Hierarchical Analysis of IC Artwork with User-Defined Rules

Louis K. Scheffer and Ronny Soetarman
Valid Logic Systems, Inc.

Hierarchical design rule check and component extract offer many advantages, but no single form of hierarchical analysis fits all situations. However, we will deal with hierarchical analyses which allow the user to specify how abstract representations of cells are formed, how they are checked, and how to respond if a violation is detected. This allows one analysis program (with different rules) to use the designer's hierarchy for a wide variety of analyses. Analyses that were difficult in previous schemes (cross-coupling capacitances, terminals in the center of cells, and multilayer interconnects) can now be handled hierarchically.

An analysis of existing custom VLSI chips reveals how designers use hierarchy. While they commonly add geometry across hierarchical boundaries, designers rarely change the circuits of subcells—implying that hierarchical analysis is practical for a wide variety of analyses. Cases where hierarchical analysis is not practical, limited in variety, can be handled automatically by using different rules depending on the cell type.

An integrated circuit layout must meet two conditions to produce working chips. First, the layout must implement the intended circuit. Second, the layout must obey ***design** rules,* the physical requirements of the fabrication process. Design rules specify the legal sizes of features on the chip; violate these rules and, depending on the magnitude of that violation, the chip will either malfunction or be difficult to produce.

When integrated circuits were small, designers checked by hand to ensure that circuits were correct and design rules not violated—an irksome and error-prone process. However, as circuits grew in size and complexity, hand checking became less practical. By this time, since IC Iayouts were normally stored in machine-readable form, programs were developed to check layouts against their constraints.

These design rule check (or DRC) programs test an IC layout against a set of physical design rules and report any violations. The designer corrects reported errors, repeating DRC until the layout is error free. At this point, the design can be fabricated. Extractors (component extraction programs) generate a component netlist, from an IC layout, which is either compared against the desired netlist or made the input to a simulator. This guarantees that the layout implements the intended circuit. DRC programs and circuit extractors are essential for the construction of VLSI chips containing tens of thousands of devices, for which hand checking would be completely impractical.

Non-hierararchical analysis programs

Early DRC and extraction programs[1,2] often had straightforward algorithms whose time requirements grew as $O(N^2)$, where Nrepresents the number of features on the chip. As chips grew larger, however, execution times of the programs also grew. Improved algorithms[3-6] were developed. In general, these had run times in the range of $O(N^{1.5})$. Since N was doubling every year, a pace that continues to the current day, these algorithms were replaced by still more sophisticated—and optimal—algorithms that are $O(N \log N)$.[7-9] More recent work has extended the $O(N \log N)$ results to circuit extraction and decreased memory requirement, and has provided more function.[10,11] While chip sizes have increased 10,000-fold, almost every paper in the field states that small problems run in a few minutes—but the largest chips take a few hours. This has held true due to a combination of faster processors, larger main memories, and better algorithms.

Basics of hierarchical analysis

Each of the above programs started by expanding hierarchy, {hereby creating a flat design. This eliminated all hierarchical structure present in the input, dramatically simplifying the checking algorithms. However, eliminating the hierarchy also introduced problems: It was difficult to report errors in terms of the cells in which they occurred, and errors in repeated cells were reported many times.

Since expanded designs contain huge numbers of primitives, analysis programs require large amounts of main memory and disk space, and run times are long. Small changes to individual cells may require rerunning the whole analysis, particularly in the case of circuit extraction, since it is difficult to determine how far the changes propagate.

Solutions to these problems use the hierarchical input structure to make analysis easier; two basic approaches are common. The first[12-14] joins the structure present in the input with some method to take care of the cases not readily amenable to hierarchical analysis. The second approach imposes constraints upon the design so that a hierarchical analysis is always possible.[15-18]

Each approach contains disadvantages. Programs that can accept an arbitrary hierarchy produce output that is either non-hierarchical, or that uses a different hierarchy than specified by the user. Schemes that restrict the hierarchy may not be acceptable if those restrictions force major changes in design style. For these schemes, using designer-preferred hierarchies is extremely important.

Hierarchies preferred by designers can only be determined by experiment. Experiments described in a later section of this paper measure a number of full-custom chips to determine which cell/cell and which cell/primitive interactions are present. The chips we chose to measure were designed without hierarchical tools, so the only constraints on the hierarchy were imposed by the designers' desire to layout, understand, and analyze the resulting chip. These experiments show that strict hierarchy (hierarchy with no overlap at all) is rarely observed; most subcells have something overlapping them. However, in most cases the overlap does not change the subcell's function. And overlap that does change the function of the subcell is normally restricted to specific cases, such as ROMs and PLAs.

An ideal analysis tool would analyze such a hierarchical design (as a human would) expanding where necessary while treating the design hierarchically insofar as possible, given the analysis to be performed. The solution proposed here allows the user to specify rules controlling hierarchical use in the analysis. These rules may differ for each form of analysis, and may vary based on the cell being analyzed. This approach allows sufficient flexibility for analyses applying user-specified hierarchies.

Examples of problems in hierarchical IC analysis

The following two examples, using the same layout and differing only in the analysis to be performed, illustrate the complexity of hierarchical analysis. The first example requires accurate overlap capacitances, implying that a. cell must be reanalyzed if a primitive is added to it. The second example requires only connectivity. In this case, the analysis must determine if the overlap changes the cell's function. If it does not, then the overlap is legal; otherwise, it is in error.

Both examples use the simple cell represented in Figure 1 and designed to be used in a single-layer metal environment. This cell contains metal on an internal signal, but also offers areas where other metal lines may cross. Since we are performing hierarchical analysis, we'll assume that the cell has been analyzed in detail. We are primarily concerned with the usage of the cell, and a typical cell use is shown in Figure 1.

In the first example, the user wants detailed circuit data including device size and capacitive loading information. Among the parasitics necessary for accurate results is capacitance caused by overlapping layers, which can only be obtained if the cell is reanalyzed with the interconnections added to it. In this case, the hierarchy has been ignored since the unique pattern of metal over the cell's top has converted the shared cell to a unique configuration. Note, however, that hierarchical analysis can still save computation time even if every cell is unique.[17]

In the second example, the extraction is performed to obtain a netlist for comparison with the desired netlist. While the exact overlap capacitances are unimportant in this case, it is important that the interconnect metal connect to the appropriate pins of the device, and that any primitives overlapping the cell do not impair the cell's functioning. In this example, the subcell can be treated the same in every instantiation—although the resulting

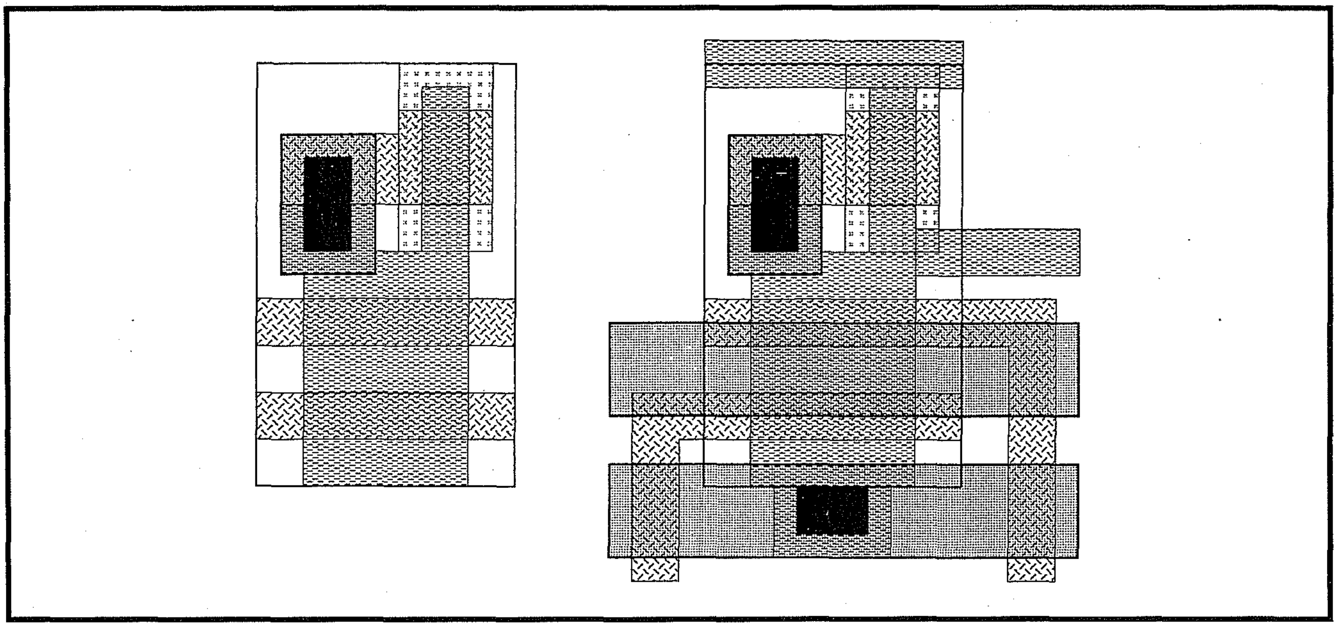


Figure 1. Cell and use in context.

circuit may not be the same for every instantiation since different instantiations may have different pins tied together. Hierarchical circuit extraction can work only if the subcell schematic can be made to represent the extracted (flat) circuit by connections made only to the pins. Thus, an internal connection that shorts together two nodes is allowed, whereas a modification dividing one node into two is forbidden.

These examples show that even with one design, at least two different hierarchy-handling methods are needed. In the first instance, we are looking for gross interference with the function of the cell. Any such interference is an error. In the second instance, any primitive overlapping of the cell forms a unique occurrence requiring complete reanalysis. These are not limitations of the current programs; the user requires two different results. A human faced with the same analysis problems would be forced to treat them in the same way.

Previous work on hierarchical analysis

One of the first hierarchical analysis techniques[12] involves identifying (1) all instances of primitives overlapping cells and (2) each instance of unique cell/cell interaction. Only one of each instance is tested no matter how often it occurs. This approach has several advantages: It works on any hierarchy, although it may not save time on an ill-structured example; it reduces the number of repeated errors in the output; it reduces the computational time by eliminating redundant checking. Major disadvantages are that the output is not hierarchically structured, and that it does not generalize easily to circuit extraction. We have several examples of work accepting arbitrary hierarchical input.[13,14]

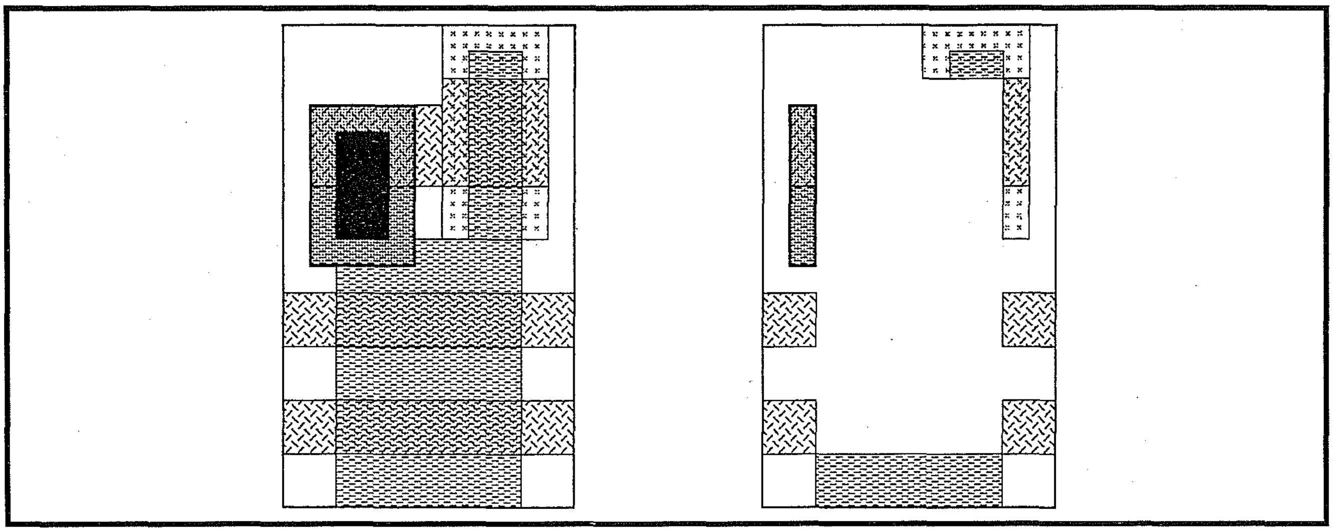


Figure 2. Cell with abstract.

Although the exact constraints differ in "hierarchical analysis with constraints,"[15-19] in general each cell has a boundary that primitives may not overlap, and cells may be of arbitrary shape. The advantages are (1) linear time analysis, (2) easy extension to circuit extraction, and (3) output in the same hierarchical form as input.

The main disadvantage is the requirement that no primitives overlap cells. This complete elimination of overlap is always possible, but experimental evidence shows that designers do not currently follow this design style. Furthermore, recent technical advances (such as multiple layers of interconnect) make it inconvenient to follow these restrictions. One implementation of this form of hierarchical analysis[19] avoids some of these problems by introducing the reanalysis of cells overlapped by primitives. This works, and is required in some cases (such as cross-coupling capacitance) but is unnecessarily slow when we desire interconnections only.

A third form of hierarchical analysis extends the idea of a boundary to the idea of one boundary per layer (called a protection frame).[20] This resembles the previous approach except that the boundary is computed on a layer-by-layer basis, providing more flexibility for primitives overlapping the edges of cells. The main disadvantage here is the difficulty in handling strong layer-to-layer interactions. For example, if cross-coupling interactions must be considered then protection frames are insufficient. Furthermore, in a typical NMOS process it is insufficient that poly not violate the poly protection frame—it must also have a certain clearance from the diffusion protection frame.

A solution to these problems

An *abstract* of a cell is any simpler representation of the cell that can replace the cell in a hierarchical analysis. The exact contents of an abstract depend on the analysis being performed; for circuit extraction, the abstract of a cell might consist of the area occupied by the cell and the pins of that cell. Protection frames, DRC "donuts," and functional models are other abstracts built for particular analyses.

Most analysis (electrical and physical) can be performed hierarchically with the proper form of abstract.[17] In general, the technique involves finding all errors that can be found without knowing the context of the cell—and then recording in the abstract all information about parts of the cell that cannot be checked until the context is known.

Consider the following two examples of different abstracts for different analyses. In the first example, assuming that primitives do not overlap cells, the abstract of a cell is the region within some distance (D) of the edge, where D is the largest design rule. For the previous example, the abstract might look like Figure 2. Note that when the abstract assumptions are violated—when primitives are run over a cell—then the cell must be reanalyzed.

Another form of abstract might be called the "occupied area and pins" model.[21] In this case, the abstract consists of a region occupied by the cell and the pins of the cell, as shown in Figure 3. This abstract is useful for computing connectivity in a process where second-layer metal often overlaps cells. Here, if a primitive overlaps the corresponding occupied layer of the subcell, then an error is declared and the user must fix it. No automatic reanalysis is performed.

In order to accommodate these two forms of hierarchical analysis, the user must specify how the abstract is to be created, and must also state under what conditions it is correct to use the abstract in place of the full cell and what must be done if these conditions are not met. If the abstract is not valid, the alternatives are to reanalyze the cell or to report it as an error and let the user fix it.

The abstract creation can be specified with the same commands used in the conventional DRC. The Figure 2 abstract was created by saving all material within some distance (D) of the boundary. This can be expressed in the operations AND, OR, ANDNOT,

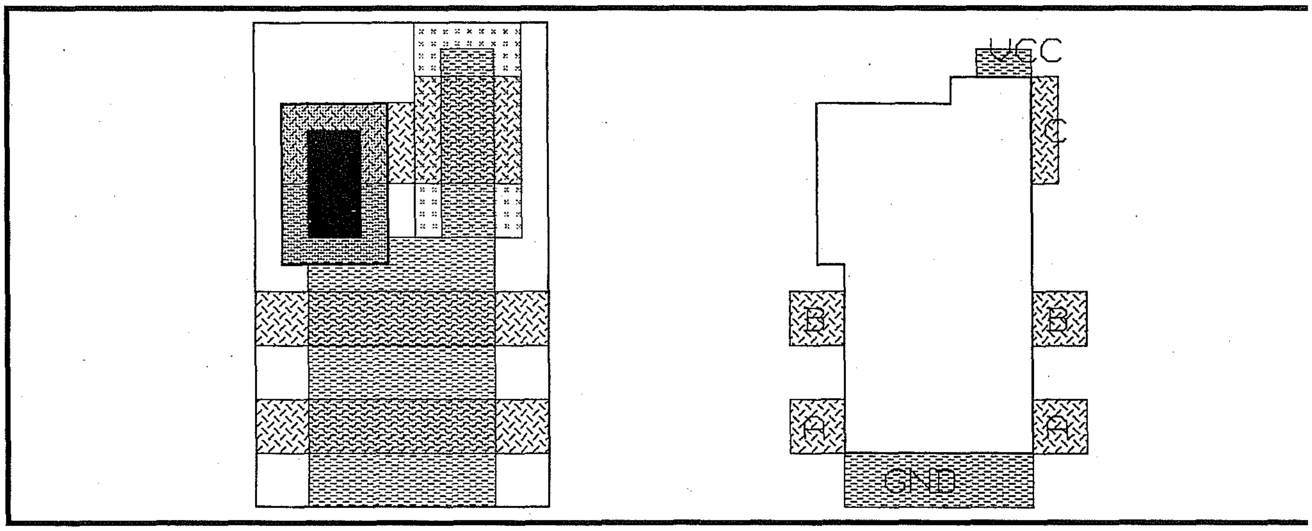


Figure 3. Occupied area and pins abstract.

EXPAND, and CONTRACT as applied to polygons. (The accompanying summary box explains these operations.) Rules for this operation are.

```
abstract = primitives AND
(boundary ANDNOT
(boundary CONTRACT D)).
```

In addition, the user must specify the rules determining which parts of electrically conductive polygons within the cell are treated as pins. This may be done by proximity to the boundary (as above) or by establishing a separate layer for the purpose. For example, if we have a *define—pins* layer and a *metal—1* layer, then the pins on this layer are defined as

pins = *metal—1* AND *define—pins.*

Determining whether the abstract can be used in place of the cell may involve many complex rules. The rule is simple when overlap capacitance is important; if any primitive overlaps a cell, then the abstract cannot be used and the cell must be reanalyzed with the primitive added. When checking connectivity, however, the situation is more complex. For example, the poly layer in an NMOS process may overlap a cell provided that (1) it doesn't get within a certain distance of any poly in the cell; (2) it doesn't get within any (different) distance of diffusion in the cell; and (3) it doesn't get within any (still different) distance of contacts in the cell. There are similar rules for all other layers.

We can check these rules in at least two ways: by recording the area that is used by the subcell, or by recording the area that must be avoided by the parent cell primitives. For example, suppose that the relevant rules are 2 microns poly/poly spacing, 1 micron poly/diffusion spacing, and 1.5 microns poly/contact spacing. We can code these rules in two different ways:

```
poly > 2 microns from subcell
          poly
poly > 1 micron from subcell
          diffusion
poly > 1.5 microns from subcell
            contact
```

or, alternatively, for each cell compute the poly_keepout layer as

```
poly—keepout = (poly EXPAND 2)
    OR
    (diffusion EXPAND 1) OR
    (contact EXPAND 1.5)
poly > 0 microns from poly—
    keepout.
```

These methods are exactly equivalent only if the same metric is used for distance measuring and the expand/contract operation. Typically, this is not the case since different metrics are used for efficiency; the differences, however, are small.

In the above example, we see two separate problems: (1) computing the layers that represent a cell at the next level of the hierarchy, and (2) stating rules that use these abstract representations.

We can express these rules in the DRC command file as follows: For every user-defined layer, we create two other layers—the "occupied" area of the subcell (computed when the subcell was analyzed), and the "occupied" area of the current cell (computed as we DRC the cell). In the examples that follow, if poly is the user-defined layer, then poly[occupied] refers to the subcell area occupied by poly if used as a source of data. If used as a destination, poly refers to the occupied area of the cell under consideration. In these examples, we expand and then contract layers by an amount that merges polygons too closely aligned to allow any routing between them, thereby reducing the amount of data that must be handled at the next layer of the hierarchy:[20]

```
   compute own occupied region:
poly[occupied] = (poly EXPAND 4)
  CONTRACT 4
contact[occupied] = (contact
  EXPAND 5) CONTRACT 5
diffusion[occupied] = (diffusion
  EXPAND 5) CONTRACT 5
  check spacing of poly in cell to poly
              of subcells:
spacing poly poly[occupied] >2.0
spacing poly diffusion[occupied] >1.0
spacing poly contact[occupied] >1.5
```

or, if the user wishes to express the rules in terms of keepout regions,

```
  dejine the region wherepoly is not
              legal:
poly[keepout] =
  (poly EXPAND 2.0) OR
  (contact EXPAND 1.5) OR
  (diffusion EXPAND 1)
 Checkpoly in cell versus keepout of
              subcells:
spacing poly poly[keepout] >0.0
```

As described earlier, both the rules and the abstracts differ for different analyses. Each form of analysis has a user-supplied name associated with it. The DRC program keeps several different abstract representations for each cell—one for each type of analysis—allowing any analysis of any cell at any time.

Error and circuit output

Error reporting is more complex for a hierarchical analysis tool than for a flat analysis tool; the use of the abstract in place of the complete subcell may cause complications. If the "donut" abstract for a cell is used, for example, the abstract will contain pieces of geometry not meeting the minimum width rules. The program must examine the location of each error to determine whether it is real and should be reported, or whether it is an abstraction process artifact and should be ignored—not a difficult task provided the width of the "donut" is greater than the largest rule to be checked. If so, each error is reported only once, no errors are missed, and no false errors are generated.[17]

Subcell reanalysis also creates potential confusion. If a subcell must be reanalyzed because a primitive has been placed over it, then any discovered errors must be reported in the parent and not the subcell. This will generate no false errors, provided that the subcell had no errors when analyzed by itself.

Hierarchical circuit extraction always generates hierarchical netlists. The schematic for each cell is described in terms of primitive components, such as capacitors and transistors, and references to subcells. If another tool requires a fully expanded netlist, then a separate program (a netlist compiler) must be used to generate the fully expanded list. If subcells have been reanalyzed because they had primitives over them, then multiple netlists exist for that cell, one for each different overlapping geometry. The netlist of the parent cell explicitly refers to these subcell versions. Such an approach allows most programs that read the hierarchical netlists to ignore the difference between cells created by the user, and cells created by reanalysis. The alternative—storing only the differences in the netlists for the different versions—would save storage space, but at the cost of added complexity.

Modifications to DRC

We added the features of the previous section to our existing hierarchical DRC/EXTRACT program. Designers then used the program to perform analyses for many different rule sets and processes—processes including NMOS, CMOS, and bipolar. The analyses employed simple DRC rules, complex DRC rules, continuity-only extraction, and full-parameter extraction.

Most users run a simple hierarchical continuity extraction, followed by netlist comparison, until they achieve the correct circuit topology. Then, they run a more detailed set of rules to obtain more accurate parasitics. Some designers, particularly those designing analog circuits, need full cross-coupling capacitances. They must use reanalysis to obtain these capacitances. Other designers, primarily those working on digital circuits, can approximate the cross-coupling capacitances with lumped capacitances to ground—an approach not requiring reanalysis of cells.

Both full extraction (with internodal capacitors) and continuity-only extraction can be coded by one program using different rules. To extract overlap capacitances, we must retain the ability to reanalyze when overlapping occurs. If reanalysis is always used, however, many different versions of simple cells are created (particularly in a process with more than one layer of metalization). We have seen up to 40 versions of a single cell. If only continuity and lumped capacitance to ground are required, then reanalysis is avoided by use of keepout layers and pins.

By using keepouts and pins, 85 to 95 percent of the cells in a typical design can be analyzed hierarchically for both design rule checking and circuit extraction. The major exceptions are ROMs and PLAs where users, intending to alter the subcell's function, have deliberately placed primitives inside subcells. These violations, although bad in principle, can be easily handled in practice. Small ROMs and PLAs can be handled by flattening their hierarchy; large ROMs and PLAs, normally machine generated, are already handled as special cases. Usually, we don't need circuit extraction since the same program generating the PLA generates the schematic. DRC of the final programmed cell is not necessary except, perhaps, for a border around the edge. The border, another form of abstract, can be generated by a rule file that only applies to PLAs.

ROMs and PLAs are handled automatically by specifying, in the command file, that certain cells are to be analyzed with different rules. For example, each cell whose name begins with "PLA" can be flattened, while all other cells can be treated hierarchically. Once ROMs and PLAs are treated separately, the percentage of instances that cannot be handled hierarchically drops to roughly three percent (for a design that was not built with hierarchical analysis tools in mind). With new designs and cooperative designers, the percentage of cells that cannot be hierarchically analyzed drops to zero.

Measurements of custom chips

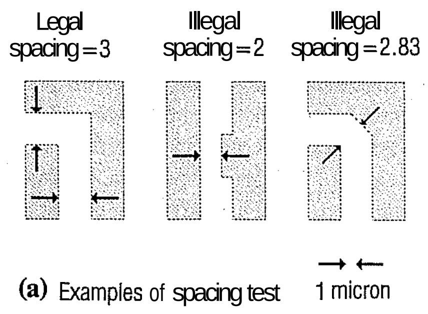
Several questions about hierarchical analysis techniques can only be answered by experiment. The most interesting of these questions is: What percentage of analyses performed on real designs can be performed hierarchically? This is difficult to quantify for several reasons. First, if the analysis method needs constraints, then the percentage will depend on the willingness of designers to follow those constraints. Furthermore, the analyses requested depend heavily upon what other tools are available. If the analog simulation programs and network comparison programs only work on flat input, for example, then the designer has no use for hierarchical circuit extraction.

Nonetheless, the hierarchichal restrictions that designers will accept can be estimated by examining chips designed using only flat analysis tools. Designers using these tools employ hierarchy only as an aid to building and understanding these circuits—not a true test of designer wants since, by analogy with programming, when hierarchical tools are available designers will use them. However, the structure designers impose provides a lower bound to the structure they will accept.

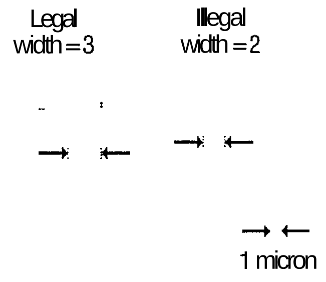
Table 1 lists statistics from two large custom chips. One is Berkeley's RISC-II chip; the other is the 9852 CRT controller from Advanced Micro Devices. Both were designed with flat analysis tools and hierarchical editors. Both are built in an NMOS process. Table 1 reports cell uses two ways: First, an array of cells is counted as one reference; second, an array of N cells is treated as N separate subcells.

The incentive for hierarchical DRC is clear from Table 1; if we analyze each cell once in the RISC-II design, we analyze roughly 24,000 trapezoids. If we analyze the flat design, we need to analyze roughly 461,000 trapezoids. Moreover, DRC algorithms are worse than linear so the difference is further magnified. On the other hand, when a cell is analyzed hierarchically information from subcells must be included and the abstract must be generated. This decreases the advantage of the hierarchical approach, but reductions of between 400 to 1000 percent in CPU time remain typical.

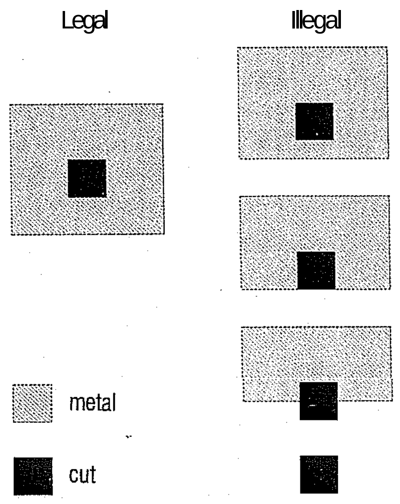
The performance of the hierarchical DRC/EXTRACT depends dramatically on the required reanalysis. If no



(a) Examples of spacing test



(b) Example of width test



(c) Examples of enclosure test

Figures la-c. Basic **measurements** performed by DRC.

Basic Geometrical Operations for Design Rule Checking

Design rule checking is built around two basic operations: measuring geometry, and creating new geometry as a function of existing shapes. Operations define which regions to test, and measurements find the errors which are reported to the user. Measurements are performed between all edges of a given type. The basic tests are illustrated in Figures 1a-c.

Geometrical operations isolate features to be tested. A nonimplanted transistor, for example, occurs only where polysilicon and diffusion coincide and the implant mask is not present. In this case, the logical operations can generate geometry corresponding to these transistors only. For operations on geometry, each region on an IC mask can be regarded as a set of points in the Cartesian plane, allowing operations of AND, OR, ANDNOT, and XOR to be performed pointwise. Figure 2 shows the results of these operations on two rectangles.

The operations EXPAND and CONTRACT work on a single layer. EXPAND makes all polygons on a layer bigger without affecting their position. CONTRACT makes them smaller. EXPAND is performed by including all points within a given distance into the existing set, possibly causing adjacent figures to merge and holes inside figures to disappear. CONTRACT is performed by deleting from a figure all points within a specified distance of any point not in the figure. CONTRACT can cause single figures to break into multiple figures, and can cause figures to disappear entirely.

The exact results of EXPAND and CONTRACT depend on the metric used. The Euclidean metric is perhaps the most natural, but a square corner will turn into an arc if expanded using the Euclidean metric. Arcs are more difficult both to deal with computationally and to reproduce with mask-making machines, most of which are based on rectangles or trapezoids; approximating circular arcs with these figures increases the cost manyfold.

By using a different metric, square corners will remain square—useful for making masks, but resulting in corners being overexpanded. A compromise, such as the octagonal metric, reduces the maximum error to eight percent while adding only one side to an original square. Figures 3a-b show the results of EXPAND and CONTRACT on severa! images, using different metrics.

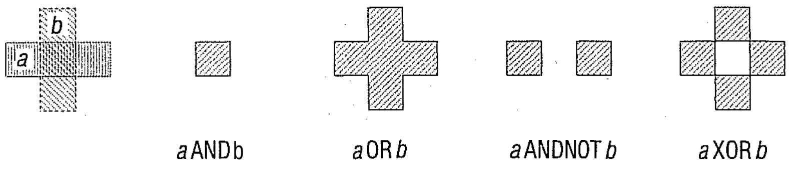


Figure 2. Logical operations on two rectangles.

reanalysis is required, performing a 15-rule circuit extraction on the RISC-II chip takes about 6000 CPU seconds. With a set of rules that forces reanalysis of every instance of every cell, the same analysis takes roughly 40,000 CPU seconds (for an 8 MHz Motorola 68000).

Table 2 enumerates cell use in the hierarchy. Each cell in the design is looked at once (no matter how many times it is used) and the subcells used are counted. This corresponds to the number of subcell references a hierarchical analysis program must analyze. For the first measurement ("with overlap") we build an abstract for each cell by expanding and then contracting each layer, and by removing holes. If any parent cell geometry overlaps this region, the cell is counted in this group.

Next, we look for overlaps changing the subcell's circuit. Since all the chips considered here are poly-gate MOS, we can do this by searching for poly overlapping the diffusion of subcells or vice versa.

Finally, we assume that ROMs and PLAs will be analyzed with different rules as discussed in the previous section. Therefore, we recalculate the per-

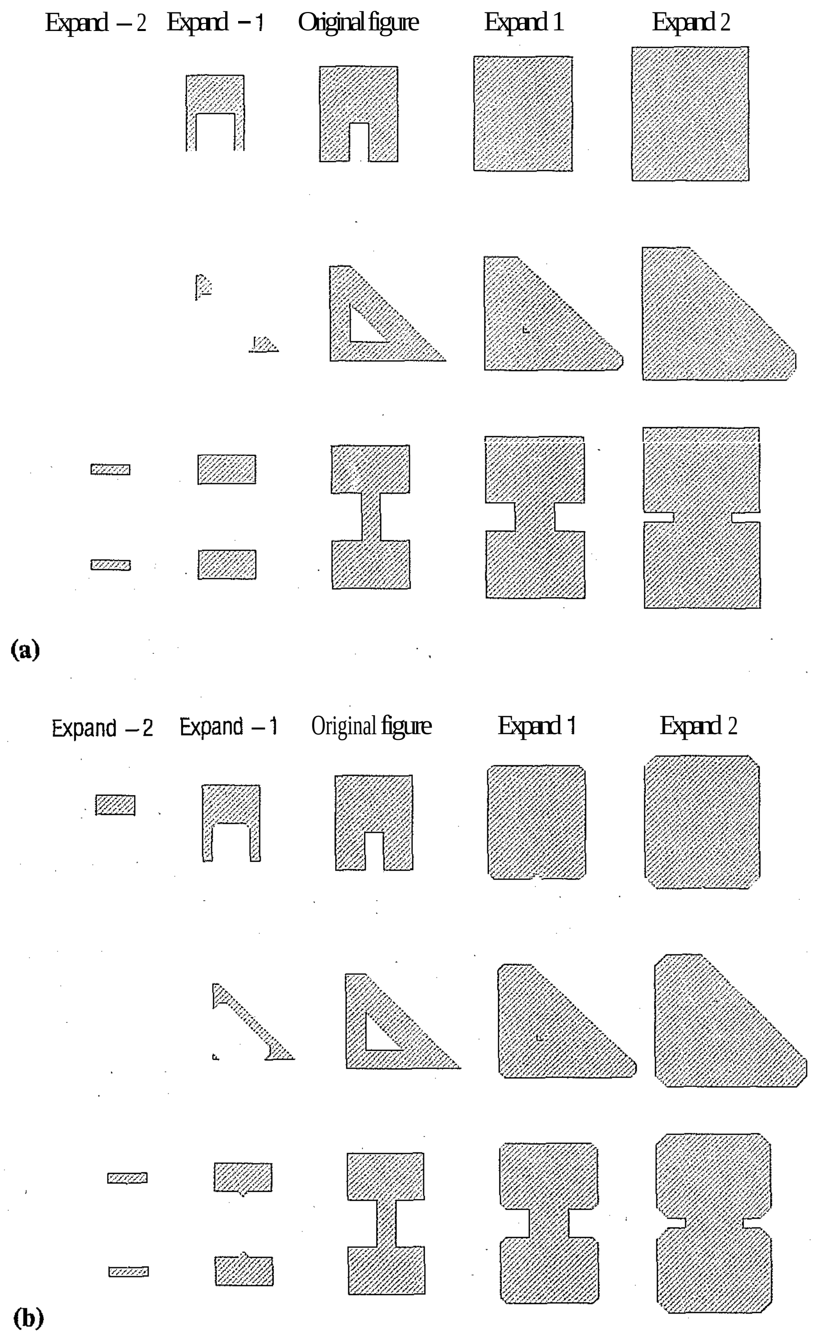


Figure 3a-b. Expansion and contraction with different metrics.

Table 1. Custom chip statistics.

chip	RISC-II	8052
Each cell once:		
number of cells	346	487
references (no arrays)	1360	3704
references (with arrays)	2966	10915
trapezoids in all cells	23565	173717
Full hierarchy:		
references (expanded)	1069	4330
references (with arrays)	11367	20693
total trapezoids	460537	1015479

Table 2. Modification of subcells.

	RISC-II	8052
references	1360	3704
with overlap	87.9%	94.5%
with circuit mods	11.6%	17.8%
ROMs, PLAs	10.0%	13.4%
non-ROM circuit mods	1.54%	4.32%

centages excluding these cells—labelled "non-ROM circuit mods," these cells cannot be analyzed hierarchically for circuit extraction.

The results clearly show that while designers often put primitives over cells, they generally change the circuits of subcells only in certain well-defined cases. Except for ROMs and PLAs, just a small percentage of cell references change the subcell circuitry. Examination of these remaining cases shows that, if necessary, they could be done as easily without circuit modification. The tools used in developing these examples provided no such incentive. It's very encouraging that only a small number of cells violate hierarchical constraints, even in the absence of explicit rules forbidding these violations.

Our experiments show hierarchical analysis of IC layouts to be practical—providing the performance benefits of hierarchical analysis, improving error reporting, and allowing the use of circuit tools needing hierarchical input—without requiring major changes in layout design. However, different approaches to hierarchical analysis are required depending on analyses requested by designers. Some requests demand subcell reanalysis for sufficient accuracy; for other requests, this reanalysis is unnecessary and time-consuming.

With the addition of four new features, a hierarchical DRC and circuit extraction program can handle all commonly required forms of analysis. These features are: user-defined abstract generation, userdefined rules stating when the abstract is valid, user specification of what to do with an invalid abstract in a given context, and user specification of analysis form depending on the cell type.

There are several advantages to this program. All common layout analysis operations can be performed with user-specified hierarchy. The user, not the program, makes tradeoffs between the performance advantages of preserving the hierarchy and additional operations (such as detection of overlap capacitance) possible with explicit subcell reanalysis. In particular, this program handles multiple interconnect layers and cells with internal terminals; these are analyzed using the user-specified hierarchy and without subcell reanalysis. The same program, with different rules, can also handle problems (such as the calculation of overlap capacitance) requiring subcell reanalysis.

Hierarchical analysis tools resemble structured programming tools. All are designed with consideration given to existing applications, but the real test comes as users grow familiar with the tools and design new applications. In each case, there are many possible tradeoffs between user constraints and ease of analysis. In each case, complexity limits the size of practical design. As has proven true with programming tools, human understanding of both the layouts and the results of the analysis tools probably determines the proper tradeoffs for IC layouts. Thus, the designs and tools most easily understood by human designers present a promising topic for future research.

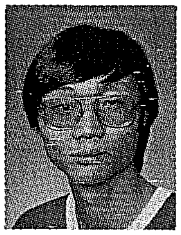
Acknowledgments

We would like to thank Advanced Micro Devices for providing examples of commercial chips. Commentary by IEEE referees was also helpful, and is appreciated.

References

1. M. Yamin, "XYTOLR—A Computer Program for Integrated Circuit Mask Design Checkout," *Bell System Tech. J.,* Vol. 51, No. 7, Sept. 1972, pp. 1581-1593.
2. B. Preas, B. Lindsay, and C. Gwyn, "Automatic Circuit Analysis Based on Mask Information," *Proc. 13th Design Automation Conf.,* San Francisco, Calif., June 1976, pp. 309-317.
3. H.S. Baird, "Fast Algorithms for LSI Artwork Analysis," *Proc. 14th Design Automation Conf.,* New Orleans, La., June 1977, pp. 303-311.
4. D. Alexander, "A Technology Independent Design Rule Checker," *Proc. 3rd USA-Japan Computer Conf.,* San Francisco, Calif., Oct. 1978, pp. 412-416.
5. K. Yoshida et al., "A Layout System for Large Scale Integrated Circuits," *Proc. 14th Design Automation Conf.,* New Orleans, La., June 1977, pp. 322-330.
6. P. Wilcox, H. Rombeek, and D.M. Caughey, "Design Verification Based on One-Dimensional Scans," *Proc. 15th Design Automation Conf.,* Las Vegas, Nev., June 1978, pp. 285-289.
7. U. Lauther, "An O(N log N) Algorithm for Boolean Mask Operations," *Proc. 18th Annual Design Automation Conf.,* Nashville, Tenn., June 1981, pp. 555-562.
8. J. Bentley and D. Wood, "An Optimal Worst-case Algorithm for Reporting Intersections of Rectangles," *IEEE Trans. Computers,* Vol. C-29, July 1980, pp. 571-577.
9. J. Bentley and T. Otmann, "The Complexity of Manipulating Hierarchically Defined Sets of Rectangles," Tech. Report, Carnegie-Mellon University, 1981.
10. T. Szymanski and C. Van Wyk, "Space-Efficient Algorithms for VLSI Artwork Analysis," *Proc. 20th Design Automation Conf.,* Miami, Fla., June 1983, pp. 734-739.
11. P. Chapman and K. Clark, "The Scan-Line Approach to Design Rules Checking: Computational Experiences," *Proc. 21st Design Automation Conf.,* Albuquerque, N.M., June 1984, pp. 235-241.
12. T. Whitney, "Description of the Hierarchical Design Rule Filter," SSP file #4027, Silicon Structures Project, California Inst. Technology, Pasadena, Calif., Oct. 1980.
13. M. Newell and D. Fitzpatrick, "Exploiting Structure in Integrated Circuit Design Analysis," *Proc. Conf. Advanced Research VLSI,* MIT, Cambridge, Mass., 1982, pp. 84-92.
14. S. Johnson, "Hierarchical Design Verification Based on Rectangles," *Proc. Conf. Advanced Research VLSI,* MIT, Cambridge, Mass., 1982, pp. 97-100.
15. J. Rowson, "Understanding Hierarchical Design," PhD dissertation, California Inst. Technology, 1980.
16. L. Scheffer, "A Methodology for Improved Verification of VLSI Designs without Loss of Area," *Caltech Conf. VLSI,* Pasadena, Calif., 1981.
17. L. Scheffer, "The Use of Strict Hierarchy for Verification of Integrated Circuits," PhD dissertation, Stanford University, 1984. (Available as a tech. report from: Integrated Circuits Laboratory, Stanford University, Stanford, CA 94305.)
18. S.N. Stevens and S.P. McCabe, "IDS—A System for Fast, Hierarchical Design of Handcrafteci VLSI Circuits," IEEE *1984 Custom Integrated Circuits Conf.,* Rochester, N.Y., May 1984, pp. 107-111.
19. "DRC/EXTRACT Manual," Valid Logic Systems, Inc., 2820 Orchard Pkwy, San Jose, Calif., Oct. 1983.
20. K.H. Keller, A.R. Newton, and S. Ellis, "A Symbolic Design System for Integrated Circuits," *Proc. 19th Design Automation Conf.,* Las Vegas, Nev., June 1982, pp. 460-466.
21. "Electronic Design Interchange Format Version 1 0 0," The EDIF Users' Group, Design Automation Dept., Texas Instruments, P.O. Box 225474, MS3668, Dallas, Tex. 75265.

Louis K. Scheffer is an engineer at Valid Logic Systems, working on hardware and software to support VLSI layout. Before joining Valid, he worked for Hewlett-Packard as an IC designer and layout tool developer. He received both his MS from the California Institute of Technology and his PhD from Stanford in electrical engineering.



Ronny Soetarman is a software engineer at Valid Logic Systems, working on the DRC/extract program. He received his BS in mechanical engineering from the University of California at Santa Barbara, and his two MS degrees in mechanical engineering and computer engineering from Stanford University.

The authors' address is Valid Logic Systems, Inc., 2820 Orchard Parkway, San Jose, CA 95134.