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

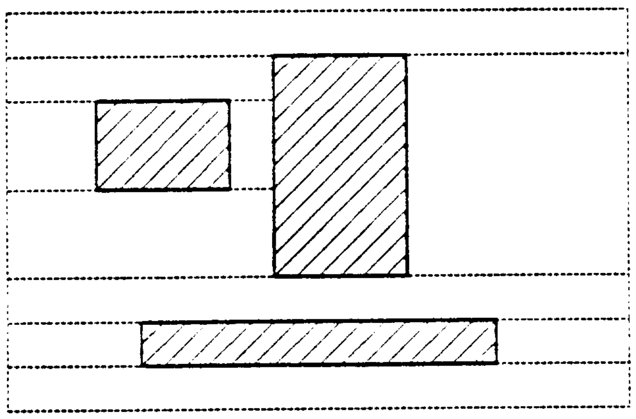

Figure 1. Every point in a corner-stitched plane is contained in exactly one tile. In this case there are three solid tiles, and the rest of the plane is covered by space tiles (dotted lines). The space tiles on the sides extend to infinity. In general, a plane may contain many different types of tiles.

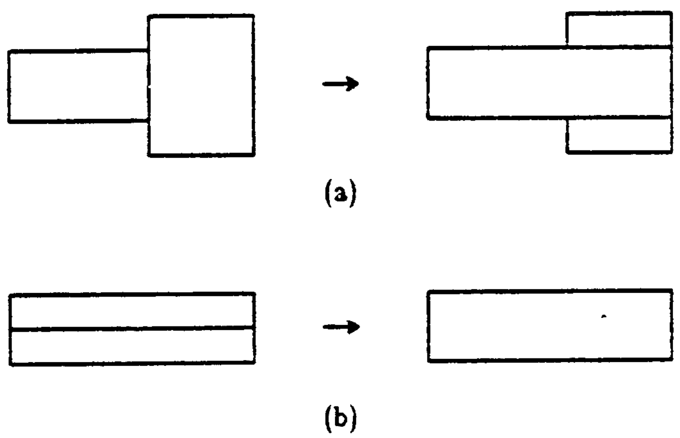

Figure 2. Areas of the same type of material are represented with horizontal strips that are as wide as possible, then as tall as possible. In each of the figures the tile structure on the left is illegal and is converted into the tile structure on the right. In (a) it is illegal for two tiles of the 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).

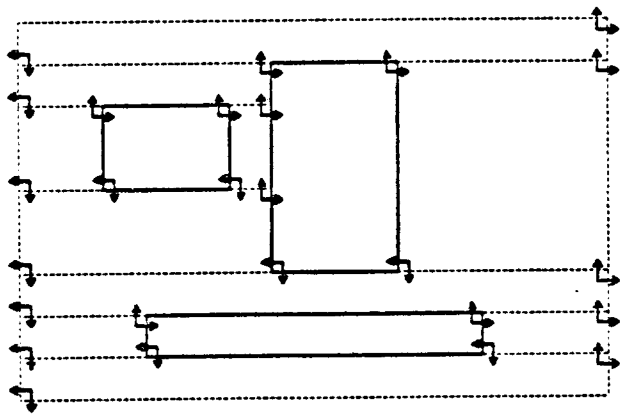

Figure 3. Each tile is linked to its neighbors with four pointers, called *corner stitches*. The corner stitches provide a form of two-dimensional sorting. They permit a variety of geometrical operations to be performed efficiently, such as searching an area or finding all the neighboring tiles on one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

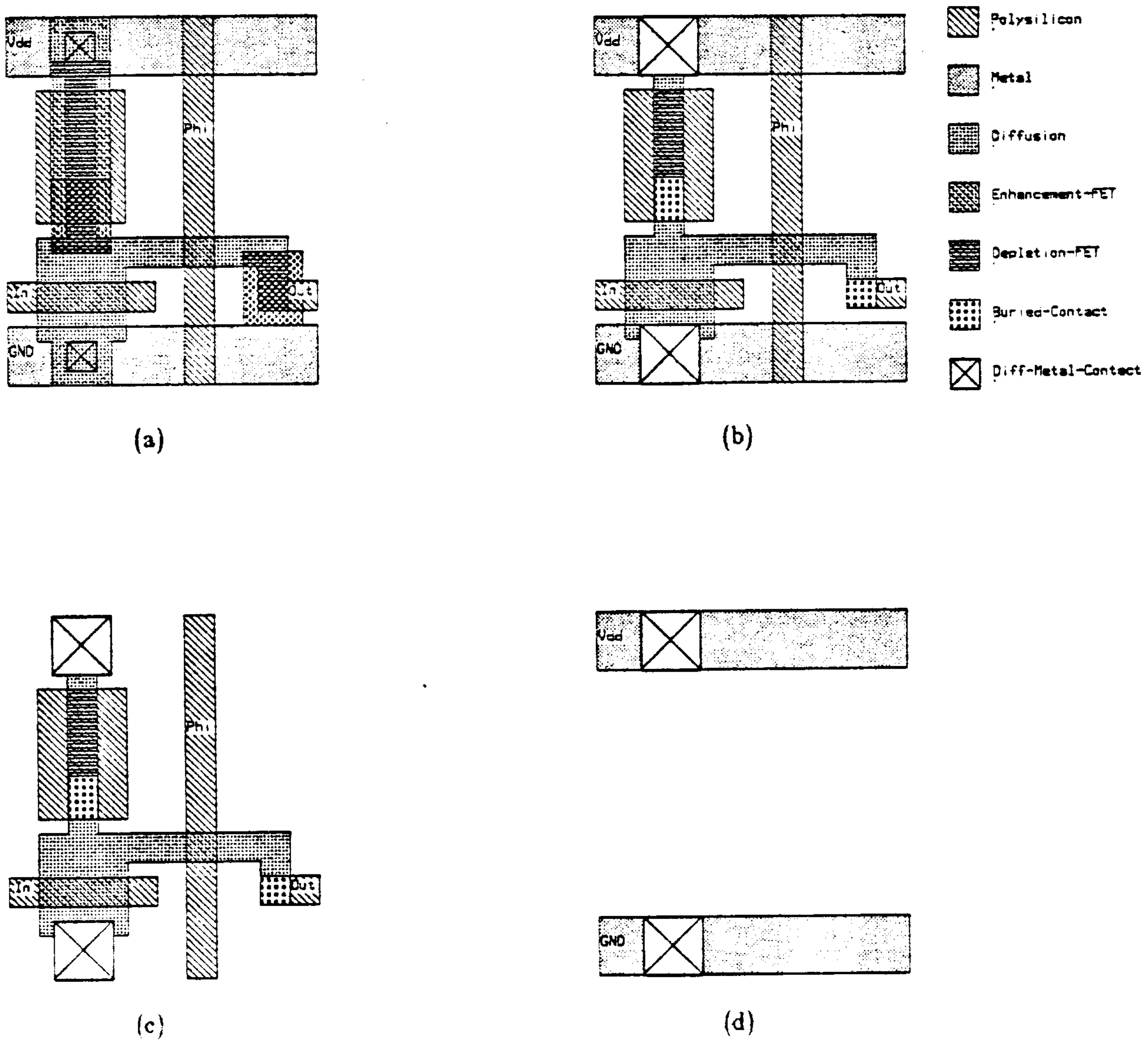

Figure 4. In Magic, transistors and contacts are drawn in an abstract form: (a) a three-transistor shift-register cell, showing actual mask layers; (b) the same cell as it is seen in Magic; (c) the information in Magic's poly-diff plane; (d) the information in Magic's metal plane. Contacts are duplicated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

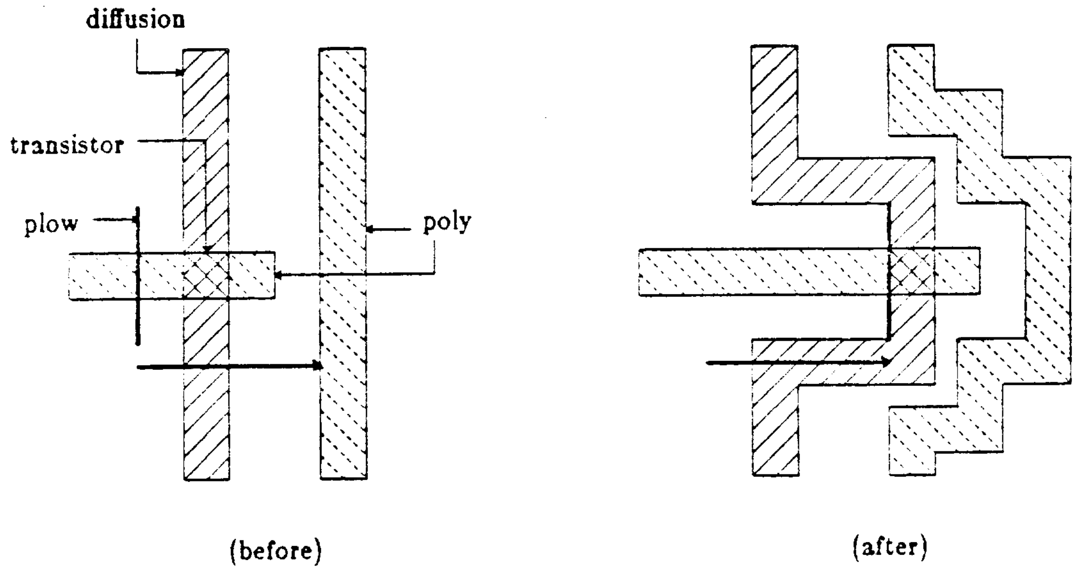

Figure 5. In plowing, a horizontal or vertical line is moved across the circuit, pushing material out of its way. Design rules and connectivity are maintained.

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.

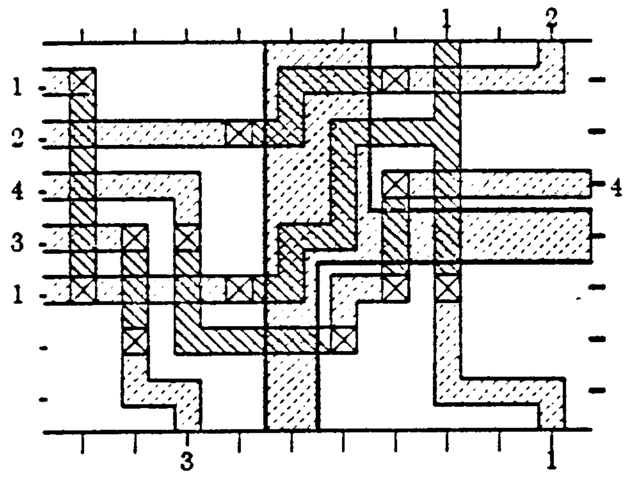

Figure 6. An example of routing with a single-layer obstacle in the channel. The router tries to avoid the thickest part of the obstacle if possible.

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 re-arrange 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

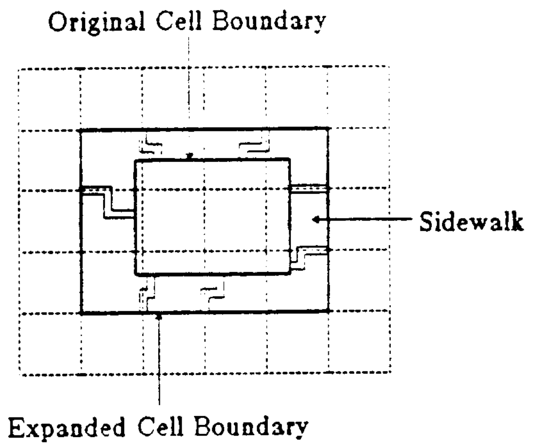

Figure 7. In the sidewalk approach, each cell is enlarged so that its boundary is grid-aligned. Then connections on the edge of the original cell are routed to grid points on the outside of the sidewalk.

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.

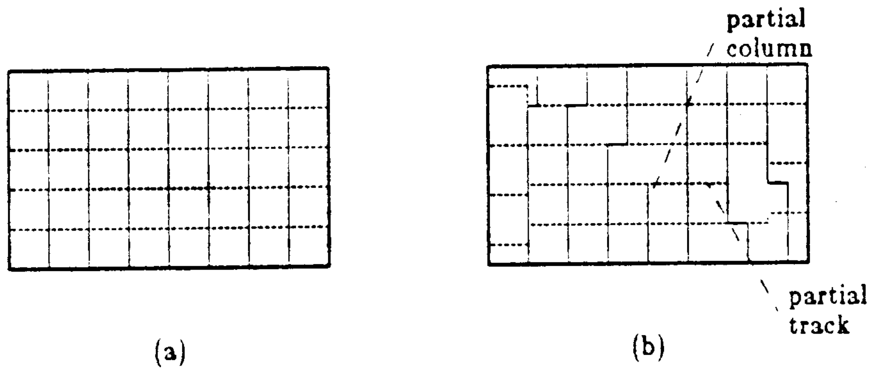

Figure 8. Rather than expand cells to grid points as in the sidewalk approach, the flexible grid approach modifies the track and column structure of the channel. The channel is grid-based in the center, but the grid lines jog at the edges to meet up with non-gridded connections. (a) shows the standard 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.