Exploiting Structure in the Analysis of Integrated Circuit Artwork

Daniel T. Fitzpatrick

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

ABSTRACT

Design of very large scale integrated circuits requires structured design approaches to deal with the ever increasing complexity of these circuits. As designs become more complex, the need for layout analysis tools, such as design rule checkers and circuit verifiers, becomes greater. However, current layout analysis tools make no use of this structure, and as a result, are taking increasingly large amounts of computer time. This thesis examines ways in which these analysis tools can exploit the structure present in the layouts of large integrated circuits in order to improve runtimes.

A major obstacle in building hierarchical analysis tools is in handling overlaps of cells. This thesis describes an algorithm that transforms a hierarchical layout description into a hierarchical layout description without overlaps. This newly created hierarchy is called the disjoint hierarchy. Analysis tools that rely on this disjoint hierarchy can effectively exploit the structure of the layout. The design of hierarchical algorithms to do circuit extraction, geometric design rule checking, and simple mask operations are also described. A program that implements the disjoint transformation and performs circuit extraction is discussed. A comparison of the runtimes of this program with conventional circuit extractors shows a considerable speedup for even relatively small circuits.

July 7, 1983

Table of Contents

Chapter I. Introduction

Over the past few years integrated circuit technology has packed an exponentially increasing number of devices onto a single chip of silicon. This trend is likely to continue over the next decade. Already chips containing nearly half a million transistors have been successfully fabricated[5]. Chips may soon contain in excess of a million transistors. To cope with the complexity of placing this many transistors on a chip, designers have been relying on structured design methodologies. Large complex designs are broken down into their major functional sections, and these sections are in turn broken down into a number of components. Thus at each level there is a manageable number of pieces to be considered by the designer. The details of the other parts are hidden by suitable abstractions.

In addition to breaking each part of the chip into subparts, designers rely on re-using previously designed cells to minimize design time. A memory circuit, although it contains many tens of thousands of devices, is made up of only a small number of different cells of a few transistors each. These cells, however, are repeated many times to build up the circuit.

As the number of transistors increases, so does the possibility of making a mistake in the layout. Several computer analysis tools are available to help catch these mistakes before submitting a chip for fabrication. These tools include *Geometrical Design Rule Checking*, which establishes that certain geometric constraints are satisfied to insure correct fabrication, and *Circuit Extraction*, which extracts the underlying circuit from the layout description. The extracted circuit can be compared to other circuit descriptions derived from the original specification. The data from the circuit extraction can also be used to drive a *Static Electrical Rules Checker*, which checks that certain electrical rules are satisfied, and to perform *Dynamic Simulation*, which predicts the behavior of the circuit under various situations. Most designers consider these tools essential to create working circuits.

Figure I-1a shows a plot of the RISC chip. This chip contains almost 45,000 transistors. Although details of the chip are not visible at this scale, one can readily see that the chip is highly regular. The designers use regularity to minimize the design time and complexity of the chip. Figure I-1b shows this same chip with every different type of cell drawn exactly once. The plot is mostly empty space. Thus, the designers have to draw only a few individual devices that are repeated many times. The biggest chore is the wiring between cells that are not placed in a regular array.

Computer analysis of this chip took eight hours of CPU time on a VAX 11/780 computer. Analysis included layout rule checking, circuit extraction, and static electrical rule checking. To understand why analysis took so much computer time we need to understand how these analysis tools work.

When checking a circuit, traditional computer tools make no use of the fact that it is an array of identical cells. The layout is viewed as a hierarchically flat arrangement of rectangles. Instead of checking only a few key cells, the computer checks each feature in the array. Thus, in effect, the same cell is checked over and over again. Although the computer can check a cell much faster than the human designer, the human designer can exploit knowledge about the structure of the layout to make a more economical choice about the checks that need to be performed.

Let us now consider the way a human designer would check an array of cells, like that shown in figure I-2. A human designer would pick an interior cell and check it for design rule violations, and make sure that it matched the intended circuit. He would then check how the cell interacts with its neighbors, whether any new design rule violations occur and that it connects in the proper way to the next cell. The designer must also check cells on the boundary of the array to see if the boundary cells introduce any errors. Because of the regularity of the array of cells the designer can be sure that those few specific checks are sufficient.

Traditional layout analysis tools make no use of this structure. Instead they immediately flatten the hierarchical description, thus throwing away any knowledge of the structure. This method of layout analysis will become impractical when chips containing over a million transistors will need to be checked. Checking such large designs would take days of computer time. Clearly, future analysis tools must exploit the hierarchy and repetition present in the layout description.

The goal of this thesis is to investigate ways that computer analysis tools can also exploit this knowledge of structure. By explicitly recognizing the hierarchical structure of the layout, the performance of the layout analysis programs can be dramatically increased. In the following chapter, the Disjoint Algorithm is introduced. This algorithm breaks a layout into a set of non-overlapping hierarchical cells. Later chapters show how the Disjoint Algorithm aids in carrying out circuit extraction, design rule checking, and mask operations in hierarchical manner. In chapter VI and VII the implementation and performance of the Disjoint Algorithm and a specific hierarchical circuit extractor is described. In chapter VIII other work that has been going on in this area is reviewed. The remainder of this chapter reviews some of the concepts and definitions used throughout this thesis.

The layout description of a circuit is usually available in a *Hierarchical Description Language*(HDL), with all the repetition and structure explicitly represented. A HDL describes the layout in terms of *symbols*. A symbol contains *geometry* and *instances*. Geometry is specified by its layer and vertices. An instance is a reference to a symbol definition with a given offset, rotation, and mirroring. The offset, rotation, and mirroring can be represented by a 3 by 3 matrix, called a *transformation*[14]. Thus, an instance can be represented by the ordered pair **(S,T)**, where **S** represents the instance's symbol, and **T** represents the instance's transformation.

A symbol may be referred to many times through instances contained in other symbols. This makes it possible to create large regular arrays of circuit elements with very small descriptions. Figure I-2 shows a memory array with over 700 transistors. This memory array was described in CIF, the Caltech Intermediate Form[7], a HDL. The CIF description is only 124 lines long because the hierarchical structure of the layout was suitably exploited.

Figure I-3a shows in graph form the symbols and instances of the layout shown in figure I-2. Symbols are represented by circles. Instances are represented by edges. The edges leave the circle that represents the symbol that contains the instance, and point to the circle that represents the referenced symbol. Figure I-3b shows a graph of how the circuit would look if no two instances referred to the same symbol. There is a considerable increase in the complexity of this graph. This is analogous to what happens in layout analysis tools that ignore the hierarchical structure of a layout. Rather than deal with the layout in the compact hierarchical form, the hierarchy is flattened and the amount of information the analysis tool must deal with explodes.

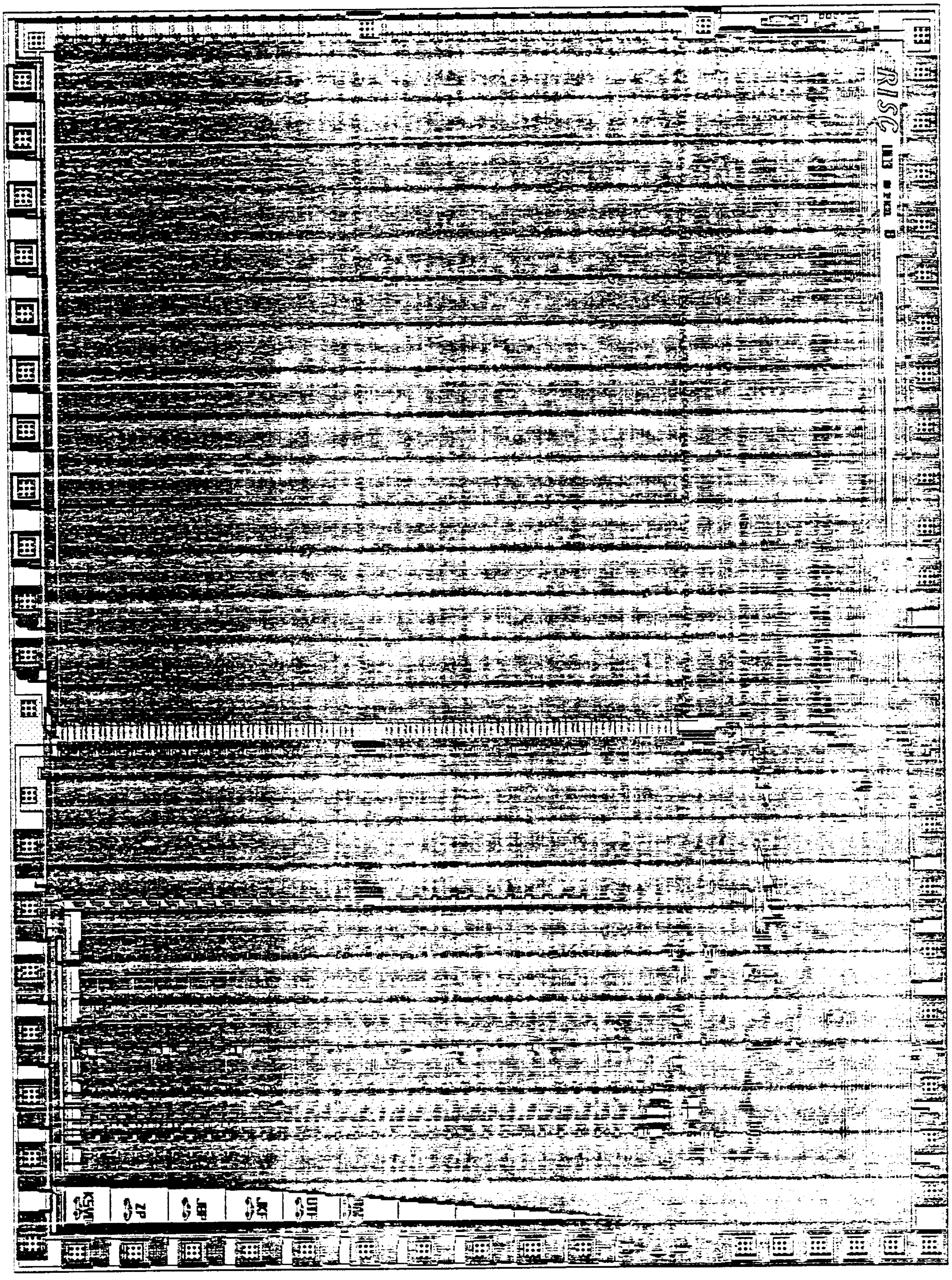

Figure I-1a. *fully instantiated RISC layout*

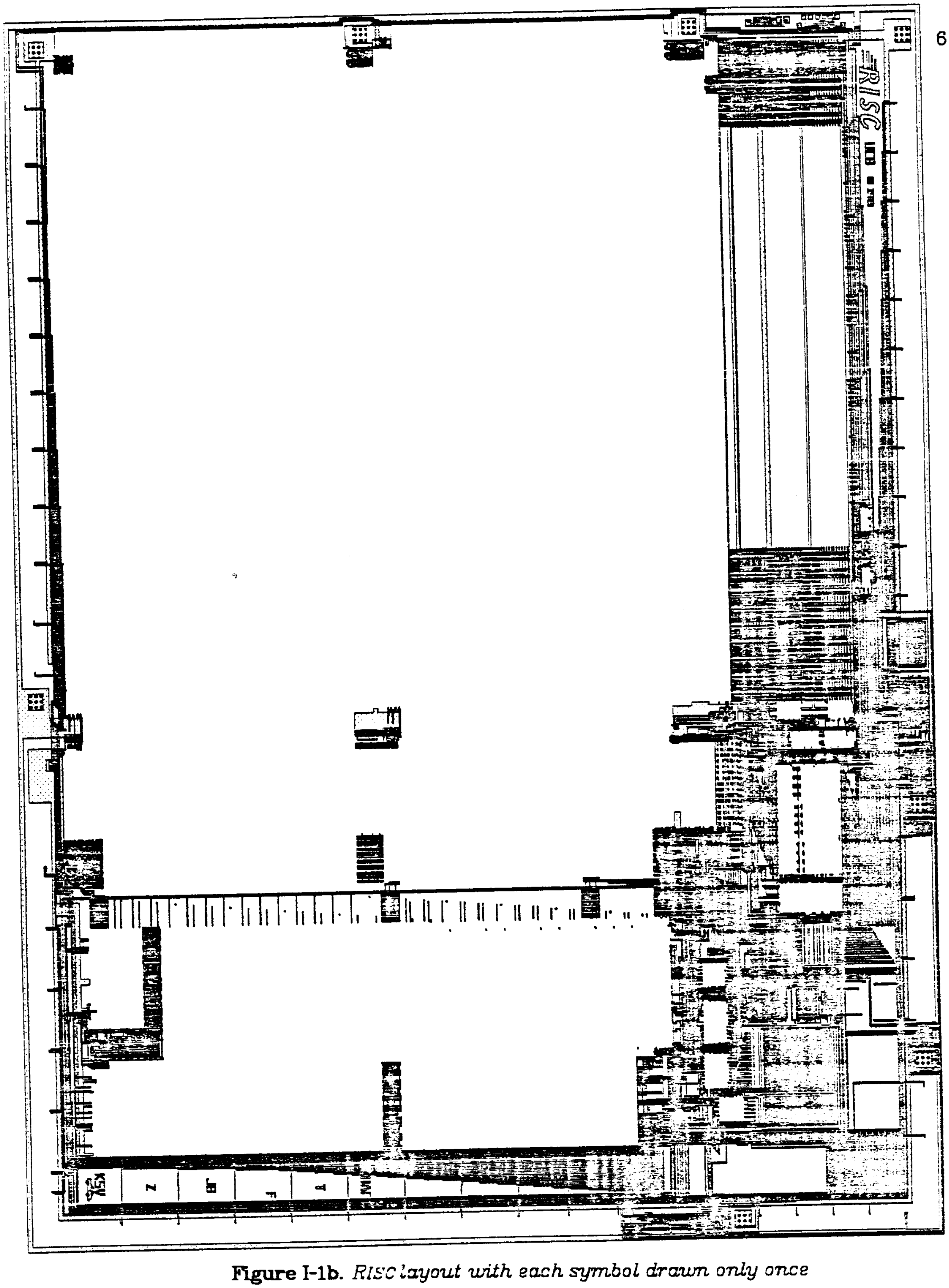
Figure I-1b. *RISC layout with each symbol drawn only once*

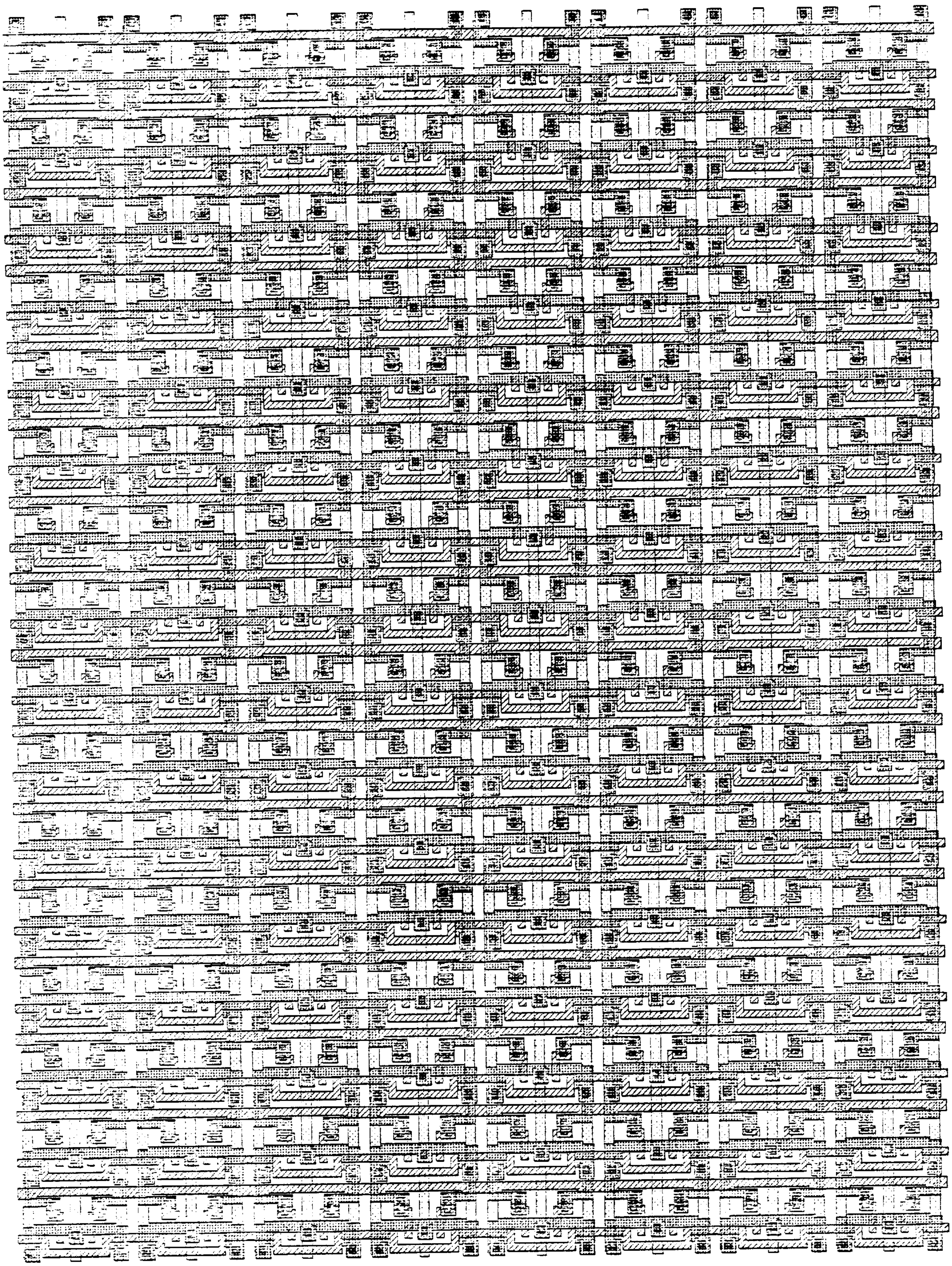

Figure I-2. *memory array*

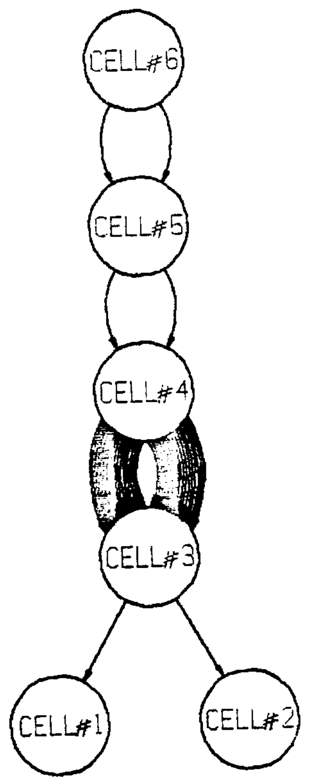

Figure I-3a. *hierarchical graph of memory array*

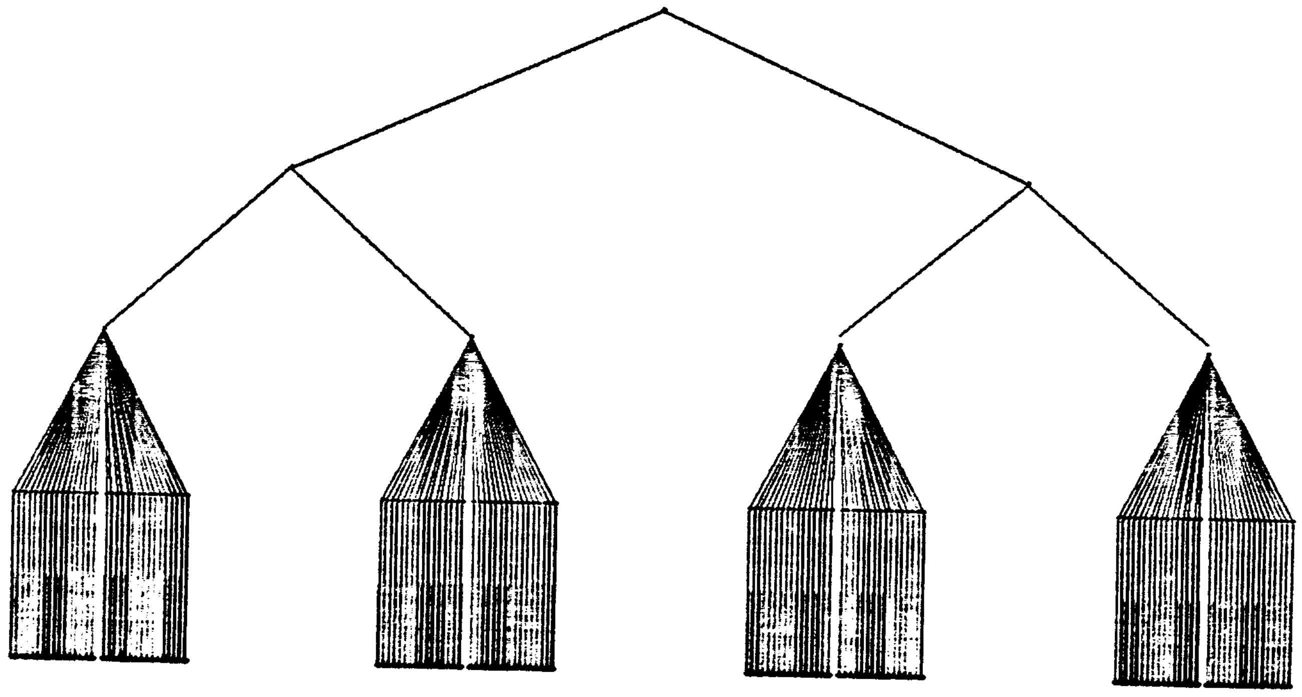

Figure I-3b. *fully instantiated graph of memory array*

Chapter II. The Disjoint Transformation

The main problem in exploiting the hierarchical structure of layouts is dealing with overlapping cells. Overlaps can invalidate any previous analysis of a cell. They can create design rule violations in a cell, or remove apparent design rule violations. In the context of circuit extraction, overlaps can radically change the circuitry of a cell. Overlapping geometry can short together two previously unconnected nodes, or it can disconnect two apparently connected nodes. Overlaps can create or even remove transistors in a cell. Figure II-1 shows some particularly drastic examples of the effect of overlaps.

On the other hand, overlaps are very useful to a circuit designer. Figure II-2 shows a way in which designers commonly use overlaps. The memory cell, even when viewed in isolation, has its full context. When these cells are combined to build a memory array, the designer overlaps the power and ground busses of adjacent cells. If the designer were prevented from overlapping these cells, he would have to either increase the pitch of the memory array, or design the basic memory cell without the full power and ground bus. Increasing the pitch of the memory cell is not desirable, since it may have considerable impact on the overall size of the circuit. Designing the memory cell without the full power and ground busses is also undesirable, because the designer would see an incomplete circuit when he examines the memory cell. The full cell could only be seen in the context of the surrounding circuitry. The designer must remember to complete the busses at the periphery and take special care to avoid design rule violations in cells on the edge of the array. Such incomplete cells may have apparent design rule violations, which only disappear when they are placed into the full circuit. Thus, disallowing overlaps either increases the area of the circuit or places a larger burden on the circuit designer, increasing the design time and the chance for errors.

Rather than restricting the designer to designs in which cells do not overlap, the approach taken here is to transform the hierarchical layout description, which may contain overlaps, into a hierarchical layout description without overlaps. There are several reasons for this approach. The main reason is that we do not want the layout analysis tools to restrict design styles in an inappropriate manner. Since, as outlined above, it is useful to overlap cells, tools need to support this design style. Even if there were strong reason to discourage overlap, it is desirable to have tools that still function when recommended practice has not been strictly adhered to. Finally, there are many existing designs that have overlapping cells. Tools should continue to be useful for these designs.

The transformation that takes a hierarchical layout description, with potential overlaps, and transforms it into a hierarchical layout description free from overlaps is called the *disjoint transformation*. The disjoint transformation works by partitioning each symbol into regions where instances overlap, and creating new symbols corresponding to those regions. Repeated occurrences of identical regions are recognized and new symbols are created only for regions that have not been seen before. In this way much of the original structure of the layout is kept intact. Once overlaps have been removed from the layout, the job of analyzing the circuit is made considerably simpler.

1. Example

Before going into details of how the algorithm works, it may be helpful to go over an example. Figure II-3 shows a symbol, A, made up of one instance each of symbols B, C, D, E, and F. Consider a layout consisting of four instances of A juxtaposed in the form of a regular array as in figure II-4a. The function of the disjoint transformation is to remove overlapping cell instances. This is done by generating new cells from combinations of given instances where they overlap. The partitioning uses the edges of the boundaries of the given instances, so that every edge of a derived cell is coincident with an edge of a given instance. The result of this partitioning of figure II-4a is shown exploded in figure II-4b. Four new cells have been generated: P, Q, R, and S, of which P and S appear once each, Q appears three times, and R appears twice.

It is now necessary to consider the contents of each cell P, Q, R, and S to fill out the next level of the derived disjoint hierarchy. The partitioning algorithm is now applied to each of P, Q, R, and S in turn. The contents of the derived cells are shown in figure II-5. The cells B', B'', C', C'', F', and F'' are parts of the cells B, C, and F contained in the original cell A. This process continues to recurse until the cells containing only geometry are met.

Of course, this example is atypical, not just in the small number of cells involved, but also in the fact that only one type of cell is involved at the top level, and that overlaps involve only two instances. A realistic algorithm has to generalize on both of these issues.

The disjoint transformation is implemented as two procedures, *Split*, and *Gather*, that call each other recursively. In the above example, Split generates the structure shown in figure II-4b, except that the repetition of R and Q is not recognized. Gather recognizes repeated occurrences and creates the cells P, Q, R, and S.

2. Split

The Split procedure takes as input a *symbol*. A symbol here is a generalization of the normally accepted use of the term. In common with conventional usage, a symbol is made up of geometry and instances of other symbols. However, also associated with the symbol is a *window*. The window is a rectagonal region (all edges parallel to the x- or y-axes) outside of which any component instances or parts of instances are to be ignored. This extended definition is convenient for the intermediate structures generated during the Split procedure. Each input cell is initially converted to such a symbol by defining its window to be its minimum bounding box.

The function of the Split procedure is to partition the parts of the symbol's geometry and instances that lie inside the window into a minimum number of disjoint rectagons regions, called *discells*. Discells are defined to be regions uniformly covered by the same combination of instances. All geometry of the cell is partitioned at the boundaries of the discells. Therefore, each discell is associated with parts of instances that overlap and with fragments of geometry of the given cell that overlap the discell. The union of all these pieces covering such a discell region is made into a new symbol.

The Split procedure uses a sweeping-line algorithm in which the set of instance windows is cut into horizontal *swaths* whose upper and lower limits coincide with vertices of the instance windows. Each swath is scanned left to right keeping track of the set of instances between each consecutive pair of vertical window edges. Each set corresponds to a partial discell to be used in subsequent processing by Gather. Discells are kept in a hash coded dictionary to make it possible to determine quickly whether a given discell has been seen before. The hash is a function of the component instances' addresses in memory. If a discell has been previously seen, the entry in the dictionary is updated to extend the window of that discell. The geometry of the given symbol is clipped to the individual discells generated by the above procedure.

For example, consider a swath, S3, from the simple example shown again in figure II-6. The individual instances of symbol A have been named A1, A2, A3, and A4. While scanning swath S3 the following potential discells will be found: {A1}, {A1,A2}, {A2}, {A2,A3}, {A3}, where {} denotes the set of instances making up a discell. Of these, the discells {A1}, {A1,A2}, and {A2} will already be in the discell dictionary from the analysis of the previous swaths below S3, and so their entries will have their windows extended by the appropriate regions of the swath S3. Discells {A2,A3}, and {A3} will make new entries in the dictionary. On completion of scanning all the swaths of the given symbol, Split transfers the discells collected in the dictionary to Gather.

3. Gather

The function of the Gather procedure is to recognize sets of discells that consist of similar juxtapositions of instances and windows, and to convert them to regular symbols. In forming a new symbol from a discell, Gather replaces the set of component instances with their contents, as shown in figure II-5 of the simple example. Gather next creates instances of each new symbol to replace the discells converted to that new symbol. The total set of instances so created replaces the contents of the symbol originally passed to Split and from which the discells were created.

The recognition of similar discells is achieved using another hash-coded dictionary, except this time the hash is a function of the set of component symbols as well as the relative transformation of those symbols. In figure II-6, Gather will recognize that discell {A1,A2}, {A2,A3}, and {A3,A4} all consist of similar juxtapositions of instances of A. The new symbol, Q in figure II-4a, will therefore be created, and will consist of instances making up component instances of the replaced discells that overlap the discell's window, as in figure II-5. Each discell is replaced by an instance of the symbol Q, shown in figure II-4b. Similarly, discells {A2} and {A3} are replaced by instances of symbol R. The set of instances of P, Q, R, and S now replace the original contents of the symbol that consisted of the four instances of symbol A.

To understand how similar juxtapositions are recognized it is important to remember that instances are represented by an ordered pair consisting of the referenced symbol and transformation. We now consider the discell itself to be an instance, where the transformation of the first element is the transformation of the entire instance. The inverse of this transformation is then applied to each element of the discell. Thus, for example, we can represent the discell {A1,A2} as $\{(A,T_1),(A,T_2)\}$. The discell is converted to an instance of the form $(\{(A,I),(A,T_2T_1^{-1})\},T_1)$. (Note: I is the identity transform.) Likewise, the discell {A2,A3} is converted into an instance of the form $(\{(A,I),(A,T_3T_2^{-1})\},T_2)$. The transform $T_2T_1^{-1}$ is the relative transform from A1 to A2, and the transform $T_3T_2^{-1}$ is the relative transform from A2 to A3. Thus, if $T_2T_1^{-1}$ equals $T_3T_2^{-1}$, it is recognized that {A1, A2} and {A2, A3} are instances referring the same symbol with different transformations. In addition to checking the symbols referenced, it is necessary to check that the windows of the discells are similar.

Finally, Gather calls Split for each newly created symbol. The mutual recursion between Split and Gather terminates when a created symbol is found to contain only geometry and no instances of other symbols.

4. Cleanup Phase

This method of breaking apart overlapped symbols usually creates a few new symbols which contain neither geometry nor instances, and several new symbols which contains only one instance or one geometric primitive. On the final phase of the disjoint algorithm, symbols that contain nothing, along with instances of them, are filtered out. Instances of symbols that contain only one instance or one geometric primitive are replaced by the referenced instance or geometric primitive appropriately transformed.

An optional pass will rearrange the hierarchy by identifying arrays and replacing them with extra levels of hierarchy. This pass helps speed up subsequent analysis procedures, but this pass is optional since the designer may not want to change the hierarchy.

5. Subsequent Analysis

Now that all overlaps have been removed, the task of the analysis procedures is greatly simplified. Analysis can proceed on each symbol without having to worry about overlapping regions changing its interior geometry. The boundary is the only area of the symbol that can interact with neighboring cells.

The following chapters discuss layout analysis with respect to circuit extraction, design rule checking, and mask operations. The broad outline of these algorithms is always the same: for any symbol, first the program looks to see if all the symbols instanced have been checked and recurses on the unchecked symbols, then it runs the analysis procedure on the current symbol's geometry and boundary areas of the instances.

6. The Sweeping Line Algorithm

Many of the algorithms presented in this thesis are derivations of a general sweeping line algorithm. The sweeping line (also called the scan line) algorithm is a method of completely traversing a layout from bottom to top in an orderly fashion. It is presented here only for the sake of completeness. It has been published and widely used by others for several years[13][15].

Let us consider using the sweeping line algorithm in order to produce a minimum set of horizontally oriented trapezoids from a number of possibly overlapping polygons. For the sake of simplicity let us assume that the polygons are not self-intersecting. (The self-intersecting case is covered in [13].) First we orient the edges in a counter-clockwise direction, marking those edges oriented downwards as starting edges, those edges oriented upwards as ending edges, and discarding horizontally oriented edges. We now sort the edges onto an edge list by their minimum-y vertex, sorting first by y then by x.

We take the minimum-y value of the first edge on the edge list and call this *yCurrent*. We take all the edges off the edge list whose minimum-y value equals *yCurrent* and put these edges onto a second list called the active list. We take care to see that the edges on the active list remain sorted in x.

It is now necessary to find the next *event point*. An event point is defined by one of the following three cases: a point at which an edge on the active list ends, a point at which two edges on the active list intersect above *yCurrent*, or a point at which an edge on the edge list starts. Since the edge list is sorted, if the next event point is on the edge list, it must be the start point of the first edge. If the next event point is the intersection of two edges on the active list, the two edges must be adjacent. Thus we can scan down the active list to find the minimum ending point of the edges, and the minimum intersection point where two adjacent edges intersect above *yCurrent*. We set *yNext* to be the minimum-y value of these three points.

Now we consider the swath defined from *yCurrent* to *yNext*. We run through the edges of the active list, maintaining a counter which we increment every time we see a starting edge, and decrement every time we see an ending edge. Whenever the counter makes a transition from 0 to 1 we have found the starting edge for a trapezoid from *yCurrent* to *yNext*. Whenever the counter makes a transition from 1 to 0 we have found the ending edge for that trapezoid. Once we have finished a swath we set *yCurrent* to *yNext*, take the edges off the edge list whose minimum-y value equals *yCurrent* and put them on the active list, sorting the list for the current value of *yCurrent*. Edges on the active list whose maximum-y value is less than *yCurrent* are removed from the active list. This whole process repeats itself until both the edge list and the active list are empty.

This yields a set of non-overlapping trapezoids which exactly covers the set of polygons. In order to minimize the number of trapezoids, we delay outputting a swath of trapezoids until we have finished processing the swath immediately above. We then look for any trapezoids on the lower swath which are continued on the upper swath. When we find such a trapezoid, we delete the trapezoid from the lower swath and add it to the trapezoid on the upper swath.

This algorithm can easily be extended to find the intersection of two layers by replacing the single counter across a swath with two counters. Whenever we cross a starting edge from layer A we increment counter A, when we cross an ending edge from layer A we decrement counter A. Likewise for layer B. Whenever the product of counter A and counter B makes the transition from 0 to 1 we have found the starting edge for a trapezoid, and when the product of counter A and counter B makes the transition from 1 to 0 we have found that trapezoid's ending edge. To find the union of the two layers, we use the sum of counter A and counter B rather than the product.

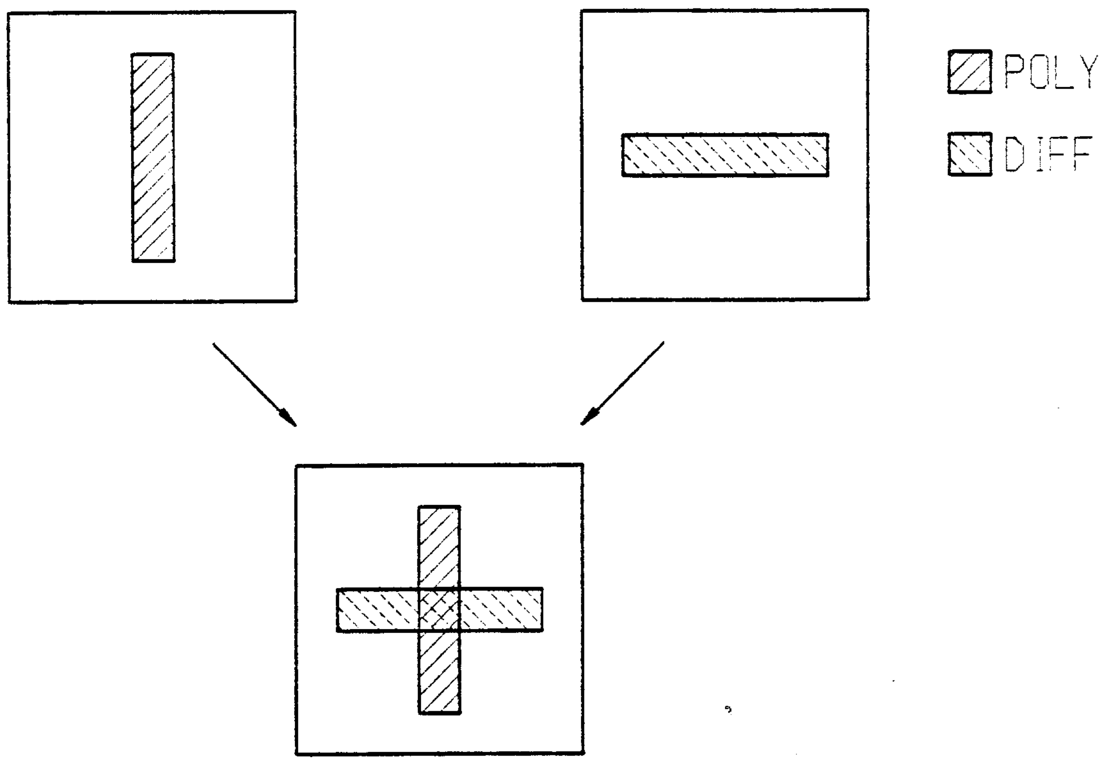

Figure II-1a. *overlaps can create transistors*

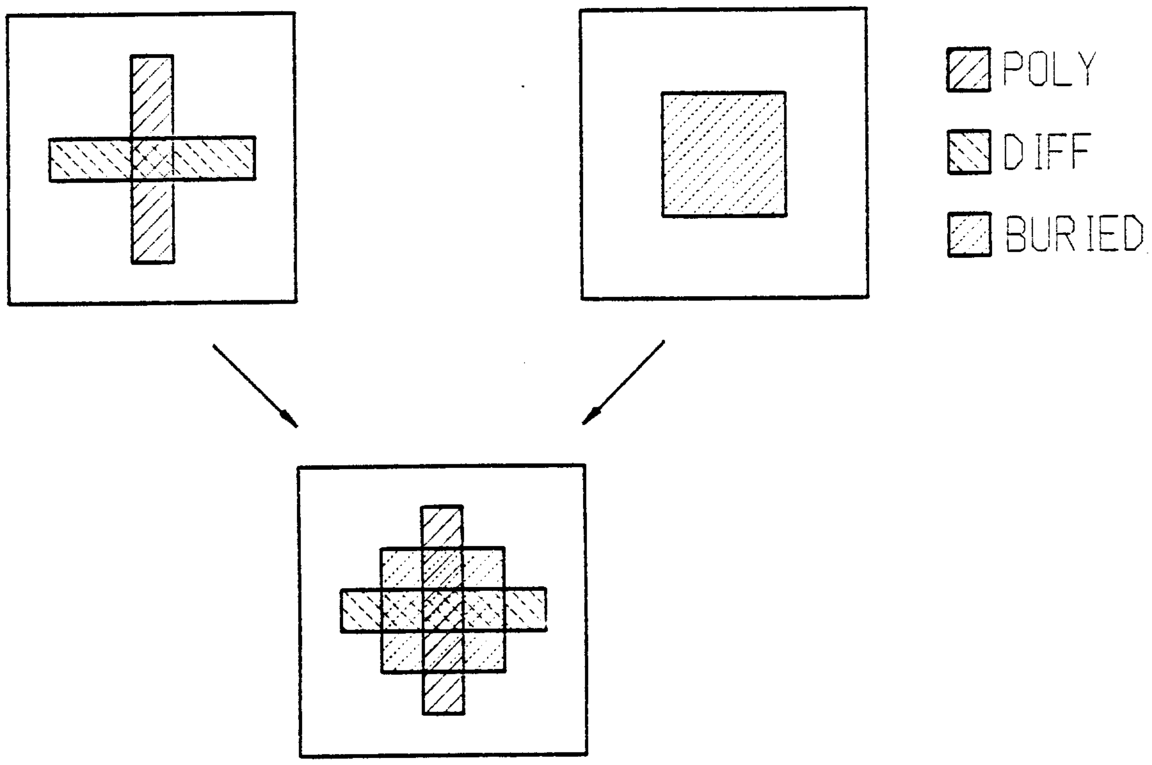

Figure II-1b. *overlaps can short out transistors*

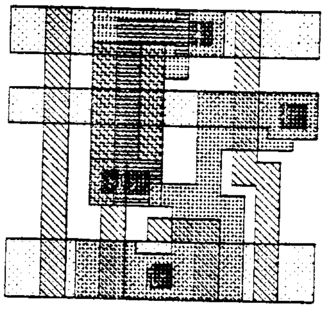

Figure II-2a. *memory cell*

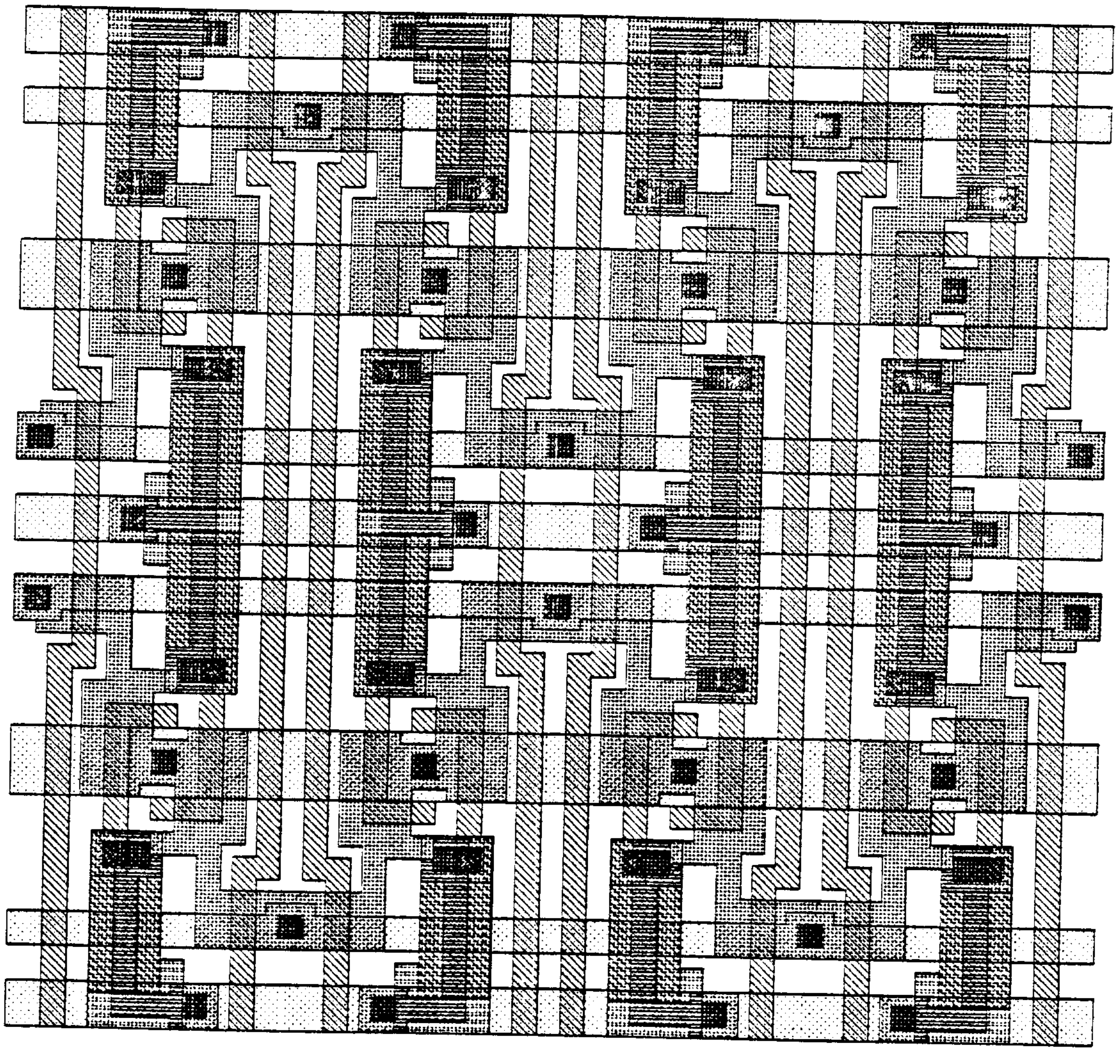

Figure II-2b. *4 by 4 array of overlapping memory cells*

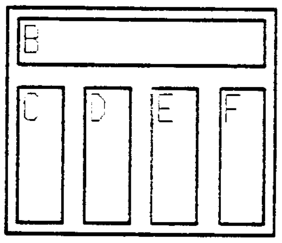

Figure II-3. *symbol A*

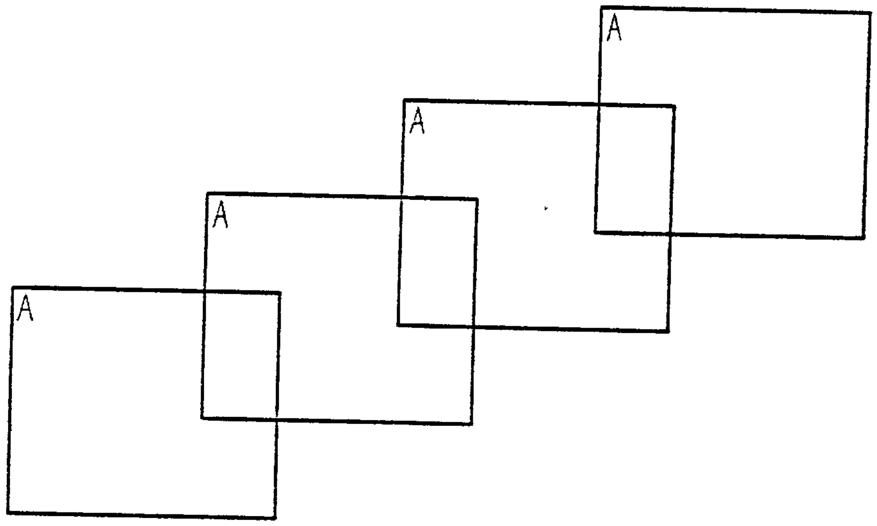

Figure II-4a. *before partitioning*

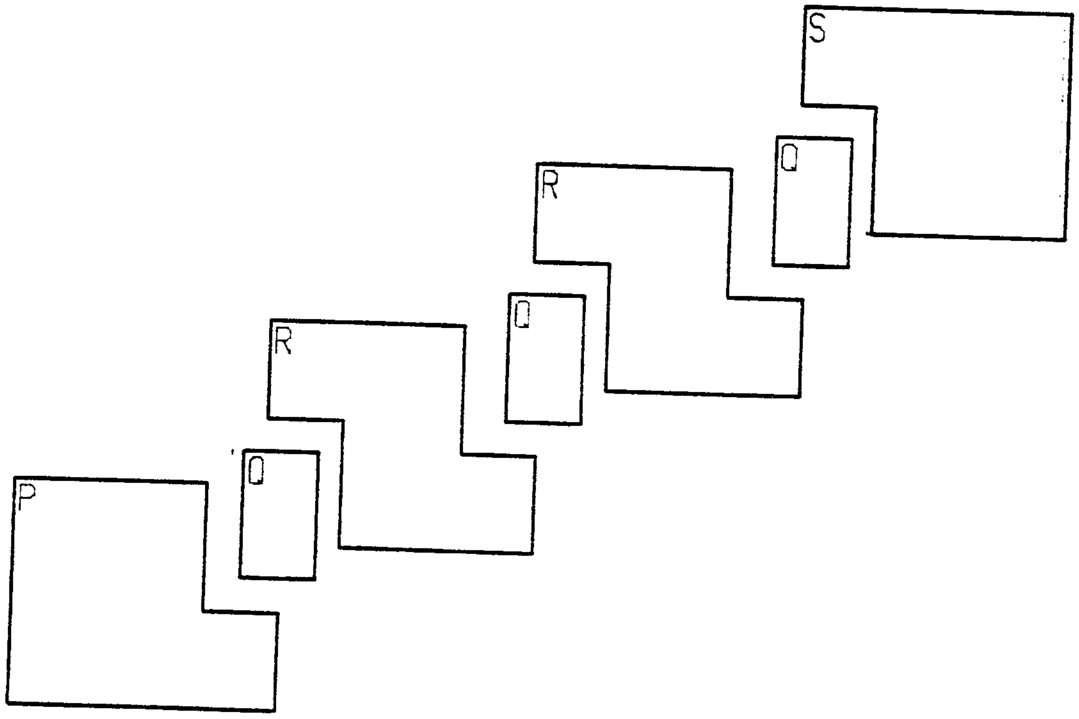

Figure II-4b. *after partitioning*

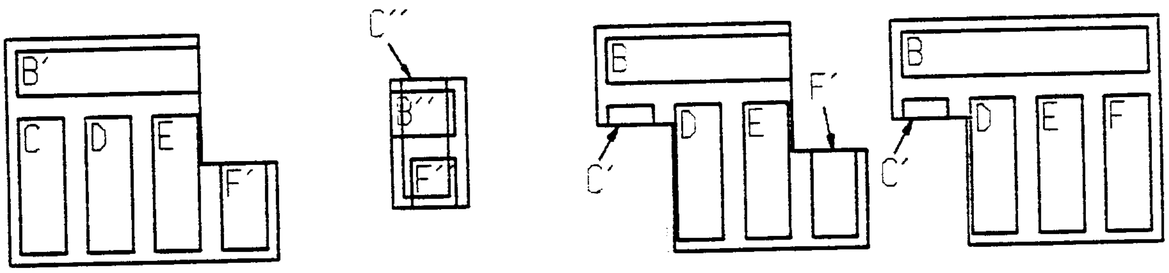

Figure II-5. *symbols P, Q, R, and S*

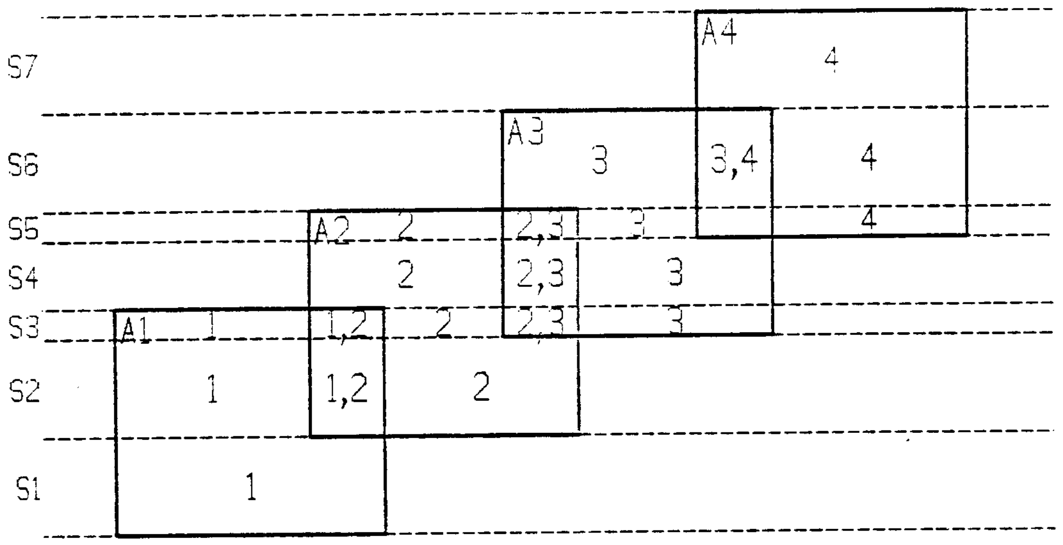

Figure II-6a. *swaths*

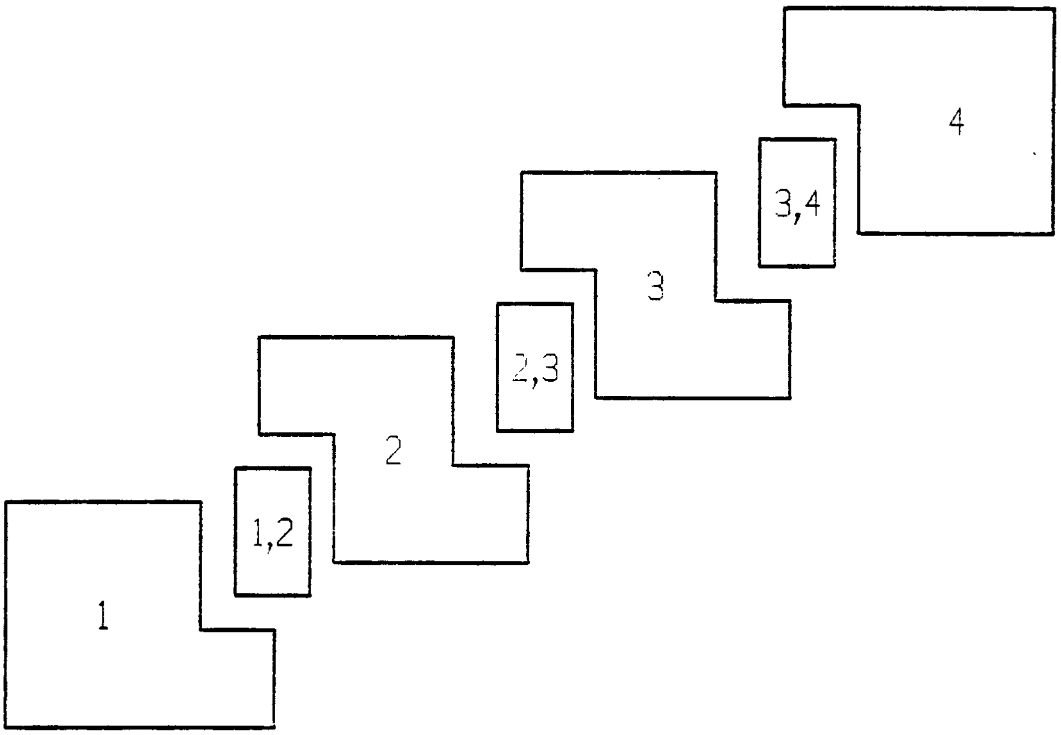

Figure II-6b. *generated discells*

Chapter III. Circuit Extraction

The task of extracting the circuit from a layout description removed is considerably easier once all overlaps are removed. The extraction of a symbol proceeds as follows. If the symbol contains any instances, all the corresponding symbols must be extracted. The extraction of these symbols yields, in addition to a circuit description for each symbol, a set of connection points for each symbol. Geometric analysis of the current symbol, similar to the analysis of a typical flat extractor, can be used to find connectivity between the symbol's geometry, circuit elements, and connection points. The peripheral connection points of the current symbol must also be found, so that later this symbol can be used as part of another symbol. The extractor traverses the calling hierarchy, extracting each symbol once, until it has extracted the entire circuit. This results in a hierarchical description of the circuit.

Since the symbols that are created by the disjoint transformation are cut up regardless of content, many transistors will straddle symbol boundaries. Thus the problem of partial transistors must be addressed by the circuit extractor even if the original layout description had no transistors crossing any symbol boundaries. Figure III-1 shows several possible configurations of a transistor gate area crossing a symbol boundary. The actual circuit may be a simple transistor as in figure III-1a, or it may be a capacitor as in figure III-1b, or it may even wrap back around on itself as in figure III-1c. From this symbol the extractor cannot tell what type of transistor will result, thus it waits until it processes a larger symbol that totally contains the transistor. With this method the extractor never makes a guess that later has to be fixed.

1. The Algorithm

The basic function of the extractor is to convert a symbol's layout description into a circuit description and a set of external interface segments. The circuit description is a list of transistors and their connections to *nodes* of the circuit. A node is any electrically conductive path not including a transistor. A node is specified by a number, which is called its *node number*.

External interface segments are those parts of the symbol boundary that are touched by geometry that forms conductive paths. They are used to specify how the circuitry of a symbol can connect to its neighboring symbols. An interface segment is specified by a line segment, a layer, and a node number. Each external interface segment is assigned a node number corresponding to the node that generated the segment. There are two types of nodes in the symbol: nodes that have external interface segments, called *interface nodes*, and those that do not, called *internal nodes*. This distinction is similar to that between parameters and local variables of procedures in programming languages.

During the extraction it is often found that two node numbers have been assigned to what turns out to be the same node. When this happens, the extractor *merges* the two node numbers, so that both numbers refer to the same node. This is done by maintaining equivalence relations between nodes. When two node numbers are merged their equivalences sets are simply combined. (Details on how this is implemented are covered in chapter VI.)

2. Extraction of Geometry-Only Symbols

First consider the simpler case of extraction of a geometry-only symbol. A sweeping line algorithm progressing in the direction of increasing y-values cuts the symbol into horizontal swaths coinciding with the vertices of the geometry within the symbol. At the bottom and top of each swath *swath segments* are created. These segments have a similar function as the interface segments mentioned above. Given two adjacent swaths, by comparing the swath segments at the top of the first swath with the swath segments at the bottom of the subsequent swath, node information is propagated from one swath to the next. Node information stored with the segments at the bottom of the swath is passed to the top of the swath by just following edges.

As an example, consider the geometry shown in figure III-2. The dashed lines represent the boundaries for the swaths of our sample geometry. Figure III-3 shows our sample geometry broken up into trapezoids. The numbers on the trapezoid are the node numbers assigned to that trapezoid. Let us consider how node numbers are passed up from swath A to swath B. Running down the segment list for swath A shows that the first segment overlaps the first segment in B. Node number 17 is assigned to the first segment of B. The second segment in B is overlapped by the second and third segment in A. The node numbers 43 and 8 are merged and the second segment in B receives the smaller number, 8.

Finally, the third segment of B is overlapped by nothing. A new node number is generates for this segment.

When an active region of a transistor is encountered it is also given a node number and this node number is propagated upwards like any other piece of geometry. Stored with the node number in each transistor record is information about its gate, source, and drain. If any of that information is not present the field is left unchanged. Since the entire transistor may not be in the symbol, this information may be only partially known when the analysis of the symbol is complete. By the time that the entire transistor has been seen, this information will be known. The extractor must wait until other symbols are extracted to get complete information about the transistor. Since there may be several partial transistor records that refer to one transistor, there needs to be a way of detecting when two transistor records should be combined. This situation can be detected by noting when two node numbers that refer to transistors are merged. When this happens it is not only necessary to merge the node numbers but also to combine the partial transistor records.

Finding the external interface segments for the symbol is a matter of simply noting where any geometry touches the boundary. External interface segments are also generated when the active region of a transistor touches a boundary. This allows transistors that cross symbol boundaries to be identified. The list of external interface segments is attached to the symbol definition along with the circuit description. The emerging circuit description contains two lists of transistors—one list of transistors fully contained within the symbol, and the second list of transistors with active regions that touch the boundary. The second list is used to record cases where transistors cross symbol boundaries.

3. Extraction with Instances

Now let us consider the case where a symbol contains not only geometry but also instances of other symbols. The extractor first checks each instance to see if its symbol has already been extracted. If not, then the extractor is applied recursively to that symbol. Even if a symbol is called several times, it is extracted only once.

After this phase the called symbols have been extracted, leaving a set of interface segments associated with each symbol. The transformation specified by each call to a symbol is applied to the starting and end points of each interface segment for that particular instance.

Before extracting the connectivity of the symbol, the extractor takes into account the connectivity information present in the interface segments. For example, if an instance had a metal bus crossing it, there would be two interface segments for that node; one on each side of the instance. These interface segments are physically separated so the physical connectivity extractor will not find any connection between them. Care must be taken to ensure that the connectivity of these two interface segments is not lost. This is done by assigning a node number to each interface segment before starting the physical connectivity extraction. This is called the ***actual node number*** of the interface segment. This contrasts it with the ***formal node number***, which is the node number assigned to it in the symbol for which it is an external interface. For each instance, if two interface segments have the same formal node number, then they must be assigned the same actual node number. Within an instance, if two interface segments are connected, they will be assigned the same formal node number. Thus, by assuring that interface segments with the same formal node number are assigned the same actual node number, we know that nodes connected through instances are assigned the same node number.

The actual node number of an interface segment is assigned by adding an offset to the formal node number of the interface segment. The offset is different for each instance and is chosen so that no actual node number generated in one instance is the same for an actual node number generated in another instance. A simple way to assure this is to set the offset to 1 and then assign actual node numbers to the interface segments of the first instance. Then set the offset to be 1 greater than the largest node number generated so far, and assign actual node numbers to the interface segments of the next instance. This process is repeated till all the instances have been processed.

Extraction can now proceed as in the case without instances. The only difference is that the physical connectivity extractor must handle interface segments as well as geometry. At the end of the physical extraction phase the list of external interface segments and the circuit description are attached to the symbol. Also a list of the node numbers assigned to each interface segment of each instance is attached to the symbol definition.

4. Example

As an example let us consider how the extraction of a four-bit shift register shown in figure III-4 would proceed. This shift register contains eight instances of the basic shift register cell. First the basic shift register cell must be extracted. This cell is shown in figure III-5. Extraction of this symbol is straightforward since it contains no instances of other symbols. Extraction provides a description of the three transistors in the circuit and how they are connected both among themselves and to the external interface segments. This description is attached to the basic shift register cell. A pictorial representation of this description is shown in figure III-6. The heavy black lines represent the interface segments and correspond to the heavy black marks of figure III-5.

Now the four-bit shift register symbol is ready to be extracted. Figure III-7 shows the original four-bit shift register with the geometry of the basic cell replaced by its interface segments. These are called ***internal interface segments*** of the four-bit shift register cell since these interface segments connect to instances rather than the current symbol. Note that the internal interface segments on the left and right ends of the four-bit shift register are also external interface segments since they touch the boundary of the shift register. Extraction of this symbol is a bit more complicated than that of the basic cell. Connectivity of geometry and interface segments must be included in the extracted description. Yet the same basic techniques for extracting a geometry-only symbol can be used to extract a symbol with interface segments. At the end of this extraction, a description of how the interface segments of the eight basic cells are connected to each other, to the two poly lines, and to the external interface segments of the shift register will have been produced.

5. Circuit Description

Figure III-8 shows the circuit representation for the shift register layout used above. In this figure the circuit is represented in a macro language similar to that generated by the extractor. The *P* names represent interface nodes, i.e. nodes that connect to interface segments. Internal nodes, those not connected to interface segments, are represented as an offset from the base. The base is used during macro expansion to assign unique numbers to nodes.

In the macro description for the basic cell there is only one internal node, which connects the three transistors. *P1* is the ground bus, *P2* is the clock signal, *P3* is the input, and *P4* is the output, and *P5* is the Vdd bus. In the macro description for *Shift4* there are seven internal nodes, which are inputs and outputs to instances of the basic cell. Notice that in the macro description for *Shift4*, 'base+1' appears in the fifth field of the first occurrence of *ShiftBit* and in the fourth field of the second occurrence of *ShiftBit*. This shows the connection of the first *ShiftBit* cell's output to the second *ShiftBit* cell's input. Throughout the macro description the node name in the fifth field in each instance of *ShiftBit* is repeated as the fourth field in the following instance. The second field of *ShiftBit* is always *P2*, and the sixth field is always *P5*; this shows the connection of the the ground and Vdd busses. The third field alternates between *P6* and *P1*, showing the alternating clock signals among the shift cells. While the input of the first *ShiftBit* cell and the output of the last *ShiftBit* cell are parameter nodes, the input and outputs of the other instances of *ShiftCell* are all internal nodes.

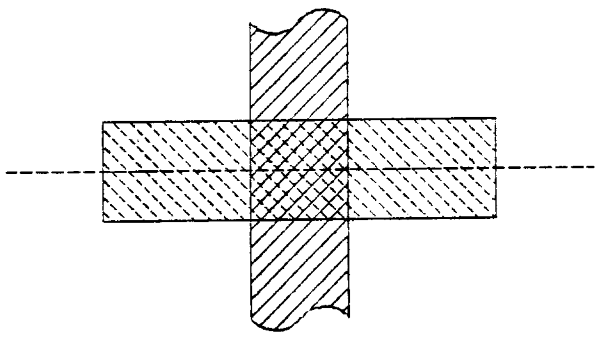

Figure III-1a. *a simple transistor*

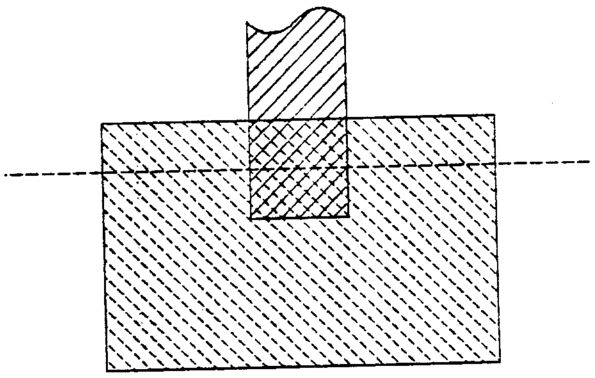

Figure III-1b. *a capacitor*

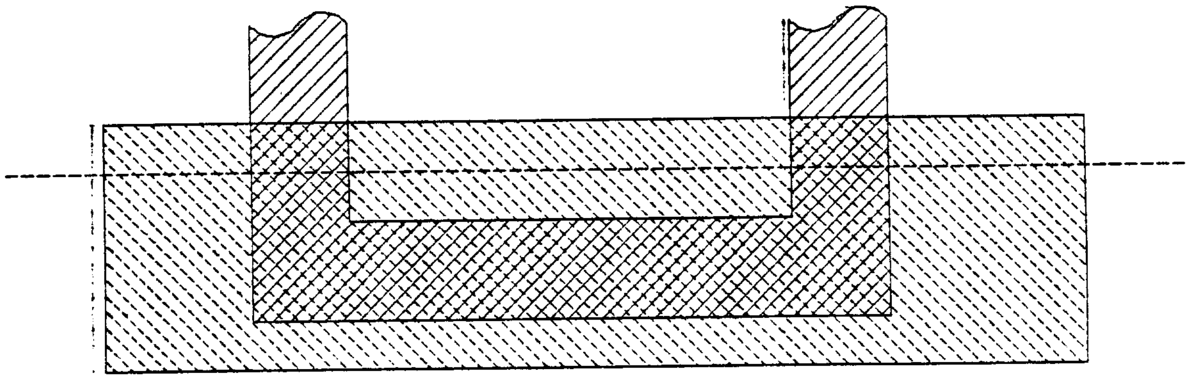

Figure III-1c. *a transistor which wraps back on itself*

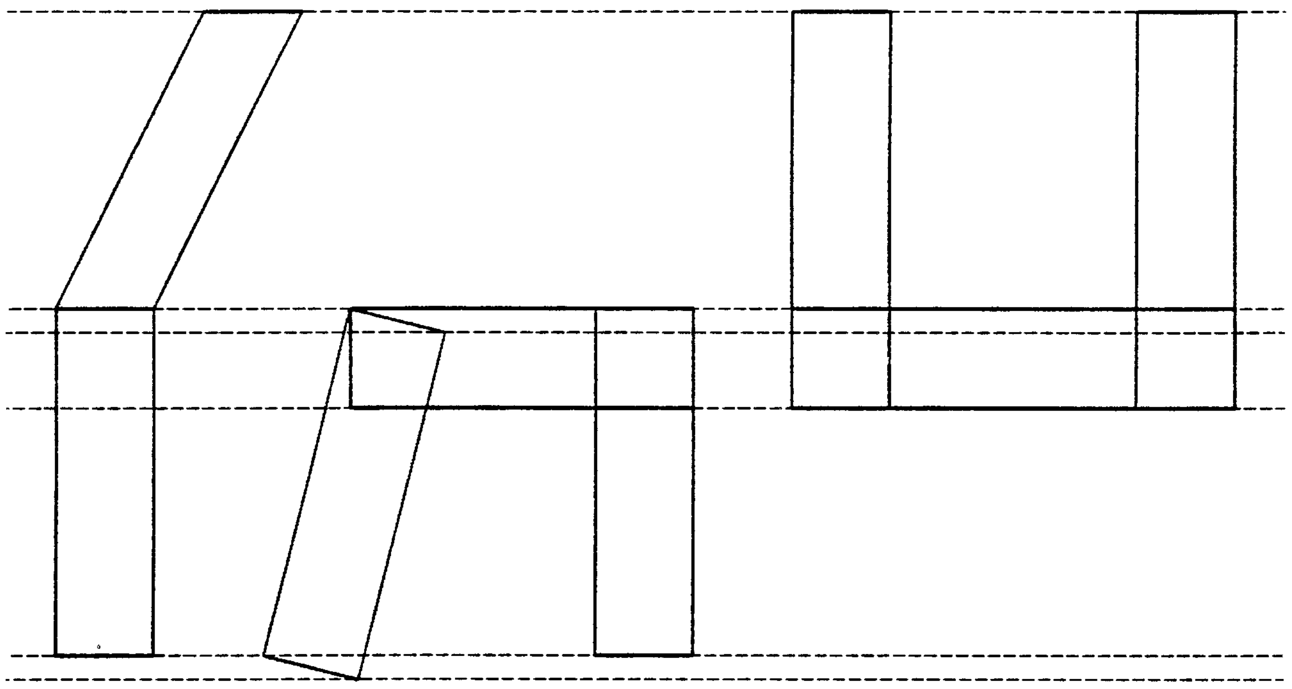

Figure III-2. *cutting geometry into swaths*

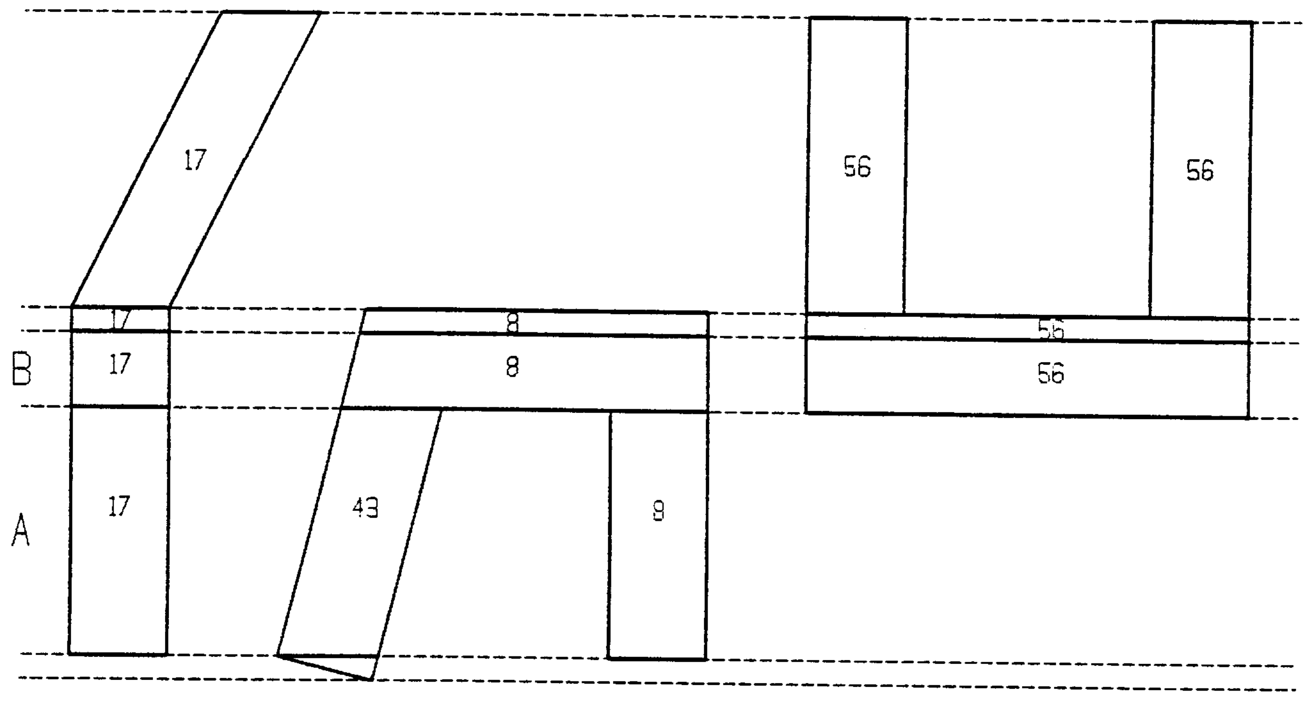

Figure III-3. *propagating node numbers*

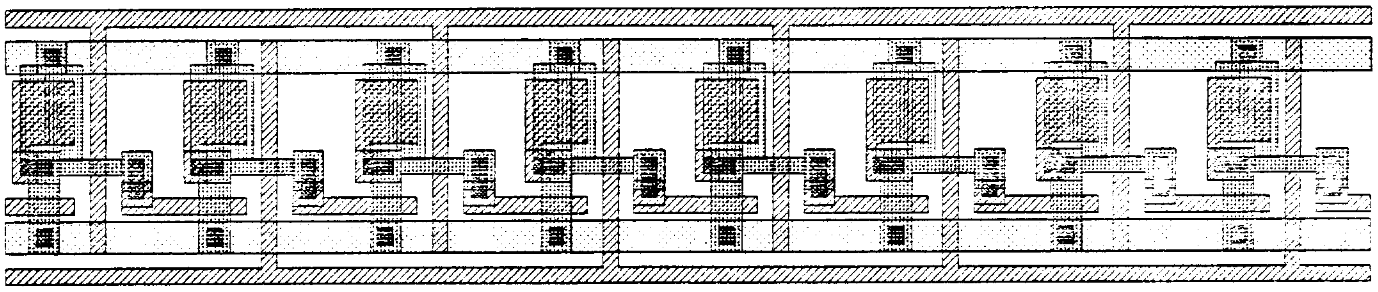

Figure III-4. *four bit shift register*

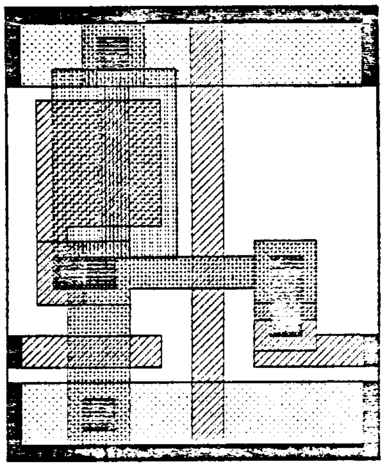

Figure III-5. *basic shift register cell*

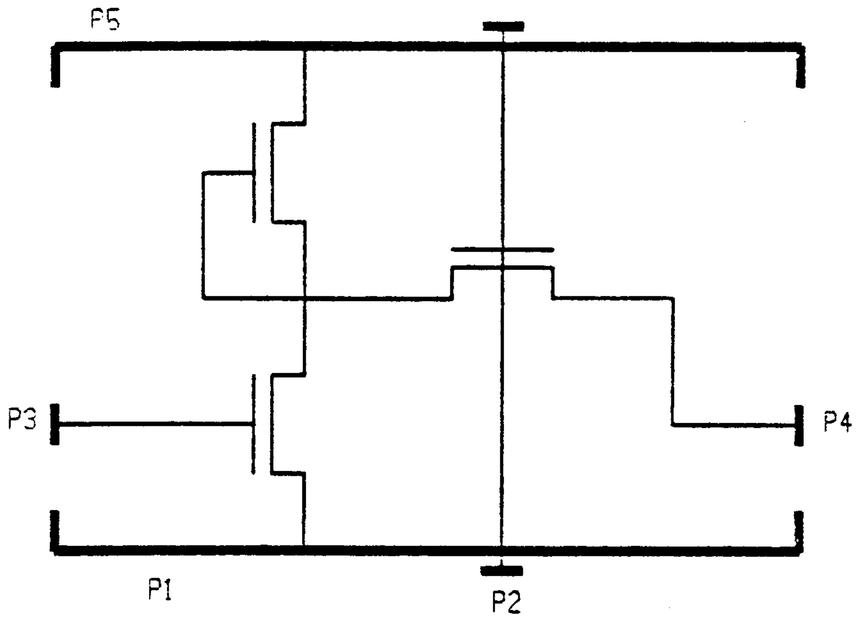

Figure III-6. *pictorial representation of basic shifter cell*

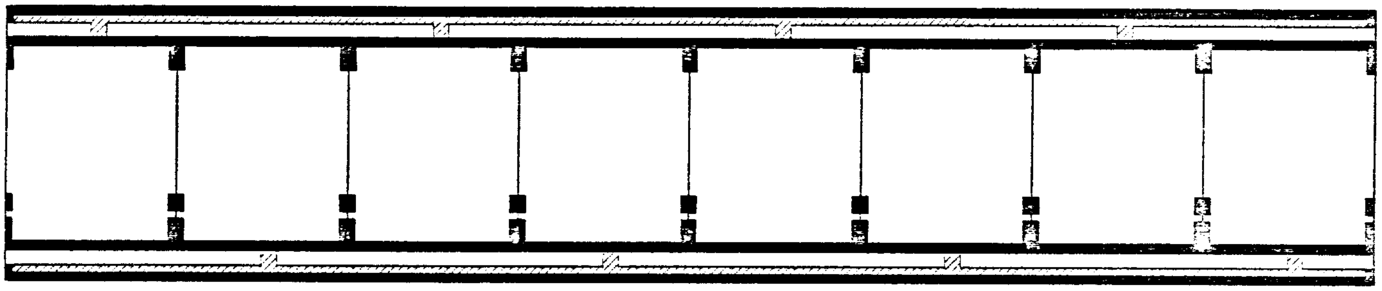

Figure III-7. *interface segments of four bit shift register*

```
define ShiftBit(base,P1,P2,P3,P4,P5);
        etrans(P3,P1,base+1);
        etrans(P2,P4,base+1);
        dtrans(base+1,base+1,P5);
end;
```

```
define Shift4(base,P1,P2,P3,P4,P5,P6);
        ShiftBit(base+7, P2,P6,P3,base+1,P5);
        ShiftBit(base+8, P2,P1,base+1,base+2,P5);
        ShiftBit(base+9, P2,P6,base+2,base+3,P5);
        ShiftBit(base+10,P2,P1,base+3,base+4,P5);
        ShiftBit(base+11,P2,P6,base+4,base+5,P5);
        ShiftBit(base+12,P2,P1,base+5,base+6,P5);
        ShiftBit(base+13,P2,P6,base+6,base+7,P5);
        ShiftBit(base+14,P2,P1,base+7,P4,P5);
end;
```

Figure III-8. *circuit description for shift register*

Chapter IV. Design Rule Checking

In any fabrication process there are limits on what can be manufactured. A line must be some minimum width in order to avoid breaks. Lines must be a minimum distance apart to avoid shorts. Rules such as these are called Geometric Design Rules. In order to pack the most circuitry into a given area a circuit designer will push these rules to their limit. Before submitting a design for fabrication, the layout is run through a program called a Design Rule Checker(DRC). The DRC looks for violations of these design rules and reports violations back to the designer. Many runs through the DRC are usually required before the designer has satisfied all the design rules.

There exist many commercially available design rule checkers. These DRC's check for violations on the fully expanded layout. Flattening the layout forces the DRC to recheck each symbol for each instance of that symbol. This approach has many problems. The first problem is the amount of time spent on checking the layout. Another problem is the large number of design rule violations that can result from a few errors in symbols that are repeated several times. The designer, lost in a sea of violation messages and knowing that most of the reported violations are due to a few errors in a few key cells, is likely to skim over the violation messages and thus may miss important errors in other symbols.

Another problem with flattening the layout is how to report violations back to the designer. One way is to present error messages graphically. A box can be placed around the offending geometry with an indication of the problem found. But, for designs consisting of a million rectangles it is quite hard to spot these violation boxes. Listing all violations with the coordinates of their occurrence is not much better. The designer must trace the approximate location of each violation on a plot of the full layout, identify the symbol in which the violation occurred, then fix the problem. Although a good layout editor can help with this process, it would be much better if the DRC could identify the symbol in which the error occurred. This can only be done, though, if the DRC explicitly deals with the structure of the layout.

1. Hierarchical Design Rule Checking

As in circuit extraction, a major problem with hierarchical design rule checking of a layout is the presence of overlaps. Overlaps can create new design rule violations in symbols in which no design rule violation were found, or can remove violations from symbols which had violations. (See figures IV-1 and IV-2.) We use the disjoint algorithm in order to remove overlaps from the layout, and check the new hierarchical description with the overlaps removed.

Once overlaps have been removed, the basic algorithm proceeds as follows. It starts with the top level symbol. In this symbol, each instance is examined to see if its symbol has been checked. If the symbol has not been checked, the algorithm recursively checks that symbol. Thus symbols containing only geometry are checked first, then symbols one level up, and so on. Hence, the algorithm proceeds in a bottom-up fashion.

Checking symbols which contain no instances is quite similar to conventional design rule checking. Geometry on the interior of the symbol is checked. If any design rule violation is found, the violation is reported with its type, location within the symbol, and the symbol name. However, violations found near the boundary of the symbol are not reported if geometry outside the symbol could remove the violation.

Checking symbols which contain instances is done by considering not only the geometry of the symbol, but also the boundary geometry from the symbol's instances. A conventional design rule check is made on this geometry. Again, violations on the interior of the symbol are reported, while violations that can be fixed using geometry from the outside are not.

Finally, when checking the top level symbol, even violations near the boundary of the cell are reported, since there is no geometry outside the circuit to fix those errors.

Notice here that the design rule checker still scans the entire area covered by the circuit. However, the amount of geometry it must check is considerably smaller. Therefore, the runtime of the geometric design rule checking algorithm should depend primarily on the amount of geometry to be checked, rather than on size of the area to be checked.

2. Geometric Checking Algorithm

There are several algorithms which can be used to do the actual geometric checking[3][1]. The algorithm presented in this section has many desirable properties for use in hierarchical checking. First, its runtime is dependent on the amount of geometry to be checked, rather than on area scanned, as in the raster approach[4]. Next, the algorithm's checks are all local, an important property for hierarchical checking. Third, the algorithm checks general polygons, not just manhattan geometry. Finally, the algorithm works well with the scanline algorithm.

The first step of the algorithm is to break the geometry up into a minimal set of horizontally oriented trapezoids using the scanline algorithm. (See section II-6.) These trapezoids are called ***mask trapezoids*** since they correspond to mask features. The algorithm generates three new types of trapezoids, ***inclusion trapezoids***, ***exclusion trapezoids***, and ***sanctuary trapezoids***. The generated inclusion trapezoids have the property that if there are no design rule violations, then the inclusion trapezoids will be completely enclosed in mask trapezoids of the same layer. The generated exclusion trapezoids have the property that if there are no design rule violations, then the exclusion trapezoids will not overlap any mask trapezoid of the same layer. Sanctuary trapezoids indicate areas where these conditions need not be met.

The method for generating these trapezoids depends on the design rules being checked. Let us consider how these trapezoids are used to check Mead and Conway separation rules[11] for the metal layer. Metal lines must be separated by 3 lambda. For each mask trapezoid on the metal layer, a three lambda wide exclusion box is generated to the right of the right edge of each trapezoid. (See figure IV-3.) This checks separation on the right of the trapezoid. Next, separation above the trapezoid must be checked. Since a metal line may continue upwards beyond this trapezoid, it is important to check only areas that are not continuations of the current node. Hence, the trapezoid is broken into sections, those sections which have metal continuing upwards, and those which do not. Exclusion trapezoids are generated above those sections which do not continue upwards. (See figure IV-4.) So far, metal separation has been checked above and to the right of the trapezoids, but there are gaps left in the corners. Now any exterior angle greater than 180 degrees between the edges of a mask trapezoid and the trapezoid directly above it is located. At these corners exclusion trapezoids are generated. Corners of trapezoids with exterior angles of greater than 180 degrees also generate exclusion trapezoids when there is no adjacent trapezoid at that corner. (See figure IV-5. For increased precision, several trapezoids could be generated to approximate a circular section.) Thus, above and to the right of the mask trapezoid has been checked. This is all that is necessary because below and to the left will be checked by trapezoids in that area. Figure IV-6 shows all the exclusion trapezoids generated for this sample geometry.

Testing minimum line width is similar to testing minimum separation. Inclusion boxes are generated to the right of the left edge of each trapezoid. (See figure IV-7.) The bottom edge is broken into sections, those sections with mask geometry beneath them, and those without. Inclusion boxes are placed above those sections with no mask geometry below. (See figure IV-8.) To check the corners, inclusion trapezoids are generated at angles between adjacent trapezoids greater than 180 degrees. (See figure IV-9.) Figure IV-10 shows all the inclusion trapezoids generated for this sample geometry. Note that these tests forbid both interior and exterior acute angles.

Other design rule violations are easily detected with this technique. Metal overlap of cut can be checked by creating inclusion trapezoids on the metal layer one lambda bigger on each side of a cut trapezoid. Polysilicon overlap of transistors can be checked, along with diffusion overlap, by creating inclusion trapezoids along each edge of a transistor trapezoid but not at the corners. The new inclusion trapezoids are placed on a special layer of 'polysilicon-or-diffusion'. These trapezoids must be covered by either polysilicon or diffusion trapezoids.

Finding violations is simply a matter of detecting when inclusion boxes are partially uncovered or exclusion boxes are partially overlapped.

3. Example of Hierarchical Checking

Now let us consider how to use the above algorithm to hierarchically check a symbol which contains both geometry and instances as shown in figure IV-11. Since the maximum extent of Mead and Conway design rules is 3 lambda, included with the geometry of the symbol is the geometry of each instance within a 3 lambda wide frame of the instance's boundary. The interior of each instance is covered with sanctuary trapezoids to within 3 lambda of the instance's boundary. The exterior of the symbol is also surrounded with sanctuary trapezoids. (See IV-12.) The geometry checking algorithm now runs. Inclusion and exclusion trapezoids are not generated from edges in sanctuary trapezoids. Violations found outside sanctuary boxes are reported. Violations found inside are ignored. Thus, in figure IV-13 there are five violations, four generated by insufficent overlap over the cuts, and one due to insufficient separation of metal lines. The four inclusion trapezoids generated by the cuts lie in the sanctuary region, so they are not reported. The exclusion trapezoid generated by the insufficient separation is not in a sanctuary trapezoid, so it is reported.

4. Non-Local Rules

A common criticism of hierarchical design rule checking is that there are certain design rules which are non-local in nature. For instance, a fabrication line may have a rule the says that polysilicon lines must have a minimum separation of 3 microns but parallel runs of polysilicon lines longer than 2 millimeters must be separated by at least 4 microns. Rules such as these are especially hard to check hierarchically.

Most design rules, however, are local. The hierarchical algorithm described here only checks local rules. Most conventional DRC's also only check local rules. It turns out that very few rectangles in a layout are candidates for these non-local rules. These non-local rules could fairly quickly be checked by a special purpose flat design rule checker. The more time consuming local rules still should be checked hierarchically.

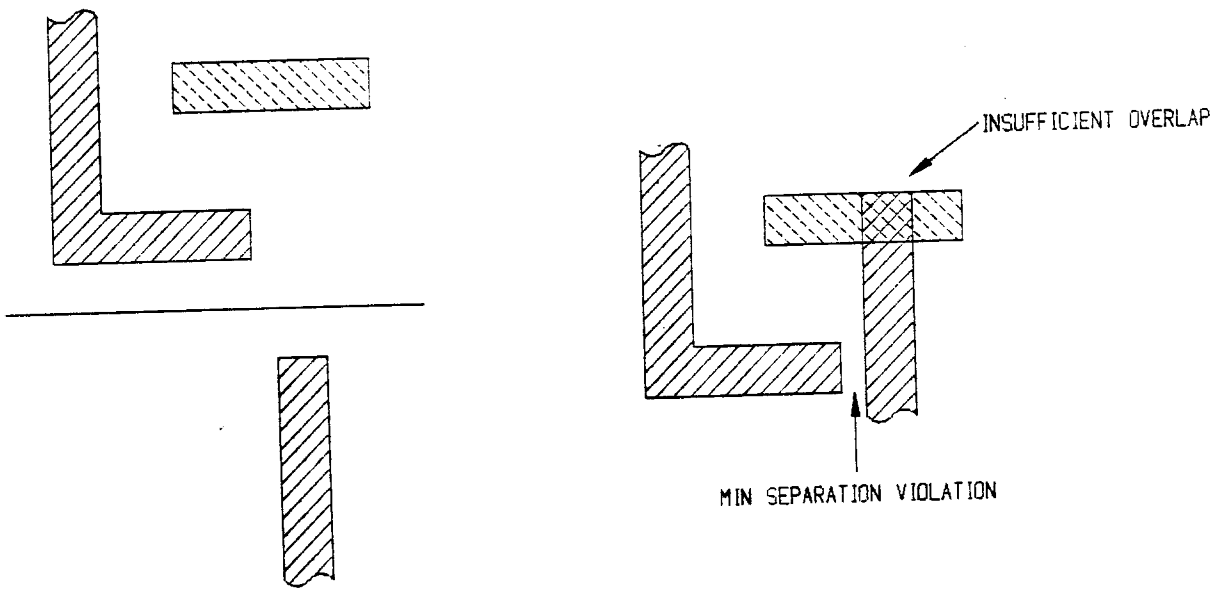

Figure IV-1. *overlaps can create design rule violations*

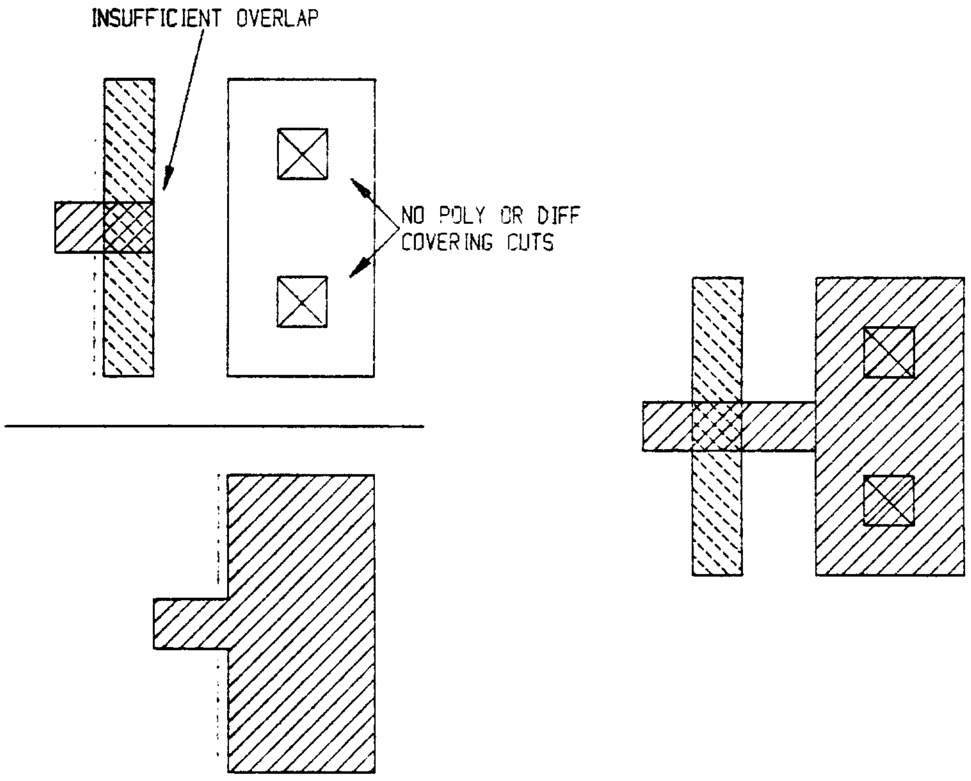

Figure IV-2. *overlaps can remove design rule violations*

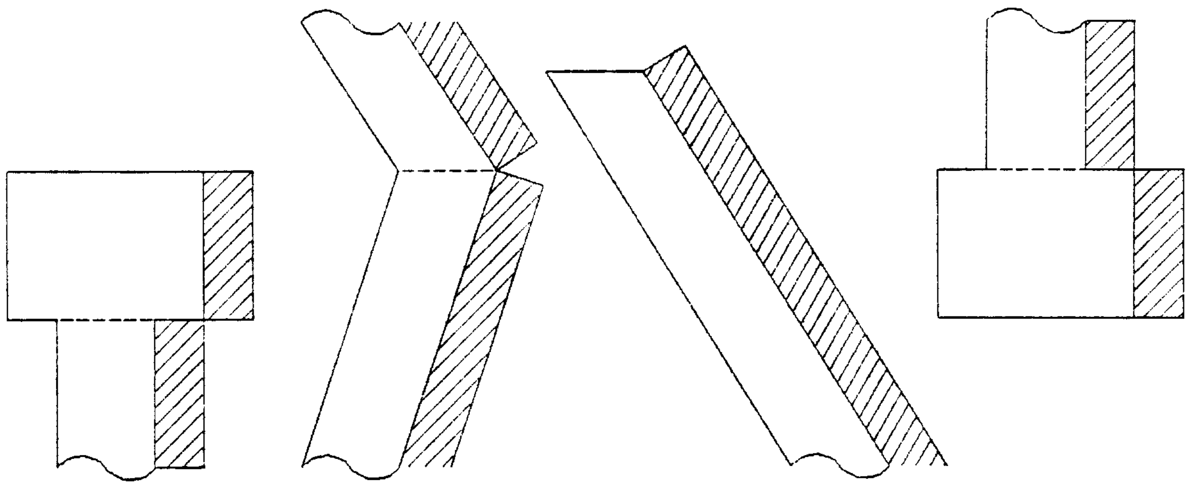

Figure IV-3. *generate exclusion trapezoids to the right of geometry*

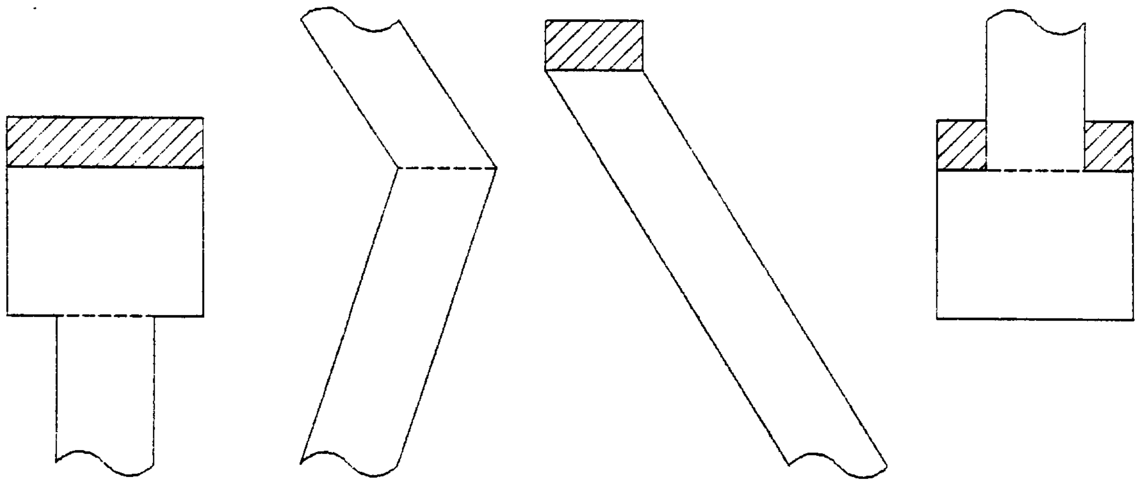

Figure IV-4. *generate exclusion trapezoids above geometry*

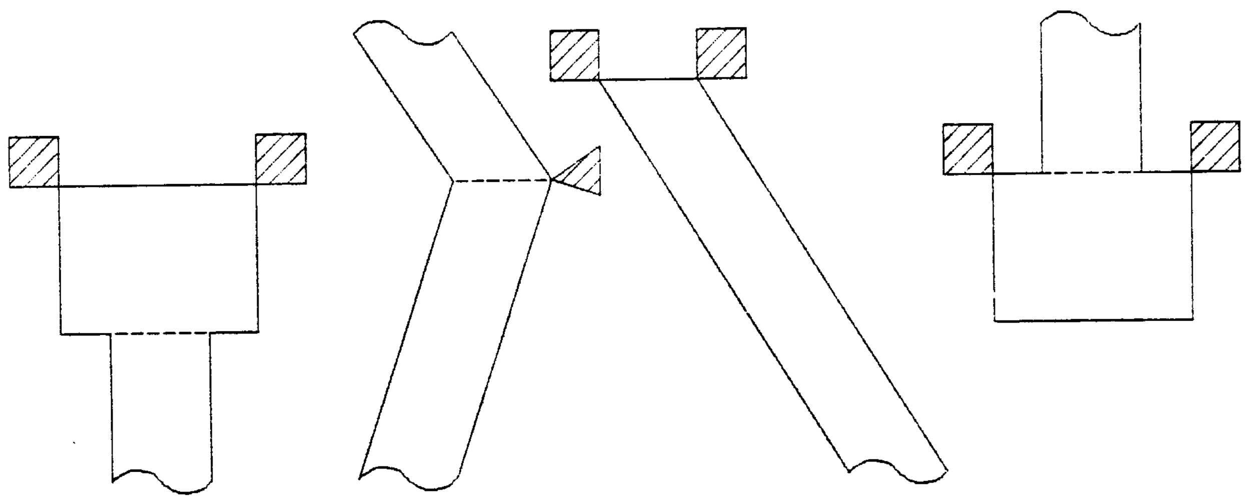
Figure IV-5. *generate exclusion trapezoids at oblique corners*

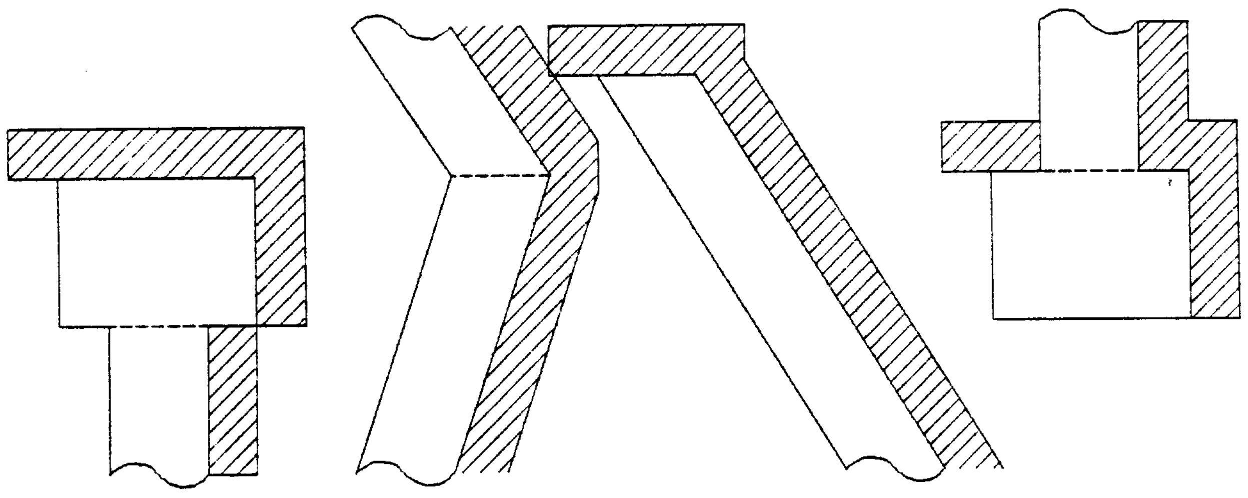
Figure IV-6. *all exclusion trapezoids*

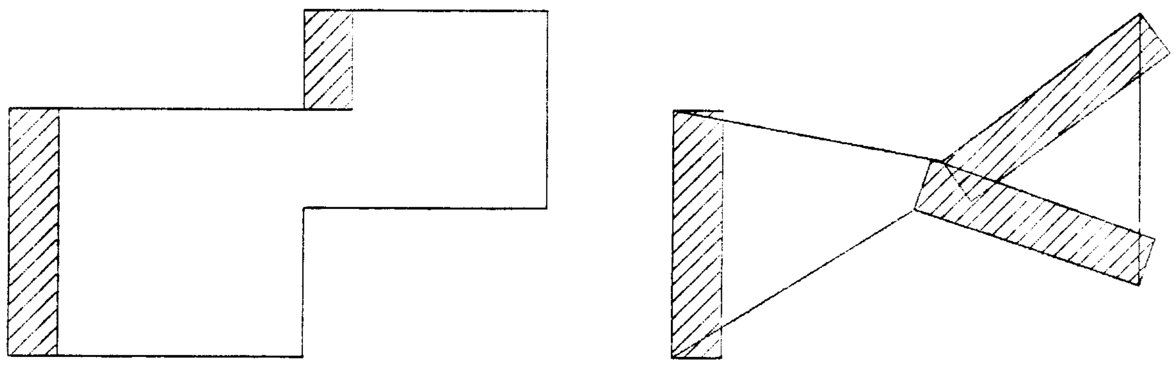

Figure IV-7. *generate inclusion trapezoids to the right of left edges*

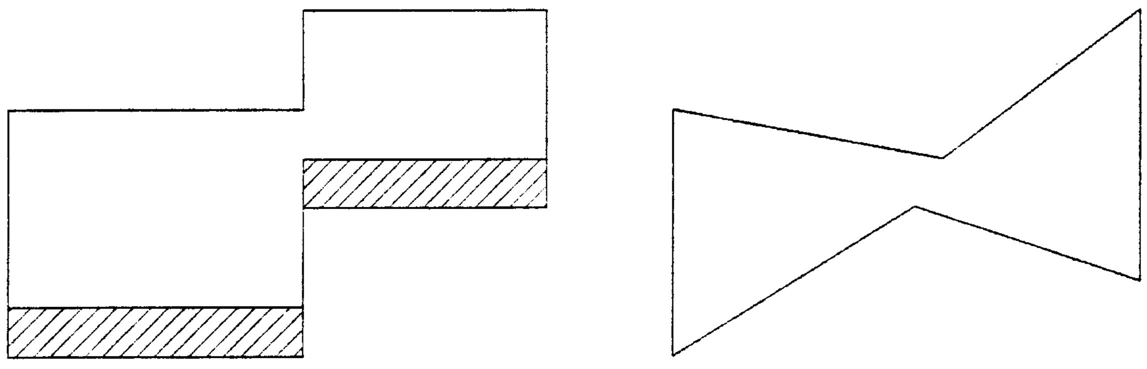

Figure IV-8. *generate inclusion trapezoids above bottom edges*

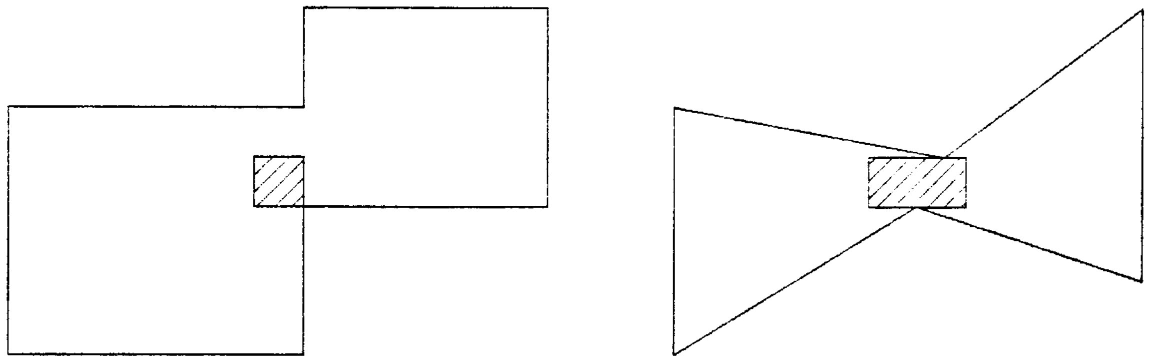

Figure IV-9. *generate inclusion trapezoids at oblique interior corners*

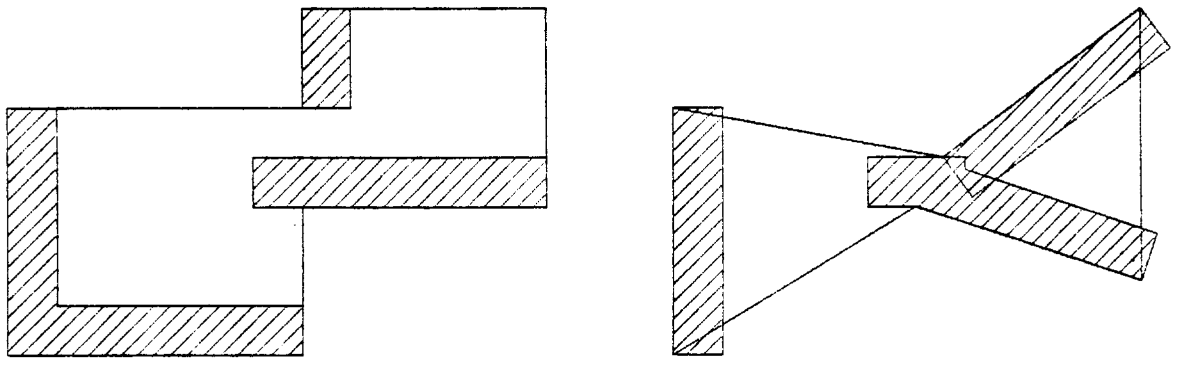

Figure IV-10. *all inclusion trapezoids*

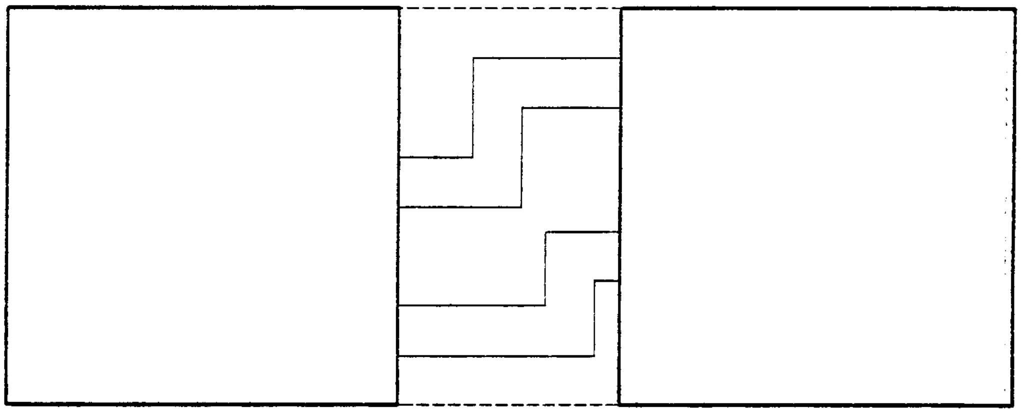

Figure IV-11. *a symbol containing instances and symbols*

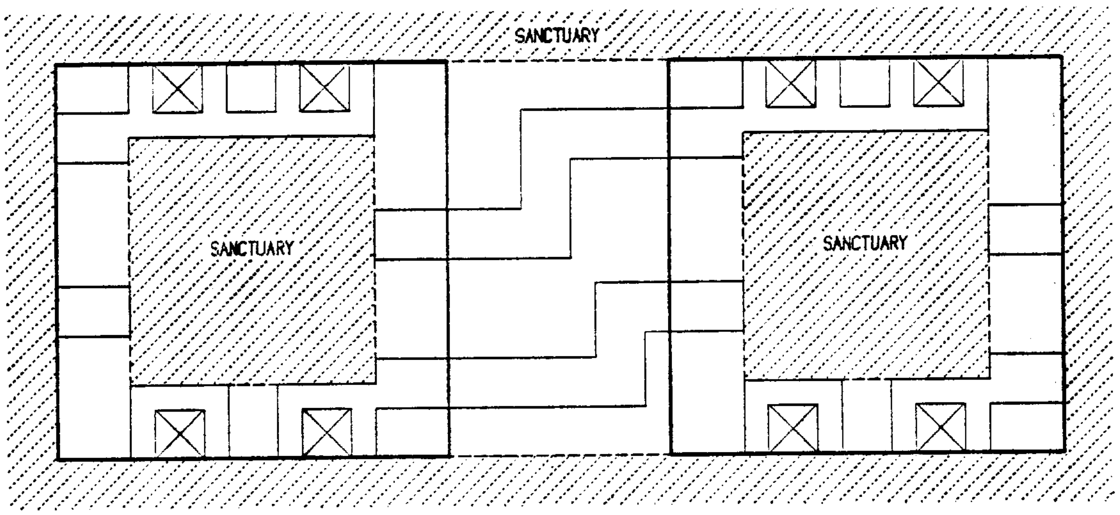

Figure IV-12. *expand geometry along perimeter of instance and create sanctuary trapezoids*

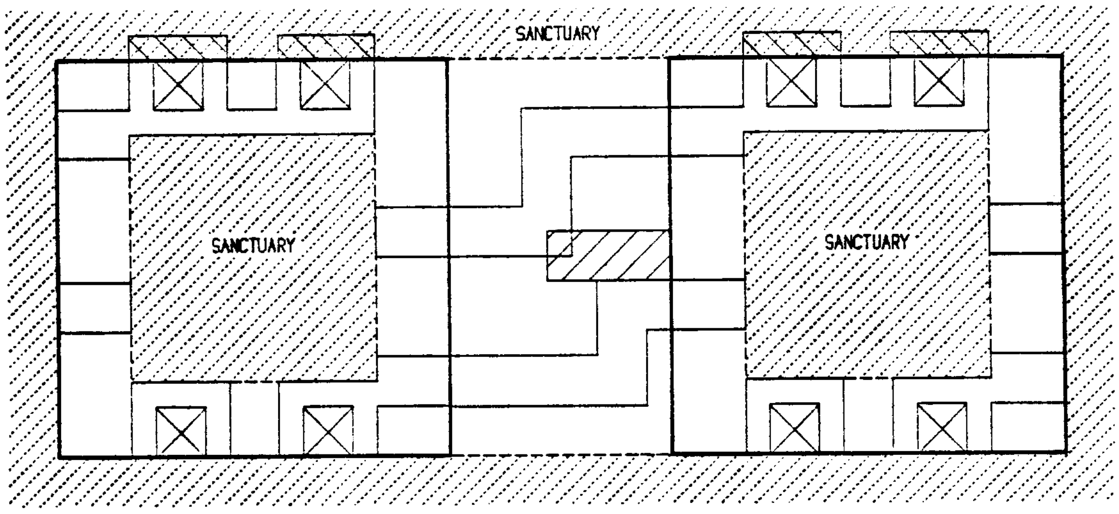

Figure IV-13. *create and check exclusion and inclusion trapezoids*

Chapter V. Mask Operations

Mask operations such as AND, OR, NOT, GROW, and SHRINK are an important set of operations performed on mask layouts. Figures V-1 through V-5 show examples of these operations applied to various layouts. These operations may be necessary to match the specific requirements of different IC processing lines, or to allow the designer to specify his layout in a more abstract manner, freeing him from worrying about the actual implementation details[18]. The methods used for performing mask operations discussed here are different than those described by Baird[3] and commonly used in commercially available software. The most notable difference, besides being hierarchically applied to a layout, is that they do not require a separate 'merge' operation. None of the algorithms described here assume that the input layers are in a merged format, nor will the newly created layers from these algorithms be in a merged format. (A merged format represents contiguous geometry on a single mask layer as a single polygon. The geometry dealt with here is assumed to be made up of several polygons which may overlap each other. This is the type of geometry most likely to come out of a design system.) It is important to note that the 'merge' operation typically destroys most of the structure present in a layout and can result in large unwieldy polygons for global signals such as Vdd and ground.

Most of the algorithms described here are based on the scanline algorithm described in detail in section II-6. The AND, OR, and NOT operators preserve the structure of the layout. However, the GROW and SHRINK operators do change the boundary of symbols and may introduce additional overlap. Therefore, if further processing is to take place on a layer created from a GROW or SHRINK operation, the disjoint transformation must be run over the layout again. Since the disjoint transformation is likely to create more symbols, care should be taken to avoid unnecessary GROW and SHRINK operations.

1. OR

The OR operator is by far the easiest mask operation considered here. It can be done hierarchically even in the presence of overlapping instances. The OR operator applied to layers A and B simply outputs a copy of any geometry it sees on layer A or layer B.

2. AND

The AND-operation is slightly more complicated than the OR operator, but once overlaps have been removed, it too is quite straightforward. To AND layers A and B, a simple scanline algorithm is applied to the geometry on layers A and B. The algorithm simply outputs a trapezoid on the regions where layers A and B intersect along each swath. To minimize the number of trapezoids, the output of the trapezoids is delayed until the following swath to see if any of the trapezoids continue upwards.

3. NOT

The NOT operator is also straightforward once overlaps have been removed from the layout. Again the scanline algorithm is used. This time, though, the algorithm is applied to the considered layer as well as the boundaries of the instances in each symbol. Trapezoids are output to cover any area within the boundaries of the symbol currently being processed that is not covered by either mask geometry or an instance. Instances are excluded from the NOT area since these will be covered from within when the instances themselves are processed.

4. GROW

The GROW operation is another straightforward procedure and can be performed in the presence of overlapping instances. To perform a GROW operation, each piece of geometry is expanded by the GROW amount. The GROW operation, therefore, does not maintain the disjoint hierarchy. The boundary of each symbol is likely to expand by the GROW amount, therefore instances which previously did not overlap will overlap. Extensive GROW operations may destroy much of the structure of the layout.

5. SHRINK

Of all the mask operations SHRINK is by far the most difficult to implement hierarchically. The difficulty with the SHRINK operation is that geometry touching the boundary of a symbol may be affected by geometry outside the symbol. And since the geometry outside the symbol is likely to be different in different instances of the symbol, the final shrunk geometry may be different in different instances of the symbol.

The SHRINK operation can be done by performing a shrink operation on the symbol as though nothing existed outside the symbol. (See figure V-6b.) In addition it is necessary to clip twice the SHRINK distance along the border of the symbol on the original geometry. Thus, if *s* is the shrink distance, any geometry within 2*s* of the symbol boundary is saved. (See figure V-6c.) Any of this clipped geometry that does not touch the symbol boundary can be discarded. (See figure V-6d.) When considering the SHRINK of a symbol which contains instances, the clipped geometry of each instance must be combined with the geometry of the current symbol. A SHRINK with this additional geometry will correctly describe the shrunk geometry. Figure V-7a shows a symbol which contains two instances of the symbols used in figure V-6. Figure V-7b shows the geometry of the symbol along with the clipped geometry of the instances. Figure V-7c shows the resulting shrink on this geometry. Finally, figure V-7d shows all the geometry after the shrink operation.

Like the GROW operation, the SHRINK operation changes the structure of the layout. Excessive SHRINK operations can destroy most of the structure that was originally in the layout. Often, though, the GROW and SHRINK operations can be arranged so that they are the final steps of a process. If this is possible to arrange, the structure of the layout can be exploited by all the mask processing operations.

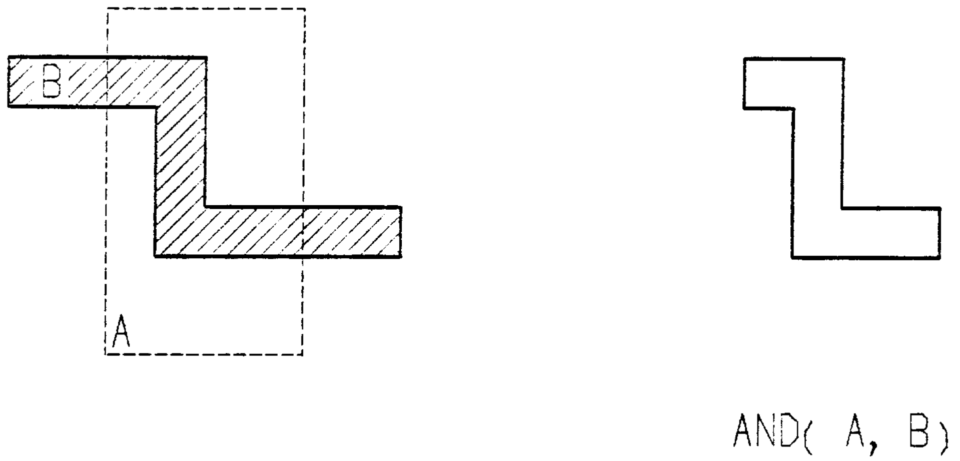

Figure V-1. *And operator*

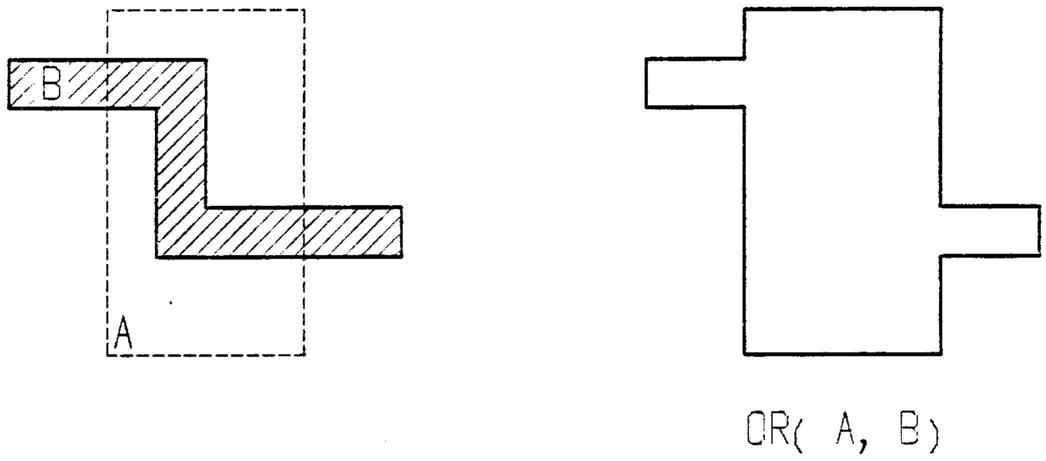

Figure V-2. *Or operator*

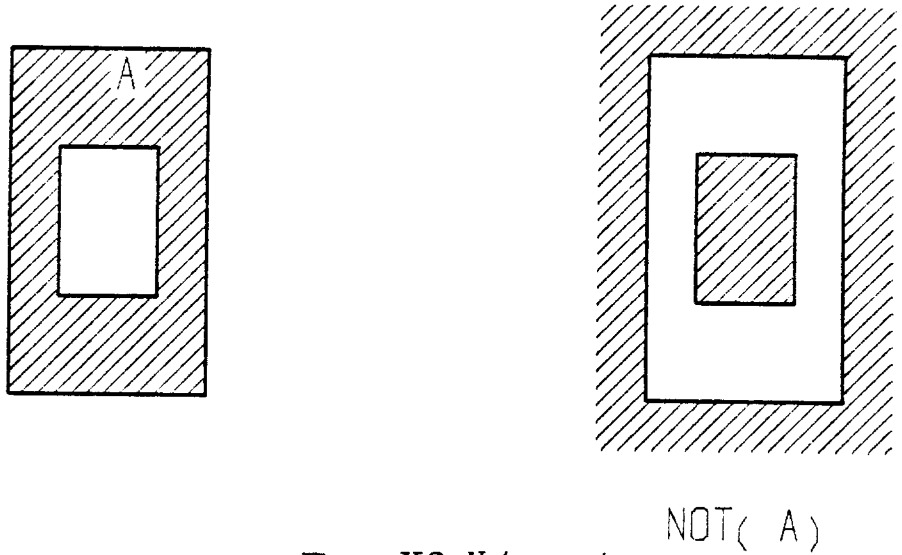

Figure V-3. *Not operator*

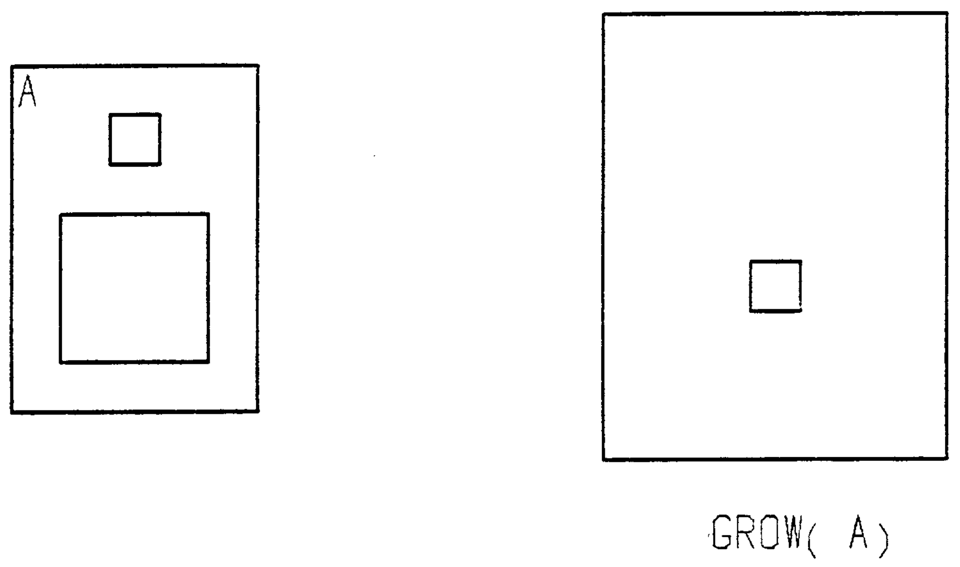

Figure V-4. *Grow operator*

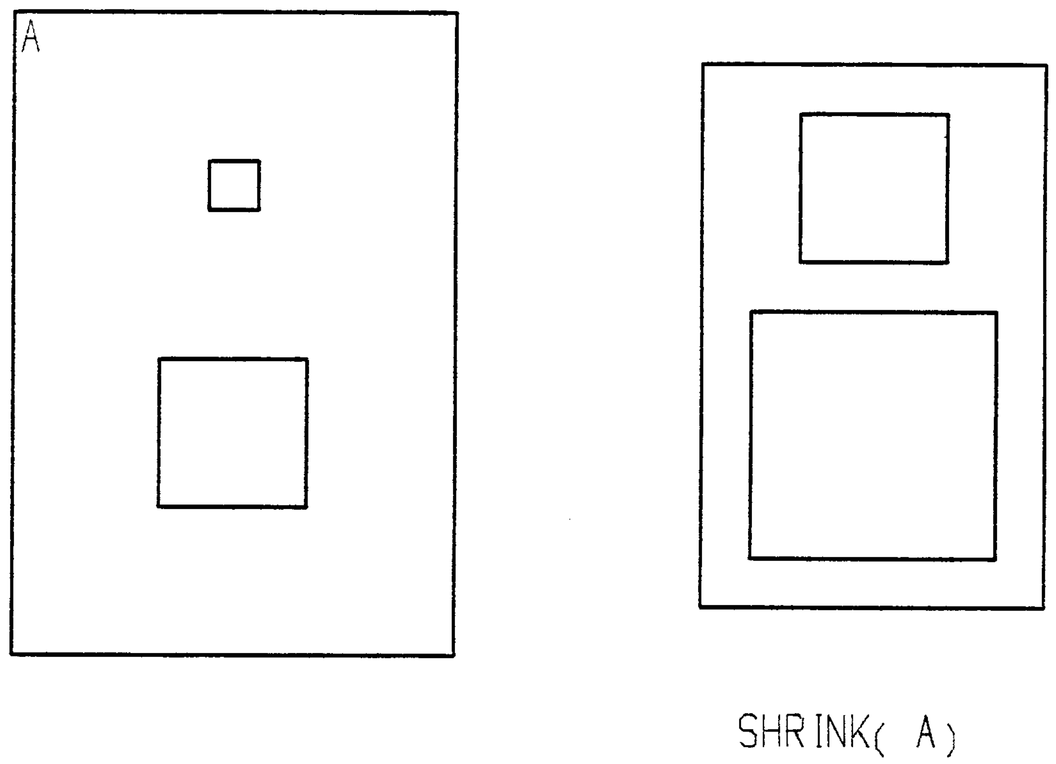

Figure V-5. *Shrink operator*

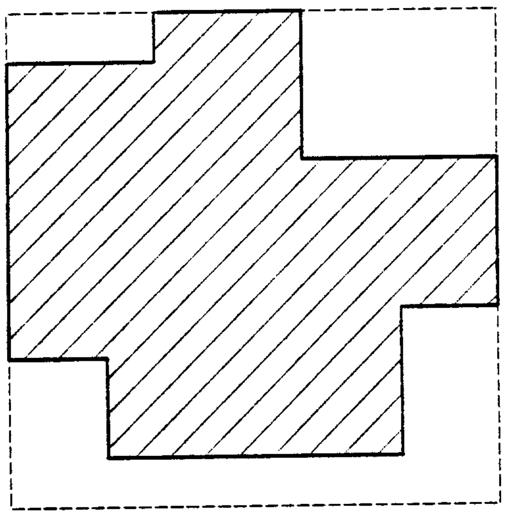

Figure V-6a. *symbol before shrink operation*

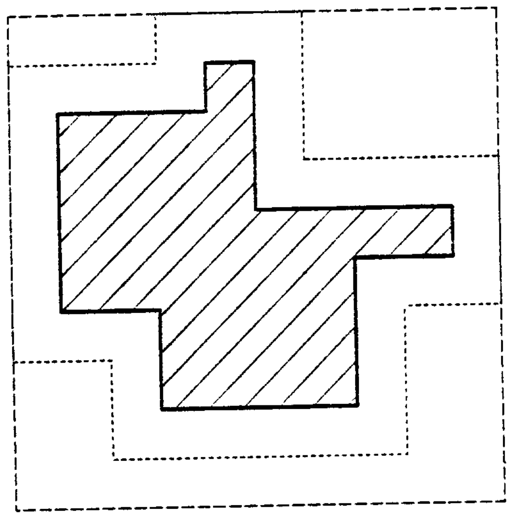

Figure V-6b. *geometry after shrink operation*

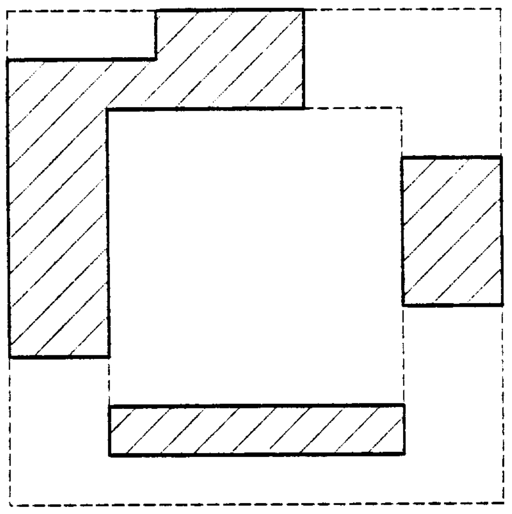

Figure V-6c. *clip geometry near symbol boundary*

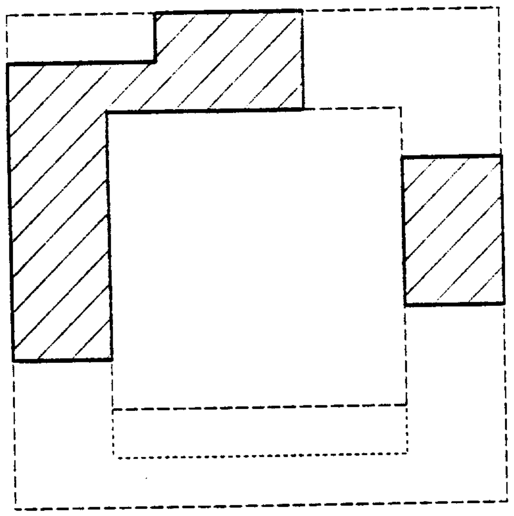

Figure V-6d. *retain clipped geometry touching symbol boundary*

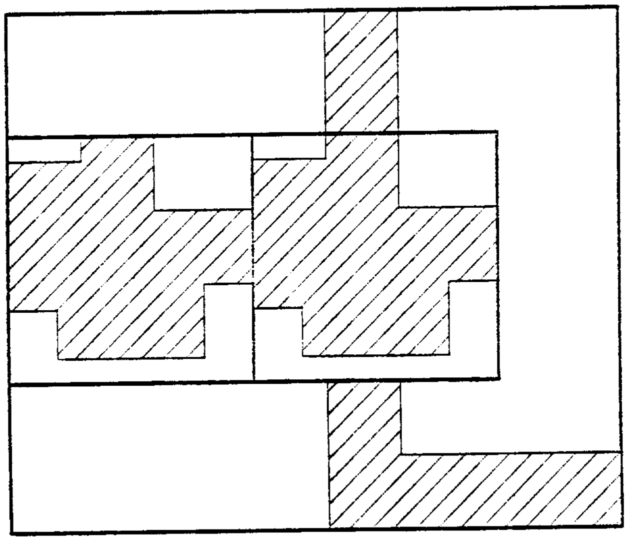

Figure V-7a. *shrink on symbol containing instances and geometry*

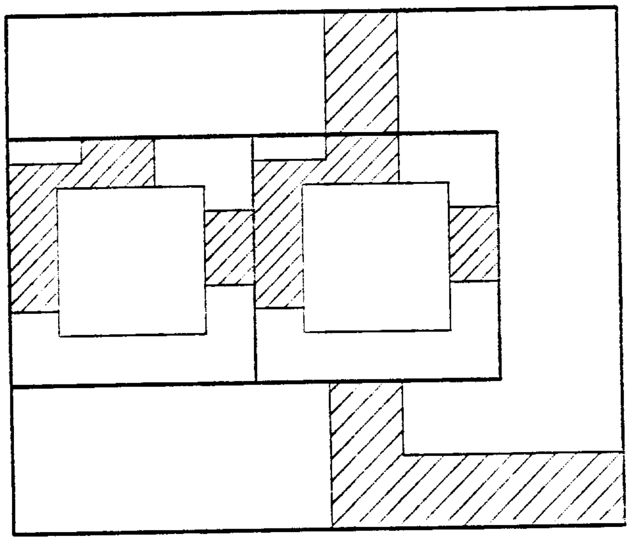

Figure V-7b. *expand clipped geometry of instances*

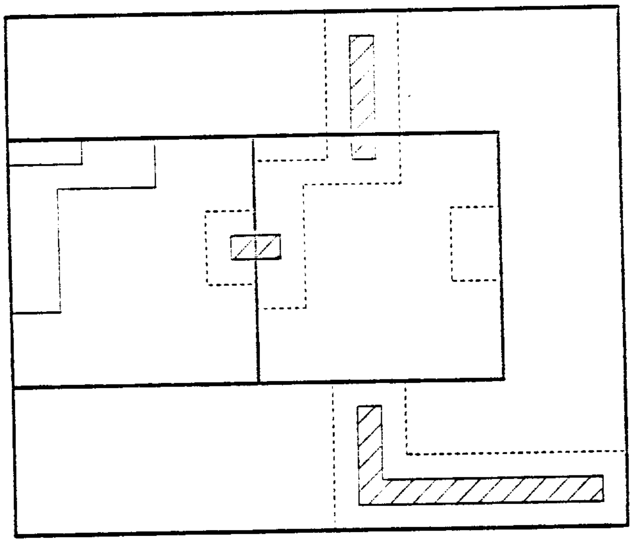

Figure V-7c. *shrink the geometry*

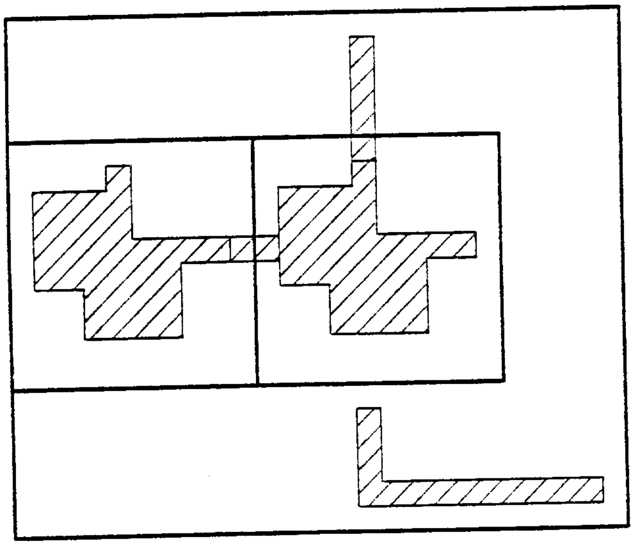

Figure V-7d. *symbol fully expanded after shrink*

Chapter VI: Implementation

This chapter discusses the implementation of a hierarchical extractor based on the disjoint transformation. This program has been implemented on two systems, one written in Mesa for the Dorado computer[9], and the other written in C to run under Berkeley VAX UNIX. Rather than discuss the implementation of both programs, I will describe an idealized implementation and then point out how the real implementations differ from this idealized description.

The implementation described here assumes that the geometry to be extracted is rectilinear. This makes the implementation considerably easier, but it is important to note that all the algorithms described here will generalize to non-manhattan geometry. This manhattan restriction also applies to symbol transformations. Thus, while symbols may be translated by an arbitrary amount, they may be mirrored only about the X or Y axis and rotated by multiples of 90 degrees. While this restriction does not easily generalize, it does cover the overwhelming majority of integrated circuits layouts.

1. Data Structures

Before discussing the algorithms, it is useful to discuss the major data types used by the program. By convention the first field in every data type is a type identifier. This aids in writing generic routines (i.e. routines that can operate on many different data types). The type field is commonly followed by a link field which is used to put the structure into a list. The link field is then followed by four integers, *left*, *bottom*, *right*, and *top*, which form the bounding box of the structure.

One of the main data types is the symbol. Its structure is shown in figure VI-1. *Name* is the symbol's name. *Instances* is a list of instances contained in the symbol. *Boxes* is a list of boxes contained in the symbol. (As noted above, this implementation deals only with manhattan geometry, so all geometry can be represented as rectangles. An implementation that deals with general geometric shapes would need a list of polygons.)

The above fields are all needed to describe the symbol itself. The following fields are used by the disjoint algorithm to transform symbols into non-overlapping region, and by the extractor to pass connectivity information

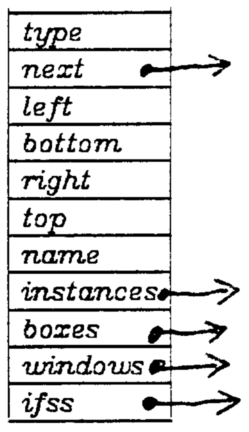

Figure VI-1. *data structure for a symbol*

between symbols. The *windows* field is a list of rectangles which define the manhattan boundary of the symbol. (Rather than represent the polygonal boundary of the symbol explicitly, the symbol is divided into rectangular regions which covers this polygonal area. The polygonal boundary can be easily reconstructed from these rectangles. No assumption is made that that these rectangles are connected.) The *ifss* field is a list of interface segments for the symbol. (Interface segments are discussed later. Basically, they are the means by which the circuit extractor passes information between symbols.) In both the Mesa and C implementations there are additional fields for maintaining statistics but they need not concern us here.

The data structure for the box is shown in figure VI-2. Boxes are stored in lists attached to the symbol they are contained in. The bounding box defines the the area that the box covers. The *layer* field indicates the mask layer for that box.

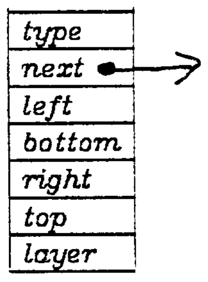

Figure VI-2. *data structure for a box*

The data structure for an instance is shown in figure VI-3. Like the boxes, instances are stored in lists attached to the symbol that contains them. The *next* field is the link field for this list. The bounding box is the minimum bounding box for the instance. The *symb* field is a pointer to the symbol which the instance references. *Transform* is a 3 by 2 transformation matrix. (A 3 by 2 matrix is sufficient to describe all transformations commonly applied to symbols, i.e. rotation, mirroring, and translation. See [14] for details.)

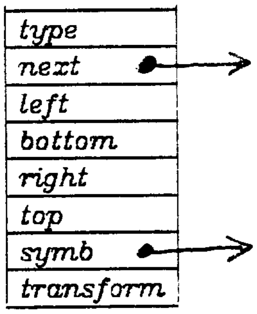

Figure VI-3. *data structure for an instance*

There are several other data types used in the program but these will be discussed later as their use becomes more apparent.

2. Disjoint Routine

The *disjoint* routine is called with a pointer to a symbol. This symbol is usually the top level symbol in the layout hierarchy. *Disjoint* simply calls *split* passing the symbol as a parameter. This call to *split* may generate new symbols. *Disjoint* keeps feeding these newly created symbols to *split* until all the symbols have been exhausted. *Disjoint* then calls a cleanup routine which performs various bookkeeping chores, such as freeing up no longer used structures.

3. Split Routine

Split is also called with a pointer to a symbol. It is *split*'s job to consider the instances of the symbol and to divide the symbol up into regions separated into different regions of overlap. As noted in chapter II, this is done with a sweeping line algorithm. *Split* first enters into a list the edges from the windows on the symbol. The edges from the instances of the symbol are then entered into the list. The window edges are used to mark which instances are contained within the symbol and which instances are totally outside the symbol and can therefore be ignored. The structure for these edges is shown in figure VI-4.

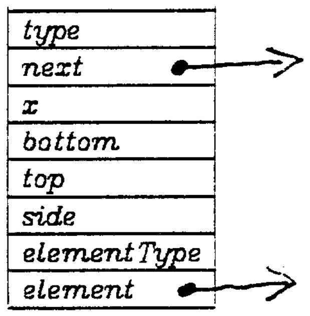

Figure VI-4. *data structure for an edge*

The *x*, *bottom*, and *top* fields describe the position of the edge. The *side* field indicates whether this is a left edge or a right edge. The *elementType* field indicates whether this is an edge from a window or from an instance. The *element* field is a pointer to the window or instance that generated this edge.

As the program scans from left to right along a swath, a *current set* is maintained. The current set is the set of instances which covers the part of the swath now under consideration. At the start of the swath, the current set is empty. On every left edge of an instance that instance is added to the current set. On every right edge the instance is removed from the set. The region covered by the current set is bounded by the bottom and top of the current swath, and by the x position of the last change to the current set and the current x position. Whenever the current set is changed, i.e. instance is added to it or removed from it, a check is made to see whether the edge is inside a window or not. If the edge is not in a window, no further action is taken. If the edge is in a window, the rectangle covering the current set is added to the rectangles that have already accumulated for this set of instances. These rectangles are kept in a structure called a *discell.* The structure for a discell is shown in figure VI-5.

The *instance* field is a pointer to the list of instances that this discell covers. The *windows* field is the list of rectangles generated as described above. The *boxes* field is a pointer to a list of boxes contained in this discell. Initially it is set to *nil.* The *symbol* field is a pointer to the symbol which matches this discell. This is described in section 4.

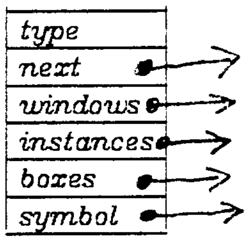

Figure VI-5. *data structure for a discell*

By the time the entire symbol has been scanned, a discell for each distinct set of overlapping regions will have been created. Any geometry in the symbol is now partitioned into the appropriate regions. For each discell, the rectangles of the symbol are clipped, creating two separate lists, rectangles inside the discell and rectangles outside the discell. The rectangles inside the discell are attached to the discell. The rectangles outside are kept to compare against the next discell.

This method of dividing up the rectangles has the potential to become explosive. Figure VI-6 represents this process of clipping rectangles. When one rectangle is clipped against another as many as 5 rectangles can be created, 1 inside rectangle and 4 outside rectangles. (See figure VI-7.) A discell may be composed of several windows, and many rectangles could be created for each original rectangle. Thus, this process could create a huge number of rectangles. Fortunately, this doesn't happen in typical layouts.

4. Gather Routine

The *gather* routine is called with a list of discells and the symbol which generated them. The job of gather is simply to change each discell passed to it, into an instance of a symbol. It must see if this discell can be changed into an instance of an already existing symbol or if it must create a new symbol. Creating new symbols should be avoided if possible, since this can ruin the structure of the layout.

Although the process of changing a discell into instances is long, it is not complicated. Associated with each discell is a list of instances. These instances are broken down into symbol-transformation pairs. The pairs are sorted by symbol names. The transformation of the first pair entered into the list is taken as the reference transformation. All the transforms of the pairs are multiplied by

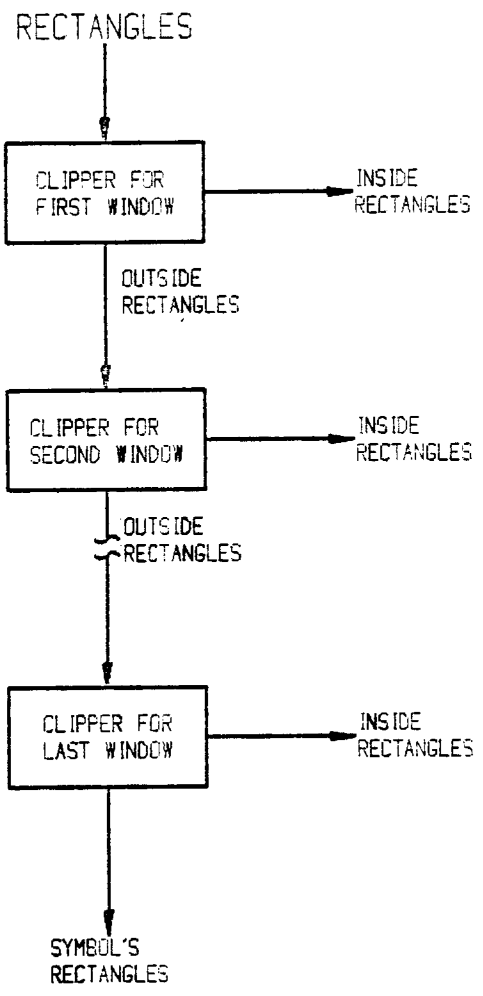

Figure VI-6. *clipping process*

the inverse of the reference transformation. In addition, all the boxes and windows of the discell are transformed by the inverse reference transform. The boxes and windows are then sorted.

Next a check is made to see if there already exists a matching symbol. Stored with each discell are the normalized symbol-transformation pairs it was made from, along with its boxes and windows. It is compared with each previously encountered discell. To speed up this matching process a hash table is used. A hash index can be computed from the symbol-transformation pairs, and the windows. While this hashing scheme may not result in any algorithmic gains, it significantly speeds up this lookup chore.

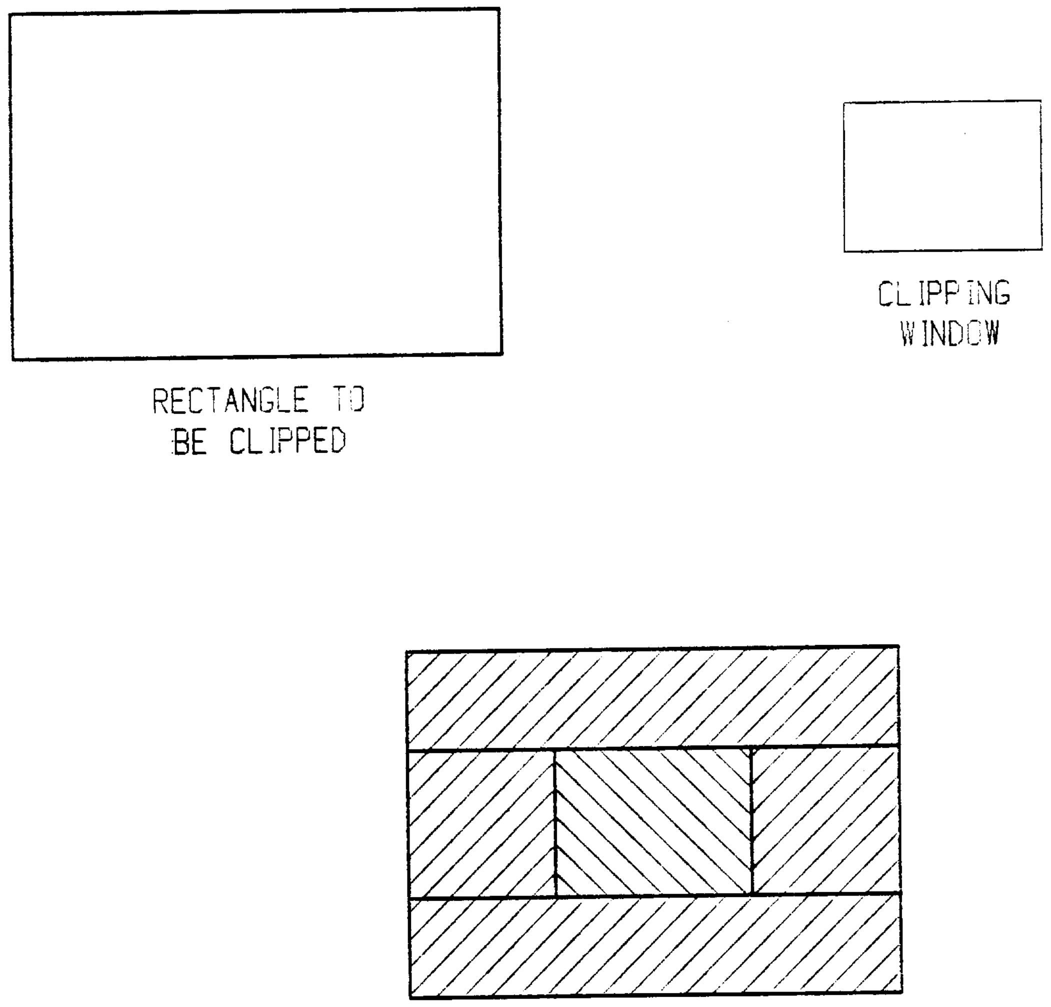

Figure VI-7. *clipping a rectangle*

If a match is found, the matched symbol is returned. Otherwise a new symbol is created, the discell is entered into the hash table, and the new symbol is returned. The returned symbol is combined with the reference transform to create a new instance. In this manner gather replaces each discell with an instance. These new instances replace the old instances of the symbol gather was working with. The important property that these new instances do not overlap is now established.

Symbols newly created in the gather procedure are placed on a new symbol list. As mentioned earlier, the disjoint routine keeps calling split with these new symbols until the list is exhausted.

A new symbol is created based on the instances covering it. The contents of these instances, boxes and other instances, become the contents of this new symbol. Boxes and instances outside the symbol's windows are not considered part of this new symbol.

The disjoint algorithm works from top down. First the top level symbol of the hierarchy is called. It is broken down into non-overlapping instances and geometry. Then the algorithm recurses and runs the split-gather routines on these symbols, thus removing overlap.

The Mesa and the UNIX versions differ in the type of symbol transformations considered in the matching. The Mesa version does not recognize rotated or mirrored instances, and the only transformations considered are translations. Rotated and mirrored versions of a symbol are considered different versions of the symbol. This may account for the differences in the statistics reported in the next chapter.

5. Circuit Extraction

After the disjoint algorithm has been run over the layout hierarchy, removing all overlap, the actual circuit extraction can be performed. While the disjoint algorithm works from the top down, circuit extraction works bottom up.

The extractor is implemented as a two phase process. In the first phase all the cells are scanned, finding transistors. For NMOS processes this is simply finding where a polysilicon box overlaps a diffusion box. Where this occurs, a transistor box is created, and the corresponding area is removed from the diffusion layer. Two different types of transistors are created, depletion or enhancement, depending on whether ion implant surrounds the transistor or not. A sweeping line algorithm is used to find these polysilicon-diffusion overlaps. This phase could not work with overlapping instances. Without overlap it is not even necessary to pass information between symbols.

The second phase of the algorithm now concentrates on finding the connectivity of the circuit. Connectivity within a symbol is a simple task, and can use the methods of conventional flat extractors. It is necessary, however, to consider connectivity information which may come from instances inside the symbol, and to pass this information to the outside. Consider the symbol shown in figure 8. In this figure, only the metal layer is shown. The symbol consists of two boxes, and an instance. One box touches the left side of the instance, the other box touches the right side of the instance. Suppose that inside instance A there is a bus which connects the two boxes. If this symbol is extracted without taking into account the connection made through A, these boxes will not appear electrically connected. Thus the connectivity information of the instances within symbols must be considered in this phase.

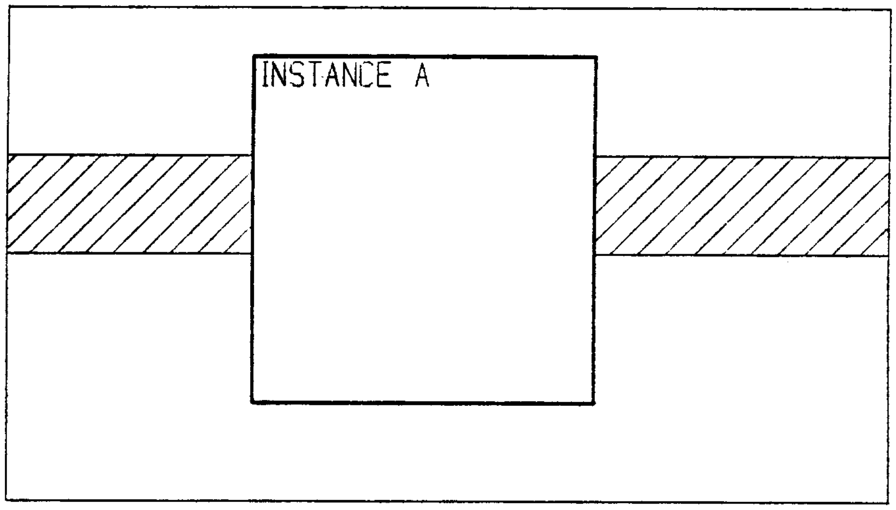

Figure VI-8. *symbol containing one instance and two boxes*

To pass connectivity information between cells, each symbol has a set of *interface segments*. Interface segments mark places along the symbol boundary where a connection can be made to the circuitry of the symbol from the outside. An interface segment is specified by a line segment, a layer, and a node number. Figure VI-9 shows the structure of an interface segment.

The first step in extracting a symbol is to check that each instance has had its corresponding symbol already extracted. If there is an instance whose symbol has not yet been extracted, the algorithm recurses and extracts that symbol. This check insures that all the instances have been extracted, and that they

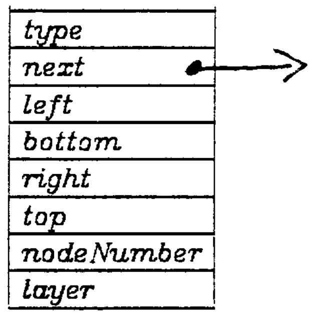

Figure VI-9. *data structure for an interface segment*

have a set of properly connected interface segments associated with them.

The next step is to instantiate all interface segments of the included instances. The instance's transformation is applied to the all interface segments in the symbol definition. This places the interface segment in its proper position in the coordinate system. A zero width (or height) box is created at this position, and its layer number is set to the layer number of the interface segment. The box is assigned a node number by assigning the node number of the interface segment plus an offset, called the *base node number*. (The base node number is initially 1.) The base node number is then set to the highest node number seen so far plus 1 and the next instance is examined.

Thus, every interface segment in every instance generates a zero width (or height) box. Boxes created from interface segments of different instances cannot be assigned the same node number. However, boxes created from interface segments of the same instance with the same node number will be assigned the same node numbers. This conveys the connectivity within an instance during the extraction phase.

After these boxes have been generated, interface segments to the outside must be created. To do this, all the boxes, including the zero width and zero height boxes created from interface segments, are checked to see if they touch the symbol's boundary. For any box that does touch the boundary, an interface segment is created along the line segment where the box touches the boundary. If the box already has a node number assigned to it, that node number is also assigned to the interface segment. If there is no node number associated with the box, a new one is generated and assigned to both the box and the interface segment.

The next step is the actual connectivity extraction. A sweeping line algorithm is used to extract the connectivity of the symbol. Care must be taken in coding this algorithm so that the zero width (and height) boxes are not ignored. By associating node numbers with these boxes, connections between the geometry and interface segments, and connections among interface segments will be recognized. If two nodes with two different node numbers are found to be connected, their node numbers are merged.

Merging is used to keep track of equivalence relationships among node numbers. If two node numbers are in the same equivalence set, then their corresponding nodes are connected. The function *merge* takes two node numbers as parameters and combines their equivalence sets. The function *lookup* takes a node number as a parameter, and returns the smallest node number in the equivalence set that contains the parameter. With this function, it is possible to tell whether two nodes are connected by testing whether *lookup* returns the same value for each of their node numbers. Thus, equivalent node numbers are all represented by the smallest number in the equivalence set.

The equivalence relationship is maintained in an array of the node numbers which is indexed by node numbers. This array maintains the property that if node number i is not the smallest node number in its equivalence set, then the i^{th} element of the array contains a node number in i's equivalence set smaller than i. If node number i is the smallest node number in its equivalence set, then the i^{th} element of the array contains i. Initially each element of the array is set to its own index.

Lookup is implemented as follows. For a given node number, that element in the array is checked. If it contains itself, then it is the smallest element in its equivalence set, and the node number is returned. Otherwise, continue this process with the number found in this slot. Since each element of the array contains a number smaller or equal to its index, the recursion must stop. Thus, *lookup* always returns the smallest node number in the equivalence class.

Merge is implemented as follows: Given two node numbers a and b, call *lookup* on these node number to get a' and b'. If a' equals b', nothing more needs to be done. If they are not equal, set the entry for the larger number equal to the value of the smaller one. This will cause all subsequent calls to *lookup* with either a' or b' to return the smaller value.

Once the extraction algorithm has swept the entire symbol, equivalence relationships have been set up between node numbers. The nodes are then renumbered, assigning a unique number to each equivalence set. The numbering is done so that the interface segments' node numbers are given the lowest numbers. Nodes which do not have interface segments (called *internal nodes*) are given higher numbers.

Finally, for every interface segment of every instance an equivalence number is assigned. This step is necessary to allow the circuit to be reconstructed later on.

Chapter VII. Performance

This chapter discusses the performance of the disjoint and circuit extraction routines. The data presented here are based on 5 layouts, *adder*, *cherry*, *ralu*, *fifo*, and *testram*. *Adder* is an eight bit adder. It is part of a larger layout, and does not contain interface circuitry or bonding pads. *Cherry* is a 4x4 bitmap manipulator, *ralu* is a 16-bit arithmetic-logic unit, and *fifo* is a first-in first-out buffer; all these are complete layouts. Finally, *testram* is a 8Kbit ram. This is again a partial layout, and it also has the property that no cell is rotated or mirrored. The description of these circuits was available in the Caltech Intermediate Form (CIF 2.0).

Table VII-1 lists for each layout the number of transistors, the number of specified rectangles, the number of rectangles in the fully instantiated layout, and the regularity. Specified rectangles are counted by considering the layout where each symbol is expanded only once. The number of rectangles in the fully instantiated layout are counted by considering the layout where each symbol is expanded each time it is called. Regularity is computed here by dividing the number of rectangles in the fully instantiated layout by the number of specified rectangles. This definition is similar to that given by [10] but is easier to compute, and is more natural to apply after the layout geometry has been transformed.

Table VII-1: *Size and Regularity of Layouts*

name	*transistors*	*specified rectangles*	*total rectangles*	*regularity*
adder	203	662	2017	3.0
cherry	881	563	7416	13.2
ralu	1853	2890	21583	7.5
fifo	1927	86120	86120	44.7
testram	20480	160	196992	1231.2

Running the disjoint algorithm over the layouts changes the hierarchical structure of the layouts. Table VII-2 shows the effect on the number of specified and total rectangles and on the regularity of the layouts, after they were run through the Mesa and C versions. The differences between the Mesa and C versions are due to the fact that the C version recognizes rotations and mirroring of symbols, whereas the Mesa version considers rotated and mirrored instances of a symbol to be different symbols. This difference shows itself in the number of

Table VII-2: *Effect of Disjoint Transformation*

	before disjoint			after disjoint					
				Mesa Version			C Version		
name	*spec rects*	*total rects*	*reg*	*spec rects*	*total rects*	*reg*	*spec rects*	*total rects*	*reg*
adder	662	2017	3.0	990	2147	2.2	876	2147	2.4
cherry	563	7416	13.2	3972	11274	2.8	3763	11282	3.0
ralu	2890	21583	7.5	7368	21947	3.0	5200	21961	4.2
fifo	1927	86120	44.7	8248	117202	14.2	4079	117213	28.7
testram	160	196992	1231.2	577	264076	457.7	577	264076	457.7

specified rectangles. While in many cases the difference is small, in some layouts that contain many mirrored cells, such as *ralu* and *fifo*, the difference is significant.

Figures VII-1 through VII-5 show the fully instantiated layouts, the layouts with each symbol instantiated once, and the layout with each disjoint symbol instantiated once.

Table VII-3 compares the runtimes of the Mesa and C versions of the extractor. The Mesa times are for a Dorado computer, and the C times are for a VAX-11/750. Times are reported in minutes:seconds.

Table VII-3: *Runtimes of Mesa vs. C Version of Extractor*

	Mesa Version		C Version	
name	*disjoint transform*	*extraction*	*disjoint transform*	*extraction*
adder	:01	:13	:06	:52
cherry	:08	2:53	:47	12:30
ralu	:05	1:48	:34	4:20
fifo	:09	2:51	:52	14:04
testram	:04	5:42	:28	13:30

The C version of the disjoint transformation takes approximately 6 times as long as the Mesa version, whereas for the extractor the C version takes approximately 3 times as long as the Mesa version. (Note that the Dorado is a much faster machine than a VAX 11/750.) The C version puts more effort into the disjoint operation trying to recognize mirror and rotated symbol combinations. This pays off at extraction time since fewer symbols need to be analyzed.

It is interesting to compare how the hierarchical extractors compare with flat extractors. Table VII-4 compares the two hierarchical extractors with two flat extractors, one written in Mesa and one written in C[6]. Again, the times for the Mesa programs are for a Dorado computer, and the times for the C programs are for a VAX-11/750.

Table VII-4: *Runtimes of Mesa vs. C Version of Extractor*

	Flat Extractors		Hierarchical Extractors	
name	*Mesa Version*	*C Version*	*Mesa Version*	*C Version*
adder	1:33	:27	:13	:52
cherry	2:37	1:40	2:53	12:30
ralu	10:33	3:57	1:48	4:20
fifo	26:54	16:03	2:51	14:04
testram	62:48	20:51	5:42	13:30

The C flat extractor is highly optimized. This explains why it does so well compared to the other extractors listed in the table. Note that although this optimized extractor does well against the hierarchical extractors for small layouts, the hierarchical extractors start to do better as the size of the layout increases.

The runtimes for the hierarchical extractor show very little correlation between the runtime and the regularity or the number of rectangles. Analysis of the algorithm shows that the runtime is very dependent on the number of interface segments which the extractor must look at. Let us assume that connectivity extraction can be done in linear time with respect to the number of rectangles and interface segments. Further, let us assume that the number of interface segments in a symbol is proportional to the perimeter of the symbol. Then, if symbol x is made up of k calls to symbol i, the expected time to extract symbol x is $O(kp_i)$, where p_i is the perimeter of symbol i. If c_i is the number of calls to symbol i in the disjoint hierarchy, then i contributes $O(c_i p_i)$ to the extraction time. Thus the contribution of all the interface segments to the extraction time is O(P), where P is the sum of all the $c_i p_i$'s. Let R be the number of rectangles in the disjoint hierarchy, then the total expected time for extraction is O(P) + O(R). We have seen that there is little correspondence between the number of rectangles in the hierarchy and runtime. Table VII-5 shows runtimes of the programs versus the number of rectangles, R, and versus the total perimeter, P.

This table indicates that the runtime is much more dependent on the total perimeter than on the number of rectangles. Figures VII-6a and VII-6b show graphs of total perimeter versus runtime and number of rectanges versus runtime for the MESA and C extractors. These graphs show that total perimeter is a

Table VII-5: *Number of Rectangles and Total Perimeter vs. Runtimes*

	Mesa Version			C Version		
name	*# of rects*	*total perimeter*	*runtime*	*# of rects*	*total perimeter*	*runtime*
adder	990	18	:13	876	19	:52
cherry	3972	425	2:53	3763	434	12:30
ralu	7368	181	1:48	5200	184	4:20
fifo	8248	303	2:51	4079	297	14:04
testram	577	464	5:42	577	471	13:30

good indication of what the runtime will be for a given layout. The total perimeter, of course, is not known until the disjoint hierarchy has been obtained. Table VII-3 indicates, though, that the disjoint transformation takes only a small fraction of the total time for extracting a layout. Thus, an estimate of the runtime for a given layout can be made relatively quickly.

The dependence on total perimeter indicates that most of the extractors time is spent on examining the interfaces between instances, very little time is actually spent examining the geometry of the layout.

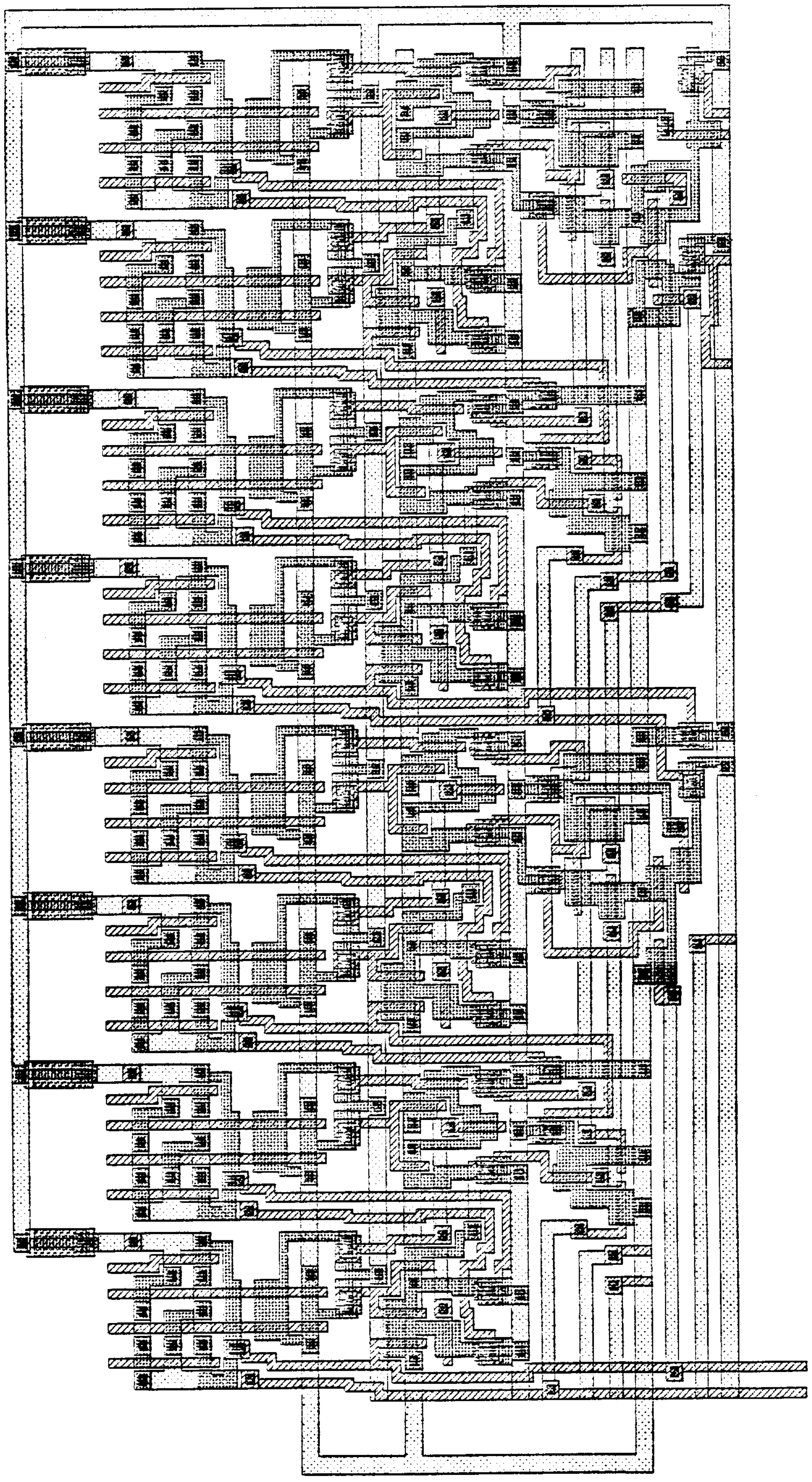

Figure VII-1a. *fully instantiated layout of 'adder'*

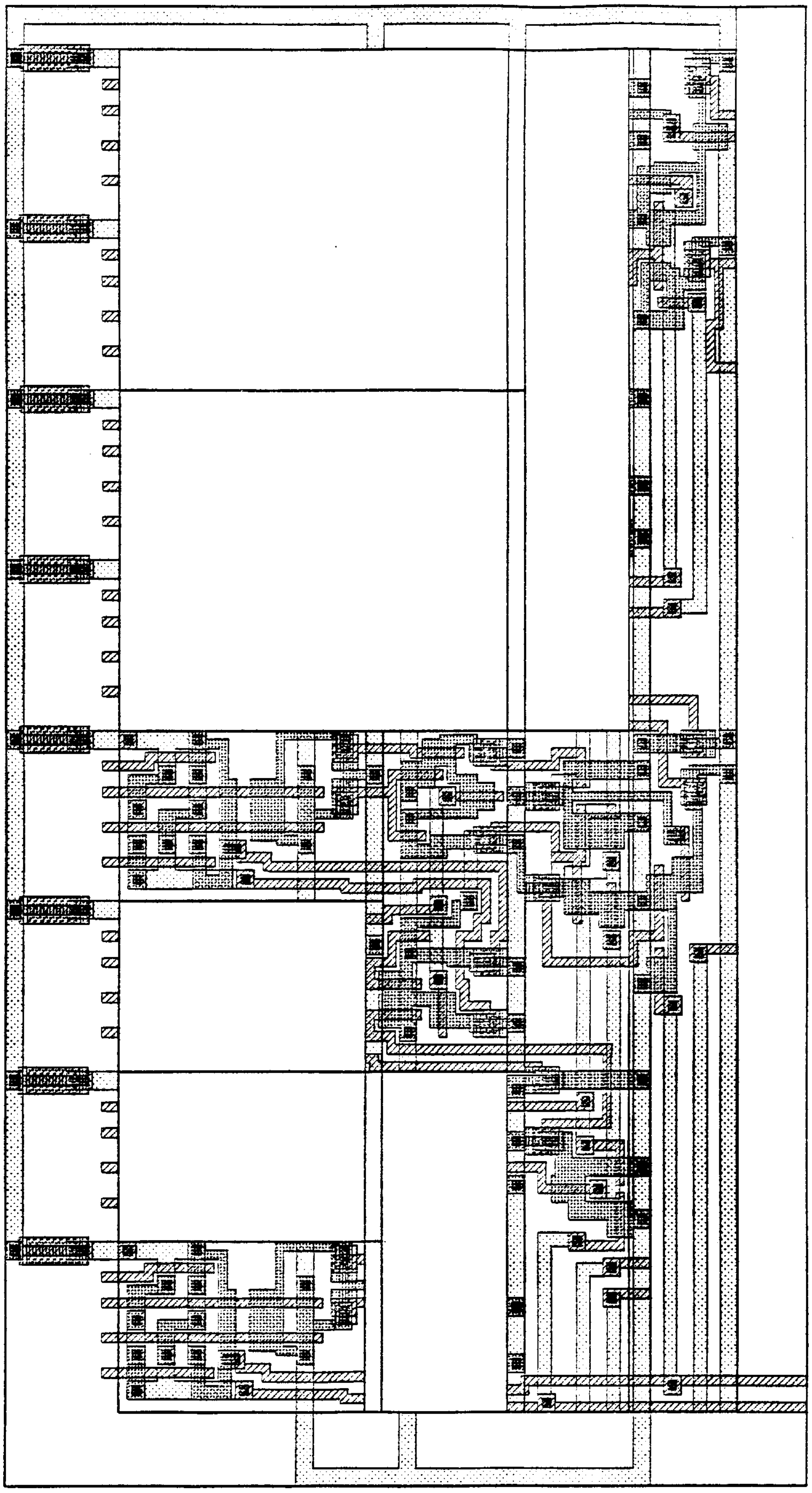

Figure VII-1b. *layout of 'adder' with each disjoint symbol drawn only once*

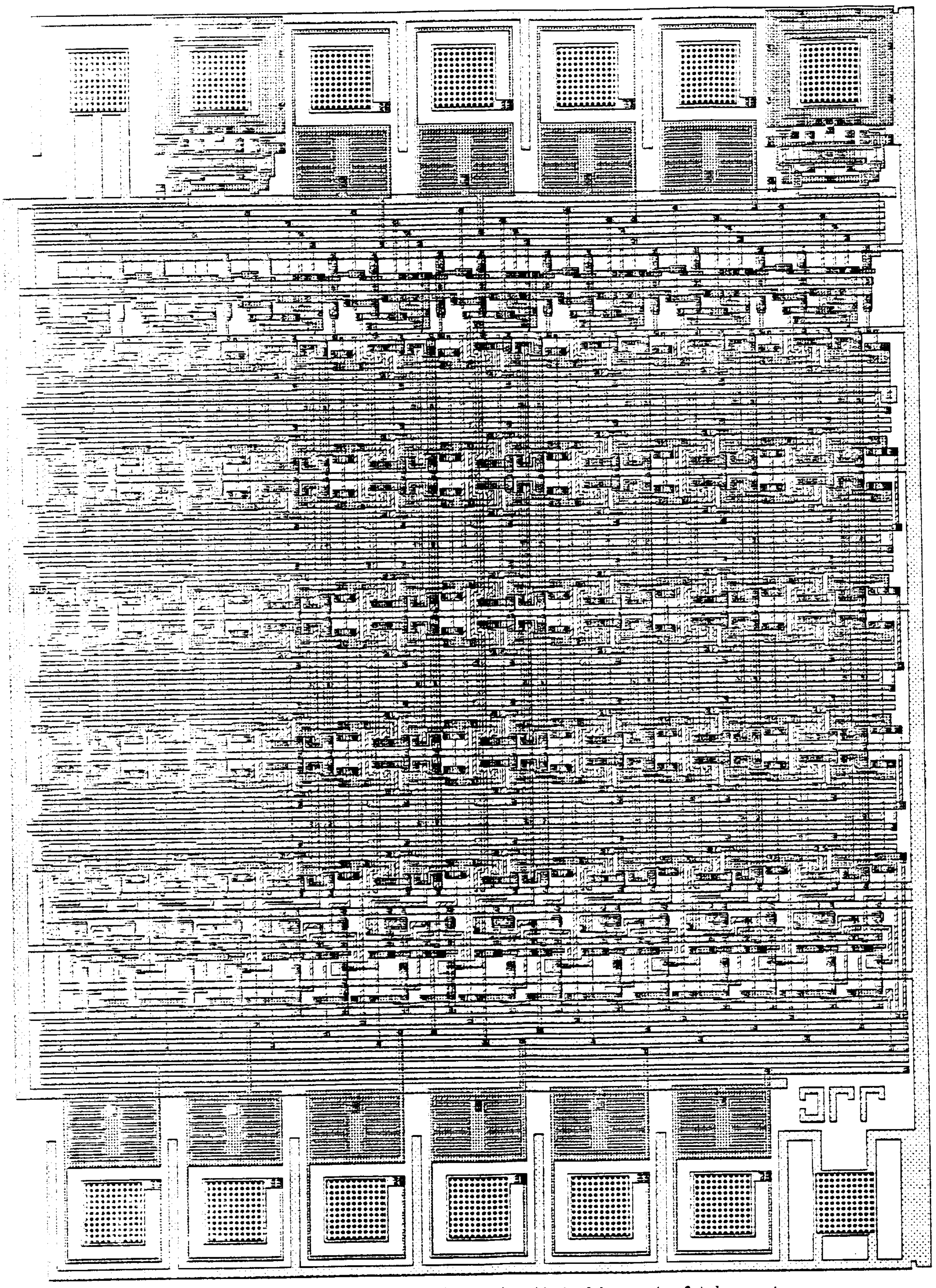
Figure VII-2a. *fully instantiated layout of 'cherry'*

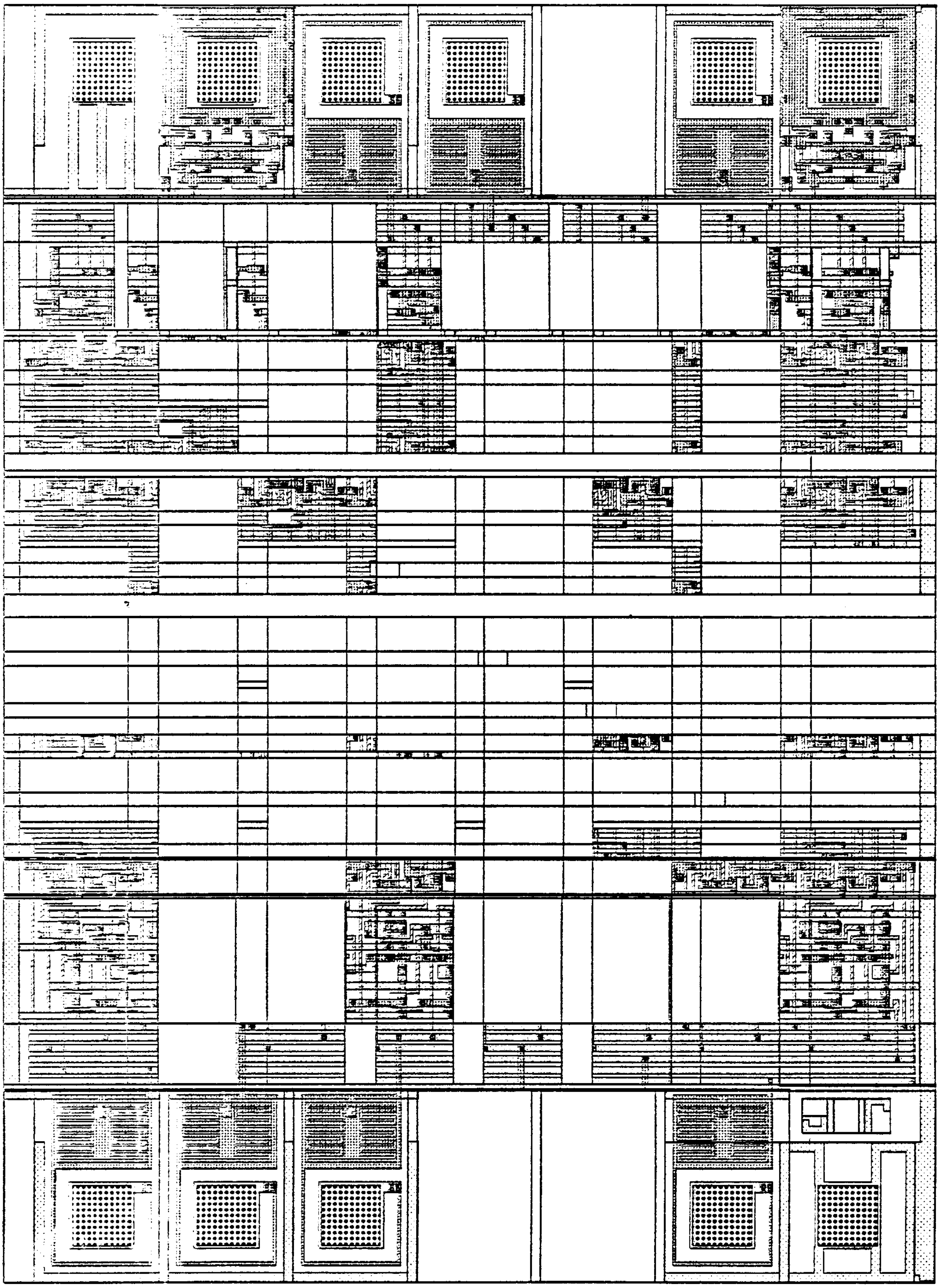

Figure VII-2b. *layout of 'cherry' with each disjoint symbol drawn only once*

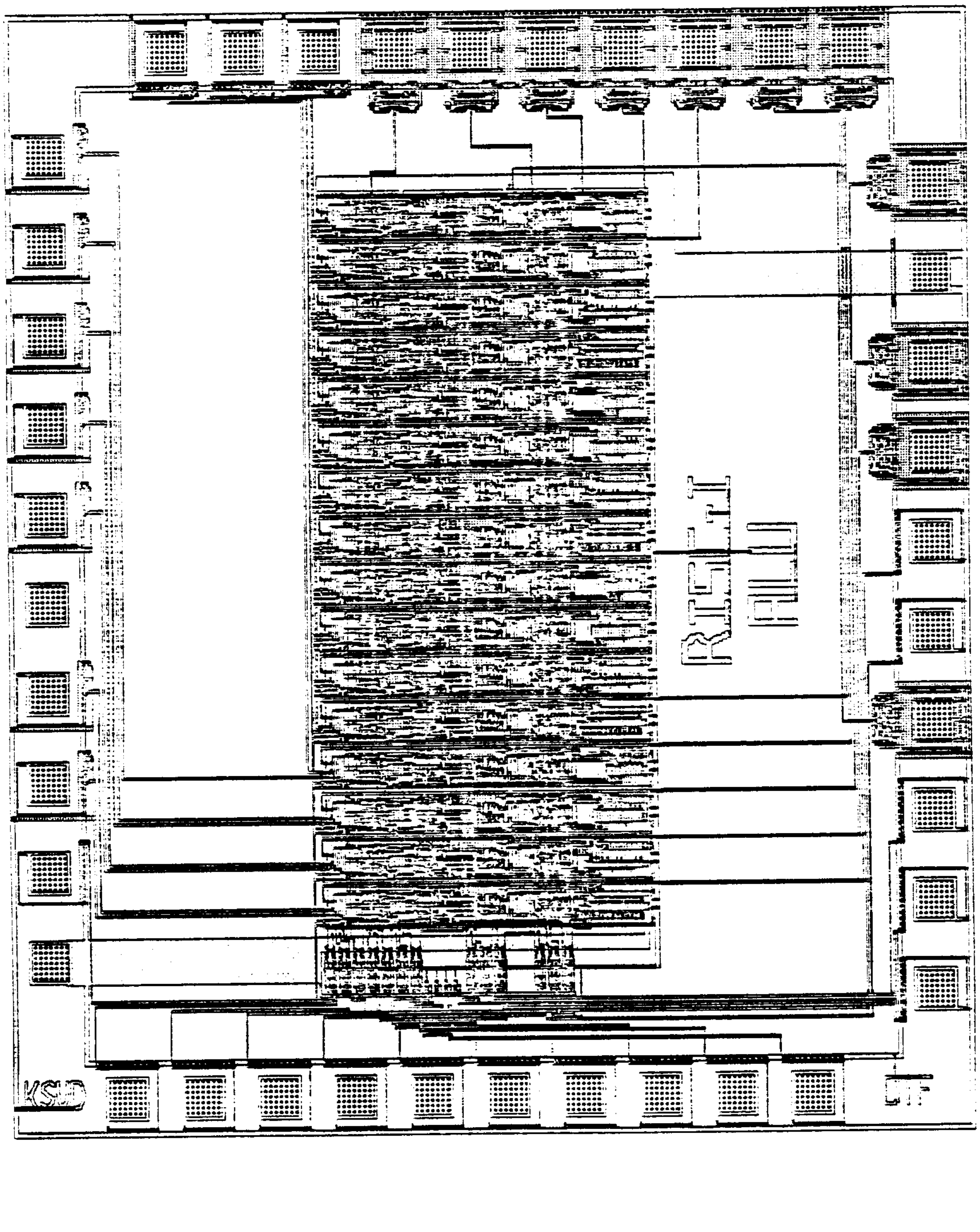

Figure VII-3a. *fully instantiated layout of 'ralu'*

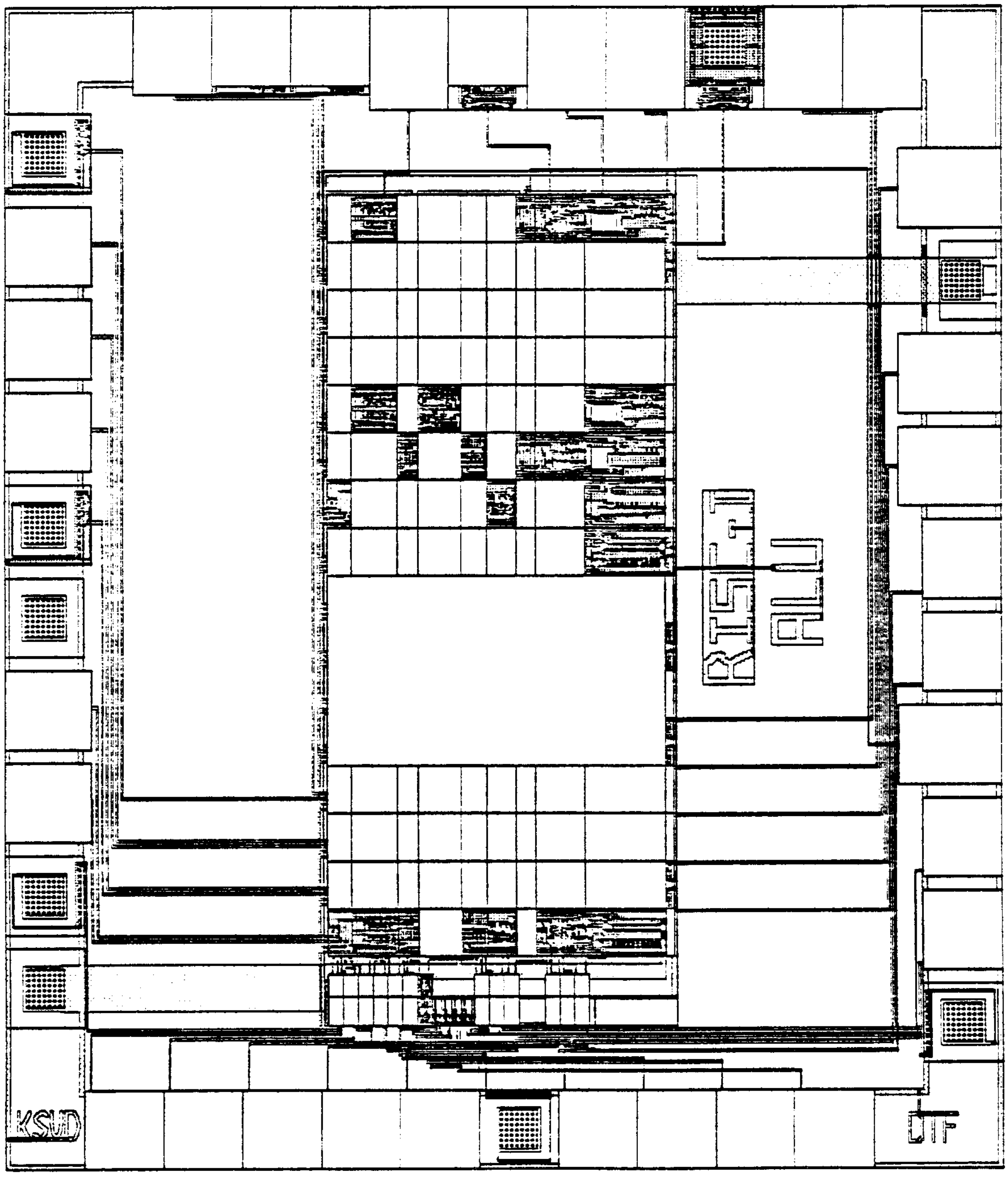

Figure VII-3b. *layout of 'ralu' with each disjoint symbol drawn only once*

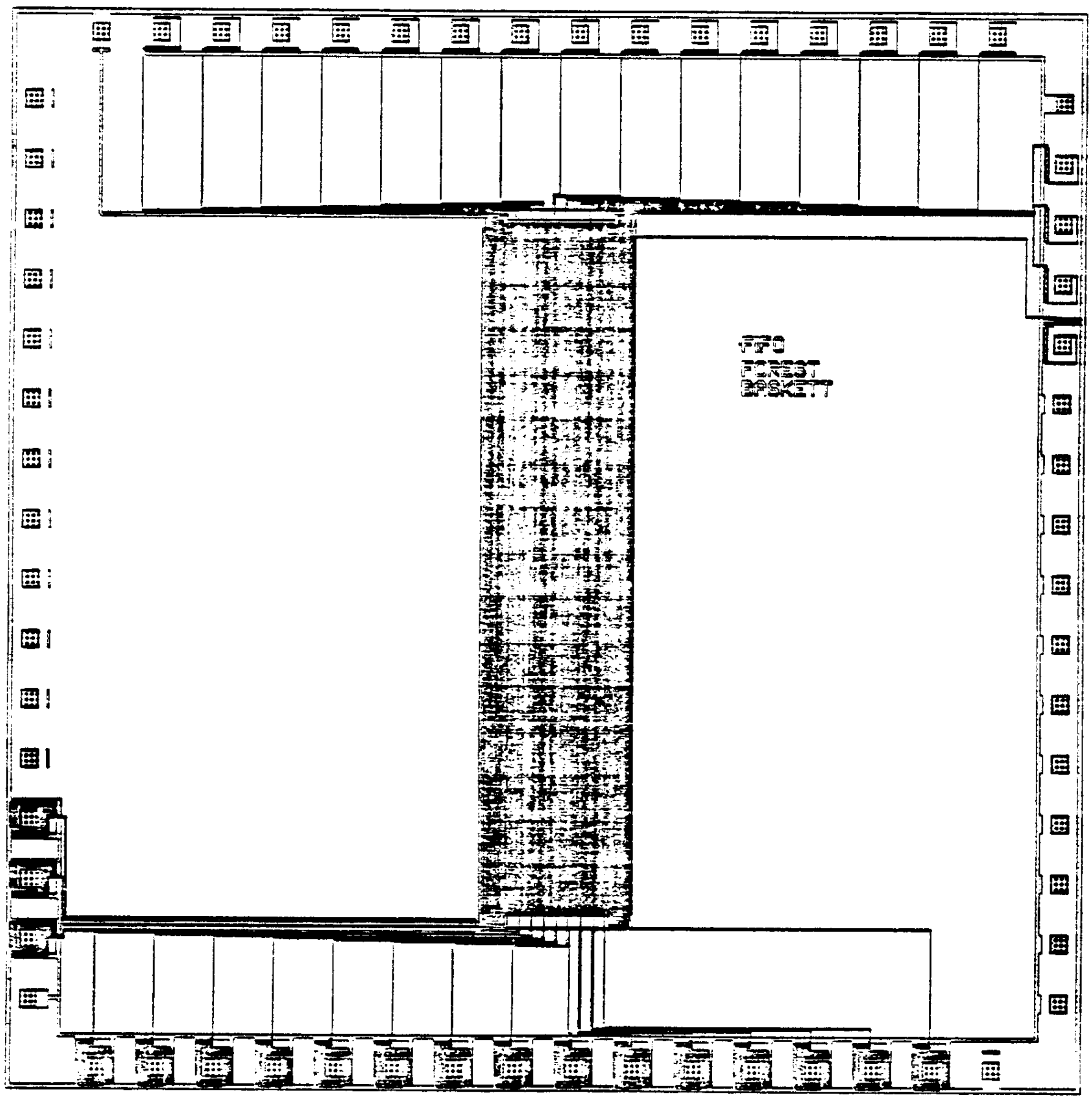

Figure VII-4a. *fully instantiated layout of 'fifo'*

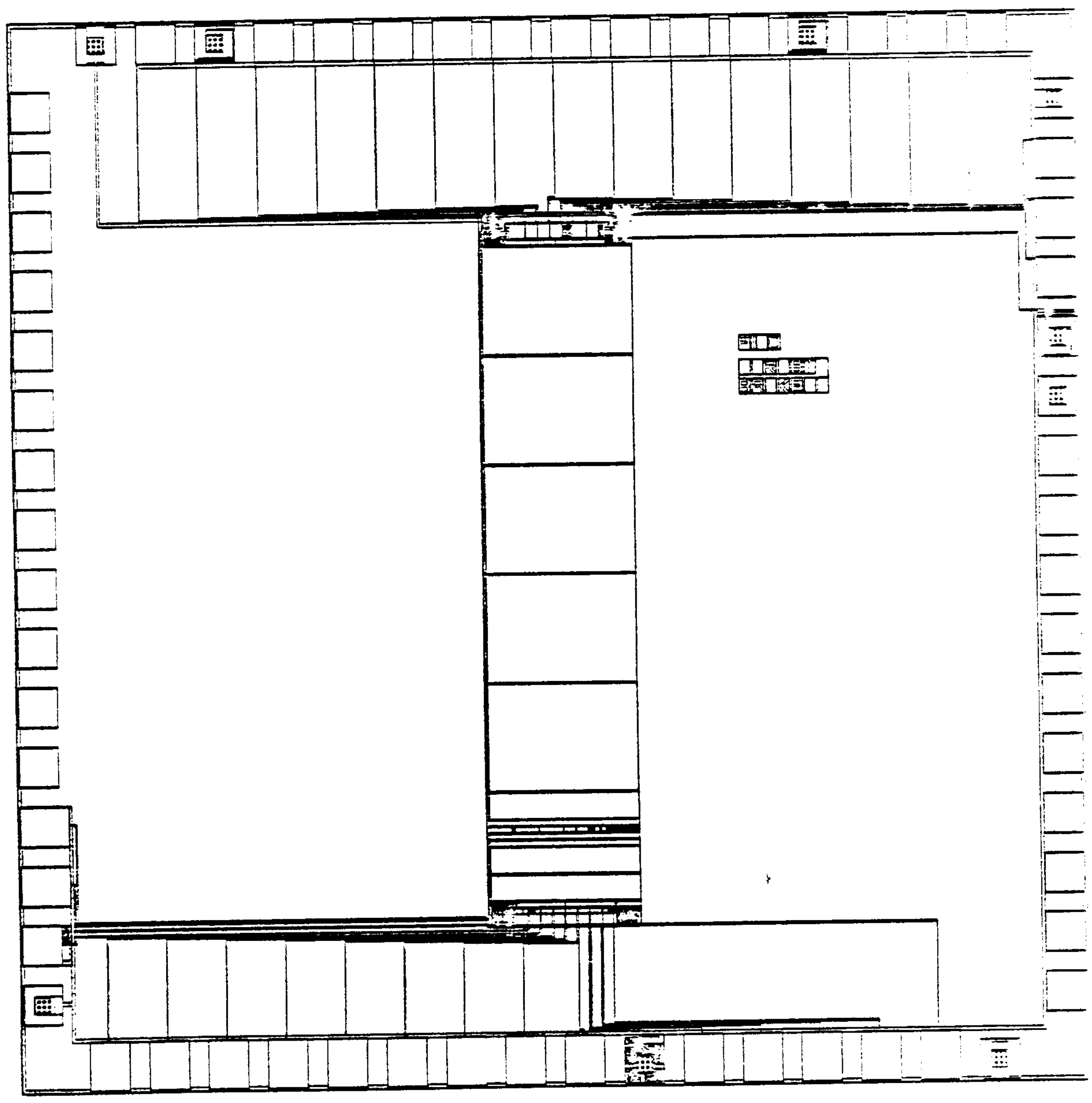

Figure VII-4b. *layout of 'fifo' with each disjoint symbol drawn only once*

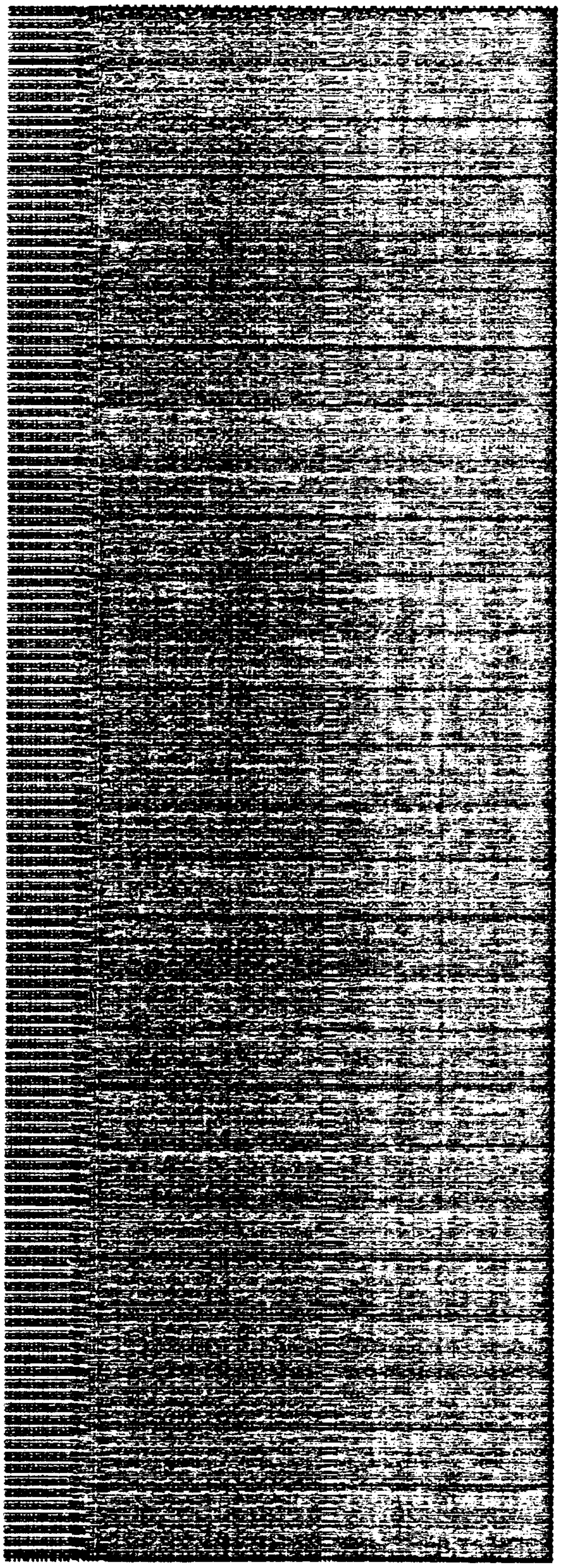

Figure VII-5a. *fully instantiated layout of 'testram'*

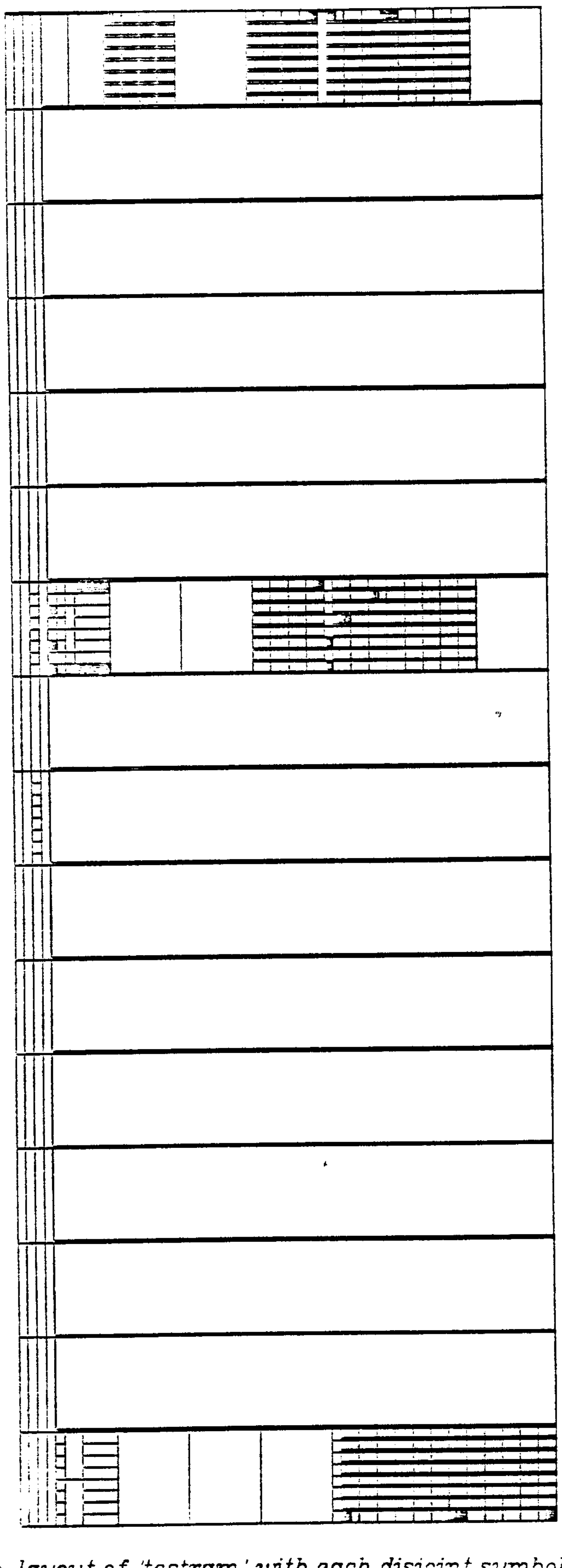

Figure VII-5b. *layout of 'testram' with each disjoint symbol drawn only once*

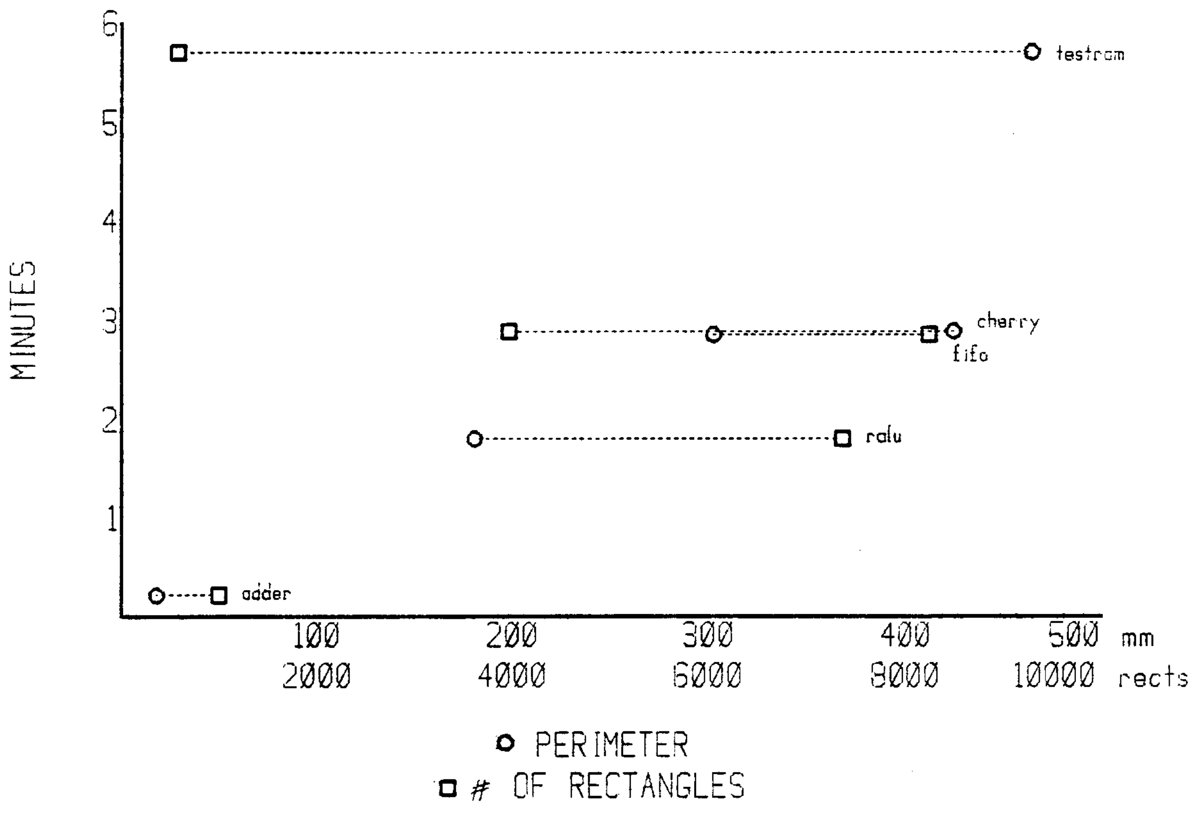

Figure VII-6a. *total perimeter vs. runtime for MESA extractor*

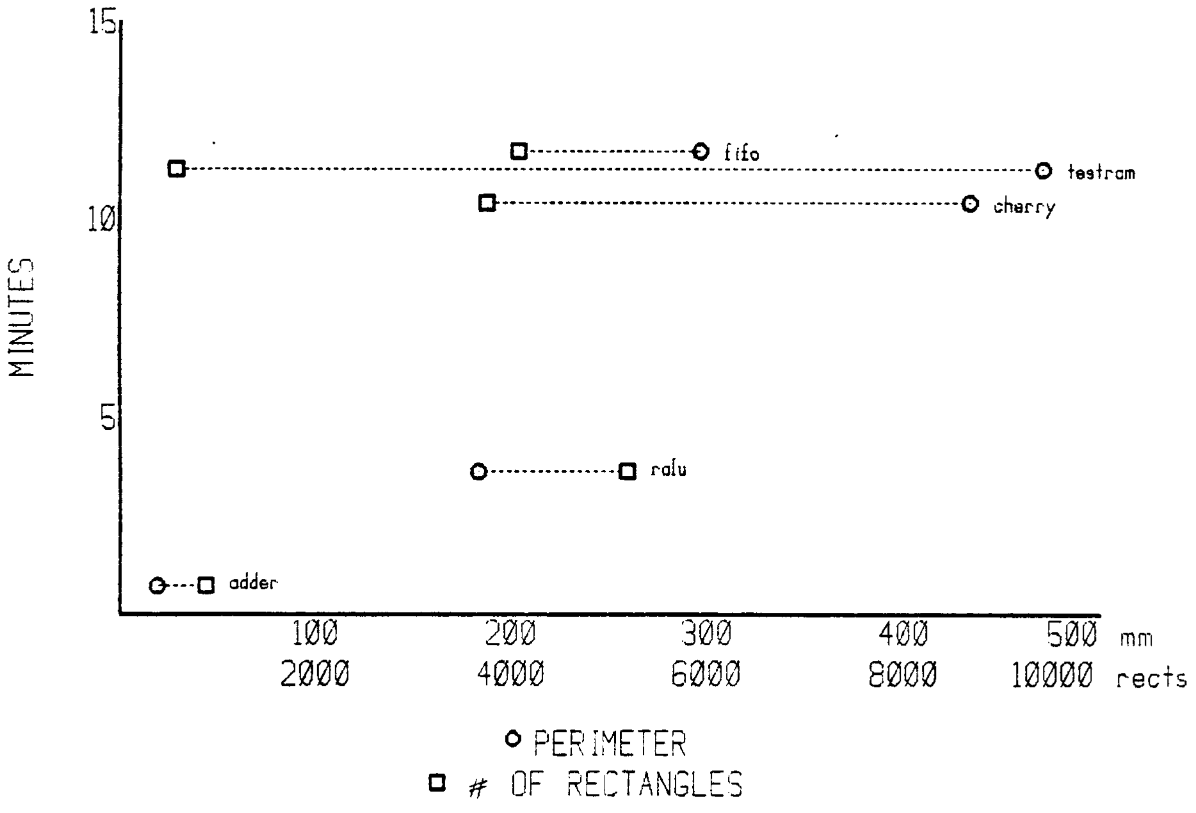

Figure VII-6b. *total perimeter vs. runtime for C extractor*

Chapter VIII. Work by Others

In the last few years a considerable amount of work has gone into finding ways to exploit the hierarchical structure of VLSI layouts. This chapter reviews some of that work and compares it with the approach taken in this thesis.

1. Hierarchical Filter

Whitney[21] developed an algorithm to do hierarchical design rule checking (DRC). Her algorithm reads a hierarchical layout description, and produces a flat description which can be run through a conventional design rule checker. The algorithm acts as a filter, passing only a subset of the geometry to the actual checking program. Repeated configurations of symbols are not generated. Thus, the algorithm's flat description has far fewer rectangles to check than the actual flat description of the layout, so it can be checked much faster.

Whitney's method is based on pairwise comparison of instances within a symbol. The algorithm starts by checking the top symbol of the layout. Each instance is examined to see if its symbol has been checked. If it has not been checked, the algorithm recurses and checks that symbol. After all the instances have been examined, all geometry at that level that needs to be checked is generated.

Next, interactions between instances and geometry, and between instances and other instances are checked. This is done by bloating the bounding boxes of the instances and geometry by one half of the maximum extent of the design rules. For each instance, its bloated bounding box is checked to see if it overlaps the bloated bounding box of any geometry or other instance. If the bounding boxes do overlap, the program checks whether it has checked this configuration before. This configuration is ignored if it has been checked it before. If not, the geometry in the overlapped region is generated and this configuration is recorded as checked.

Checked configurations are recorded by storing with each instance's symbol, the other instance's symbol and the instance's relative transformation. For example, in figure VIII-1 instance **a**'s bloated bounding box overlaps instance **b**, where **a** is an instance of symbol **A** with the transformation $\mathbf{T_a}$, and **b** is an instance of symbol **B** with the transformation $\mathbf{T_b}$. Thus, with symbol **A** the configuration $(\mathbf{B}, \mathbf{T_a}^{-1}\mathbf{T_b})$ is stored, and with symbol **B** the configuration $(\mathbf{A}, \mathbf{T_b}^{-1}\mathbf{T_a})$ is stored.

There are problems with this approach, however. One problem is that the algorithm does not generalize to other layout analysis operations such as circuit extraction or mask operations. A more serious problem is that one can construct pathological cases where this algorithm fails to report actual design rule violations. Figure VIII-2 is an example of where this algorithm fails for simple Mead and Conway design rules. Figures VIII-2a through VIII-2c shows symbols **A**, **B**, and **C**. Figure VIII-2d shows symbol D, which is composed of two instances of symbols **A** and **B**, and one instance of symbol **C**. If the algorithm checks the bottom cluster first, it will output the **A-B-C** cluster finding no design rule violations. When the algorithm sees the top cluster, it will not output anything since **A** and **B** have been seen in an identical configuration. Thus, the design rule violation caused by the absence of the diffusion layer of symbol **C** is not caught. Whether cases such as the above arise in actual layouts I do not know, but their possible existence makes this algorithm less attractive.

2. Front-End Processor

Hon[8] has considerably extended Whitney's algorithm to make it applicable to other applications such as circuit extraction. His algorithm concentrates on a front end processor for hierarchical analysis. The front end processor is responsible for dividing the layout into non-overlapping rectangular regions called *windows*. Different analysis modules may be plugged into this front end processor. The front end processor requires each analysis module to implement two operations: analyze a single window, and compose two previously analyzed non-overlapping windows.

The front end processor uses a heuristic algorithm to divide the layout into non-overlapping windows. The algorithm starts by creating a window that contains the top level symbol. This window must now be broken into sub-windows. A window is divided by first checking to see if any instance completely covers the window. If there is such an instance it is expanded and replaced with its component instances and geometry. Now the window is checked to see if any of the instances within it are overlapping. The instances within the window are divided into *groups*. A group is two or more instances that overlap. If a group is found whose bounding box does not overlap any other group, a subwindow is created at the group's bounding box. If a group's bounding box completely surrounds one or more other groups, a sub-window is created at the bounding box of that group. The instances and geometry that are contained in these newly created sub-windows are removed from the original window.

If any instances are left that overlap in the window, an instance is selected to be expanded. Various heuristic techniques may be used to pick which instance to expand. A heuristic that works well in most cases is to pick the instance overlapping the most other instances. After the selected instance has been expanded, it is necessary to repeat the checks for groups.

At the end of this process the layout will have been divided into a hierarchical set of windows. Each window will contain only non-overlapping instances and geometry. The front end processor will then call upon the analysis module to analyze each window, and to compose adjacent windows. This process continues until the highest level window has been analyzed.

This technique for dividing up the check provides a function similar to the disjoint transformation. This method does take more time than the disjoint algorithms but it usually produces fewer cells that need to be examined by the back end processor. Although both Hon's technique and the disjoint transformation change the hierarchy, the disjoint transformation is driven solely by the hierarchical structure of the original layout. This results in a derived hierarchy that is closer to the original layout.

3. Restricting Overlaps

Scheffer[17] describes a system which exploits hierarchy by disallowing overlapping cells. He shows that by allowing the circuit designer to specify a polygonal outline of his cells, no area loss results. A layout with no overlapping cells has several advantages for the analysis tools. Besides easing the job of circuit extraction and design rule checking for the entire layout, incremental checking can be performed at the end of each editing session with a cell. This is extremely valuable to the circuit designer in that errors can be reported while the design is still fresh in his mind.

Although disallowing overlaps results in no area loss, it can complicate the design process. The designer is forced either to avoid sharing busses between adjacent cells or to contort his design so that only half the shared bus resides in each cell. Tucker and Scheffer[19] relax the constraint by allowing the designer to overlap regions along the perphiery of the cell as long as certain restrictions are met, such as no transistors are formed by the overlap. (A similar approach is used by Weise[20].) By making suitable restrictions on the overlapping areas they show that most of the advantages claimed for non-overlapping cells remain. This approach is found much more acceptable to circuit designers.

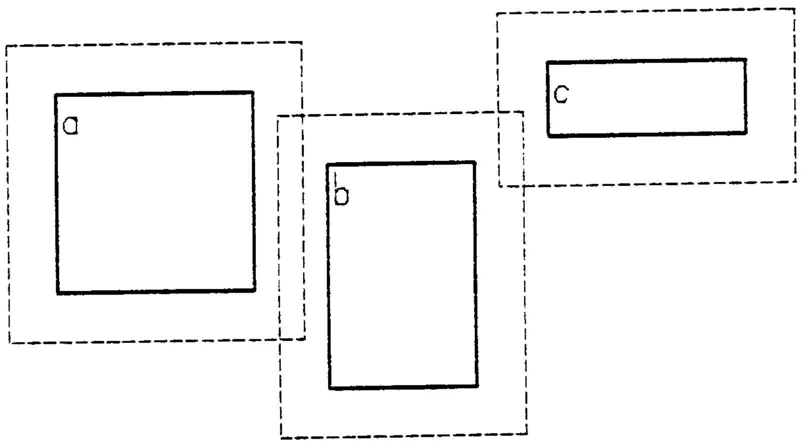

Figure VIII-1. *bloated bounding boxes of instances a, b, and c*

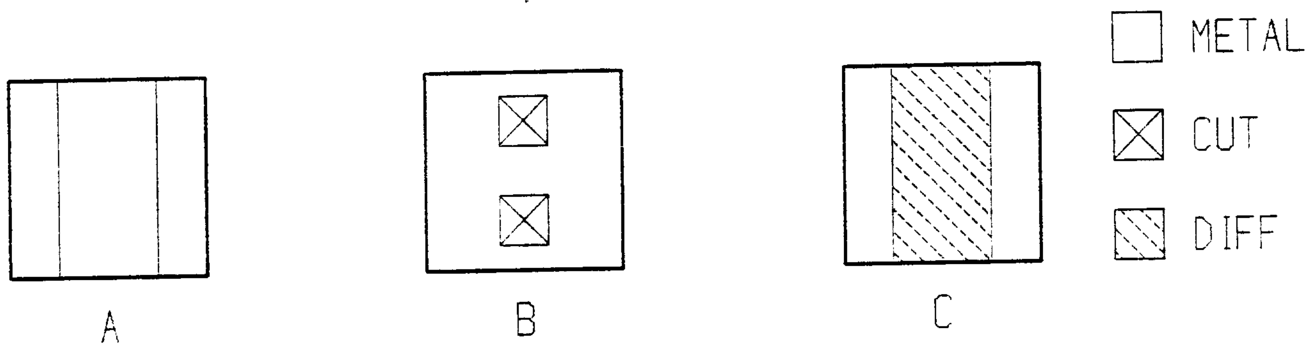

Figure VIII-2a. *symbols A, B, and C*

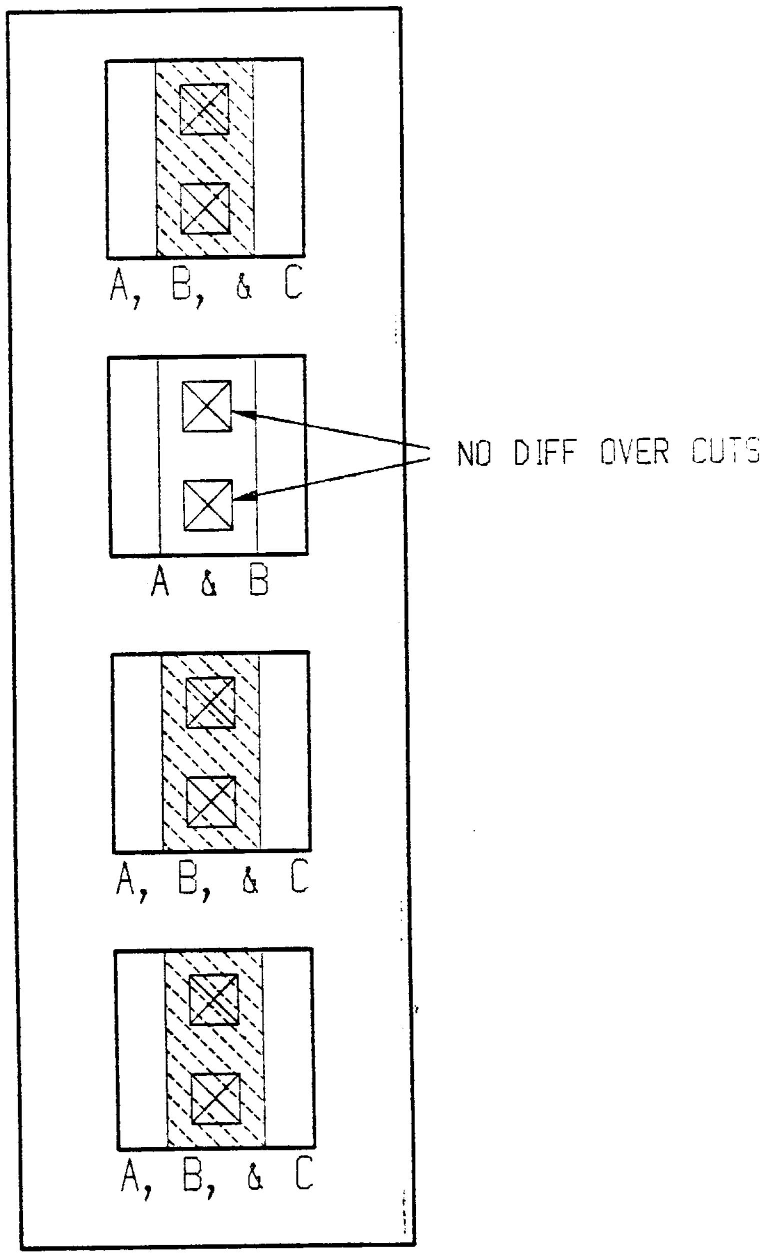

Figure VIII-2b. *design rule violation in second cluster may be missed*

Chapter IX. Summary

Over the past few years much progress has been made in integrated circuit technology. Better fabrication techniques have made it possible to fit close to a million transistors onto a single integrated circuit. The problems of designing a circuit with this many devices have led chip designers to adopt structured design approaches. Analyzing layouts of this size has presented a major problem, though. The conventional approach of flatting the symbol hierarchy, and thereby ignoring all structure present in integrated circuit layouts, is doomed to fail as the size of circuits continue to increase. Layout analysis tools must take advantage of the structure of the layout to solve the problem of analyzing layouts of this complexity.

This thesis has discussed approaches to exploiting the structure of integrated circuit layouts. The main problem in doing this is the presence of overlapping instances. The disjoint transformation solves this problem by transforming a hierarchical integrated circuit layout description, which may contain overlapping instances, into an equivalent hierarchical layout description that does not contain any overlapping instances.

Once the overlap has been removed from the layout, the analysis tools can proceed much more efficiently. An algorithm hierarchical circuit extraction is discussed. By creating connection points along the boundary of each symbol this extractor is able to maintain connectivity information through each level of the hierarchy. The output of this extractor is a hierarchical description of the circuit. An algorithm that performs conventional design rule checking based on only local information is also presented. This algorithm works for generally shaped polygons and is not limited to manhattan shapes. This algorithm is then extended to check layouts hierarchically. Algorithms for performing the general mask operations of AND, OR, NOT, GROW, and SHRINK hierarchically are also presented.

Programs that hierarchically extract circuits have been implemented in the MESA and C programming languages. Comparing the performance of these programs to conventional flat extractors shows that while the flat extractors do well on smaller circuits, the hierarchical extractors perform better on larger circuits. The runtimes of the hierarchical extractors were shown to be mostly dependent on the total perimeter of the instances in a layout, while the runtimes of the flat extractor were dependent more on the total number of rectangles in the fully instantiated layout. The disjoint transformation took only a small fraction of the total time for extracting the circuit.

The benefits of exploiting the structure of layouts are more than just faster runtimes for layout analysis tools. The data files created by hierarchical tools can be much smaller, because the data files can be made hierarchical. Smaller data files, in turn, mean that less time is spent reading these files by programs which operate on them. These data files will more likely be meaningful to the chip designers since their structure is much closer to the structure that the designer created for his layout.

Another benefit of hierarchical analysis tools is the possibility for incremental analysis. In conventional layout analysis systems the whole layout must be rechecked if the designer makes a change to any part of the layout. With a hierarchical system, only the symbols that change, and the symbols that reference these symbols, need to be checked. This can result in great saving near the end of the design as the designers fine-tune parts of the layout. Small changes will mean only those parts of the design affected by changes will be checked instead of the entire layout.

References

1. M. H. Arnold and J. K. Ousterhout, "Lyra: A New Approach to Geometric Layout Rule Checking", *Design Automation Conf.*, 1981

2. M. H. Arnold, "Research in Corner-Based Design Rule Checking", PhD. thesis proposal, U.C. Berkeley, 1982

3. H. S. Baird and Y.E. Chao, "An Artwork Design Verification System", *Proc. 12th Design Automation Conf.*, 1975.

4. C. M. Baker and C. Terman, "Tools for Verifying Integrated Circuit Designs", *Lambda Magazine*, fourth quarter 1980.

5. J. W. Beyers, *et al*, "A 32b VLSI CPU Chip" ISSCC Digest of Technical Papers, 1981.

6. D. Fitzpatrick, "MEXTRA: A Manhattan Circuit Extractor", ERL Memo M82/42, U.C. Berkeley, 1982.

7. R. Hon and C. H. Sequin, "A Guide to LSI Implementation", Xerox PARC, 1980.

8. R. Hon, "The Hierarchical Analysis of VLSI Designs", PhD. thesis proposal, VLSI Memo V073, CMU, 1981.

9. B. Lampson and K. Pier, "A Processor for a High Performance Personal Computer", *Proc. 7th Symp. on Computer Architecture*, 1980.

10. W. Lattin, "VLSI Design Methodology: The problem of the 80's for microprocessor design", *Proc. Caltech Conf. on VLSI*, 1979.

11. C. Mead and L. Conway, *Introduction to VLSI Systems*, Reading MA: Addison-Wesley, 1980

12. M. Newell and D. Fitzpatrick, "Exploiting Structure in Integrated Circuit Design Analysis", *Proc. Conf. on Advance Research in VLSI*, 1982.

13. M. Newell and C. H. Sequin, "The Inside Story of Self-Intersecting Polygons", *LAMBDA*, Second Quarter 1980.

14. W. M. Newman and R. F. Sproul, *Principles of Interactive Computer Graphics, Second Edition*, McGraw-Hill 1979.

15. J. Nievergelt and F. Preparata, "Plane-Sweep Algorithms for Intersecting Geometric Figures", *CACM*, October 1982.

16. J. Rowson, "Understanding Hierarchical Design", PhD. thesis, Caltech, 1980.

17. L. Scheffer, "A Methodology for Improved Verification of VLSI Designs Without Loss of Area", *Proc. Second Caltech Conf. on VLSI*, 1981.

18. C. H. Sequin, "Generalized IC Layout", *VLSI '81*, 1981

19. M. Tucker and Lou Scheffer, "A Constrained Design Methodology for VLSI", *VLSI Design*, May/June 1982.

20. D. Weise, "Hierarchically Based Analysis Tools for Computer Assisted Design of VLSI Ciruits", Masters Thesis, MIT, May 1982.

21. T. Whitney, "A Hierarchical Design Rule Checking Algorithm", *LAMBDA*, First Quarter 1981.