

# VLSI Layout as Programming

RICHARD J. LIPTON, JACOBO VALDES, and GOPALAKRISHNAN VIJAYAN

Princeton University

STEPHEN C. NORTH

Bell Laboratories and Princeton University

and

ROBERT SEDGEWICK

Brown University

---

The first component of a VLSI (very large-scale integration) design environment being built at Princeton University is described. The general theme of this effort is to make the design of VLSI circuits as similar to programming as possible. The attempt is to build tools that do for the VLSI circuit designer what the best software tools do for the implementer of large software systems.

Categories and Subject Descriptors: B.7.1 [Integrated Circuits]: Types and Design Styles—*VLSI (very large-scale integration)*; B.7.2 [Integrated Circuits]: Design Aids—*layout*; D.3.2 [Programming Languages]: Language Classifications—*design languages*

General Terms: Design, Languages

Additional Key Words and Phrases: Hierarchical design

---

## 1. INTRODUCTION

In this paper we describe a very important component of any VLSI (very large-scale integration) design environment: a tool to automate the layout of circuits. This work is part of an effort to create an integrated environment for VLSI design (including layout systems, device and switch-level simulators, and testing facilities) currently under way at Princeton University.

---

Portions of this paper appeared in the Proceedings of the 1982 ACM Symposium on Principles of Programming Languages [12] and in the Proceedings of the 1982 Design Automation Conference [11]. The work of Richard Lipton has been partially supported by grants MCS8023-806 from the National Science Foundation and N00014-82-K-0549 from the Defense Advanced Research Projects Agency and the Office of Naval Research. Stephen C. North is being supported by Bell Laboratories. Robert Sedgewick's work was partially supported by National Science Foundation grant MCS80-17579. The work of Jacobo Valdes has been supported by Defense Advanced Research Projects Agency and Office of Naval Research grant N00014-82-K-0549.

Authors' addresses: R. J. Lipton, S. C. North, J. Valdes, and G. Vijayan, Department of Electrical Engineering and Computer Science, School of Engineering/Applied Science, Engineering Quadrangle, Princeton University, Princeton, NJ 08544; S. C. North, Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974; R. Sedgewick, Computer Science Department, Brown University, Providence, RI 02912.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1983 ACM 0164-0925/83/0700-0405 \$00.75

Our main thesis is that the VLSI design task can be profitably thought of as a *programming task*, as opposed to a geometric editing task. We believe that much is to be gained by consciously attempting to apply our knowledge about programming to this new activity. We have thus tried to create tools for the VLSI designer that incorporate the most useful features of the software development tools that we are familiar with.

Although we feel that we have had moderate success in this endeavor, we are well aware of how much room for improvement remains, and we would like to help convince the community of people interested in the design of programming languages and programming environments that there are fresh and important challenges in this relatively new direction.

A prototype of the procedural layout language described in this paper has been operational for some months. All figures given in this paper were generated by the language, and all the code fragments have been used as part of larger programs.

## 2. ALI: A PROCEDURAL LANGUAGE TO DESCRIBE LAYOUTS

The main feature of ALI as a layout language is that it allows its user to design layouts at a *conceptual level* at which neither sizes nor positions (absolute or relative) of layout components may be specified. Mostly as a consequence of this feature, ALI simultaneously (1) makes the layout task more like programming than editing, (2) eliminates the need for design rule checking after the layout is generated, (3) permits the creation of easy-to-use cell libraries, and (4) provides the designer with the mechanisms to describe a layout hierarchically so that most of the detail at one level of the hierarchy is truly hidden from all higher levels.

The notion of not assigning sizes or positions to any object in a layout until the complete layout has been described (similar to the idea of *delayed binding* in programming languages) sets ALI apart not only from just about all of the graphics-based layout editors we know of [1, 3, 6, 16, 18] but also—with the exception of [15]—from most of the procedural languages for the layout task currently in use or recently proposed, whether or not they include a graphics interface [1, 4, 5, 7–10, 14].

The issues that we tried to address with ALI are the following.

(1) The creation of an *open-ended tool*. Graphics editors tend to be closed tools in that it is hard to automate the layout process beyond what the original design of the system allowed. Procedural languages are generally much better in this respect. However, the fact that most such languages require the specification of absolute sizes and positions makes the creation of a general-purpose library of cells a hard task, since information about the sizes and positions of the cell elements that can interact with the outside world has to be apparent to the user of the library. The absence of absolute sizes and positions makes this problem much less severe in ALI. The extensibility of ALI derives from the fact that it has been built on top of PASCAL, thereby making the full power of PASCAL available to the designer. The generation of tools to automate the layout process, such as simple routers or PLA (Programmable Logic Array) generators, involves writing PASCAL routines to solve some abstract version of the problem and, after having done so, invoking ALI cells to generate the layouts.

(2) Creating tools that are *simple to use* and *easy to learn*. In particular, we want to avoid tools whose behavior is unpredictable. Many programs that rely heavily on sophisticated heuristics respond to small changes in their input with wholesale changes in their output. We have maintained a simple correspondence between the text of an ALI program and the resulting layout so that changes in the layout can be easily related to changes in the program. This decision has simplified the system at the cost of making it less knowledgeable about MOS (metal-oxide semiconductor) circuits.

(3) Facilitating the *division of labor*. Large layouts have to be produced by more than one designer. If the piece produced by each designer is specified in absolute positions, serious problems are likely to arise when the different pieces are put together, unless very tight interaction—with its attendant penalties in productivity—is maintained throughout the design. ALI allows the partitioning of tasks in such a way that the designer of a piece of the layout does not need to know anything about the sizes of other pieces of the complete layout. For instance, at the top of Figure 1 three simple cells are shown, with the intended connections between them shown by dotted lines; at the bottom of the figure, the pieces have been brought together to form a larger piece. The stretching that has taken place has occurred without the designer having to plan for it explicitly while considering each individual cell.

(4) Facilitating *hierarchical design*. Even when a single designer is involved, the ability to view a layout as a hierarchy, with much information about lower levels completely hidden from higher levels, is extremely useful. In ALI, the information about a given level of the hierarchy needed at the level immediately above is reduced by the absence of absolute sizes and positions to just the topological relations among the layout elements of the lower level visible to the higher one.

(5) Reducing the *life cycle* cost of layouts. Modifying a layout to be fabricated on a new process, or to make it conform to a new set of design rules, is currently a costly operation. Yet, successful designs seem to be more or less continuously updated as improved processes become available during their lifetime. Figure 2 shows two instances of a simple layout produced with ALI. The instances are the result of running an ALI program twice, changing *exactly four constants in the program* between runs (those that specified the sizes of power and ground buses). This type of flexibility addresses the problem directly. An ALI program can be written naturally so that all layouts produced by it are completely free of design rule violations, no matter what the values of the constants used in the programs. Therefore, the need for costly design rule checking of different instances of a layout (see Figure 2) can be avoided. Using the same ALI program, one can also generate layouts using different design rules by running the program with a new module incorporating the new design rules.

(6) To avoid the *need for special-purpose computing equipment*. ALI can be used effectively from a standard ASCII terminal in combination with a small plotter shared by several designers. All the algorithms used in the inner cycle of ALI require *linear time*, therefore permitting the use of just about any machine and guaranteeing fast turnaround on small layouts. Furthermore, ALI replaces design rule checking by a hierarchical process that can be performed separately on the individual pieces of the layout. For example, after checking that each of

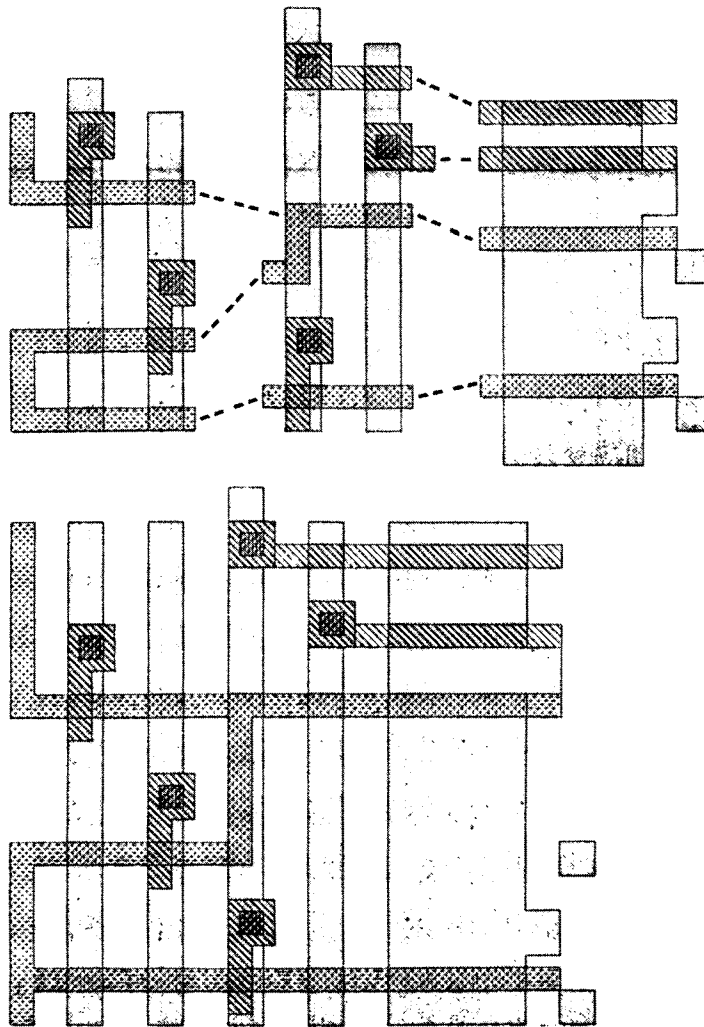


Fig. 1. Three separate cells and the result of connecting them along the dotted lines.

the pieces shown at the top of Figure 1 is free of design rule violations, ALI guarantees their combination shown at the bottom of the same figure to be free of rule violations regardless of the stretching that takes place as a consequence of connecting them. ALI, in fact, requires far fewer computing resources than many design-rule-checking programs.

We feel that ALI succeeds in partially solving most of these problems. We do not claim, however, to have made the layout task trivial. To use a software metaphor, we feel that ALI elevates the work of the layout designer from absolute machine-language programming to programming in a relocatable assembler with subroutines. This not only makes the task more pleasant but makes new and

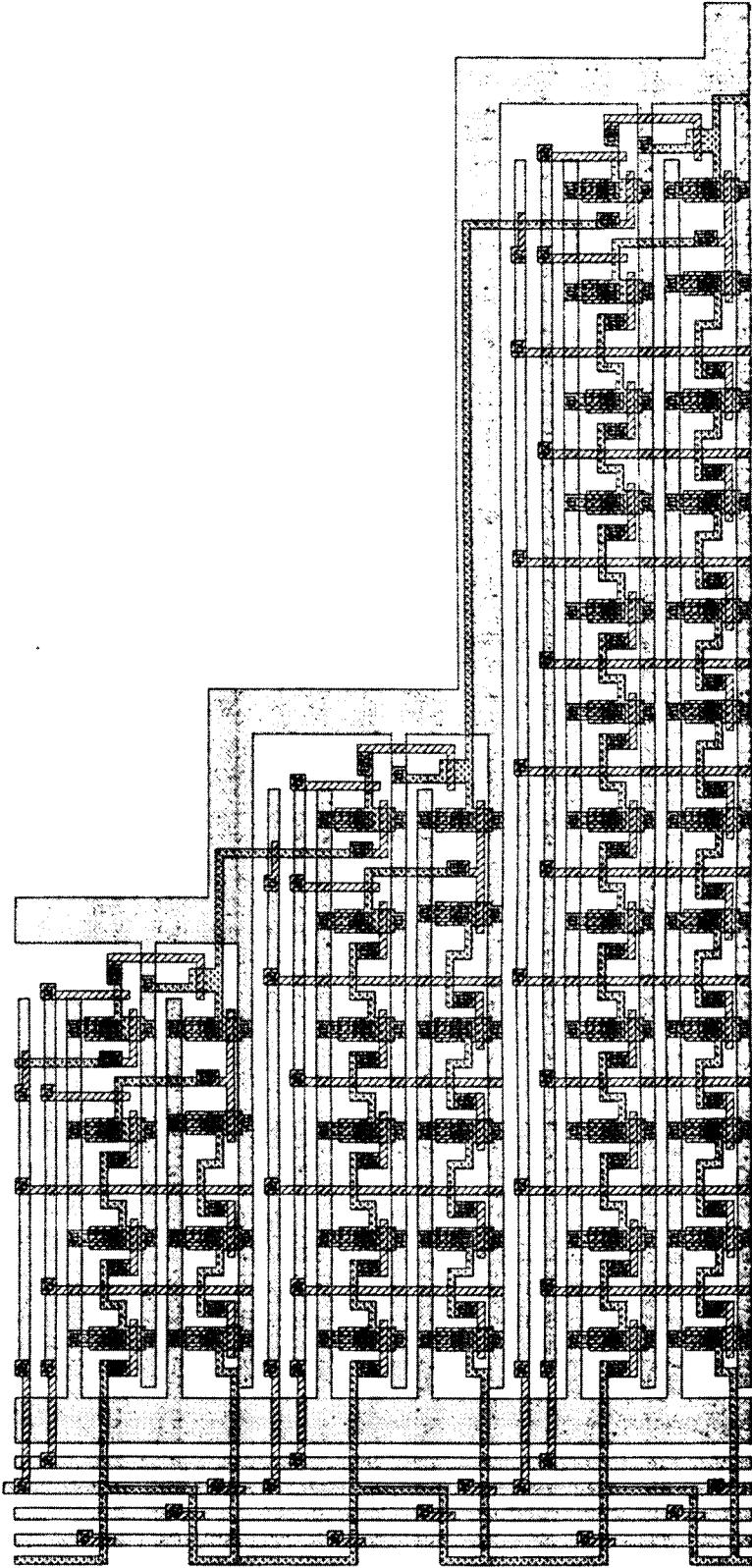


Fig. 2. Two ALI layouts generated by programs differing only in the values of four constants.

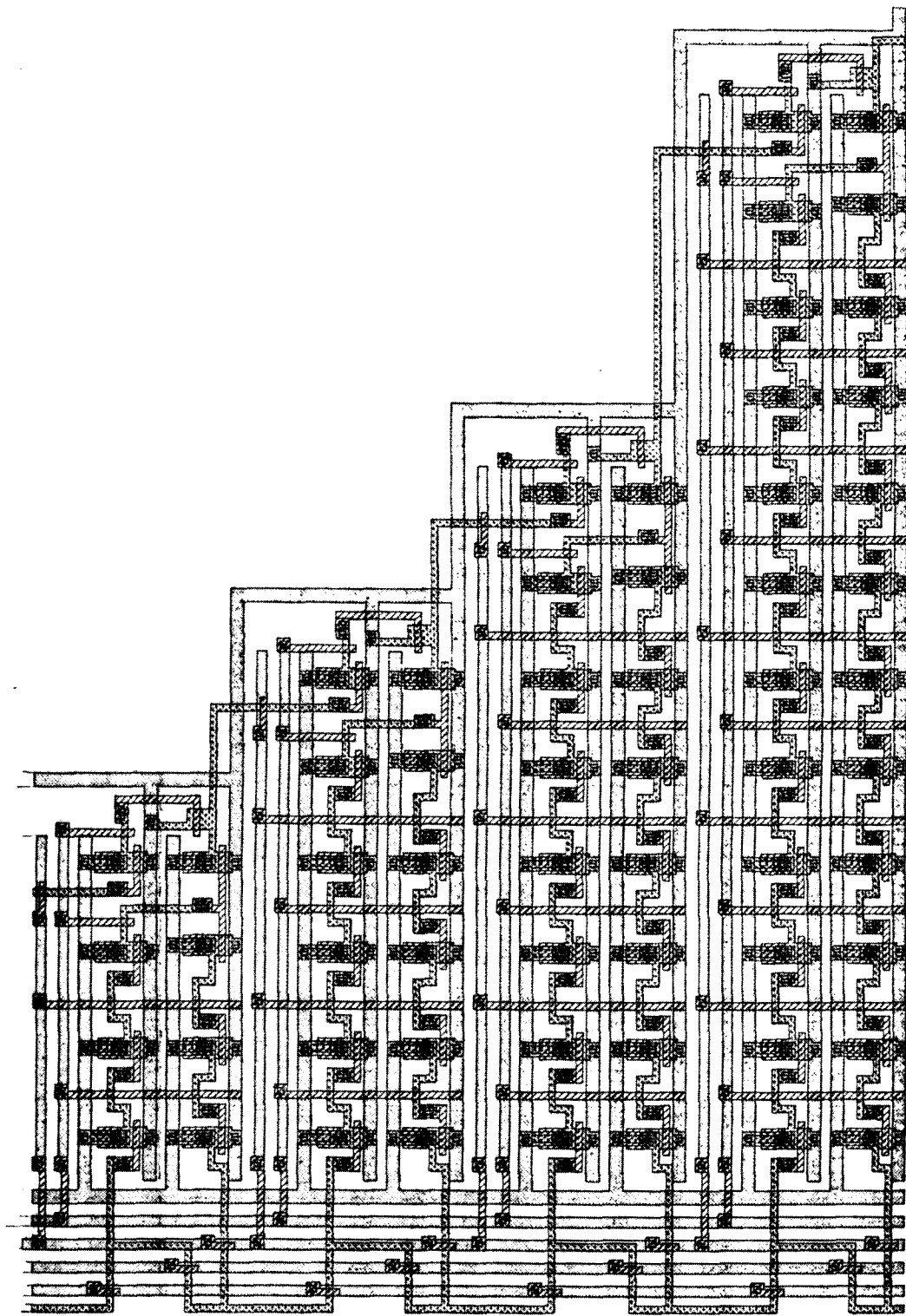


Figure 2 continued.

```

chip simple;
  const
    hnumber = 10;
    length = 20;
    width = 6;
  boxtype
    htype = array [1 .. hnumber] of metal;
  var
    i: integer;
  box
    horizontal: htype;
    vertical: metal;
  begin
    for i := 1 to hnumber - 1 do begin
      above(horizontal[i], horizontal[i + 1]);
      glueright(horizontal[i], vertical);
      xmore(horizontal[i], length)
    end;
    glueright(horizontal[hnumber], vertical);
    xmore(horizontal[hnumber], length);
    xmore(vertical, width)
  end.

```

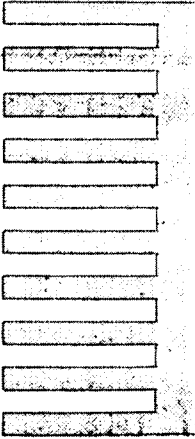


Fig. 3. A simple ALI program and the layout it produces.

more powerful tools possible, such as loaders, linkers, and compilers in the case of software. Similar tools for the VLSI world—which would indeed simplify the layout task enormously—remain, however, to be written. ALI should stand or fall with its ability to allow such tools to be built: whether we are right in believing that we have a framework in which these tools can be more easily implemented will not be known until our efforts in that direction succeed or fail.

The remainder of this section is devoted to a survey of the main features of ALI and a brief discussion of its current implementation.

### 2.1 An Overview of ALI

The basic principles of ALI are quite simple. A layout is regarded as a collection of rectangular objects (with their sides oriented in the directions of the axes of a Cartesian coordinate system) and a set of relations among these rectangles. The ALI user specifies a layout by declaring the rectangles (also called *boxes*) of which it is composed and stating the relations that hold between them. ALI then generates a *minimum-area* layout that satisfies all the relations between the boxes specified in the program. For example, Figure 3 shows a trivial ALI program and the layout it produces. This program looks very much like a PASCAL program: it consists of a declarative part followed by an executable part. To declare a box the user specifies its *name* (*horizontal* or *vertical* in the example) and its *type* (*metal*—a predefined type—in the example). The standard box types correspond to the layers of the physical layout. As the example also shows, the ALI user can define structured objects (an array in the example). Further details on the type structure of ALI can be found in Section 2.2.1.

The relations between the rectangles that make up a layout are specified in ALI through calls to a small set of *primitive operations* in the executable part.

All such operations take as arguments boxes and possibly values of standard PASCAL types (integers in our example). In our example *above*, *glueright*, and *xmore* are primitive operations. The primitive *above* specifies that its first argument must appear above the second one in the final layout, the primitive *glueright* extends its first argument to the right to intersect its second argument, and *xmore* makes the size along the *x* axis of its first argument at least as large as the value of its second argument. Note that in this example ALI has determined the minimum separation between the horizontal elements, as well as the minimum sizes of boxes not specified by *xmore* (such as the height of the horizontal metal lines), by accessing a table of design rules. More information about the type structure and the primitive operations of ALI is given in the next section.

When an ALI program is executed, it generates two kinds of information. It produces a set of linear inequalities involving the coordinates of the corners of the boxes in the layout as variables. These inequalities, which embody the relations between the rectangles of the layout, are then solved to generate the positions and sizes of the layout elements. A brief description of the problems involved in this step can be found in Section 2.3.2. The program also produces connectivity information about the rectangles in the layout. This information can then be used by a switch-level simulator that predicts the behavior of the circuit as laid out without having to perform the usual “node extraction” analysis.

In order for the layouts produced by an ALI program to be free of design rules, the program must be *complete*, in that every pair of rectangles in it must be related in some way. Two rectangles may be related explicitly in the user program by virtue of being arguments to a primitive operation, or they may be related through the transitivity of the separations. The reason for this strong requirement is to prevent the area minimization process from shoving together rectangles that were intended to be separate (see Section 2.3.3 for a discussion of completeness).

ALI helps the designer to achieve this goal by generating certain relations between layout elements in an automatic fashion and by checking on request whether this condition is satisfied. It is, however, the responsibility of the user to make an ALI program complete in this sense, as the computational cost of doing any sophisticated inference (beyond the transitivity of relations such as *above*) is prohibitive [17].

## 2.2 Main Features of ALI

This section describes how ALI appears to its user. The three subsections below deal, in turn, with the *type structure*, the *primitive operations* of the language, and the *cell mechanism*. ALI has been built on top of PASCAL and has inherited most of its features. In the interest of shortening this section we have assumed a certain familiarity with the general features of PASCAL.

**2.2.1 Type Structure.** As the example of Figure 3 shows, the objects manipulated by ALI are declared by stating their *name* and their *type*. The types of ALI have the same structure as the PASCAL types. Objects can be of a *simple type* (boxes) or of a *structured type*.

There are a small number of *standard types*, all of them simple. The standard types correspond to the layers of the process to be used to fabricate the layout (*metal*, *poly*, *diff*, *impl*, *cut*, and *glass* in the NMOS (n-channel MOS) version



currently implemented) plus the type *virtual*, used to name bounding boxes and having no physical reality in the fabricated circuit. For example, in the program of Figure 2, the declaration

```
vertical:metal
```

specifies that the rectangle named *vertical* on the final layout should be on the metal layer. ALI will use this information to generate constraints on its minimum size and its separation from other layout elements.

Structured types are of two flavors: *array* (a collection of objects of the same type) and *bus* (a collection of objects of heterogeneous types, much like a *record* in PASCAL), which correspond directly to the array and record structured types of PASCAL. ALI, like PASCAL, permits the creation of new *user-defined* types, which can be either simple or structured. For example, in Figure 3, the fragment

```
htype = array [1 .. hnumber] of metal
```

inside the **boxtype** section of the program creates a new type, *htype*, each object of that type being made up of a number of metal rectangles; and the fragment

```
horizontal:htype
```

inside the **box** section creates an object of type *htype* named *horizontal*.

In a similar fashion, the type declaration

```
shiftbus = bus  
ph1, ph2:metal;  
vdd:metal;  
data:diff;  
gnd:metal  
end
```

creates a user-defined type, allowing the user to create objects that consist of four metal boxes and a diffusion box. The types of the components of structured types are arbitrary: the user can define arrays of buses, or buses containing arrays.

The accessing of the elements of arrays and buses is done as in PASCAL. Thus, if *x* is of type *htype*, then *x[i]* refers to the *i*th element of *x*, and, if *y* is of type *shiftbus*, then *y.data* refers to the diffusion box of *y*.

Although the structured objects are generally used by ALI simply as a naming mechanism, they are also used in conjunction with the cell mechanism (discussed in Section 2.2.3) to automatically generate separations between boxes. We are more precise on this point when we describe the cell mechanism of ALI.

Like PASCAL, ALI is a *strongly typed* language. The primitive operations know about certain type restrictions and generate type-mismatch errors if operations are attempted with rectangles of inappropriate types.

**2.2.2 Primitive Operations.** The relations between the rectangles that make up a layout are specified in ALI through calls to a small set of primitive operations. All such operations take boxes (i.e., objects of simple types) as arguments. In the program of Figure 3, *above*, *glueright*, and *xmore* are primitive operations.

It is not important to know the actual primitive operations of the current version of ALI to understand its operation. As a gross measure of its complexity we can say that the system currently implemented—based on NMOS as described in [12]—has about twenty primitive operations, which can be arranged in the

following groups:

- (1) *Separation primitives*, such as *above* in Figure 3, which specify that their arguments must be separated in a certain direction in the final layout. The minimum amount of space between boxes separated in this manner depends on their types and is supplied by ALI from a table of design rules.
- (2) *Connection primitives*, such as *glueright* in Figure 3, to specify that their arguments—which must be boxes in the same layer—are to be joined in a particular manner.
- (3) An *inclusion primitive*, *inside*, which specifies that one box is to be placed inside another. The minimum distances between their edges are again supplied by ALI from a table of design rules.
- (4) *Minimum-size primitives*, such as *xmore* in Figure 3, which specify the minimum size of a box along a certain direction. Default minimum sizes are provided by ALI from a design rule table.
- (5) *Transistor primitives*, which create depletion mode and pass transistors.
- (6) *Contact primitives*, which create contacts between layers and connect boxes to them.

Note that no absolute positions or dimensions for any rectangle can be specified with these primitives. All the rectangles of a layout can be stretched and compressed (up to a minimum size), and all can float in any direction. If one single characteristic is to be used to separate ALI from other layout systems, this must be it. Most of the power of ALI and most of the problems one faces in its implementation are consequences of this fact.

It is important to remember that, in order for a layout produced by ALI to be free of design rule violations, any two rectangles in it must be related in some way. ALI makes no inferences as to the relations between boxes beyond those implied by the transitivity of some primitive operations (i.e., if *above*(*a*, *b*) and *above*(*b*, *c*) are stated, *above*(*a*, *c*) need not be stated). Although the system generates a good number of relations automatically for the user, particularly in connection with the cell mechanism (see the next subsection), there is still a fair amount of drudgery left for the user in making sure that this requirement is met. A brief discussion of the computational complexity of the automatic generation of relations between boxes can be found in Section 2.3.3.

**2.2.3 Cells.** Perhaps the most powerful feature of ALI is its procedurelike mechanism for the definition and creation of *cells*. A cell is a collection of related rectangles enclosed in a rectangular area. Rectangles that are inside a cell are of two types: *local*, which are invisible from the outside, and *parameters*, which can interact in a simple and well-defined manner with rectangles outside the cell.

A cell is *defined* by specifying its local objects, its formal parameters, and the relations among all of them. Once a cell has been defined, it can be *instantiated* as many times as desired by specifying the actual parameters for the instance, much the same way as one invokes a procedure or function in a procedural language. The result of instantiating a cell is to create a brand-new copy of the prototype described in the cell definition with the formal parameters connected to the actual parameters.

cell *shift* (left *l:shiftbus*; right *r:shiftbus*)

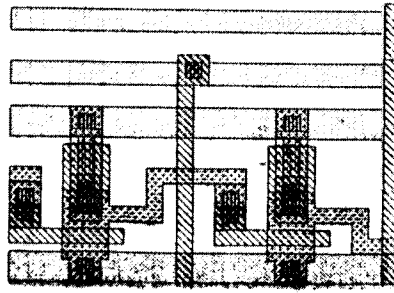


Fig. 4. A sample cell definition header and an instance of the cell defined.

A cell definition is made up of a *header*, in which the formal parameters are described, a set of *local box declarations*, and a *body* in which the relationship between the parameters and the local boxes, as well as those among local boxes, is specified.

The header describes the names and types of the parameters and the side of the bounding rectangle through which they come into contact with the inside of the cell. The header of a cell (using the type *shiftbus* defined in Section 2.2.1) and an instance of it are shown in Figure 4.

Cells may have any number of parameters on each of their four sides. The order in which they are listed in the cell header describes their relative positions. Horizontal parameters (i.e., those touching the cell on the left or right) are assumed to be listed in top-to-bottom order and vertical parameters in left-to-right order.

The body of a cell is very much like an ALI program. For example, Figure 5 shows a complete cell definition that consists of a variable number of *shift* cell instances connected sequentially together with two of its instances. Note that cells are instantiated by the **create** statement and that the parameter list of the cell contains both box parameters and other parameters (an integer in this case) in separate lists. Note also that recursion has been used to define this cell; this highlights the fact that ALI has the full power of PASCAL at its disposal.

When an instance of a cell is created, it can be given a name, provided that the name given has been declared as a rectangle of the standard simple type *virtual*. The relationship of the rectangle bounding a newly created cell to any other rectangle of the layout can be specified in the standard manner by calls to the primitive operations. This is a vital feature since, in many cases (i.e., *above*, *below*), stating a relation between two cell instances  $c_1$  and  $c_2$  immediately implies a relation between every pair of rectangles  $r_1$  and  $r_2$  such that  $r_1$  is part of  $c_1$  and  $r_2$  part of  $c_2$ .

There are two important ways in which the cell mechanism helps in the automatic generation of constraints between boxes. When an object of a structured type is passed as a parameter to a cell, its component boxes are separated from top to bottom (if it is a **left** or **right** argument) or from left to right (if it is a **top** or **bottom** argument). The order of the separation is determined by

```

cell shiftregister (left inbus:shiftbus;
                  right outbus:shiftbus)
  (length:integer);
box
  temp:shiftbus;
begin
  if length = 1 then
    create shift (inbus, outbus)
  else begin
    create shift (inbus, temp);
    create shiftregister (temp, outbus) (length - 1)
  end
end
end.

```

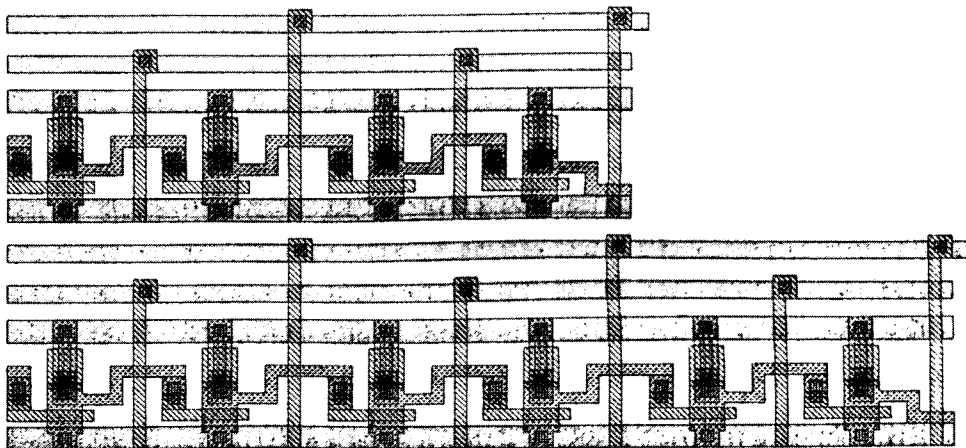


Fig. 5. A cell definition and two instances of it generated by a simple ALI program.

applying recursively the following rules: array elements are separated in the order of their indices, and bus elements are separated in the order in which they were specified in the bus declaration. Thus, in the example of Figure 5, the components of parameter *inbus* would be separated from top to bottom. The second mechanism involves the automatic separation of cells that share a parameter; thus, in the example of Figure 5, the individual instances of *shift* are separated automatically, since adjacent instances share a parameter.

The cell mechanism gives the ALI user the ability to describe layouts in a truly hierarchical manner. A proper ALI design, very much like a well-structured program, consists of a hierarchy of cell instances, with only a small amount of information at a given level (the parameters of the cell instances at that level) being visible from the immediately higher level. For example, the layout given in Figure 2 consists of four instances of the same cell stacked vertically. That cell in turn is defined in terms of three other cells, one of them being the cell shown in Figure 1, which is in turn defined in terms of three other cells.

Much of the power and generality of the cell mechanism of ALI comes from the absence of absolute positions and sizes in a layout specification. In particular,

two instances of the same cell may have radically different sizes depending on the actual parameters used to create them, as exemplified by Figures 1, 2, and 5. We believe that no cell mechanism can be said to be truly general unless the sizes of its arguments and local rectangles, as well as their relative distances, are determined at the time the cell is instantiated.

There are some penalties involved in the use of the cell mechanism. In particular, ALI generates separations between cells in a manner that is oblivious to what is inside them. That is, the minimum separation between cells, as far as ALI is concerned, is the maximum of all the minimum separations for two layers in the design rules, thus creating a certain wastage. We believe that the advantages gained by the ability to separate cell instances as units are well worth this penalty, which will generally be a small percentage of the total area.

Another source of wastage is the fact that cells are restricted to be bounded by a rectangle, so the packing of cells that have irregular shapes results in a certain amount of unused space. The rectangular shape of the cells is a fundamental characteristic of ALI: The introduction of irregularly shaped cells is simply not possible without completely redesigning the language. However, the waste introduced because of this restriction can be avoided in most particular cases through some code modifications.

### 2.3 Implementation Issues

The previous section described the user's view of ALI. In this section we discuss briefly some of the problems to be solved when trying to go from an ALI program to a layout that satisfies the relations stated in it. We first give an overall description of the system as currently implemented; then we discuss the method used to assign locations and sizes to the layout elements; and, finally, we describe the concept of *completeness* and how it is checked.

**2.3.1 Overall Implementation.** The current version of our system has been implemented as follows. The ALI program is first translated into standard PASCAL. The resulting PASCAL program is then compiled and linked with a precompiled set of procedures that implement the primitive operations, and the resulting object module is then run. The output of this program (generated entirely by the primitive operations) is a set of linear inequalities and connectivity relations among the layout elements. The inequalities are then solved to generate a layout or examined by a program that checks their logical completeness, and the connectivity information can be used to simulate the circuit laid out.

The design rules are incorporated as a table, which is used by the primitive operations to produce the linear inequalities. Thus, changing the design rules for our system requires only changing this table.

**2.3.2 Placement.** As explained above, one of the results of running an ALI program is a set of linear inequalities that embody the relations between the layout elements. These inequalities are of the following simple form:

$$x_i - x_j \geq d \quad (d \geq 0)$$

where the variables are the coordinates of the corners of the boxes that form the layout and the constants are either user supplied (e.g., as in the second argument

of the *xmore* primitive) or extracted from the table of design rules by the system itself.

This set of inequalities should be solved so as to generate placements for the boxes that compose the layout in such a way as to minimize its total area. In order to perform this task efficiently, we require that no inequality in the set involve both  $x$  and  $y$  coordinates. This restriction allows us to minimize the total area by minimizing the maximum  $x$  and  $y$  coordinates of any point independently, at the cost of reducing the range of the relations between boxes that we can express. We cannot, for instance, handle rectangles whose sides are not parallel to the Cartesian axes or express aspect ratios directly.

We have now a sufficiently simple problem so that it can be solved in *time proportional to the number of inequalities in our set*. (All layouts that can be expressed in ALI can be generated by a program that produces a constant number of inequalities per rectangle.) This is done by a version of the topological sort algorithm applied to the  $x$  and  $y$  coordinates independently. This algorithm assigns to each point the lowest possible coordinate while minimizing the largest coordinate of all points.

The form of the inequalities that we allow is rather restrictive; it is sufficient, however, to describe the design rules given in [13] for NMOS, and the efficiency gained in return for this simplicity seems to us to be a good trade-off. A more subtle consequence of the simplicity of the inequalities and the method we use to solve them is that undesirable stretching can occur, since we have no way to specify a maximum size for any object. This is not a common occurrence, and the user can in all cases guard against such stretching by careful selection of the primitive operations used. It is nonetheless an additional burden placed on the designer.

The choice of an efficient placement algorithm over expressibility power and a reduced degree of user convenience has been quite conscious in this particular case. We feel that every reasonable measure should be taken to keep the complexity of the placement problem linear, given that the size of layouts is currently large ( $10^7$  rectangles) and is growing fast. Widening the class of linear inequalities acceptable is almost certain to make linear-time solutions impossible [2].

**2.3.3 Completeness.** ALI programs do not involve absolute sizes or positions of boxes and are, to a great extent, independent of the design rules. These characteristics make it clearly desirable to ensure, in a way other than checking the finished layout, that the layout described by a program will be free of design rule violations. The following paragraphs describe a way of ensuring freedom from design rule violations in a manner that is independent of the actual design rules used to generate the final placement. The description may be somewhat cryptic; the interested reader is referred to [17] for further details.

A layout generated by an ALI program is *complete* if, for any two boxes  $a$  and  $b$  whose types make it possible for them to interact in the final layout, either

- (1)  $a$  and  $b$  are explicitly stated to be in contact by some primitive operation or
- (2)  $a$  and  $b$  are, explicitly or through the transitivity of primitive relations, stated to be separated in either the  $x$  or the  $y$  direction by a minimum amount that depends on their types.

From this definition, it should be clear that testing completeness of a cell instance involves computing the transitive closure of a graph. Therefore, the complexity of the operation will be  $O(n^3)$  where  $n$  is the number of boxes in the cell. It is thus not feasible to test a large layout for completeness in a direct way.

Fortunately, completeness can be checked hierarchically. Let us look only at the objects at the highest level of the hierarchy of boxes that defines a layout, that is, those boxes (including cell boundaries) defined globally in the ALI program that generated the layout. If these objects are related in a complete manner and the cell instances used at this level are also complete, then the whole layout is complete.

Thus one can check the completeness of a layout by successively checking cell instances for completeness, thereby reducing the complexity of the process to  $O(m^3)$  where  $m$  is the largest number of boxes local to a cell instance in the layout. This process can be reduced further, since not every cell instance needs to be checked. For example, if a cell is defined by a straight-line program, checking one instance for completeness suffices, as one instance of the cell will be complete if and only if all of its instances are [17]. The case of cells with branches and iteration is not quite so simple. Yet we are confident—and our experience tends to confirm this belief—that checking the completeness of a few carefully selected instances of any cell definition will be enough to guarantee that the cell definition is complete.

The end result is that completeness has the flavor of a static, almost syntactic, property for all nonmalicious examples, and it is much easier to check in a well-structured layout than it is to check design rule violation by the standard means on the final layout.

Finally, a word about the possibility of taking an incomplete layout specification and automatically completing it. The general problem of generating an optimal completion is NP-complete, but the simpler version of generating any completion for graphs embedded in a grid (as our layouts are) can be solved in  $O(n^2)$  steps [17]. The question of how much area will be wasted by such a completion algorithm will have to wait for some experimentation, but there is no question of its usefulness.

#### 2.4. Experience with ALI

The current implementation of ALI has shown the soundness of most of our original ideas. The system is efficient and the language easy to learn, and the layouts the system produces are relatively dense (for example, an ALI program written without concern for area optimization produced a layout which was about 30 percent larger than a similar layout packed by hand on a graphics editor). Unfortunately, this evidence has been gathered mostly from people who had a hand in designing or implementing ALI. Perhaps a more reliable evaluation of ALI must wait until a substantial number of users not involved in its design can give an informed opinion. We hope to obtain this evidence before long, since ALI is currently being used in a VLSI design course.

Since, for the sake of expediency in getting a prototype running, very little effort was invested in error recovery, and since no mechanism for integrating separately produced layout pieces was provided, the current system is useful mostly for teaching purposes and experimentation. It must be emphasized that

this is a result of implementation choices and not of any intrinsic limitation on the approach we have taken.

The problems of the current system that we plan to address with the next version are the following:

- (1) *Memory Requirements.* The solution of the system of linear inequalities requires large amounts of memory. We will use a different algorithm that is slightly less efficient in terms of time but requires an order of magnitude fewer memory locations for a typical large layout.
- (2) *PASCAL Problems.* The current ALI has exactly the same type structure as PASCAL. The lack of generic types and dynamic arrays has made the task of writing general-purpose tools (PLA generators, routers, etc.) inside ALI more difficult than it ought to be. The next ALI will have the notions of generic types and dynamic arrays.
- (3) *Connecting Primitives.* Certain objects, such as contacts, are used frequently enough to warrant making them part of the language.
- (4) *Separate "Compilation" Facilities.* Clearly, large layouts will have to be generated in pieces, which is something that our current system cannot do.

#### ACKNOWLEDGMENTS

We would like to thank Jose Mata, Vijaya Ramachandran, and Jerry Spinrad for their help in the implementation of ALI and Jean Vuillemin, Scot Drysdale, and the referees of the paper for their comments. We also thank Bruce Arden for his advice and support. Finally, special thanks go to David Dobkin and Andrea LaPaugh for their help throughout the ALI project.

#### REFERENCES

1. ACKLAND, B., AND WESTE, N. A pragmatic approach to topological symbolic IC design. In *VLSI '81*, J.P. Gray (Ed.). Academic Press, New York, 1981, pp. 117-129.
2. ASPVALL, B., AND SHILOACH, Y. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. In 20th Annual Symposium on Foundations of Computer Science, San Juan, P.R., Oct. 29-31, 1979, pp. 205-217.
3. BATALI, J., MAYLE, N., SHROBE, H., SUSSMAN, G., AND WEISE, D. The DPL/Daedalus design environment. In *VLSI '81*, J.P. Gray (Ed.). Academic Press, New York, 1981, pp. 183-192.
4. DAVIS, T., AND CLARK, J. SILT: A VLSI design language (preliminary draft). Unpublished manuscript, Digital Systems Laboratory, Stanford Univ., Stanford, Calif.
5. EICHEMBERGER, P. Lava: An IC layout language. Unpublished manuscript, Electronics Research Laboratory, Stanford Univ., Stanford, Calif.
6. FAIRBAIRN, D.G., AND ROWSON, J.A. ICARUS: An interactive integrated circuit layout program. In Proceedings of 15th Annual Design Automation Conference (Las Vegas, Nev., June 19-21, 1978), pp. 188-192.
7. FRANCO, D., AND REED, L. The cell design system. In Proceedings of the ACM IEEE 18th Design Automation Conference (Nashville, Tenn., June 29-July 1, 1981), pp. 240-247.
8. HOLT, D., AND SAPIRO, S. BOLT—A block oriented design specification language. In Proceedings of the ACM IEEE 18th Design Automation Conference (Nashville, Tenn., June 29-July 1, 1981), pp. 276-279.
9. JOHANNSEN, D. Bristle blocks: A silicon compiler. In Proceedings of 16th Design Automation Conference (San Diego, Calif., June 25-27, 1979), pp. 310-313.
10. JOHNSON, S.C. The LSI design language *i*. Bell Laboratories, Murray Hill, N.J. Unpublished manuscript.



11. LIPTON, R.J., NORTH, S.C., SEDGEWICK, R., VALDES, J., AND VIJAYAN, G. ALI: A procedural language to describe VLSI layouts. In Proceedings of 19th Design Automation Conference (Las Vegas, Nev., June 14-16, 1982), pp. 467-474.
12. LIPTON, R.J., SEDGEWICK, R., AND VALDES, J. Programming aspects of VLSI (preliminary version). In Conference Record of 9th Annual Symposium on Principles of Programming Languages (Albuquerque, N. Mex., Jan. 25-27, 1982), pp. 57-65.
13. MEAD, C., AND CONWAY, L. *Introduction to VLSI Systems*. Addison-Wesley, Reading, Mass., 1980.
14. MOSLELLER, R.C. REST: A leaf cell design system. In *VLSI '81*, J.P. Gray (Ed.). Academic Press, New York, 1981, pp. 163-172.
15. SASTRY, S., AND KLEIN, S. PLATES: A metric free VLSI layout language. In Proceedings of the 1982 Conference on Advanced Research in VLSI, 1982, pp. 165-169.
16. TRIMBERGER, S. Combining graphics and a layout language in a simple interactive system. In Proceedings of the ACM IEEE 18th Design Automation Conference (Nashville, Tenn., June 29-July 1, 1981), pp. 234-239.
17. VIJAYAN, G. Completeness of VLSI layouts. VLSI Memo 1, Dept. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, N.J., Sept. 1982.
18. WILLIAMS, J.D. STICKS—A graphical compiler for high level LSI design. In *AFIPS Conference Proceedings*, vol. 47: 1978 National Computer Conference (Anaheim, Calif., June 5-8, 1978). AFIPS Press, Arlington, Va., 1978, pp. 289-295.

Received May 1982; revised November 1982; accepted December 1982