Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA, Oct. 1975.
[6] E. Khalily, "Transistor electrical characterization and analysis program (TECAP)," *Hewlett-Packard J.*, vol. 32, no. 6, pp. 16-17, June 1981.
[7] D. E. Ward and R. W. Dutton, "A charge-oriented model for MOS transistor capacitances," *IEEE J. Solid-State Circuits*, vol. SC-13, no. 5, pp. 703-708, Oct. 1978.
[8] R. W. Knepper, "Dynamic depletion mode: An E/D MOSFET circuit method," in *ISSCC Dig. Tech. Papers*, pp. 16-17, 1978.

*

**Dileep A. Divekar** (M'78) was born in Pune, India. He received the B.E. degree in 1970 from the University of Pune, India, the M.E. degree in 1972 from the Indian Institute of Science, Bangalore, India, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1975 and 1978, respectively.

From 1972 to 1973, he worked at Indian Telephone Industries, Bangalore, India, on communication systems. From 1976 to 1978, he was a Graduate Student Research Assistant in the Department of Electrical Engineering, Stanford University, where he worked on device characterization and statistical circuit simulation. From 1978 to 1981, he was with the CAD group of the Signetics Corporation. He then joined the Design Aids Group of Hewlett-Packard working on semiconductor device modeling and circuit simulation. He is presently with ZyMos Corporation, Sunnyvale, CA.

*

**Richard I. Dowell** (S'69-M'74) studied electrical engineering at the University of California at Berkeley, receiving the B.S., M.S., and Ph.D. degrees in 1966, 1969, and 1972, respectively. His Ph.D. research was on computer-aided simulation and optimization.

He came to Hewlett-Packard from Bell Laboratories in 1977 to work on simulation and modeling with Hewlett-Packard's design aids group, and is now Project Manager for development of simulators and post-processors. He has authored 3 papers on computer-aided design and has lectured on computer-aided circuit simulation at the University of California at Berkeley.

# Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools

JOHN K. OUSTERHOUT

*Abstract*—**Corner stitching is a technique for representing rectangular two-dimensional objects. It is especially well suited for interactive VLSI layout editing systems. The data structure has two important features: first, empty space is represented explicitly; and second, rectangular areas are stitched together at their corners like a patchwork quilt. This organization results in fast algorithms (linear or constant expected time) for searching, creation, deletion, stretching, and compaction. The algorithms are presented under a simplified model of VLSI circuits, and the storage requirements of the structure are discussed. Corner stitching has been implemented in a working layout editor. Initial measurements indicate that it requires about three times as much memory space as the simplest possible representation.**

## I. Introduction

INTERACTIVE LAYOUT tools for integrated circuits place special burdens on their internal data structures. The data structures must be able to deal with large amounts of information (one-half million or more geometrical elements in current layouts [7]) while providing instantaneous response to the designer. As the complexity of design increases, tools must give more and more powerful assistance to the designer in such areas as routing and validation. To support these intelligent tools, the underlying data structures must provide fast geometrical operations, such as locating neighbors for stretching and compaction, and locating empty space for routing. The data structures must also permit fast incremental modification so that they can be used in interactive systems.

Corner stitching is a data-structuring technique that meets these needs. As described here, it is limited to designs with Manhattan features (horizontal and vertical edges only); but within that framework it provides a variety of powerful operations, such as neighbor-finding, stretching, compaction, and channel-finding. The algorithms for the operations depend only on *local* information (the objects in the immediate vicinity of the operation). Their expected running times are generally linear in the number of nearby objects; in pathological cases (which are unlikely for actual layouts) the running times may be proportional to the overall design size or to the product of nearby objects and design size. Corner stitching is especially

Manuscript received January 5, 1983; revised June 20, 1983. This work was supported in part by the Defense Advanced Research Projects Agency (DoD), DARPA Order 3803, monitored by the Naval Electronic System Command under Contract N00039-81-K-0251.

The author is with the Computer Science Division, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

0028-0070/84/0100-0087$01.00 

effective when the objects are relatively uniform in size, as is the case for low-level mask features. However, it also works well when there is variation in feature size. This occurs, for example, in a hierarchical layout where one cell might contain a few large subcells and many small wires to connect them together.

Corner stitching permits modifications to the database to be made quickly, since only local information is used in making the updates. Most existing systems that provide powerful operations such as routing and compaction do not provide inexpensive updates: small changes to the database can result in large amounts of recomputation. Corner stitching's combination of powerful operations and easy updates means that many powerful tools previously available only in "batch" mode can now be embedded in interactive systems.

## II. A Simplified Model of VLSI Layouts

A VLSI layout is normally specified as a hierarchical collection of cells, where each cell contains geometrical shapes on several mask layers and pointers to subcells. As a convenience in presenting the data structure and algorithms, a simplified model is used in this paper. There is only a single mask layer, and hierarchy is ignored. For this paper, the author defines a "circuit" to be a collection of rectangles. There is a single design rule in the model: rectangles may not overlap. The simplified model makes it easier to present the data structure and algorithms. Section VII discusses how the simple model can be generalized to handle real VLSI layouts.

## III. Existing Mechanisms

### 3.1. Linked Lists

The simplest possible technique for representing rectangles is just to keep all of them in a linked list. This technique is used in the Caesar system [6]: each cell is represented by a list of rectangles for each of the mask layers. Even though operations such as neighbor-finding require entire lists to be searched, the structure works well in Caesar for two reasons. First, large layouts are broken down hierarchically into many small cells; only the top-most cells in the hierarchy ever contain more than a few hundred rectangles or a few children [7]. Second, Caesar provides only very simple operations like painting and erasing. More complex functions such as design rule checking and compaction could not be implemented efficiently using rectangle lists.

### 3.2. Bins

The most popular data structures for VLSI are based on *bins* [2]. In bin-based systems, an imaginary square grid divides the area of the circuit into bins, as in Fig. 1. All of the rectangles intersecting a particular bin are linked together, and a two-dimensional array is used to locate the lists for different bins. Rectangles in a given area can be located quickly by indexing into the array and searching the (short) lists of relevant bins. The bin size is chosen as a tradeoff between time and space: as bins get larger, it takes longer to search the lists in each bin; as bins get smaller, rectangles begin to overlap several bins and hence occupy space on several lists.

Bin structures are most effective when rectangles have nearly uniform size and spatial distributions; they suffer from space and/or time inefficiencies when these conditions are not met. A pathological case is a cell with a few large child cells and many small rectangles to interconnect them. If bins are small, there will be many empty bins in the large areas of the subcells, resulting in wasted space for the bins; if bins are large, the bins in the wiring area will have many rectangles, resulting in slow searches. Hierarchical bin structures [4] have recently been proposed as a solution to the problems of nonuniformity. Although bins can be used to locate all the objects in an area, they do not directly embody the notion of *nearness*. To find the nearest object to a given one, it is necessary to search adjacent bins, working out from the object in a spiral fashion. Furthermore, bin structures do not indicate which areas of the chip are empty; empty areas must be reconstructed by scanning the bins. The need to constantly scan bins to recreate information makes bin structures clumsy at best, and inefficient at worst, especially for operations such as compaction and stretching.

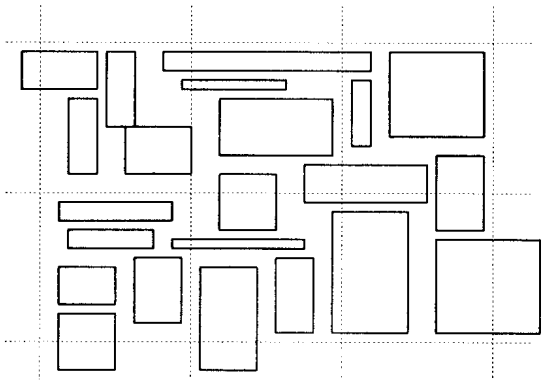


Fig. 1. In bin-based data structures, the circuit is divided by an imaginary grid, and all the rectangles intersecting a subarea are linked together.

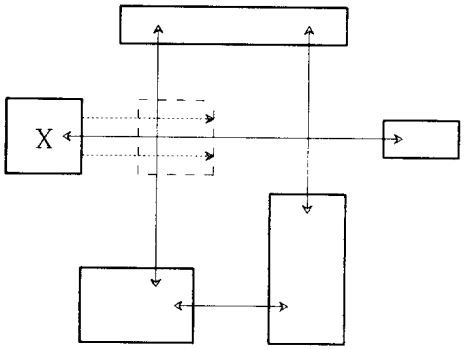



Fig. 2. Neighbor pointers can be used to indicate horizontal or vertical adjacency. However, if tile $X$ is moved right, it is hard to update the vertical pointers without scanning the entire database.

### 3.3. Neighbor Pointers

A third class of data structures is based on neighbor pointers. In this technique, each rectangle contains pointers to rectangles that are adjacent to it in $x$ and $y$ (see Fig. 2). Neighbor pointers are a popular data structure for compaction programs such as Cabbage [3], since they provide information about relationships between objects. For example, a simple graph traversal can be used as part of compaction to determine the minimum feasible width of a cell.

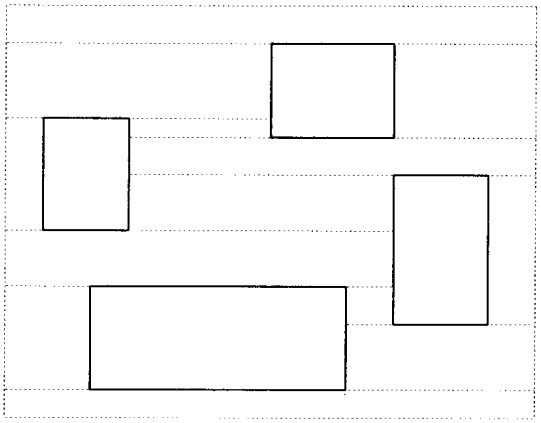


Fig. 3. An example of tiles in a corner-stitched data structure. Solid tiles are represented with dark lines, space tiles with dotted lines. The entire area of the circuit is covered with tiles. Space tiles are made as wide as possible.

Neighbor pointers have two drawbacks. First, modifications to the structure generally require all the pointers to be recomputed. For example, if an object is moved horizontally, as in Fig. 2, vertical pointers may be invalidated. There is no simple way to correct the vertical pointers short of scanning the entire database. The second problem with neighbor pointers is that they provide no assistance in locating empty space for routing, since only the occupied space is represented explicitly. For these two reasons, neighbor pointers do not appear to be well-suited to interactive systems or those that provide routing aids.

## IV. Corner Stitching

Corner stitching arose from a consideration of the weaknesses of the above mechanisms, and has two features that distinguish it from them. The first important feature is that all space, both empty and occupied, is represented explicitly in the database. The second feature is a novel way of linking together the objects at their corners. These *corner stitches* permit easy modification of the database, and lead to efficient implementations for a variety of operations.

Fig. 3 shows four objects represented in the corner stitching scheme. The picture resembles a mosaic with rectangular tiles of two types, space and solid. The tiles must be rectangles with sides parallel to the axes. Tiles contain their lower and left edges, but not their upper or right edges, so every point in the plane is present in exactly one tile. The entire plane is covered from -infinity to +infinity in both $x$ and $y$ (in practice, the largest representable positive and negative numbers are used for the infinities). Coverage to infinity is achieved by extending the outermost space tiles; no extra tiles are required.

The space tiles are organized as *maximal horizontal strips*. This means that no space tile has other space tiles immediately to its right or left. When modifying the database, horizontally adjacent space tiles must be split into shorter tiles and then joined into maximal strips, as shown in Fig. 4. After making sure that space tiles are as wide as possible, vertically adjacent tiles are merged together if they have the same horizontal span. The representation of space is of no consequence to the VLSI layout or to the designer, and will not even be visible in real systems. However, the maximal horizontal strip representation is crucial to the space and time efficiency of the tools, as we shall see in Sections V and VI. Among its other properties, the horizontal-strip representation is unique: there is one and only one decomposition of space for each arrangement of solid tiles.

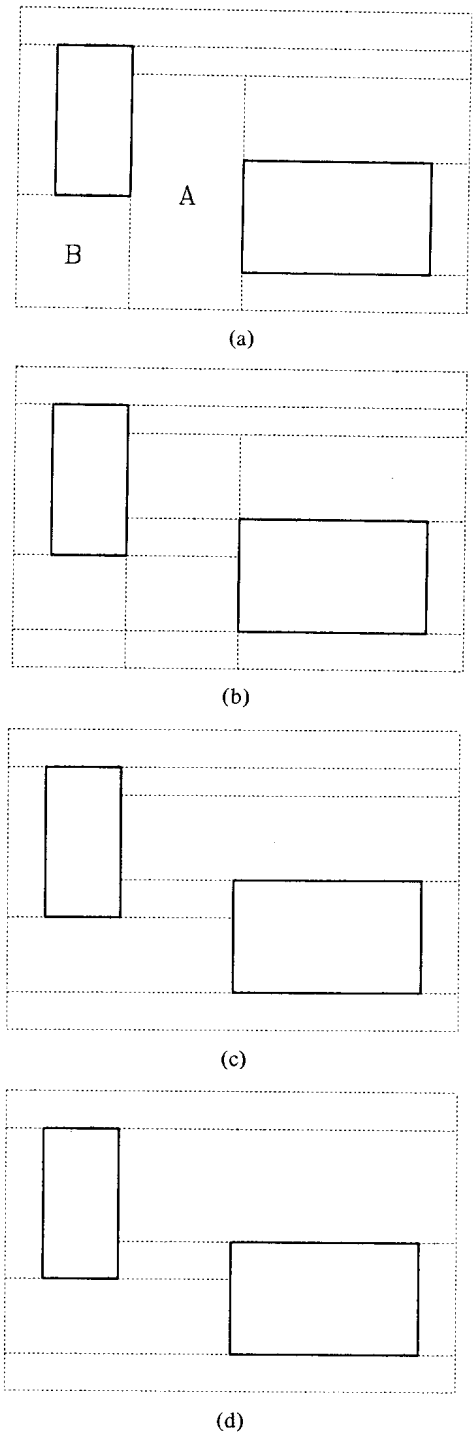



Fig. 4. No space tile may have another space tile to its immediate right or left. In this example, tiles $A$ and $B$ in (a) must be split into the shorter tiles of (b), then merged together into wide strips in (c), and finally merged vertically in (d).

Tiles are linked by a set of pointers at their corners, called *corner stitches*. Each tile contains four stitches, two at its lower-left corner and two at its upper right corner, as illustrated in Fig. 5. Since there is one pointer in each of the four directions, the stitches provide a form of sorting that is equivalent to neighbor pointers. Originally, eight stitches were used, two

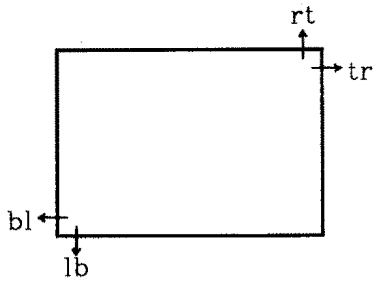



Fig. 5. Each tile is connected to its neighbors by four pointers called corner stitches. The names of the stitches indicate the tiles they point to: the **tr** stitch points to the tile's topmost right neighbor, the **lb** stitch points to the tile's leftmost bottom neighbor, and so on.

at each of the four corners, but four turned out to be sufficient for the algorithms presented here. The choice of these particular four stitches is important.

The tile/stitch representation has several attractive features, which will be illustrated in the sections that follow. First, the mechanism combines both horizontal and vertical pointers in a single structure. The space tiles provide a form of registration between the horizontal and vertical information and make it easy to keep all the pointers up to date as the circuit is modified. Because the space tiles may vary in size (as opposed to fixed-size bins), the structure adapts naturally to variations in the sizes of the solid tiles. The maximal horizontal strip representation of space results in clean upper bounds on the number of space tiles and also on the complexity of the algorithms. All tiles have the same number of pointers to other tiles, so they occupy the same number of bytes of storage; this simplifies the database management and reduces the "constant factors" in algorithms.

## V. Algorithms

This section presents algorithms for manipulating the tiles and corner stitches. The most important attribute of all the algorithms is their locality: each algorithm depends only on information in the immediate vicinity of the operation. None of the algorithms has an expected running time any worse than linear in the number of tiles in the affected area. Pathological cases will be shown where the algorithms require time linear, or even quadratic, in the overall layout size, but in practice (particularly for VLSI layouts, which tend to be densely packed) their running times are small and independent of the size of the layout.

In discussing the performance of the algorithms, the corner stitches provide a good unit of measure. The complexity of the algorithms will be discussed in terms of the number of stitches that must be traversed (or, alternatively, the number of tiles that must be visited) and/or the number of stitches that must be modified.

### 5.1. *Point Finding*

Several different kinds of searching are facilitated by corner stitching. One of the most common operations is to find the tile at a given $(x, y)$ location. Fig. 6 illustrates how this can be done with corner stitching. The algorithm iterates in $x$ and $y$, starting from any given tile in the database:

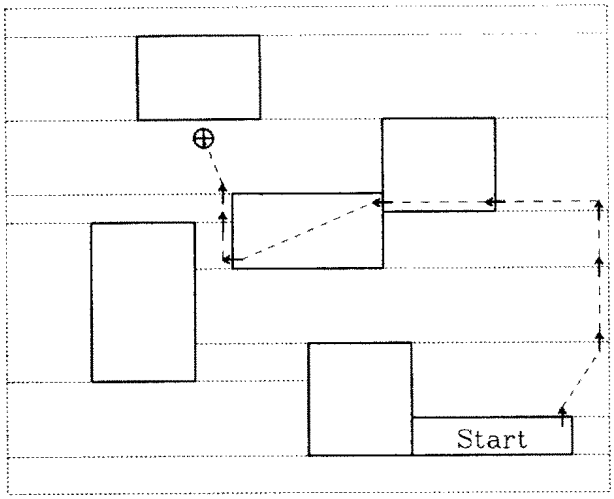



Fig. 6. To locate the tile containing a given point, alternate between up/down and left/right motions.

1) First move up or down, using right top (**rt**) and left bottom (**lb**) stitches, until a tile is found whose vertical range contains the desired point.

2) Then move left or right, using **tr** and **lb** stitches, until a tile is found whose horizontal range contains the desired point.

3) Since the horizontal motion may have introduced a vertical misalignment, steps 1) and 2) may have to be iterated several times to locate the tile containing the point. The convexity of the tiles guarantees that the algorithm will converge.

In the worst case, this algorithm may require every tile in the entire structure to be searched (this happens, for example, if all the tiles in the structure are in a single column or row). Fortunately, the average case behavior is much better than this. If there are a total of $N$ space or solid tiles and they are of relatively uniform size, then on the order of $\sqrt{N}$ tiles will be passed through in the average case. For a layout containing a million tiles (which is typical of the fully expanded mask sets of current VLSI circuits), this means a few thousand tiles will have to be touched.

In interactive systems, there is a simple way to reduce the time spent in point finding: keep a pointer around to any tile in the approximate area where the designer is working. When a large design is being edited, the designer's attention is generally focused on a small piece of the design (e.g., a piece that can be viewed comfortably on a graphic device). If a *hint* tile in this area is remembered for reference, the search time depends only on how much is on the screen, not how large the design is.

The point-finding algorithm illustrates a general feature of most of the algorithms: misalignment. While searching horizontally, it is possible to lose the vertical alignment, so the algorithm must iterate over horizontal and vertical motions. See Fig. 6 for an example. In general, large tiles can cause the algorithms of this paper to wander arbitrarily far outside their areas of interest. When this happens, the algorithms must traverse stitches to get back to the desired area again. Extreme misalignment results in worst-case behavior for many of the

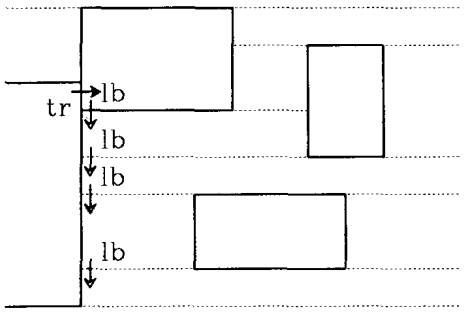



Fig. 7. The corner stitches provide a simple way to find all the tiles that touch one side of a given tile.

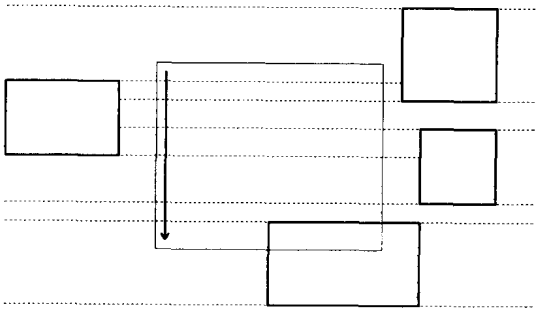


Fig. 8. To search a rectangular area for a solid tile, work down along the left edge of the area. Each tile along the edge must be either a solid tile, a space tile that spans the entire area, or a space tile with a solid tile just to its right.

algorithms. Fortunately, severe misalignment is unlikely for densely packed designs.

### 5.2. *Neighbor Finding*

Another common searching operation is neighbor finding: find all the tiles that touch one side of a given tile. Neighbor finding is useful for design rule checking, compaction, circuit extraction, and tracing out connected nets. Fig. 7 illustrates how to find all the tiles that touch the right side of a given tile:

1) Follow the **tr** stitch of the starting tile to find its topmost right neighbor.

2) Then trace down through **lb** stitches until all the neighbors have been found (the last neighbor is the first tile encountered whose lower $y$ coordinate is less than or equal to the lower $y$ coordinate of the starting tile).

Similar algorithms can be devised to search each of the other sides. The time for the search is linear in the number of neighbors. As shown in Appendix I, the expected number of neighbors is one or two along each side. In layouts where tile sizes vary greatly, the number of neighbors will, on average, be proportional to the length of the side.

### 5.3. *Area Searches*

A third form of searching is to see if there are any solid tiles within a given area. This can be accomplished in the following manner using corner stitches (see Fig. 8):

1) Use the point-finding algorithm to locate the tile containing the upper left corner of the area of interest.

2) See if the tile is solid. If not, it must be a space tile. See if its right edge is within the area of interest. If so, it is the edge of a solid tile.

3) If a solid tile was found in step 2), then the search is complete. If no solid tile was found, then move down to the next tile touching the right edge of the area of interest. This can be done either by invoking the point-finding algorithm, or by traversing the **lb** stitch down and then traversing **tr** stitches right until the desired tile is found.

4) Repeat steps 2) and 3) until either a solid tile is found or the bottom of the area of interest is reached.

As with the other operations, the time necessary for this operation depends only on local features: the number of tiles in and around the area of interest. The cost can be measured by counting the number of stitches that must be traversed. The number of iterations through the algorithm will be proportional to the height of the area (assuming, as always, a relatively uniform size distribution). In each iteration, it may be necessary to traverse one stitch in step 2). In addition, step 3) will cause a misalignment of about 1/2 tile in the average case. Thus the total running time is linear in the height of the search area, and does not depend at all on the width of the search area. In worst-case situations like the one shown in Fig. 9(a), misalignments could cause the running time to be proportional to the total number of tiles in the layout.

### 5.4. *Directed Area Enumeration*

The algorithm in Section 5.3 determines if there are any solid tiles in an area. However, for many applications, such as compaction and layout rule checking, it is useful to enumerate *all* the tiles in a given area, i.e., to "visit" each tile exactly once. Furthermore, it is often useful to do this in a particular direction. For example, during a left-to-right compaction, it is important that a tile not be processed until all tiles on its left have been processed. This section presents an algorithm wherein each tile is visited only after all the tiles above it and to its left have been visited. I call such an enumeration a *directed enumeration*. Corner stitching makes this a linear time operation. Fig. 10 shows the enumeration order for an example case.

1) As for the area-searching algorithm, use the point-finding algorithm to locate the tile at the top left corner of the area of interest. Then step down through all the tiles along the left edge, using the same technique as in area searching.

2) For each tile found in step 1), enumerate it recursively using the R procedure given in lines R1) through R5).

R1) Enumerate the tile (this will generally involve some application-specific processing).

R2) If the right edge of the tile is outside of the search area, then return from the R procedure.

R3) Otherwise, use the neighbor-finding algorithm to locate all the tiles that touch the right side of the current tile and also intersect the search area.

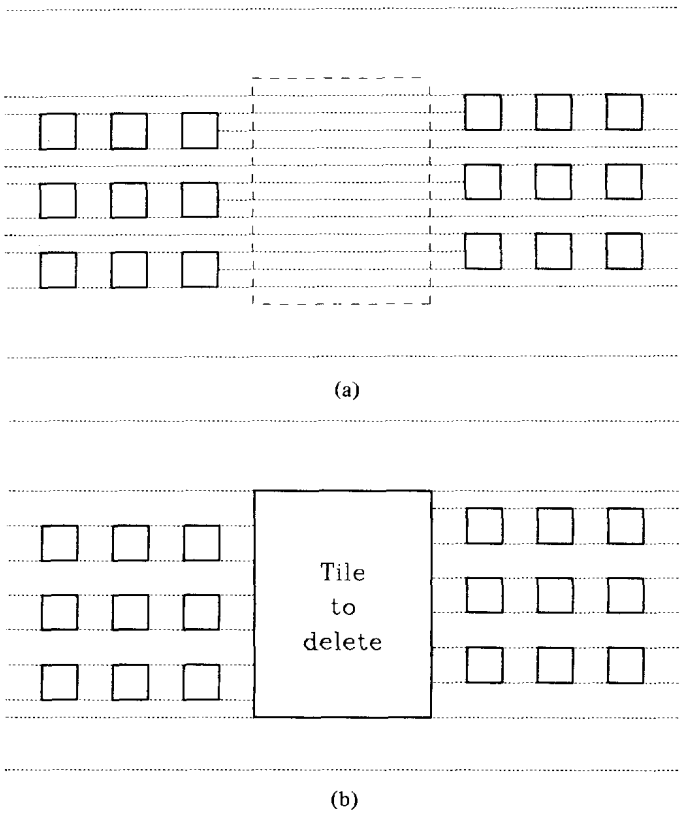



Fig. 9. Two pathological structures. In (a), area searches of the dashed area are slow because of severe misalignment during step 3) of the algorithm. It is also slow to create a tile in the dashed area at (a), which produces the situation in (b), or delete the labeled tile in (b) to get back the situation in (a): when splitting and merging space tiles, corner stitches must be modified in every solid tile in the circuit.

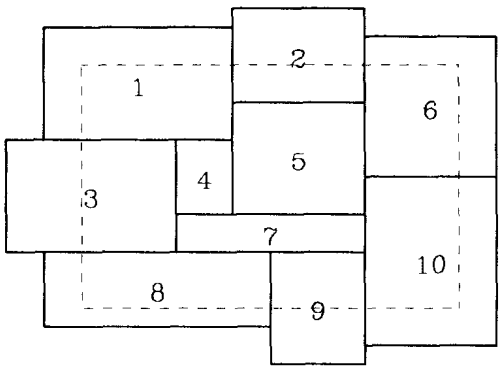



Fig. 10. An example of directed enumeration. When doing an upper left to lower right enumeration of the dashed area, the tiles will be visited in order of their numbers.

R4) For each of these neighbors, if the bottom left corner of the neighbor touches the current tile then call R to enumerate the neighbor recursively (for example, this occurs in Fig. 10 when tile 1 is the current tile and tile 2 is the neighbor).

R5) Or, if the bottom edge of the search area cuts both the current tile and the neighbor, then call R to enumerate the neighbor recursively (in Fig. 10, this occurs when tile 8 is the current tile and tile 9 is the neighbor).

The expected running time of the directed enumeration algorithm is linear in the number of tiles intersecting the search area. This can be shown by the following arguments. The checks in steps R4) and R5) guarantee that each tile is enumerated exactly once. However, a tile may be checked several times before satisfying the checks in step R4) or R5): it will be checked once for each tile that touches its left side. The total expected running time of the algorithm is thus proportional to the total number of adjacencies within the search area. Appendix I uses the properties of planar graphs to prove that the number of adjacencies must be linear in the number of tiles.

In the worst case, directed area enumeration could require every tile in the circuit to be examined. This happens if tiles stick out far above the top edge of the area being enumerated: all of their neighbors must be enumerated in step R3), even though most of them do not intersect the area of interest.

The algorithm for directed enumeration does not depend on the fact that space tiles are maximal horizontal strips. In fact, it does not even distinguish between solid and space tiles. A similar algorithm can be devised to reverse the direction of enumeration (from lower right to upper left). But, it is much more difficult to recode the algorithm to operate from lower left to upper right, or from upper right to lower left (this is because there are no corner stitches emanating from the lower right or upper left corners of tiles).

### *5.5 Tile Creation*

The first step in creating a new solid tile is to check to see that there are no existing solid tiles in the desired area of the new tile. The area-search algorithm can check this. The second step is to insert the tile into the data structure, clipping and merging space tiles and updating corner stitches as shown in Fig. 11. The insertion algorithm is as follows:

1) Find the space tile containing the top edge of the area to be occupied by the new tile (because of the strip property, a single space tile must contain the entire edge).

2) Split the top space tile along a horizontal line into a piece entirely above the new tile and a piece overlapping the new tile. Update corner stitches in the tiles adjoining the new tile.

3) Find the space tile containing the bottom edge of the new solid tile, split it in the same fashion, and update stitches around it.

4) Work down along the left side of the area of the new tile, as for the area-search algorithm. Each tile along this edge must be a space tile that spans the entire width of the new solid tile. Split the space tile into a piece entirely to the left of the new tile, a piece entirely to the right of the new tile, and a piece entirely within the new tile. This splitting may make it possible to merge the left and right remainders vertically with the tiles just above them: merge whenever possible. Finally, merge the center space tile with the solid tile that is forming. Each split or merge requires stitches to be updated in adjoining tiles.

The speed of the creation algorithm is determined by the cost of splitting and merging the space tiles that cross the area. The number of space tiles depends on the number of solid tiles in the left and right shadows of the new tile. One can devise cases where the number of space tiles is arbitrarily high, but in practice, the expected number is proportional to the relative height of the new tile in comparison to the tiles around it. Appendix B discusses the cost of splitting and merging tiles. In the average case it is constant; for very large tiles it is proportional to the circumference of the tile. This means

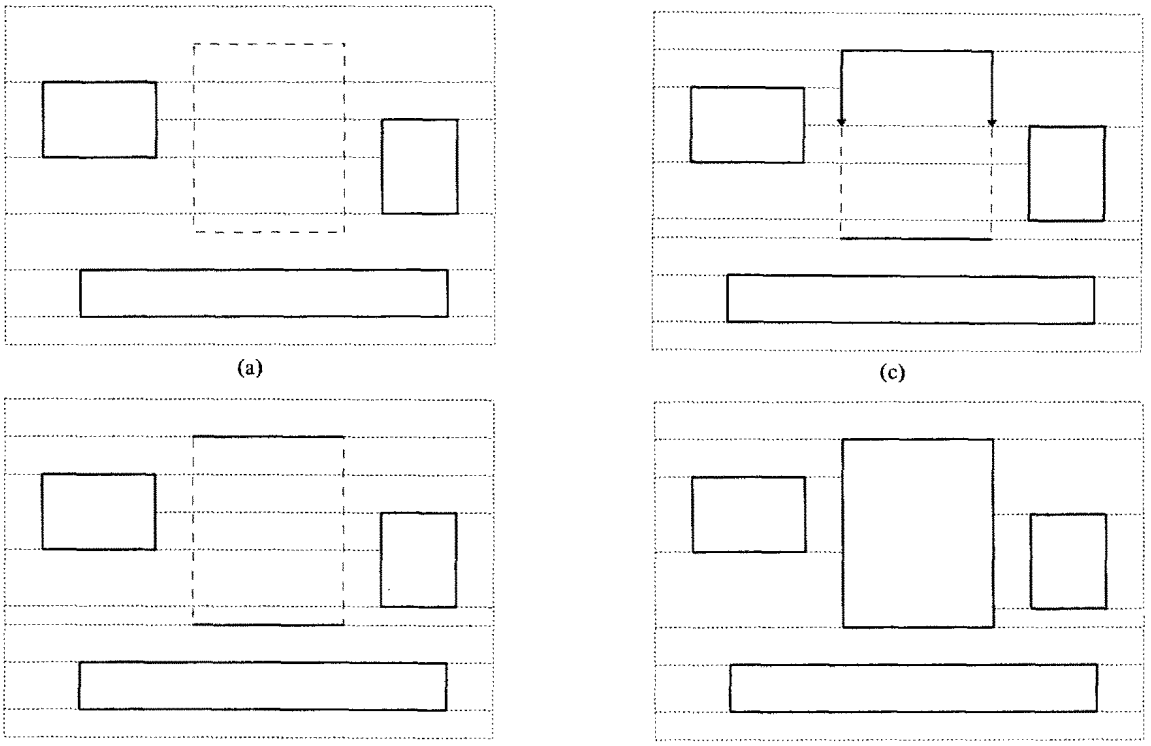



Fig. 11. Inserting a new solid tile into the data structure. (a) shows the desired location of the new tile. In (b) the space tiles containing the top and bottom edges of the new solid tile are split. In (c) and (d) the area of the new tile is traversed from top to bottom, splitting and joining space tiles on either side and pointing their stitches at the new solid tile.

that in the worst possible case, the cost of creating a new tile could be proportional to the total number of tiles in the layout (see Fig. 9(a)). In the average case, the running time is constant if the new tile is about the same size as the tiles around it; if the new tile is much larger than its neighbors, then the running time is proportional to the height of the new tile and independent of its width.

### 5.6 *Tile Deletion*

Tile deletion is complicated by the need to split and merge space tiles so as to maintain the horizontal-strip representation. The algorithm below works in a mostly clockwise fashion around the tile being deleted, which is referred to as the *dead tile*. See Fig. 12 for an example.

1) Change the type of the dead tile to "space".

2) Use the neighbor-finding algorithm to search from top to bottom through all the tiles that adjoin the right edge of the dead tile.

3) For each space tile found in step 2), split either the neighbor or the dead tile, or both, so that the two tiles have the same vertical span, then merge the tiles together horizontally.

4) When the bottom edge of the original dead tile is reached, scan upwards along the left edge of the original dead tile to find all the space tiles that are left neighbors of the original dead tile.

5) For each space tile found in step 4), merge the space tile with the adjoining remains of the original dead tile. Do this by repeating steps 2)-3), treating the current space tile like the dead tile in steps 2)-3).

6) It is also necessary to do vertical merging in step 5). After each horizontal merge in step 5), check to see if the result tile can be merged with the tiles just above and below it, and merge if possible.

As with the other algorithms, deletion could require a great deal of time in pathological cases. For example, Fig. 9(b) shows a situation where corner stitches will have to be examined and modified in every single tile in the layout, so running time will be proportional to the overall layout size. However, situations like this are not likely in integrated circuits. If the tiles are roughly uniform in size and distribution, then the number of splits and joins will be constant and the running time will also be constant. When a large tile is being deleted, the running time will be proportional to the number of left and right neighbors of the tile, which is proportional to the tile's height.

### 5.7 *Plowing*

Plowing is an example of an important operation that cannot easily be implemented with most existing data structures. When one piece of a large design is moved, it is often desirable for other pieces of the design lying in the path of motion to move as well, as if the original piece were a plow. Ideally, such a motion will stretch or shrink the design while maintaining design rules and connectivity. Plowing can be accomplished with corner stitching in the following way:

1) Determine the rectangular area that will be swept out by the motion of the original tile (see Fig. 13).

2) Use the area-finding algorithm to see if there are any solid tiles in the plow area. If a solid tile is found, invoke the plow algorithm recursively to move the tile out of the plow area. Repeat this step until no solid tiles are found.

3) Delete the original tile from its old location and create it at the new position.

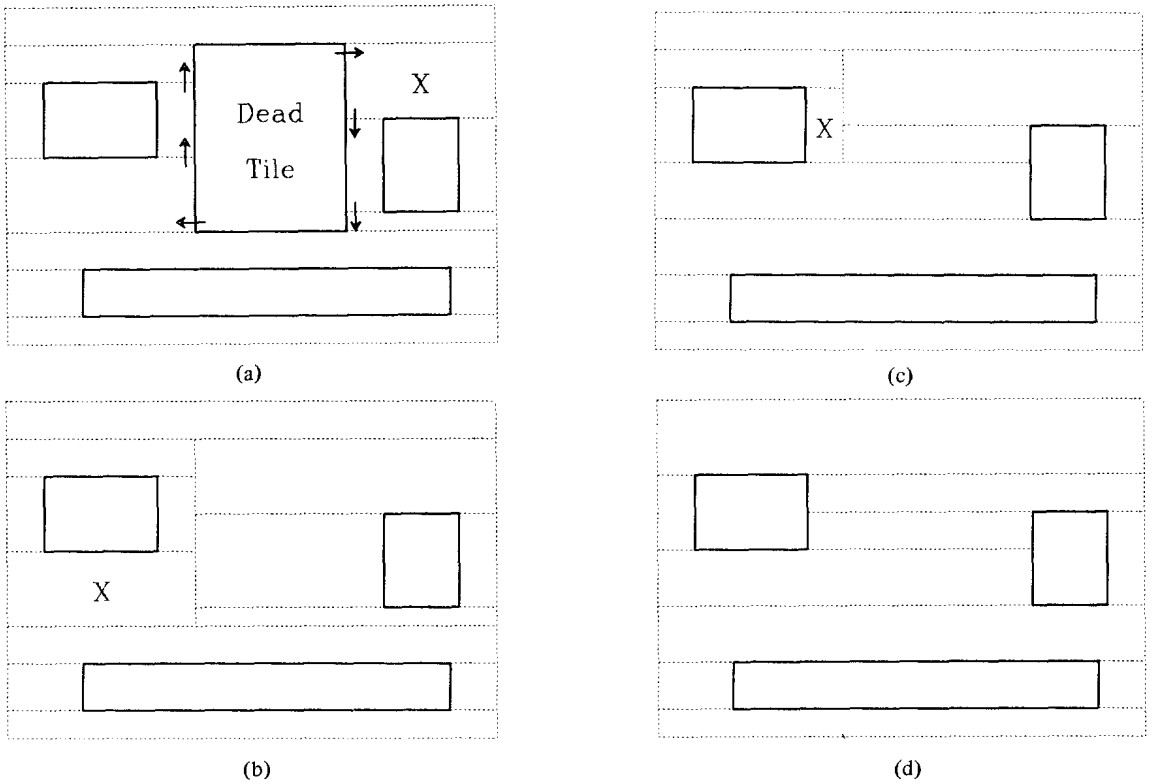



Fig. 12. An example of tile deletion. In each figure, tile $X$ is the next one to be processed. (a) Shows the initial tile arrangement and the clockwise order in which stitches will be traversed around the dead tile to merge it with adjacent space tiles. In (b) the downward sweep along the right edge has been completed (note that the left edge of the dead tile is still intact). In (c) the upward sweep along the left edge is partially complete, and (d) shows the final situation.

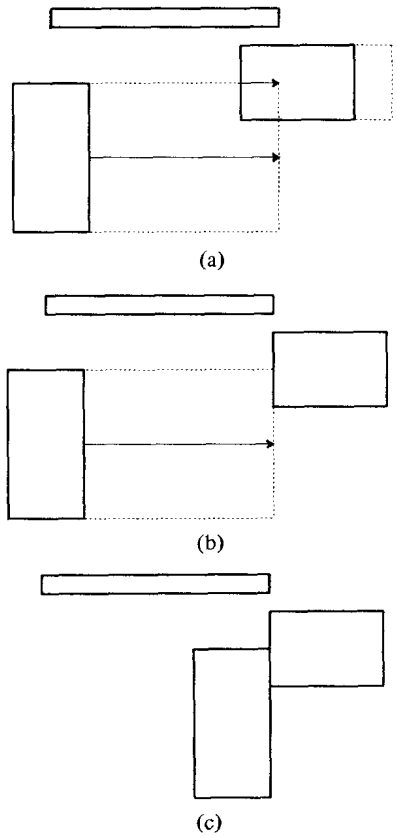



Fig. 13. An example of plowing: (a) determine the area to be swept out by the motion; (b) recursively move all solid tiles out of this area; and (c) move the original tile.

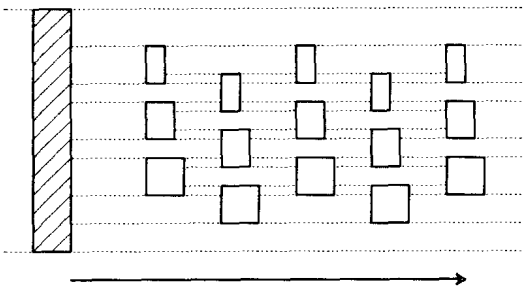


Fig. 14. Using a top-to-bottom area search with the simple plowing algorithm, this structure will cause the rightmost tiles to be moved many times when the cross-hatched tile is plowed to the right. Total running time will be exponential in the circuit size.

Unfortunately, this simple algorithm suffers from terrible worst-case behavior. Lattice structures like the one in Fig. 14 can require up to $2^N$ recursive tile moves to clear $N$ tiles out of the plow area. It seems likely that structures similar to the one in Fig. 14 may occur in actual circuits. Fortunately, the algorithm can be made to run in linear expected time by ordering the recursive processing so that a tile is not moved until its final position is known (i.e., it is not processed until all the tiles that can affect its final position have been processed). The code is somewhat complex, and is different for horizontal plowing than for veritcal plowing. Appendixes C and D develop the linear time algorithm in detail. In the worst case, the algorithms of Appendixes C and D could require $MN$ time, where $M$ is the total number of tiles that have to be moved

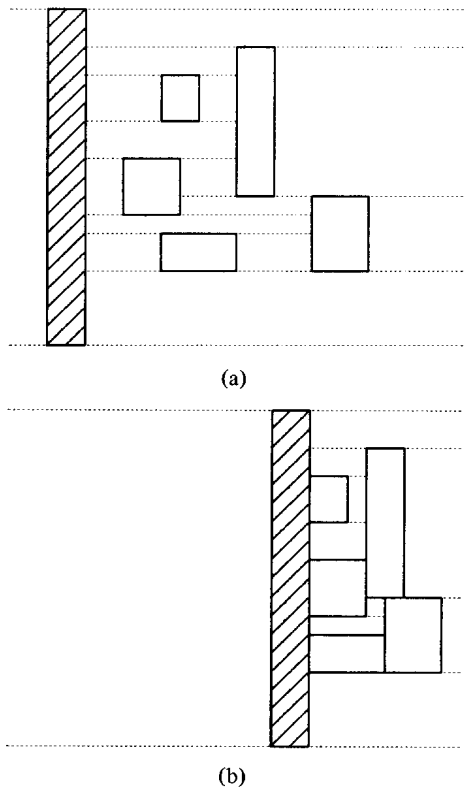



Fig. 15. To compact a layout horizontally, plow a large additional tile (cross-hatched in the figure) across the layout: (a) shows the configuration before the plow, and (b) shows the compacted configuration afterwards. The tile acts like a broom and compacts as it sweeps.

and $N$ is the size of the circuit. In the average case they require time linear in $M$.

### 5.8 Compaction

Most existing algorithms for compaction require $N^2$ time in the worst case for a layout containing $N$ elements, and have been empirically observed to have average running time close to $N^{1.2}$ [8]. With corner stitching, compaction is linear in the size of the layout. Compaction in a single direction can be achieved in a simple way by plowing a large tile across the layout, as shown in Fig. 15. The linear expected time for plowing results in linear expected time for compaction. The worst-case compaction time is still $N^2$ using corner stitching.

There are two keys to the speed of compaction in corner stitching. The first, and most important, is that all the dependencies between tiles are maintained dynamically. In other compaction systems, the dependencies must be reconstructed after each change to the layout; the algorithms for generating dependencies limit the overall speed of compaction. The second key is that the layout is planar. This means that the number of adjacencies is linear in the number of tiles, and hence, the whole layout can be scanned in time proportional to the number of tiles.

### 5.9 Channel Finding

Channel information is constantly available in the form of the space tiles. The corner stitches make it possible to find connected channels and thereby trace out signal paths. Of course, some routers may prefer a different representation of channels than maximal horizontal strips; if this is the case, then conversion will be necessary to cast the space tiles into a form suitable for routing.

TABLE I
Corner stitching requires about 40 percent more storage per tile than linked list systems like Caesar. Only the lower and left coordinates of each tile need be stored in corner stitching, since the upper and right coordinates can be gotten by examining the lower and left coordinates of neighboring tiles.

| | Caesar | Corner Stitching |
|---|---|---|
| Coordinates | $x_1,y_1,x_2,y_2$ (16 bytes) | $x_1,y_1$ (8 bytes) |
| Pointers | 1 link (4 bytes) | 4 stitches (16 bytes) |
| Tile Type | not needed | (4 bytes) |
| Total | 20 bytes | 28 bytes |

## VI. Space Requirements

Because of the enormous size of VLSI designs, a data structure used for VLSI CAD must be space efficient if it is to be effective. For example, the hierarchical representation of a 45 000-transistor chip requires about $1.5 \times 10^6$ bytes of main memory in Caesar. Corner stitching requires more information to be kept in the data structure than systems like Caesar. Table I compares corner stitching to the linked-list scheme of Caesar. Corner stitching requires three more pointers than Caesar, plus a type field (in linked-list systems all the tiles on a given list are of the same type). Corner stitching saves space by storing only the lower and left coordinates of each tile, instead of four coordinates: the upper and right coordinates of a tile can be gotten from the lower and left coordinates of neighboring tiles. As a result, corner-stitched tiles are about 40 percent larger than Caesar tiles. In addition, there are many more tiles in corner stitching than in other systems since corner stitching requires empty space to be represented. If there are many space tiles, then corner stitching will require too much space to be practical. Furthermore, most of the algorithms depend on the *total* number of tiles in an area, including both space and solid tiles; if there are many space tiles, the algorithms will be inefficient.

In a circuit with $N$ solid tiles, there will never be more than $3N + 1$ space tiles. Furthermore, the horizontal-strip representation is at least as efficient (in the worst case) as any other rectangle-based representation of space. In actual circuit layouts, the number of space tiles is about equal to the number of solid tiles.

The proof of the $3N + 1$ upper limit is due to C. Séquin. To see that no more than $3N + 1$ space tiles are needed for $N$ solid tiles, place the solid tiles one at a time in order from right to left as shown in Fig. 16. Initially there is a single space tile. When each solid tile is placed, it can result in no more than three new space tiles: the top and bottom edges may each cause a space tile to be split, and a new space tile will be created in the shadow to the left of the solid tile. Because we place the solid tiles in order, there can be no solid tiles in the shadow. This means that only a single space tile will be created there. Although the solid tiles were placed in a particular order to demonstrate the $3N + 1$ limit, the final configuration is independent of the order in which the tiles

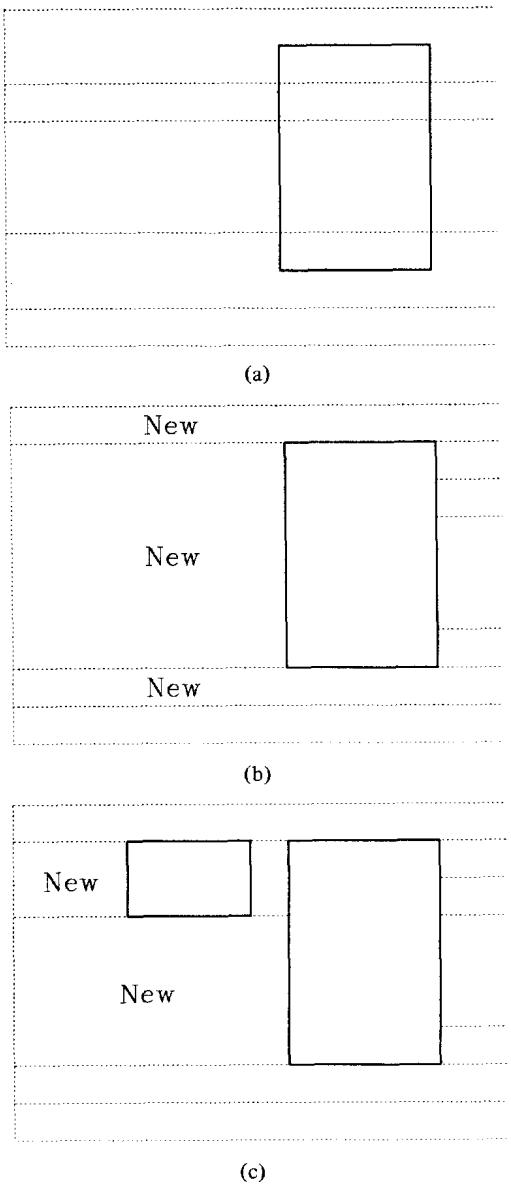



Fig. 16. Both (a) and (b) show that if solid tiles are inserted in order from right to left, each tile causes no more than three additional space tiles to be created. However, if edges of the new tile align with edges of old tiles, an in (c), less than three additional space tiles will be required.

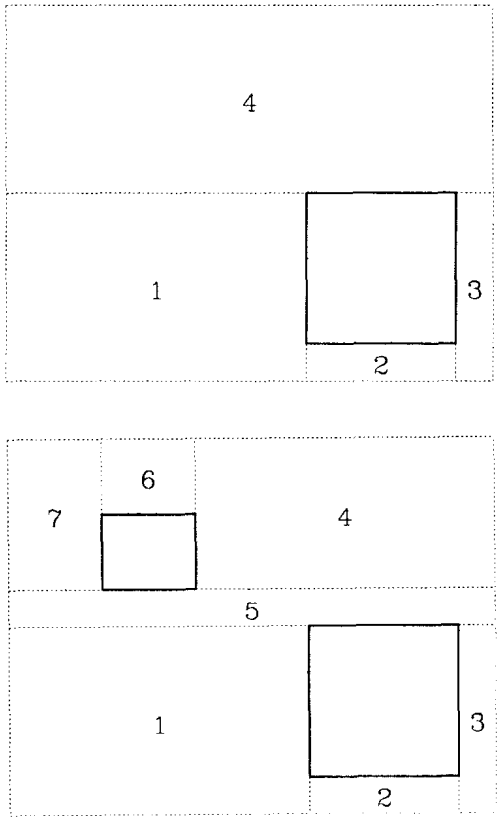



Fig. 17. In pathological situations where no two solid tiles have colinear edges, at least $3N + 1$ tiles must be used to represent space, regardless of whether or not horizontal strips are used.

TABLE II
For actual layouts, corner stitching requires about one space tile for each solid tile. The first case consists of all the global routing for the RISC I microprocessor (i.e., all the rectangles in the topmost cell of the hierarchy. The routing is sparse. The second and third cases consist of cells of another microprocessor under development.

| Circuit | Solid Tiles | Space Tiles | Space/Solid |
|---|---|---|---|
| Global Routing | 8037 | 8473 | 1.05 |
| ALU Latch | 177 | 174 | .98 |
| Register Cell | 77 | 65 | .84 |

are placed (the horizontal-strip property guarantees this). Thus the result is valid regardless of the order of solid-tile creation.

There are many other ways to organize space tiles besides maximal horizontal strips. However, in the worst case, no representation of space can use less than $3N + 1$ space tiles. This worst case occurs when no two solid tiles have colinear edges. Fig. 17 shows one such situation.

Substantially fewer than $3N + 1$ space tiles are needed for actual VLSI applications. Fewer space tiles are needed whenever edges of neighboring solid tiles align. For example, Fig. 16(c) shows a situation where the placement of a solid tile only adds two space tiles instead of three. In integrated circuits, the solid tiles must touch each other to achieve electrical connectivity, so the number of space tiles actually needed is much less than $3N$. Table II shows sample data gathered from three cells using a layout editor based on corner stitching. On the average, about one space tile is required for each solid tile. This means that the total storage required for geometry in corner stitching will be between two and a half and three times as great as in systems like Caesar. This result applies even when the mask layers are sparse, as in the global routing example.

## VII. Using Corner Stitching for Real VLSI

The scheme presented here must be extended in several ways to make it practical for real integrated circuits. This section presents some of the important issues and discusses possible solutions. To date, there have been two implementations of corner stitching. A toy implementation was built using exactly the model and algorithms of this paper, in order to test the basic viability of the ideas. About 1100 lines of C code were required to implement all the algorithms, including compaction, and for small test cases (100 tiles) response was instantaneous for all operations.

As a result of the successful toy implementation, we have undertaken the development of a full-fledged VLSI layout editor based on corner stitching. It has just recently become operational. Although stretching and compaction have not been implemented yet, the current system is at least as powerful as its predecessor, Caesar, and is being used by chip designers at the University of California at Berkeley.

The first generalization of the simple scheme is to provide for multiple mask layers. There are several ways to accomplish this. One alternative is to permit many different types of solid tiles, one type for each possible combination of mask layers. Unfortunately, this scheme will result in enormous numbers of tiny tiles in places where several mask layers cross each other. Many of the layer crossings are not relevant, so the fragmentation of the tile structure wastes space unnecessarily (for example, it doesn't matter where metal crosses polysilicon or diffusion, unless there are contact cuts present). Another alternative is to keep a separate corner-stitched "plane" for each mask layer. This scheme will be relatively space efficient, but will require frequent cross-registration between planes during operations such as plowing and design-rule checking that deal with layer interactions.

For our layout editor based on corner stitching, we used a combination of these two schemes. The polysilicon, diffusion, and implant layers are kept together in a single corner-stitched plane with different types of solid tiles for each layer combination. This makes sense because most of the different combinations of these layers are distinct electrically. Each metal layer is kept in its own corner-stitched plane, since they interact only weakly with each other and with the rest of the circuit. Because contacts provide a connection between layers, they are duplicated in each of the planes that they connect. Under this scheme, the corner-stitched representation corresponds almost exactly to the electrical circuit, since the transistors (combinations of polysilicon and diffusion and implants) are represented by special tile types. Furthermore, this particular division of mask layers among planes allows each plane to be design-rule checked independently.

To handle hierarchical designs, our layout editor keeps a separate set of tile planes for each cell in the design. An additional corner-stitched plane per cell is used to keep track of the cell's subcells. A different tile type is used in this plane for each distinct subcell or overlap area between subcells.

Design-rule checking is trivial in the simple model. The only design rule is that there can be no solid-tile overlap; this condition is enforced by the creation and plowing routines. In actual IC designs, the design rules will include more complex spacing and separation rules that are different for different tile types. For the corner-stitched editor, we have implemented a simple design-rule checker similar to Lyra [1] except that it is edge based instead of corner based. It scans a corner-stitched plane, generates constraints at each edge based on the tile types on either side of the edge, and uses area enumeration to check the constraints. To handle areas of overlap between subcells, the design-rule checker extracts information from the separate planes of the subcells into an auxiliary corner-stitched structure and then checks the auxiliary structure.

The plow algorithm is also affected by more complex design rules, and must deal with connectivity as well. Although the implementation of plowing is not yet complete, real VLSI design rules appear to be accomodated by selectively expanding the plow area to maintain proper spacings. For example, if the metal-metal spacing must be three units, then, when plowing a metal tile, all unrelated metal must be cleared from an area three units larger on all sides than the area swept out by the tile's motion. Connectivity appears to be handled by selectively stretching or shrinking some tiles, rather than moving them.

TABLE III
TYPICAL AND WORST-CASE RUNNING TIMES FOR THE ALGORITHMS
$M$ refers to the number of tiles of direct interest to the algorithm (e.g., the number of tiles being enumerated in area enumeration, or the number of tiles removed in plowing). $N$ refers to the total number of tiles in the circuit.

| Algorithm | Expected Time | Worst-case Time |
|---|---|---|
| Point Search | $\sqrt{N}$ | N |
| Point Search (with hint) | constant | N |
| Neighbor Search | M | M |
| Area Search | M | N |
| Directed Area Enumeration | M | N |
| Tile Creation | constant | N |
| Tile Deletion | constant | N |
| Plowing | M | MN |
| Compaction | N | $N^2$ |

In some industrial environments, the Manhattan restriction may be intolerable. Where this is the case, it may be possible to accomodate 45°-angles by using trapezoids instead of rectangles. Degenerate trapezoids can be used to represent triangles. We do not plan to implement non-Manhattan features in our system, since in our environment, the Manhattan restriction is acceptable (and even desirable, since it tends to simplify designs and make tools run two to ten times faster). The Manhattan design style seems to be gaining more and more acceptance in the integrated circuit design community as a whole. For example, the Caesar editor, which is Manhattan, is now being used at nearly 200 industrial and university sites.

## VIII. CONCLUSION

Corner stitching is a powerful technique for representing geometrical data. Its two most important features are a) it represents empty space explicitly, and b) it links together tiles of various types at their corners. These two features make it possible to implement a variety of important operations that operate purely locally. The efficiency of the algorithms depends only on local information and not on the overall circuit size. The database can be modified incrementally, so that one portion of the design can be changed without invalidating the pointer information of any other piece of the design. Corner stitching is effective both for densely packed circuits and for sparse ones. See Table III for a summary of the complexity of the various algorithms.

The main drawback of the mechanism is that it requires approximately three times as much storage as simple mechanisms. Fortunately, designers tend to focus their attention on a small portion of a layout at any given time; since corner stitiching uses only local information, it will have good paging behavior in a demand-paded environment.

## Appendix A
## Adjacencies

The running time for several of the algorithms depends on the number of neighbors an individual tile has. One can construct situations where a tile has an arbitrarily large number of neighbors, so it is not possible to state any absolute upper bounds. However, graph theory can be used to determine the average number of adjacencies. In any connected planar graph

$$n - e + f = 1$$

where $n$ is the number of nodes, $e$ is the number of edges, and $f$ is the number of faces contained by the edges. A face corresponds to a tile, a node to a corner of a tile, and an edge to a distinct adjacency between two tiles. For $T$ tiles, $f = T$. The number of distinct nodes $n$ can be at most $4T$, but in the interior of the tile structure, each corner of one tile must coincide with at least one corner of another tile (a "T" structure). Thus $n \leqslant 2T$ and the total number of adjacencies is

$$e = n + f - 1 \leqslant 3T - 1.$$

Note that at the outside of the structure there may be corners that don't coincide with other corners, but for each of these there is also as least one edge that doesn't represent an adjacency (because there is no tile on the other side). Hence the $3T - 1$ upper limit is not affected.

The $3T - 1$ limit counts each adjacency only once for the two tiles that are adjacent. To compute the number of neighbors per tile, the figure must be doubled. This means that on the average, an individual tile will have about six neighbors, or about one or two on each side. This is regardless of the arrangement of tiles. Of course, if there are many tiles of different sizes, the large tiles may have many more than six neighbors. The average number of neighbors of a tile in a situation like this will be roughly proportional to the perimeter of the tile, which is less than linear in its area.

## Appendix B
## Splitting and Merging

This section discusses the cost of splitting one tile into two adjacent tiles, or merging two adjacent tiles into a single tile. A tile can be split into two tiles as follows:

1) Make an exact copy of the original tile.
2) Update the coordinates of each tile to reflect the split, and set the tiles' corner stitches to refer to each other.
3) Update the corner stitches in tiles that are now adjacent to the new tile. To do this, use the neighbor-finding algorithm to locate the neighbors on three sides of the original tile, then update the stitches that must point to the new tile.

The algorithm for merging two adjacent tiles into a single larger tile is similar: stitches must be updated along three sides of the tile that is eliminated.

The cost of each algorithm consists of constant factors (copying a tile or changing an $x$ or $y$ coordinate) and the search of neighbors on three sides. Appendix A showed that the number of neighbors was constant when averaged across a whole design, but increases for those tiles that are much larger than their neighbors. In this case, the average number of neighbors will be approximately proportional to the perimeter of the tile. Thus the cost of a split or merge is constant if the tile being split or merged is about the same size as its neighbors. If the tile is much larger than its neighbors, then the cost increases in proportion to the tile's perimeter, which is less than linear in its area.

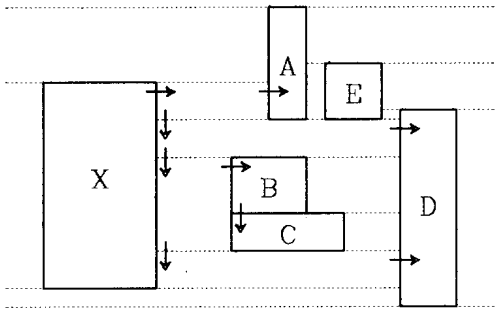



Fig. 18. An example of visibility searching. From tile $X$, tiles $A$, $B$, $C$, and $D$ are visible to the right. Tile $E$ is not visible from $X$. Tile $D$ has two distinct windows of visibility to $X$, one between $A$ and $B$ and one below $C$. During a horizontal visibility search from $X$, the pictured corner stitches will be traversed.

## Appendix C
## Visibility Searching

This section gives algorithms that locate all solid tiles *visible* on one side of a given solid tile. Two solid tiles are mutually visible if it is possible to draw a horizontal or vertical line between them without crossing any other solid tiles. Fig. 18 gives examples of visible and invisible tiles. Visibility searching is used during compaction and stretching. Unfortunately, the horizontal-strip representation of space requires different algorithms for horizontal and vertical searches.

Horizontal-visibility searching is based on the neighbor-finding algorithm of Section V-5.2. The following algorithm is for searching on the right side of the original tile; it can be modified to search on the left side.

1) Use the neighbor-finding algorithm to enumerate the tiles that touch the right side of the starting tile. For each tile found, execute step 2) or step 3), depending on the tile's type.
2) If the neighbor is solid, then it is automatically visible.
3) The neighbor is a space tile. If it extends all the way to the edge of the circuit (infinity) ignore it. Otherwise, use the neighbor-finding algorithm once again to enumerate all the tiles that touch its right side. Each of these must be a solid tile. All of the tiles whose bottom edges are lower than the top edge of the starting tile are visible.

In this algorithm, a single solid tile may be enumerated several times, once for each distinct window of visibility with the starting tile (see, for example, tile $D$ in Fig. 18). The time required for the horizontal search is linear in the total number of tile adjacencies in the search area, which was shown in Appendix A to be linear in the number of tiles. Since there must be at least one solid tile enumerated for every space tile enumerated, the expected running time of the search is linear in the number of solid tiles found. For tiles in a relatively uniform distribution, the number of visible neighbors

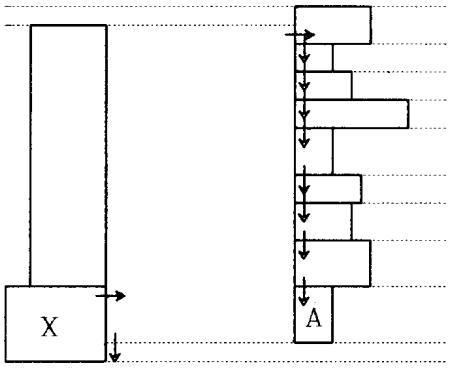



Fig. 19. A pathological case for horizontal visibility searching. When searching for tiles visible to the right of $X$, all of the tiles above $A$ will have to be passed through and skipped over.

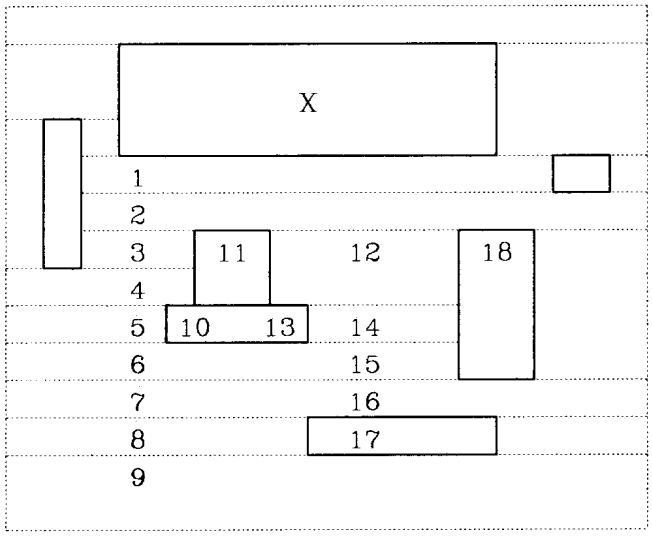



Fig. 20. In a downward visibility search from $X$, columns of space tiles are traversed until solid tiles are found or the end of the circuit is reached. In this case, the numbers give the order in which the tiles will be traversed (the numbering ignores realignments that must occur when advancing down the side of a column). Tiles that fall under more than one column are traversed more than one time.

of a given tile is small and independent of the size of the circuit. However, if a space tile found in step 3) extends above the starting tile, as in Fig. 19, it could have any number of out-of-range solid tiles along its right edge; since these have to be skipped over, the upper limit on the running time for the algorithm is the total number of solid tiles in the circuit.

For vertical visibility searches, the algorithm is an extension of the area-search algorithm of Section V-5.3. It consists of a recursive set of searches of successively thinner columns. The following algorithm will find all the solid tiles visible below the starting tile; it can be modified to find all those above the starting tile. See Fig. 20 for an example.

1) The intial column being searched extends downward from the bottom of the starting tile. Use the approach of the area-searching algorithm to advance one by one through the tiles lying under the left edge of this column.

2) When a space tile is found in step 1), check to see if it extends across the whole column. If so, then advance downwards to the next tile (this is the case for tiles 1 and 2 in Fig. 20). If the space tile extends downward to -infinity, then return.

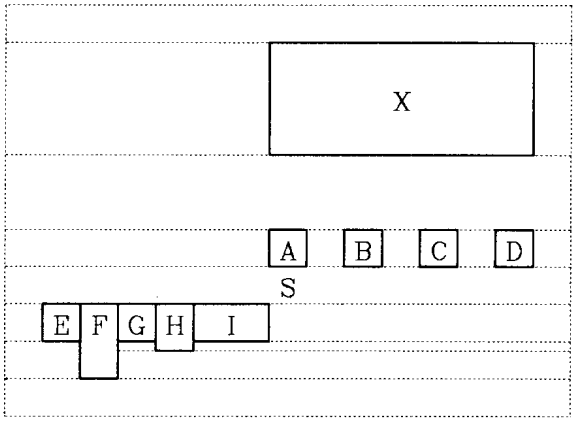



Fig. 21. Tile $S$ causes severe misalignment during downward visibility searches from $X$. Each of tiles $E$-$I$ will have to be traversed for each column between tiles $A$-$D$.

3) If a solid tile is found, or if the space tile does not extend across the entire column, then do not continue down any further. Instead, scan across the top of the column (following **tr** stitches from the tile found in step 1). Each of the solid tiles found in this way is visible to the starting tile. For each space tile found in this scan, invoke a recursive search on the column underneath this space tile (tiles 3 and 12 in Fig. 20 are examples of space tiles that start new column searches).

The algorithm terminates when all of the columns have been closed off by continuous solid tiles across the columns or when the end of the circuit (infinity) is reached. As with the horizontal search, tiles are enumerated once for each window of visibility with the starting tile. Since each of the visible tiles is visited once for each window of visibility, the expected running time is linear in the number of visible tiles (for relatively uniform tile distributions). However, the same misalignment that was illustrated in Fig. 9 for area searching can occur here, as shown in Fig. 21. In the unlikely event that most of the tiles in the circuit are piled up like tiles $E$-$I$ in Fig. 21, they will all have to be traversed as part of each column, and the total running time will be proportional to the product of total circuit size and number of visible tiles.

Each of the visibility algorithms works in a particular direction. The directed nature is important to other algorithms that use visibility searches. The right visibility search enumerates visible tiles in order from top down, and the bottom visibility search enumerates visible tiles in order from left to right.

## Appendix D
## Plowing in Linear Time

The poor worst-case behavior of the plowing algorithm in Section 5.7 occurred because the algorithm processed tiles in a haphazard order. As a result, some tiles could be moved many times as the algorithm discovered that more and more space was needed to move other tiles out of the plow area. The linear-time algorithm makes two passes over the circuit. In the first pass, it computes how far each tile must move; tiles are processed in topological order so that a given tile

is not processed until its final position is known. In the second pass, the tiles are actually moved. Each tile is moved exactly once.

The linear-time algorithm requires an extra data value to be stored in each tile. This additional value is called the tile's *delta,* and gives the distance that the tile must be moved. Initially, all of the deltas are zero; when the plowing algorithm is finished, it leaves all the deltas zero for future plowing. The linear-time algorithm also requires the use of a linked list of tiles to be moved. Tiles are added to the linked list in the first pass; in the second pass, tiles are moved in list order.

Pass 1 consists of setting the delta of the initial tile to its plow distance and calling the following recursive procedure to process the tile. The basic algorithm is independent of the plow direction.

1) Add a pointer to the current tile onto the front of the list of tiles to be moved. This tile will be moved *before* all previously encountered tiles.

2) Use the tile's delta and location to compute the area that this tile will plow out as it moves.

3) Use the visibility search from Appendix C to enumerate all visible solid tiles in the plow area. Execute steps 4) and 5) for each neighbor tile found in this way.

4) Compute the delta required to move the tile out of the plow area. If this delta is greater than the tile's current delta, then update the tile's current delta.

5) If this is the last time we will ever see the neighbor, then call this procedure recursively to process the neighbor. The determination of "last time" depends on the directed nature of the algorithms for visibility searching. For example, in a left-to-right compaction, the "last time" is when the neighbor's bottom edge is in the window of visibility, or when the bottom edge of the overall plow area is in the window of visibility. The bottom edge of the overall plow area must be passed down in the recursive calls; it is the lowest bottom edge for any plow on the recursive stack.

Pass 2 scans the list in order from front to back. Each tile on the list is erased, then recreated at a new position determined by its delta. The ordering of the list guarantees that the final position of each tile is empty at the time it is moved. When moving the tiles, the deltas are zeroed out again in preparation for the next plowing operation.

If the total number of tiles moved is $M$ and the total number of tiles in the circuit is $N$, then each of the two passes has an expected running time that is of order $M$, with worst-case running time proportional to $MN$. In pass 1, the recursive procedure is invoked exactly $M$ times (once for each tile that must be moved). The overall running time for pass 1 is determined by the time spent in enumerating all the visible neighbors for all the tiles that are moved. In the average case, each tile's visible neighbors can be found in constant time, so the total running time is proportional to $M$. In the worst case, the cost of the visibility searches may be $MN$, so the worst-case running time of pass 1 is of order $MN$.

For pass 2, the expected time to delete or create each tile is constant, so the expected running time is linear in $M$. However, the worst-case deletion or creation time for a tile is proportional to the overall circuit size, so the worst-case running time for pass 2 is of order $MN$.

## Acknowledgment

Michael Arnold, Carlo Séquin, David Ungar, and David Wallace all took part in the discussions that led to the formulation of corner stitching. C. Séquin developed the proof that $3N + 1$ space tiles are always sufficient in a design with $N$ solid tiles. Gordon Hamachi, Bob Mayo, Walter Scott, and George Taylor implemented the layout editor based on corner stitching. In addition to these people, Leo Guibas, David Patterson, Alberto Sangiovanni-Vincentelli, and the referees all provided helpful comments on drafts of this paper.

## References

[1] M. H. Arnold and J. K. Ousterhout, "Lyra: A new approach to geometric layout rule checking," in *Proc. 19th Design Automation Conf.*, pp. 530-536, 1982.

[2] J. L. Bentley and J. H. Friedman, "A survey of algorithms and data structures for range searching," *ACM Computing Surveys*, vol. 11, no. 4, 1979.

[3] M. Y. Hsueh, "*Symbolic layout and compaction of integrated circuits*," University of California, Berkeley, Tech. Rep. UCB/ERL/M79/80, Dec. 1979.

[4] G. Kedem, "The quad-CIF tree: A data structure for hierarchical on-line algorithms," in *Proc. 19th Design Automation Conf.*, pp. 352-357, 1982.

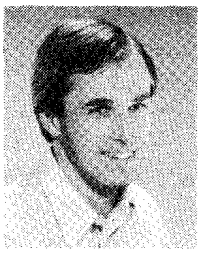
[5] K. H. Keller and A. R. Newton, "KIC2: A low cost, interactive editor for integrated circuit design," in *Dig. Papers for COMPCON Spring 1982*, pp. 305-306.

[6] J. K. Ousterhout, "Caesar: An interactive editor for VLSI," *VLSI Design*, vol. II, no. 4, pp. 34-38, fourth quarter 1981.

[7] J. K. Ousterhout, and D. M. Ungar, "Measurements of a VLSI design," in *Proc. 19th Design Automation Conf.*, pp. 903-908, 1982.

[8] A. Sangiovanni-Vincentelli, private communication.

*



**John K. Ousterhout** received the B.S. degree in physics from Yale College, New Haven, CT, in 1975 and the Ph.D. degree in computer science from Carnegie-Mellon University, Pittsburgh, PA, in 1980.

Since 1980 he has been an Assistant Professor of Electrical Engineering and Computer Sciences at the Berkeley campus of the University of California. His research interests include computer-aided design, VLSI architecture, and operating systems.