The User Interface and Implementation of Caesar

John K. Ousterhout
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720
415-642-0865

Abstract

This paper describes several novel aspects of Caesar, a layout editor for Manhattan-style integrated circuits. The program's user interface is similar to painting. By hiding many irrelevant details, the painting mechanism provides a powerful yet simple user interface. Its implementation using horizontal strips is efficient in both time and space. To handle large circuits efficiently, Caesar represents them hierarchically and capitalizes on their hierarchical structure to avoid excess computation and I/O. The rendering of mask information on color displays is done with a novel combination of transparent and opaque layers that clarifies layer interactions even in the presence of a large number of mask layers.

Keywords: Computer-aided design, integrated circuits, graphics, user interfaces.

1. Introduction

Caesar is an interactive program that allows integrated circuit designers to enter and modify mask layouts using a color display and a graphics tablet. The program was designed and implemented at the University of California at Berkeley. It has been distributed to over 200 university and industrial sites around the U.S. and is the basis for two commercial layout editors. Caesar has been used successfully to implement several large scale circuit designs [2,5,10,11].

The program's success is due to three factors. First, it has a simple and powerful user interface that is easy to learn and effective to use. Second, it is efficient: in typical use, even for large designs, it requires only twice as much processor time as text editors. Third, it is based on standard hardware and software (DEC VAX-11's, Berkeley 4.1 Unix, and any of a variety of inexpensive commercially-available color displays) so it is easy to port to other sites.

This paper presents several of the novel aspects of Caesar's user interface and implementation. Most important of these is the way users edit mask patterns in a style similar to painting. Sections 2 and 3 present the painting interface and the algorithms used to implement it efficiently. The system's efficiency for large designs is due to its use of hierarchy; the hierarchical structure and techniques for achieving efficiency are described in Sections 4 and 5. A third novel aspect of Caesar is the way it displays mask information on the color display. Section 6 presents this mechanism, which permits a large number of layers to be displayed clearly and efficiently on inexpensive displays. Section 7 evaluates Caesar's strengths and weaknesses.

2. The Painting Interface

In many ways, Caesar is similar to other existing VLSI layout editors [1,3,7]. Each design in Caesar consists of a hierarchical collection of *cells*, with each cell containing subcells and patterns on the various mask layers. Caesar's commands for manipulating subcells are similar to the subcell commands of other systems, and include copying, moving, mirroring, rotating, and arraying. However, Caesar's commands for manipulating geometrical information are quite different from those of other systems.

In most VLSI layout editors, users treat mask patterns as geometrical *objects*: polygons, rectangles, or wires. This view is a direct reflection of the internal data structures used in the systems. Operations are provided to create, delete, and modify the objects. Unfortunately, the geometrical structure is largely irrelevant. When a circuit is fabricated, its characteristics depend only on the patterns and not on their composition in terms of rectangles or polygons (see Figure 1). Polygons and rectangles do not correspond directly to meaningful circuit elements like transistors or gates, so they are of little logical use to the designer. If patterns are displayed as

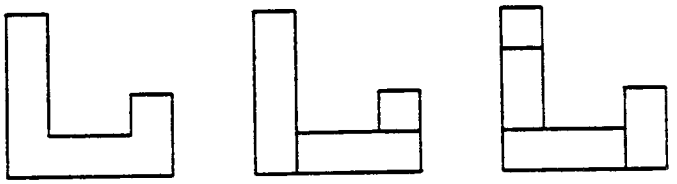

Figure 1. Although each of the three patterns has a different geometrical structure, they will produce identical circuits. In Caesar the user sees only the shape (as on the left), while Caesar manages the representation.

solid filled areas, it is not even possible to distinguish the internal structure of the patterns; nonetheless, users of most editors are forced to deal with that structure when issuing commands.

In Caesar, users do not concern themselves with the structure of mask patterns. They manipulate only the *shapes*. Instead of thinking in terms of objects, Caesar users think in terms of *painting*. Two graphical tools, a box and crosshair, are positioned over the circuit using an electronic tablet. They are used to invoke five painting commands, as shown in Figure 2. In general, the box selects an area to be operated upon, and the crosshair selects particular mask layers to be affected within that area. Caesar han-

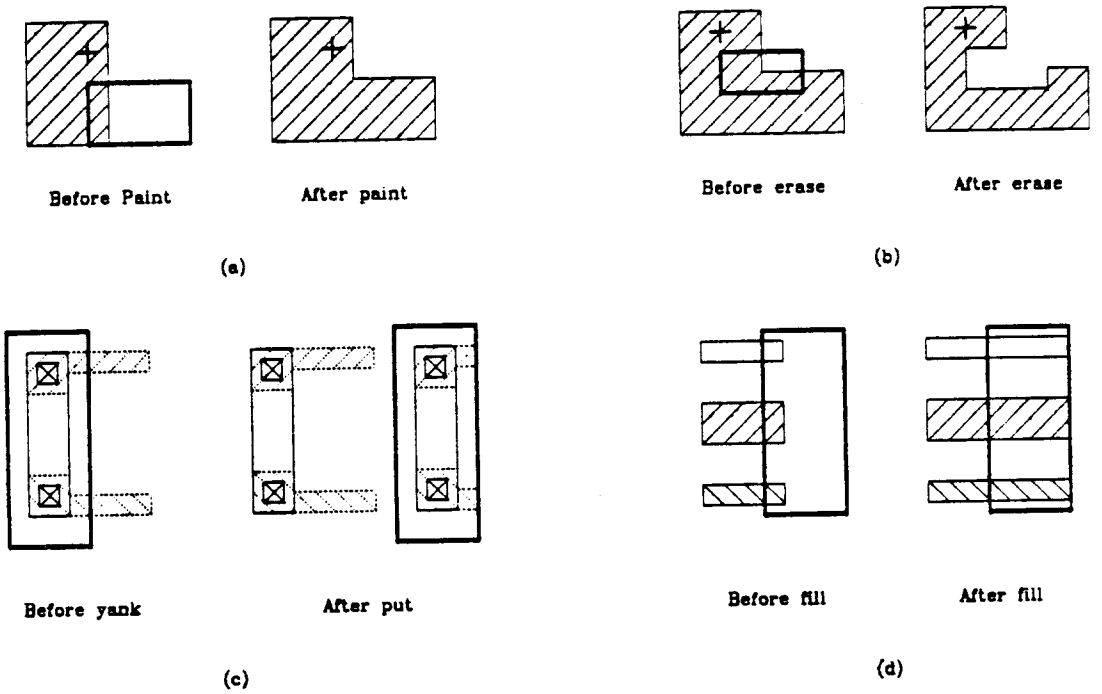

Figure 2. Caesar has five painting operations, controlled by a box and a crosshair: (a) **paint**: the box selects an area to be painted, and the crosshair selects the layer(s) to be painted; (b) **erase**: the box selects an area to be erased, and the crosshair selects the layer(s) to be erased; (c) **yank** saves a copy of the information underneath the box, and **put** copies the saved information back at the box's current position; (d) **fill right** samples all the paint underneath the left edge of the box, then uses that pattern as a paint brush to sweep from left to right (filling can also be done left, up, or down).

dles all the details of how to represent the patterns; users can thus concentrate on what is being designed rather than how it is being represented.

The painting paradigm provides both a very simple and natural user interface (students and visitors typically learn the system in a few minutes), and unusual power in manipulating the circuit. As an example of the power of the painting interface, Figure 3 shows how a sequence of three commands can be used to stretch a cell. Painting is powerful because it allows users to pick up and put down pieces of the circuit without regard for any underlying representation. In "object-based" systems users must worry about maintaining the consistency of objects such as polygons. This often results in different commands for each different type of geometrical object and makes it difficult for users to express non-trivial changes to the circuit.

Caesar accepts only Manattan designs: all features must be horizontal or vertical. Because the box used for painting is a rectangle with horizontal and vertical sides, there is no way to specify non-Manhattan features. Manhattan circuits tend to be slightly less dense than non-Manhattan ones (most designers estimate that the penalty is around 10%), but our designers have readily accepted the Manhattan style. One of the main reasons for this acceptance is the efficiency of CAD tools. Tools specialized for Manhattan shapes execute as much as ten times as fast as those designed for arbitrary angles. Manhattan designs also tend to be simpler and less prone to errors than non-Manhattan ones. Because of these advantages, the Manhattan design style is used almost exclusively in University environments, and is gaining acceptance in industrial settings also (for example, there are between fifty and one hundred companies using Caesar).

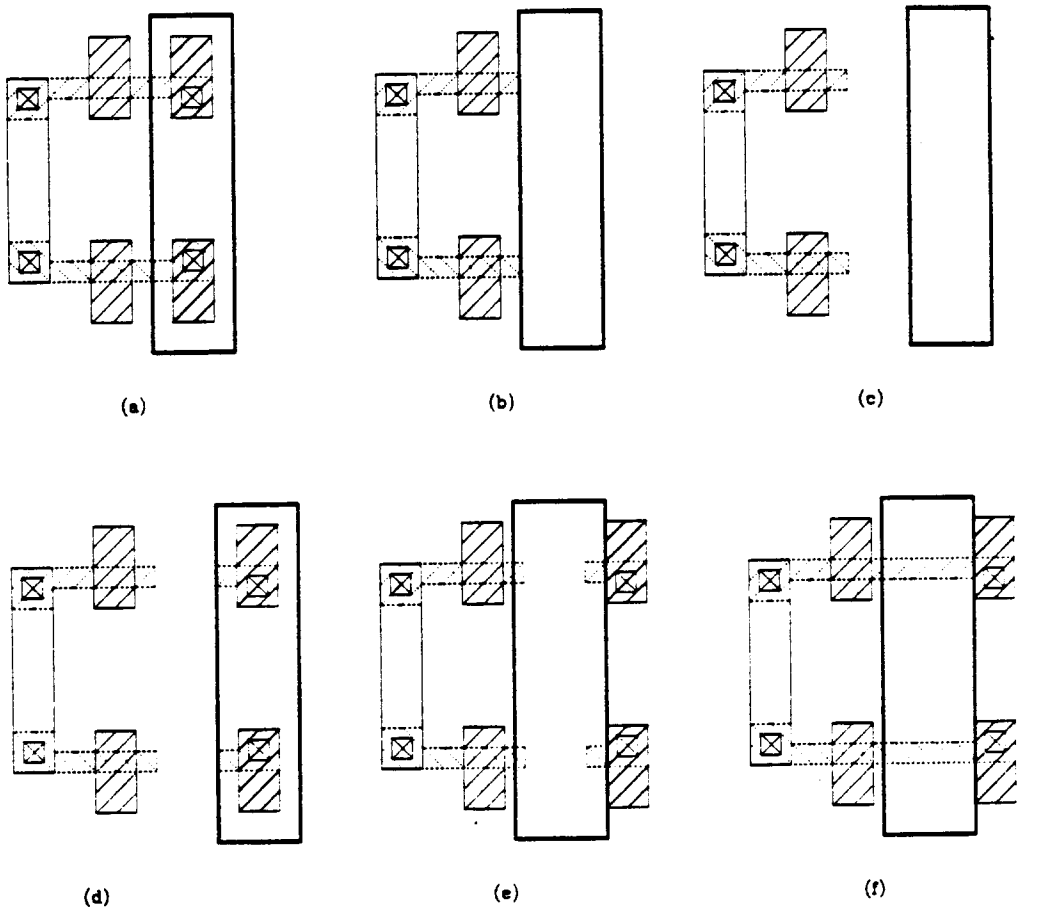

Figure 3. **Erase** followed by **put** followed by **fill right** is sufficient to stretch this cell: (a) before **erase**; (b) after **erase** (note: **erase** saves a copy just like **yank**); (c) before **put**; (d) after **put**; (e) before **fill right**; (f) after **fill right**.

The box is a convenient tool for specifying a variety of commands in addition to painting. These range from zooming (where the box specifies an area that is to be zoomed to full-screen) to making subcell arrays (where the dimensions of the box specify the x- and y-spacings between adjacent elements).

3. Painting Implementation

Since the painting user interface hides the internal representation of shapes, the implementation can use whichever representation turns out to be simplest and most efficient. There are two potential difficulties in having

the layout editor automatically manage the representation of mask patterns. The first danger is that it may be difficult or impossible for the system to find an exact representation for the shapes specified by the user. For example, if circular arcs and lines at arbitrary angles were permitted, there would be no exact representation; floating-point approximations would have to be used, and roundoff errors and other numerical problems would have to be considered. Fortunately, the Manhattan property of Caesar designs, combined with the requirement that all edges lie on a grid, makes it easy to find an exact representation. In Caesar, the mask patterns are represented by a separate linked list of rectangles for each mask layer in each cell. The rectangles within each list are unordered and have integer coordinates.

The second danger is that a long series of edits to a cell might cause the cell's representation to decay into a very large number of unnecessarily small objects, resulting in wastage of memory space and processing time. Caesar eliminates this potential problem by representing the mask patterns with *maximal horizontal strips*. The maximal horizontal strip property requires that no cell ever contain two rectangles on the same layer that share any portion of a vertical edge (see Figure 4). Furthermore, any two rectangles that share an entire horizontal edge (as in Figure 4b) must be merged into a single larger rectangle.

Using horizontal strips, there is exactly one representation for any given mask pattern, regardless of the sequence of editing operations used to create that pattern. Thus the database cannot decay into a large number of small rectangles unless the mask pattern itself becomes very detailed. Initially, Caesar used a different representation, one that did not have these properties. As a result, circuits that were modified frequently tended to

fragment into smaller and smaller rectangles. This cannot happen with the horizontal strips. Because of the automatic merging, maximal horizontal strips result in databases of nearly minimal size; Caesar cells generally have fewer geometries than the same cells layed out with systems where users manage the object structure.

Special algorithms are used by the **paint** and **erase** operations to preserve the horizontal strip property. When the **paint** command is invoked, three steps are taken, as illustrated in Figure 5. First, Caesar scans the list of rectangles on the painted layer to eliminate any new paint that is already present in the cell (this is done by splitting the paint area into a number of smaller horizontal strips). Second, Caesar merges the new paint with old paint, splitting rectangles and merging them horizontally in order to generate maximal horizontal strips. This requires another scan through all the rectangles on the painted layer. Finally, a third scan is made to merge rectangles vertically.

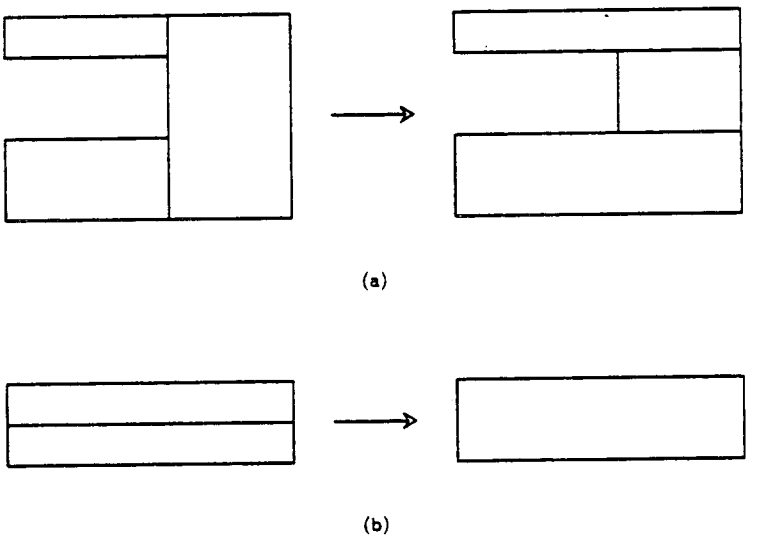

Figure 4. The maximal horizontal strip rule makes the structures on the left illegal; they are changed by Caesar into the structures on the right.

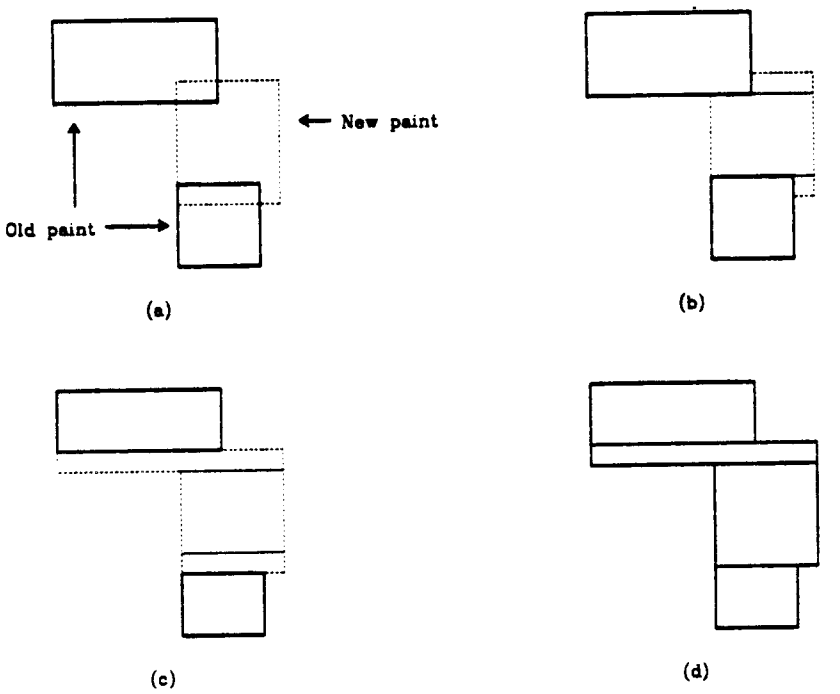

Figure 5. Three steps are used to regenerate maximal horizontal strips during the **paint** command. (a) shows the original pattern and the area to be painted; in (b) the new paint has been clipped against existing paint; in (c) the new and old paint have been merged horizontally; (d) shows the final structure after new and old paint are merged vertically.

When the **erase** command is invoked for a particular mask layer, Caesar must scan the rectangle list for that layer and clip existing rectangles against the area being erased. See Figure 6. When a rectangle is split, the remaining pieces must be merged into the database using the last step described above for painting. This is necessary because the split may have made a vertical merge possible.

Once the algorithms were chosen, the implementation of painting was quite simple. Only about 1000 lines of C code are needed to implement all of the painting operations.

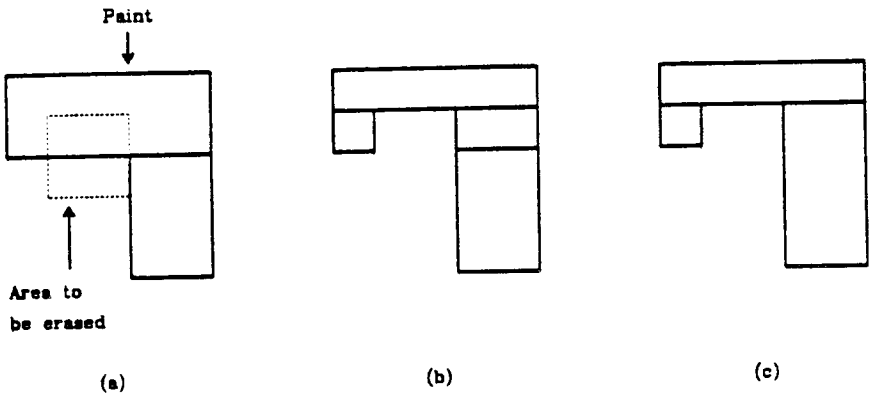

Figure 6. The **erase** operation has two steps: clipping and merging. (a) shows the original structure and the area to be erased; (b) shows the structure after clipping existing rectangles; (c) shows the final structure after vertical merging.

4. Using the Hierarchy

Caesar's space and time efficiency is accomplished almost entirely by exploiting the hierarchical structure of VLSI layouts. Typical cells contain either a few subcells or a few hundred rectangles; only rarely are cells substantially larger than this. See [9] for detailed measurements of the structure of one design; our experiences with several other designs are similar. Caesar uses very simple data structures and algorithms almost everywhere, with a few key techniques (discussed below) to take advantage of hierarchy.

Caesar keeps its space requirements low by storing the circuit only in hierarchical form. Cells are described by two structures, *celldefs* and *celluses*. A celldef describes the contents of a cell: it contains a separate list of paint rectangles for each mask layer, a list of textual labels, and a list of celluses, one for each of the subcells contained in the cell. A celluse describes the way a particular celldef is used in a particular parent; it contains a geometric transformation and a pointer to the child celldef. If a cell is used in several places in a circuit, Caesar stores a single copy of its celldef, with

one celluse for each instance.

Two-dimensional arrays are used extensively in VLSI circuits, so they are handled as a special case to reduce the number of celluses. An array is represented by a single celluse. Information about the array (starting and stopping indices in x and y, and horizontal and vertical spacings between elements) is stored in the celluse. As a result of the hierarchical structure, databases are small enough to be loaded entirely into main memory. Circuits with 40000-50000 transistors require between 1 and 1.5 megabytes of virtual address space when completely loaded.

However, Caesar rarely loads the entire hierarchy for a large circuit. At the beginning of an editing session, Caesar only reads in the top-most cell of the hierarchy being edited. Lower-level cells are displayed as bounding boxes and need not be read from disk. This results in very fast startup of editing sessions (only a second or two of CPU time for even the largest circuits). Caesar reads in lower levels of the hierarchy as the user asks for more detailed information; since this happens incrementally in small pieces, there is almost never any noticeable overhead for this. Typically, the edits in any one session deal with a small portion of the whole chip so Caesar never reads in most of the design. In the RISC I project, on average, only one fourth of the database was read in during any given work session.

Hierarchical structure is used in several other ways to gain speed. At any given time, a single cell is being edited; it is called the *edit cell*. Only the paint within the edit cell and the placement of its subcells may be modified by the user. It is not permissible to modify paint or subcell placements in cells other than the edit cell. Thus Caesar need consider only information in the edit cell when making database changes. Individual cells are

almost always small, so the editing operations are fast.

There are a few occasions when Caesar must locate all paint in a particular area, regardless of the cell structure, for example when displaying information on the screen. In this kind of database search Caesar still uses the hierarchical structure to eliminate unnecessary searching. The database is searched recursively starting at the root of the hierarchy. For each cell searched, two things happen. First, each paint rectangle and label in the cell is examined to see if it lies in the area of interest. If so, it is displayed. Second, the bounding boxes of subcells are examined: if the bounding box of a subcell intersects the area of interest, then it must be examined recursively. If the bounding box of a subcell is outside the area of interest, there is no need to search it or any of its children. This simple pruning technique is quite effective at eliminating from consideration the material that is outside the area of interest.

In order for hierarchical pruning to work correctly, accurate bounding boxes must be maintained for each cell. The bounding box for each cell must reflect all the paint in the cell, and the paint in its children, its grandchildren, and so on. If a cell grows or shrinks, it may be necessary to change the bounding boxes of its ancestors in the hierarchy. See Figure 7. If the bounding boxes of parents are not automatically updated, then the search pruning mechanism described above may decide not to search a cell even though one of its children contains information in the area of interest. In order to update parent bounding boxes, it is necessary to keep upward pointers in the database from each celldef to all the celluses that reference that celldef, and from each celluse to its parent celldef.

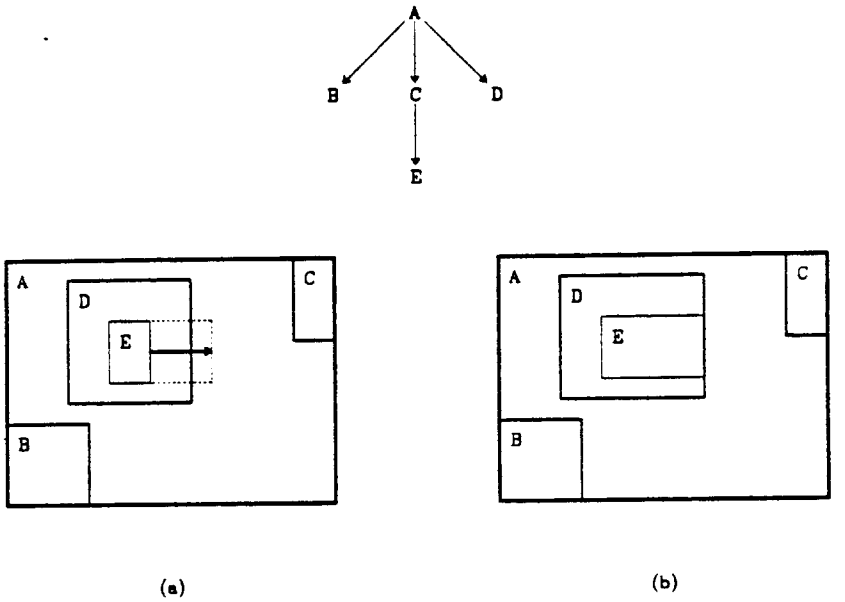

Figure 7. Caesar must maintain accurate bounding boxes for each cell. If cell E grows as shown in (a), the bounding box for its parent, cell D, must be modified as shown in (b). If the bounding box of D isn't updated, then the portion of E lying outside of D may be overlooked in searches.

The techniques for capitalizing on hierarchical structure have worked so well that there has been no need to use special geometrical data structures such as bins or quad-trees [6]. For example, even though the **paint** operation requires three complete searches of the rectangle list for one of the mask layers, the list usually has only a few tens of elements so the operation is instantaneous. The largest known Caesar cell has lists containing about 3000 rectangles each. But because of the simple data structures, the paint and erase algorithms have low overhead and are efficient even in these extreme cases. For example, less than a second of CPU time is required to paint a new rectangle into a cell with 3000-element lists.

5. Edit in Place

When assembling the subcells of a large circuit, subcells often have to be modified in order to mesh properly with their neighbors. In most layout editors, each subcell has to be modified in isolation. In Caesar, the subcell may be edited *in place*: the designer modifies the cell while viewing it in its final position in the overall design.

Without the ability to edit a cell in place, it is very hard to assemble large circuits. As a result, designers tend to lump many unrelated pieces of information into a single large cell. The first version of Caesar did not permit editing in place, and as a result all of the global wiring was placed in a single large cell with 3000 paint rectangles in each list. In subsequent designs, edit in place has made it possible for designers to divide the routing up into several smaller cells.

When editing in place, the edit cell may be a subcell in the middle of the hierarchy being displayed rather than the root cell of the hierarchy. It may be rotated or mirrored so that its internal coordinate system is not the same as the world coordinate system in which the user is viewing the circuit. Caesar has to take the user's commands, which are specified in the coordinate system of the root cell of the hierarchy, and transform them into the native coordinate system of the cell being modified. This in lurn requires that Caesar be able to compute inverses of cell transformations. Fortunately, since everything in Caesar is Manhattan, there are exact inverses for all transformations. If arbitrary feature angles were allowed, it would not be possible to compute exact inverse transformations, and edit in place would be much more difficult to implement.

6. Color and Texture

One of the interesting problems in implementing a CAD system is deciding how to render the design on a color display. The most important goal in rendering is to display the design's structure clearly. In VLSI, this is difficult because there are many mask layers that overlap; where there are complex overlaps, it is often hard to distinguish the different layers. An additional goal is to be able to redisplay the design quickly; ideally, it should be possible to redraw any one layer without having to redraw all the other layers. Finally, it is ideal if the above two goals can be achieved with inexpensive display hardware. Caesar uses a combination of three different techniques, described in the paragraphs below, to meet the first goal and most of the second with inexpensive graphics equipment.

6.1. Display Hardware

Caesar can be used with any of a number of color displays that store eight bits of color information for each pixel on the screen. When the screen is refreshed, the eight bits from a pixel are used to select one of 256 24-bit colors stored in a writable color map. The color map value determines the actual red, green, and blue intensities displayed. This means that there may be no more than 256 distinct colors on the screen, but there is considerable latitude in choosing the exact intensities of each of these colors. The color displays provide operations such as filling rectangular areas of pixel memory with particular values, drawing lines and text, and changing the color map values.

6.2. Transparent Layers

One way to use the pixel memory is to dedicate one of the bits of each pixel for each mask layer, with the bit indicating the presence or absence of the mask layer at that point. An area where a layer is present appears as a solid color. Every possible combination of layers is represented by a different pixel value, and hence can be displayed as a different color. I call this a *transparent* scheme, since the presence of one layer does not hide other layers from view. For example, a red color can be displayed when a pixel has only the bit for the R layer set, a blue color when the pixel has only the bit for the B layer set, and a purple color (as if there were a transparent blue foil on top of a red foil) if both bits are set. In dealing with complex layouts, the transparency property provides a substantial advantage in visual power.

Transparent layers have the advantage of automatic *color-mixing*. Most color displays allow the bits of pixels to be modified individually. For example, it is possible to specify a command of the form "set bit 4 of every pixel in the area (x1, y1) to (x2, y2) to 1, but leave all other bits of the pixels unchanged." Using this feature, rectangles on different layers may be drawn independently, without any concern for overlaps. Where overlaps occur, the overlap color is automatically selected by the combination of bits in the pixel. In the red-blue example, the blue layer can be drawn in an area without affecting the red layer bits in each pixel. If any of the pixels initially have the red bit set, they will end up with both the red and blue bits set, which will cause the purple color to be displayed.

6.3. Opaque Layers

Unfortunately, with only eight bits per pixel the transparent scheme can only accomodate eight mask layers. An alternative is to use all the bits of the pixel to represent a single mask layer. In this scheme, which I call *opaque*, eight bits can represent up to 255 different mask layers. However, the whole pixel can only represent a single mask layer, so if more than one layer is present at a point, a decision must be made about which one to draw. There are no special colors for overlaps. Typically, the layers are prioritized, with each layer given preference over lower priority layers. This is accomplished by drawing the layers in increasing order of priority: higher-priority layers overwrite lower-priority ones. The visual effect is one of opaque pieces of paper placed on top of each other. Although the opaque scheme can handle complex processes, it makes it very difficult to view complex structures, since important features may be hidden from view by higher-priority layers.

It is possible to display a few key overlaps in the opaque scheme by using some of the available layers. For example, three of the 255 layers could be used for polysilicon and diffusion: one for polysilicon, one for diffusion, and one for polysilicon-diffusion overlap. Unfortunately, no automatic color-mixing occurs as with the transparent scheme: the display routines must compute all the overlaps and display them with a different layer number. This inter-layer registration requires more complex data structures and algorithms than are present in Caesar.

6.4. Stippling

A third technique is to stipple the areas corresponding to each mask layer. Where mask layers overlap, their stipple patterns blend together. The patterns are chosen to blend in harmonious ways that make the overlaps

clear. Although stippling was originally developed for black-and-white systems, it is now coming into use in color systems as well. In an opaque display scheme, if a high-priority layer is stippled then it is possible to see lower-priority layers through the holes in the stipple. Stipples provide an important visual cue not present in solid filled areas, namely *texture*. However, they have three disadvantages. First, since stipples only color a few of the pixels in an area, it is more difficult to distinguish color in a stippled area than in a solid filled area. Second, if many stippled layers overlap, then the picture becomes so busy that it is difficult to distinguish features. Third, stippling is only effective if the features are large enough to contain a full repetition of the stipple pattern. Small features drawn with similar stipples may be indistinguishable.

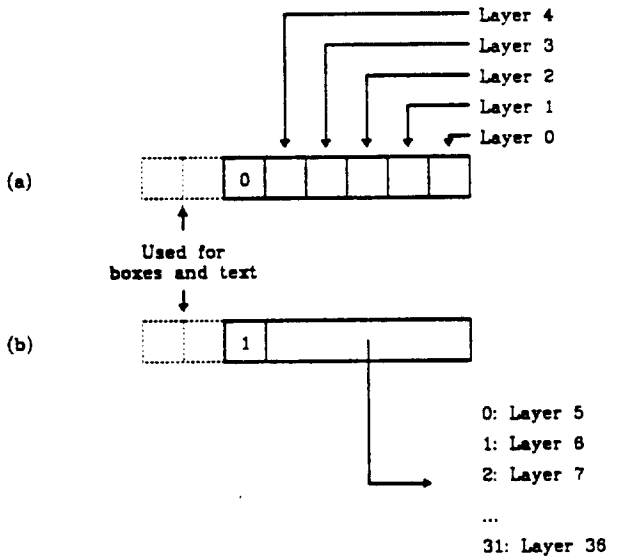

Figure 8. Caesar uses six bits per pixel to represent mask layers: (a) if bit 5 is 0, then the low-order five bits each represent one transparent layer; (b) if bit 5 is 1, then the low-order five bits together represent one opaque layer. The high-order two bits are used to display text, bounding boxes, and the grid.

6.5. Caesar's Solution

Caesar uses a combination of all three of the above schemes in order to maximize the visual power of the system. Only six of the eight bits per pixel are used to display mask information, and they are used as shown in Figure 8. If the high-order of these six bits is zero, then each of the low-order five bits indicates the presence or absence of a transparent mask layer. Each possible combination of these layers has a different color, with the colors chosen to present the appearance of transparent colored foils. Solid fill is usually used for the transparent layers. If, however, the high-order bit of a pixel is one, then the low-order five bits do not represent transparent layers. Instead, they select one of 32 opaque layers. When an opaque layer is present at a pixel, it is not possible to see transparent layers at that pixel. Opaque layers are usually drawn in stippled fashion so that they don't completely block out the transparent layers underneath.

The decision about which layers are transparent, which are opaque, and when to use stippling is made by the system maintainer. The most common layers are made transparent. Less-common layers, or those where transparency is less important (e.g. overglass), are made opaque. The information is stored in a file for each such configuration, and users can choose any of the available configurations.

The combined scheme has worked out well. It allows a total of up to 37 mask layers, of which 5 may be transparent. When stippling first became available on our displays, we experimented with using stipples for all mask layers, but found this to be unsatisfactory for the reasons mentioned in Section 6.4. On the other hand, solid filled transparent layers also become hard to distinguish when more than three or four layers overlap, so there seems

little value in having additional transparent layers. The combination of color (from the solid filled transparent layers) and texture (from the stippled opaque layers) makes it possible to distinguish more layers than either technique used alone.

The combined scheme is less efficient for redisplay than the one with only transparent layers. In a pure transparent scheme, a single layer may be redisplayed independently of any other layer. In Caesar's scheme, a single transparent layer may be redisplayed independent of any other transparent layer, but all opaque layers must be redisplayed whenever any transparent layer is modified. If an opaque layer is painted, all higher-priority opaque layers must be redisplayed, but not lower-priority opaque layers or transparent layers. If an opaque layer is erased, all layers must be redisplayed since the erasure may have exposed information that was previously obscured. Caesar's scheme is always more efficient than one with only opaque layers.

7. Evaluation

Caesar's simple data structures and algorithms, combined with a few techniques for taking advantage of hierarchy, have resulted in a very efficient program. Averaged over typical work sessions, Caesar rarely uses more than 5% of the total CPU time of a VAX-11/780 [9], or about twice as much CPU time as typical screen-oriented text editors [4]. On moderately loaded machines (Unix load factor of four or five), response is instantaneous to all commands, even when editing circuits with 50000 or more transistors. The elapsed time to load up a circuit and begin editing ranges from a second or two for small cells up to about a minute for the largest circuits. On very

heavily loaded machines, Caesar's performance degrades in about the same fashion as text editors; the program gives tolerable response up to a load factor of about twenty.

Caesar is also efficient in space. Typical editing sessions for small cells require about 150 kbytes of virtual address space, which is comparable to the space requirements of text editors. For large designs, Caesar normally requires 200-400 kbytes of virtual address space; because of the incremental database loading, this figure does not vary much with the size of the circuit being designed. The only time when the whole circuit must be memory-resident is when CIF files are created for mask generation and circuit extraction; when this happens, the virtual address space requirements will get as high as one or two megabytes. Caesar works well in paged systems: even when its address space becomes large, it rarely needs more than a few hundred kbytes to reside in main memory.

The painting paradigm has worked out quite well. It provides a simple and natural model for users to deal with, and makes the system predictable and easy to learn. The painting command set contains only five commands yet provides substantial power. By hiding the internal representation from the user, it was possible to use a simple internal data structure (lists of rectangles); this resulted in simple and fast algorithms. If more complex objects had to be represented, such as wires or polygons, the clipping and redisplay algorithms would have been slower and more complex.

The combination of transparent and opaque layers has also worked out well. Although there are a few situations where Caesar's scheme requires more layers to be redisplayed than a transparent-layers-only scheme, the difference has not been noticeable to users. The only time when redisplay

time is significant is when redrawing the whole screen to view a different piece of the circuit; in this case, any displaying scheme will require all layers to be redrawn, so they all take the same amount of time. Caesar's scheme permits more complex IC processes than a transparent-layers-only scheme, and provides greater visual power and clarity than an opaque-layers-only scheme.

Caesar does, of course, have shortcomings. The most serious of these is its lack of support for routing. Designers at Berkeley consistently have found routing to be the most laborious and least enjoyable part of circuit layout. Caesar provides no particular assistance aside from its standard painting operations. The available commands work well for creating leaf cells and joining them together into cell blocks. However, the global wiring process is difficult because each wire must be painted individually. Even a simple river router or maze router would make a substantial difference if embedded in a comfortable interactive environment.

Caesar would also benefit from a few enhancements to its user interface. Currently, it only displays a single view of the circuit at one time. Multiple windows are provided by several other systems such as Icarus [1] and KIC [7] and seem to be very useful, especially for global operations such as routing. Even a simple windowing scheme such as a split screen would be very helpful.

Although a hierarchical collection of cells is displayed on the screen, at any given time the user is editing just one of those cells. In Caesar, the paint in the edit cell is drawn in the same way as paint in other cells, so it is hard to tell where the edit cell ends and other cells begin. This results in accidents where users paint or erase in one cell when they really should have painted or erased in a different cell. The edit cell should be displayed

differently than other cells (perhaps with greater intensity) in order to make clear what is editable and what isn't. The need for this feature did not become obvious until after editing in.place was implemented, and appears to be hard to retrofit into the system.

For some industrial applications such as ROMs and RAMs, where density is critical, the Manhattan nature of Caesar may be intolerable. Caesar can be generalized to handle 45-degree angles by using maximal trapezoids instead of rectangular strips, and by permitting the box to turn into a diamond shape. Work is already underway to implement such a scheme in one of the commercial products based on Caesar. However, for most applications I believe that it is preferable to stay within a Manhattan framework, since this tends to result in fewer errors, faster CAD programs, and ultimately shorter design times.

8. Conclusions

Two overall lessons emerge from the Caesar experience. The first lesson is that simplicity and efficiency go hand-in-hand. Simple algorithms and data structures have low overhead. As a result they are often faster in practice than algorithms that have good theoretical behavior but require complex data structures and algorithms. Typical cells in VLSI circuits are small enough that the constant factors in algorithms tend to dominate the algorithmic factors.

The second lesson is that user interfaces need not necessarily match internal representations. The structure that is best for representing information inside a computer may not provide the best way for humans to think about the information. In Caesar, users think about mask patterns in terms

of paint, which is natural for them, while the information is represented internally in terms of rectangles, which is simple and efficient for Caesar. The rectangles have no logical significance for the design, so their presence is hidden from the user. On the other hand, the cell hierarchy is an example of a structure that is useful both to user and system. For the user, the hierarchy provides a mechanism for partitioning large designs into manageable units. It serves a similar function for the system, providing a space efficient representation and a mechanism for pruning searches.

9. Acknowledgements

Gordon Hamachi, Bob Mayo, and Dave Patterson all provided helpful comments on drafts of this paper. The work was supported in part by the Defense Advanced Research Projects Agency, DARPA Order No. 3803, monitored by the Naval Electronic System Command under Contract No. N00039-81-K-0251.

10. References

[1] Fairbairn, D.G. and Rowson, J.H. "ICARUS: An Interactive Integrated Circuit Layout Program." *Proc. 15th Design Automation Conference*, 1978, pp. 188-192.

[2] Foderaro, J.K. Van Dyke, K.S., and Patterson, D.A. "Running RISCs." *VLSI Design*, Vol. III, No. 5, September/October 1982, pp. 27-32.

[3] Infante, B., *et al.* "An Interactive Graphics System for the Design of Integrated Circuits." *Proc. 15th Design Automation Conference*, 1978, pp. 182-187.

[4] Joy, W.N.. Private communication.

[5] Katevenis, M., Sherburne, R. Patterson, D., and Sequin, C.S. "The RISC II Micro-Architecture." to be presented at *VLSI 83*, Trondheim, Norway, Aug. 83.

[6] Kedem, G. "The Quad-CIF Tree: A Data Structure for Hierarchical On-Line Algorithms." *Proc. 19th Design Automation Conference*, 1982, pp. 352-357.

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

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

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

[10] Patterson, D.A. and Sequin, C.H. "RISC I: A Reduced Instruction Set VLSI Computer." *Proc. Eighth International Symposium on Computer Architecture*, May 1981, pp. 443-457.

[11] Patterson, D.A., et al. "Architecture of a VLSI Instruction Cache." Internal Memo, Univ. of California, Berkeley, August 1982.