/metal edges must always be avoided as tracks extend from left to right, since it is not possible to bridge these obstacles in any layer. Single layer obstacles extending horizontally for more than one column's width also generate threshold areas. Single layer obstacles extending horizontally for only one column's width do not generate thresholds since the vertical wiring gained in the obstructed area is offset by the vertical wiring wasted in jogging around the obstacle.

Depending on the height of the obstacle, many nets may have to be jogged around it. Not all nets can make vacating jogs in the same column because the vertical wiring for one vacating jog blocks another net from making its vacating jog. On the other hand, vacating tracks long before they near obstacles wastes channel routing area. In recognition of this, the Magic router makes vacating jogs

around an obstacle depending on how far away and how high the obstacle is. Higher obstacles, which block more tracks, cause nets to start vacating jogs farther away, while shorter obstacles can be approached more closely before vacating jogs commence. The width of the threshold is the product of a parameter, *obstacle threshold constant*, and the height of the obstacle. This parameter allows some control over how soon the router attempts to vacate obstructed tracks. A typical value for this parameter is 1.

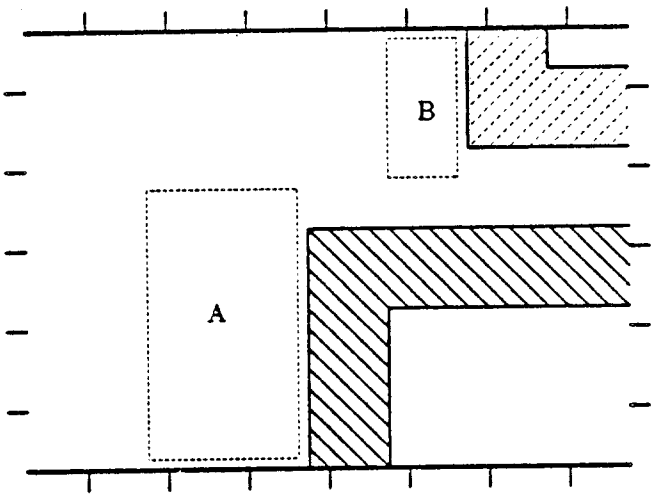

Figure 9. Taller obstacles may require more nets to vacate their thresholds; therefore taller regions have wider thresholds.

The obstacle threshold also extends one track above and below the obstacle. Nets do not get assigned to these tracks unless no other track is free. This allows contacts to be placed if vertical wiring has to switch layers to bridge the obstacle (Figure 7).

6.2. Wiring Rules

This section presents the set of rules the Magic router uses to control the placement of contacts and vertical jogs. The following discussion omits details that are identical in the greedy router. The rules are:

a. *Place Track Contacts*: As the first step in wiring a column, place a contact in each unobstructed track, if either the next column or the previous column has an obstruction in the preferred horizontal track layer. The contact serves one of three purposes: (a) it switches the net from the preferred

horizontal track layer (metal) to the alternate layer (poly) when the net enters a river-routed region; (b) it switches the net from the alternate layer back to the preferred horizontal layer when the net leaves a river-routed region; or (c) it switches the track to the preferred vertical layer in preparation for jogging the net to another track.

b. *Make Minimal Top and Bottom Connections*: Do not bring a net into an unobstructed track that is blocked in the next column. This step may bring a net into an obstructed track. If this occurs, step (f) will attempt to jog the net to an unobstructed tracks. Report failure if some net could not be brought into the channel.

c. *Collapse Split Nets.*

d. *Reduce the Range of Tracks Assigned to Split Nets*: Do not move a net from a free track to a track that needs to be vacated.

e. *Raise Rising Nets and Lower Falling Nets*: Do not jog from a free track to one that needs to be vacated.

f. *Vacate Obstructed Tracks*: Identify tracks from which nets should be vacated. These are tracks which are either in the threshold of an obstacle or are reserved to make some end connection. Try to vacate to the nearest empty, unobstructed track. Do not vacate to another obstructed or reserved track unless the source track is blocked (ie. runs into a multi-layer obstacle) and the destination track is not blocked. Give preference to vacating jogs that move rising and falling nets closer to their next pin.

g. *Split Nets to Make Multiple End Connections*: If within *channel end constant* columns of the end of the channel, attempt to split nets to make multiple connections at the end of the channel. This is the opposite of the collapsing step c above. The best pattern is that which splits the most tracks.

h. *Extend Active Tracks to the Next Column*: Report an error if some track is prevented from extending into the next column by the presence of a multi-layer obstacle that could not be avoided.

6.3. Metal Maximization

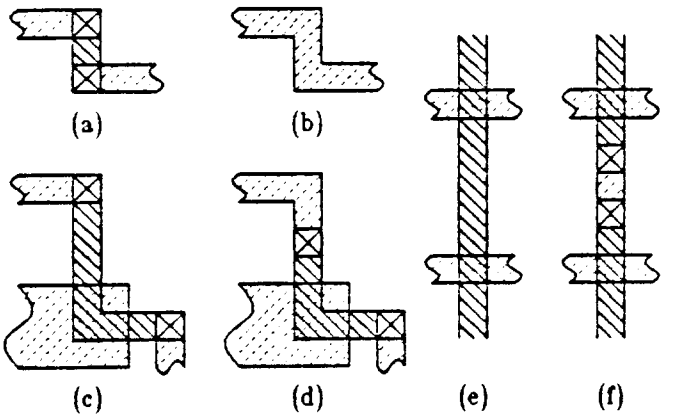

Figure 11. A postprocessing step maximizes metal. This may delete or move vias. It may also introduce vias.

After the subchannels are routed, the Magic router concludes with a metal maximization step. (Figure 11). Since the router already routes metal horizontally wherever possible, this step replaces vertical wiring in polysilicon with vertical wiring in metal, subject to constraints imposed by obstacles in the channel. Vias are deleted wherever they become unnecessary.

7. Channel Splitting

The Magic router also extends the greedy router by including a channel splitting feature. It splits a channel in two at a point of maximum density, assigns tracks to nets crossing the split, then routes both subchannels outwards from the column of the split. The intent of channel splitting is to improve the routability of the two resulting subproblems by (1) assigning tracks to the nets crossing the split to remove conflicts between vertical wiring, and (2) removing split nets at the column where the channel is divided, to guarantee that there are enough available tracks to accommodate the nets that must cross this column Channel splitting is done if the length in columns of each of the resulting subproblems is greater than or equal to the parameter *minimum channel size*, and if the density of the routing problem is close to the size of the channel. If the channel can not be split, then the router routes it from left to right or from right to left, at the discretion of the user.

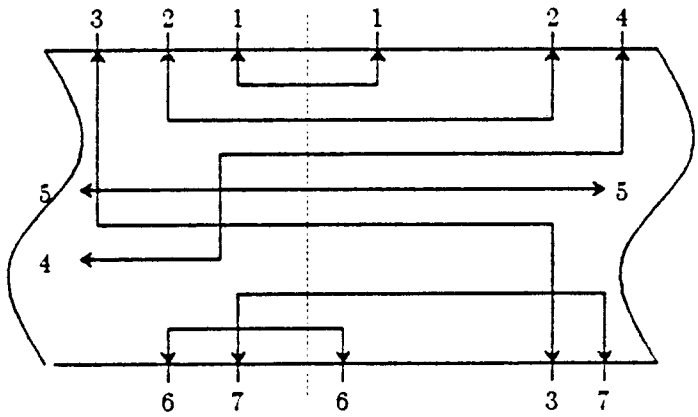

Figure 12. To increase the routability of the two subchannels the router assigns tracks to nets crossing the split. Nets are ordered based on their rising/falling status and the distance to their closest left and right pins.

Channel splitting is not recursive -- it is done at most once. The idea is to route away from the point of maximum density. Splitting each subchannel at its point of maximum density would result in subchannels routing from one highly constrained region to another.

After deciding where to split the channel, the Magic router assigns tracks to the nets crossing the split. The ranking procedure assigns each net a ranking number which is the average of the distance from the center track of the channel to the net's target tracks in the left and right subchannels. The top tracks go to nets which rise to pins on the top edge of both subchannels. The bottom tracks are assigned to nets which fall to pins on the bottom edge of both subchannels. All other nets, including those rising or falling an intermediate distance, and those steady in both subchannels, get distributed between the first two groups.

Another discriminator is used among nets rising to the top or falling to the bottom of both subchannels. A net *a* ranks above another net *b* if both *a*'s nearest left pin and its nearest right pin are closer to the split column than *b*'s corresponding pins. If the distances overlap (ie. *a*'s left pin is closer than *b*'s, and *b*'s right pin is closer than *a*'s), then the net with the smaller sum of distances is placed above the other. A similar procedure is used for falling nets. The intent is to order the nets to eliminate crossings wherever possible. If nets must cross, this procedure favors the net traveling the shorter distance.

8. Implementation and Performance

For channels without obstacles the Magic router produces results similar to those produced by other good channel routers such as the hierarchical router [BuP], the greedy router [RiF], and Algorithm #2 [YoK]. In spite of omitting the track insertion step from the greedy algorithm, it routes Deutsch's difficult in the same number of tracks as the the greedy router. The results are summarized in Table 1.

Router	Tracks	Vias	Wire Length	Time (sec)	Machine
Magic (no obstacles)	20	376	4099	1.5	DEC VAX 11/780
Magic (with obstacles)	20	376	4099	3.0	DEC VAX 11/780
Algorithm #2	20	-	-	2.1	DEC VAX 11/780
Greedy	20	347	4150	7.93	DEC KA-10
Hierarchical	19	270	3983	24	IBM 370/3033

Table 1. Router Results for Deutsch's Difficult Example

Most of the numbers in Table 1 were taken from [BuP]. The first table entry refers to our implementation of a modified greedy switchbox router before obstacle avoidance was added. The reported number of vias for the Magic router does not show the results of metal maximization.

The table shows that the Magic router is competitive with other channel routers on conventional routing problems. It produces nearly optimal solutions quickly, which may be more valuable in practice than programs such as the Hierarchical router which produce slightly better results after significantly greater computation. Adding obstacle avoidance nearly doubled the running time of our router.

Our figures provide a good comparison between Yoshimura and Kuh's Algorithm #2 and Rivest and Fiduccia's greedy router. Rivest and Fiduccia's router

was implemented in LISP on a KA-10. The Magic router without obstacle avoidance (which is almost identical to the greedy router) is implemented in the *C* programming language. Algorithm #2 is implemented in FORTRAN. Both the Magic router (without obstacle avoidance) and Algorithm #2 run on VAX 11/780s running Berkeley Unix. The early version of our router runs faster than the already fast Algorithm #2, and produces a result using the same number of tracks.

Experience with channel splitting has so far been disappointing. It has turned out to be useful mostly for assigning crossings in river routed regions. In other cases splitting the channel typically increases the number of tracks required to route the channel. Better rules for ordering the nets crossing the boundary between the subchannels might change this. Another idea would be to use different criteria to decide where to split the channel.

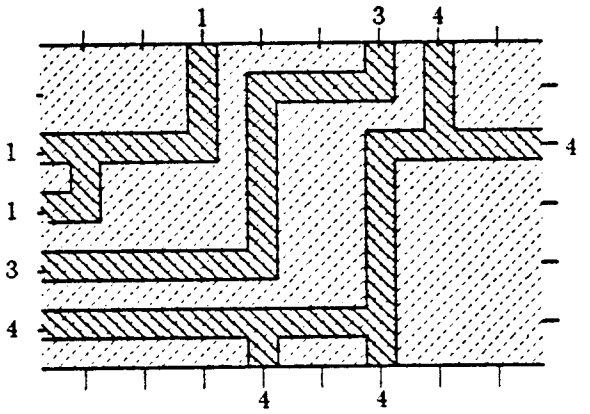

Figure 13. The Magic router river-routes in areas completely blocked in a single layer.

As an example of the range of problems handled by the Magic router, Figure 13 shows a channel completely covered with metal. Our router does a reasonable job of routing this problem.

Postprocessing to increase metal and remove vias appears to significantly improve the quality of the routing.

9. Conclusions

Our obstacle avoiding channel router adds flexibility to our design environment. It allows designers to route critical signals by hand or with separate routing steps. After critical signals are routed, the router makes the remaining connections.

The Magic channel router provides this obstacle avoiding capability, while also considering tradeoffs and interactions between nets. It accomplishes this using a rule based, column sweep routing algorithm which is simple, flexible, and fast. The simplicity of this approach makes it an attractive vehicle for further experimentation.

10. Acknowledgements

Robert Mayo, Walter Scott, and George Taylor all participated in discussions resulting in this work and provided comments on drafts of this paper. Mark Hill, Randy Katz, Carlo Sequin, and David Wallace also reviewed drafts and provided helpful comments. The work described here was supported in part by SRC under grant number SRC-82-11-008.

11. References

[BuP] Burstein, M., and Pelavin, R., "Hierarchical Channel Router", *Proc. 20th Design Automation Conference*, Miami (1983)

[Che] Chen, H., Private communication with authors.

[CHK] Chen, N. P., Hsu, C. P., and Kuh, E. S., "The Berkeley Building-Block Layout System for VLSI Design", ERL memo UCB/ERL M83/10, University of California at Berkeley, (Feb. 1983).

[Lee] Lee, C. Y., "An Algorithm for Path Connections and its Application", *IRE Transactions on Electronic Computers*, pp. 246-365 (September 1961).

[Hig] Hightower, D., "A Solution to the Line Routing Problem on the Continuous Plane", *Proceedings Design Automation Workshop*, pp. 1-24, (1969).

[OHM] Ousterhout, J. K., Hamachi, G. T., Mayo, R. N., Scott, W. S., and Taylor, G. S., "Magic: A VLSI Layout System". In this technical report.

[Riv] Rivest, R. L., "The 'PI' (Placement and Interconnect) System", *Proc. 19th Design Automation Conference*, Las Vegas (1982).

[RiF] Rivest, R. L., and Fiduccia, C. M., "A Greedy Channel Router", *Proc. 19th Design Automation Conference*, Las Vegas (1982), pp. 418-424.

[Sou] Soukup, J., "Circuit Layout", *Proceedings of the IEEE*, Vol. 69, No. 10 (Oct. 1981), 1281-1304.

[YoK] Yoshimura, T., and Kuh, E. S., "Efficient Algorithms for Channel Routing", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*", Vol. CAD-1, No. 1, (Jan 1982).