ALI: a Procedural Language to Describe VLSI Layouts

Richard J. Lipton†
Stephen C. North†
Robert Sedgewick‡
Jacobo Valdes†
Gopalakrishnan Vijayan†

†Department of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ

‡Computer Science Department
Brown University
Providence, RI

Abstract: ALI is a procedural language to specify VLSI layouts. It allows the designer to describe layouts without reference to the sizes and positions of the layout elements or to the distances between them. Among the interesting characteristics of ALI are that it does not need design rule checking, is easy to extend, facilitates the division of labor and permits the easy update of a layout to new design rules or to new processes. The general features of the language and the experience gained with a preliminary implementation of it are described.

1. Introduction

This paper describes a procedural language to specify VLSI layouts. The main feature of this language is that it allows its user to design layouts at a *conceptual level* in which neither sizes nor positions (absolute or relative) of layout components may be specified. Mostly as a consequence of this, ALI simultaneously (i) makes the layout task more like programming than editing, (ii) eliminates the need for design rule checking after the layout is generated, (iii) permits the creation of easy to use cell libraries and (iv) 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 ([4], [7], [8], [14], [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], [5], [6], [9], [10], [11], [16]).

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

- 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 generators, involves writing Pascal routines to solve the problem that invoke ALI cells to generate the layouts.
- Creating tools that are *simple to use and easy to learn.* In particular, we want to avoid tools whose behavior is unpredictable. Many programs which 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 circuits.
- 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, on the top of fig. 1 three simple cells are shown with the intended connections between them shown by

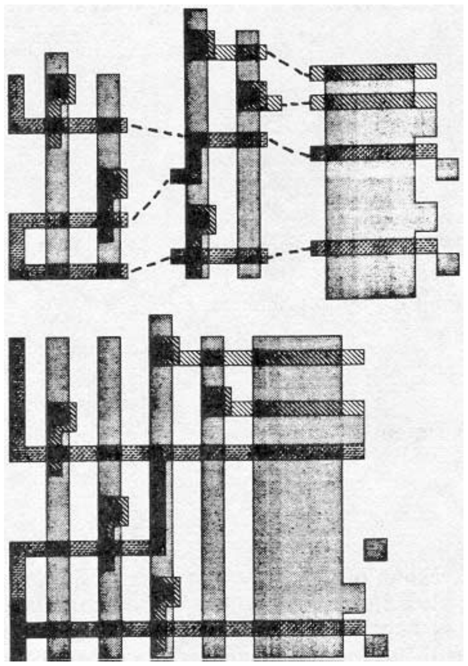

Fig. 1
Three separate cells and the result of connecting them along the dotted lines

dotted lines; on 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.

- Facilitating *hierarchical design.* Even when a single designer is involved, the ability to view a layout as a hierarchy, with as 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 topological relations among the layout elements of the lower level visible to the higher one.
- 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. Fig. 5 (see the end of the paper) 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* in 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 fig. 5) can be avoided. The same ALI program can also generate layouts using different design rules by running it with a new module incorporating the new design rules.
- To avoid the *need for expensive, 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 avoiding the need for large mainframes and permitting 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 the pieces shown on the top of fig. 1 is free of design rule violations, their combination shown on the bottom of the same figure will be guaranteed by ALI to be free of rule violations regardless of the stretch that takes place as a consequence of connecting them. ALI in fact requires far fewer computing resources than many design rule checking programs.

If one theme is to summarize our approach, it is that the VLSI layout task can be profitably thought of as a *programming task,* and that much is to be gained by consciously attempting to apply knowledge about of programming to this activity. 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 more powerful tools possible such as loaders, linkers and compilers.

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

2. 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 direction of the axis 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 boxes specified in the program. For example, fig. 2 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* for instance). 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 3.1.

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

```
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.2
A simple ALI program and the layout it produces

glueright extends its first argument to the right to intersect its second argument, and *xmore* makes the size of its first arguments along the x axis at least as large as the value of the 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 on the primitive operations of ALI is given in section 3.2.

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 4.2. The program also produces connectivity information about the rectangles in the layout. This information is then used by a switch level simulator that predicts the behavior of the circuit as laid out.

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. 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. The concept of completeness of layout descriptions is discussed briefly in section 4.3.

3. Main features of the language

This section describes how ALI appears to its user. Its three subsections deal, in turn, with the *type structure*, the *primitive operations* of the language and the *cell mechanism*. Familiarity with the general features of Pascal will help the reader greatly, because ALI has been built on top of Pascal and has inherited most of its features. We have tried however, to make the section as self contained as possible without going beyond the scope of this paper.

3.1. The type structure of ALI

As the example of fig. 2 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 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 fig. 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), which correspond directly to the array and record structured types of Pascal. ALI, like Pascal, permits the creation of new *user defined* types that can be either simple or structured. For example, in fig. 2, 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 made up of a number of metal rectangles, and the fragment

horizontal : htype

inside the **box** section, creates an object of that type 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 which 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 used in conjunction with the cell mechanism to automatically generate separations between boxes. We will be 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.

3.2. The primitive operations of ALI

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 fig. 2, *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 [13] -- has about twenty primitive operations which can be arranged in the following groups:

1 *Separation primitives*: such as *above* in fig. 2, 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 fig. 2, 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*, that specifies that one box is to be placed inside another. The minimum distances between their edges are again suplied by ALI from a table of design rules.

4 *Minimum size primitives*: such as *xmore* in fig. 2, 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 will make 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 on the computational complexity of the automatic generation of relations between boxes can be found in section 4.3.

3.3. The cell mechanism of ALI

Perhaps the most powerful feature of ALI is its procedure-like 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 to the outside, or *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.

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, are 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 3.1) and an instance of it are shown in fig. 3.

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

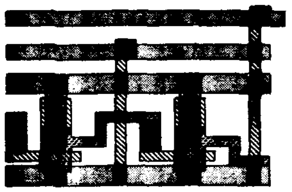

Fig. 3
The header of a cell definition
and an instance of the cell

The body of a cell is very much like an ALI program. For example, fig. 4 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.

When an instance of a cell is created it can be given a name, provided that the name given has been declared as

```
cell shiftregister (   left inbus : shiftbus;
                       right   outbus : shiftbus )
                     ( length : integer );
    const
        maxlength = ...;
    var
        i : integer;
    box
        conn : array [1..maxlength] of shiftbus;
    begin
        if length = 1 then
            create shift ( inbus, outbus )
        else begin
            create shift ( inbus, conn[1] );
            for i := 2 to length-1 do
                create shift ( conn[i-1], conn[i] );
            create shift ( conn[length-1], outbus )
        end
    end;
```

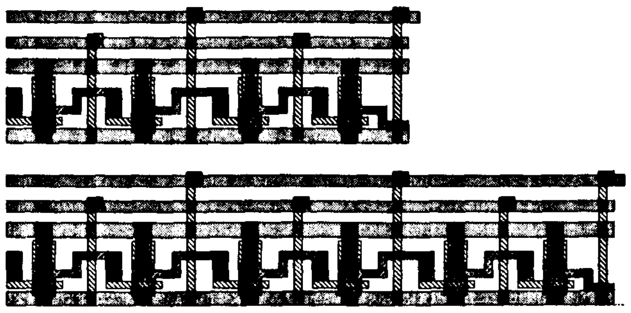

Fig. 4
A cell definition and two instances of it

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 a composite object is passed as a parameter to an 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 applying recursively the following rules: array elements are separated ordered by their indices and bus elements in the order in which they were specified in the bus declaration. Thus, in the example of fig. 4, 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 fig. 4, 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, will consist 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 fig. 5 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 fig. 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 figs. 1, 4 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 which 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 this penalty will be generally a small percentage of the total area and is well worth the advantages gained by the ability to separate cell instances as units.

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.

4. Implementation issues

The previous section attempted to describe the user 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 discuss the method used to assign locations and sizes to the layout elements and then the concept of *completeness* and how it is checked.

4.1. 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 to change this table and to recompile the module of primitive operations.

4.2. 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:

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

The variables are the coordinates of the corners of the boxes that form the layout. The constants are either user supplied (as in the second argument of the *xmore* primitive, for instance) or extracted from the table of design rules.

The set of inequalities have to be solved 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 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 to be solved in *time proportional to the number of inequalities in our set.* This is done by a version of the topological sort algorithm [12] 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 seem to us like a good tradeoff. 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 the 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].

4.3. 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 insure that the layout described by a program will be free of design rule violations in a way other than checking the finished layout. The following paragraphs describe a way of insuring 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

(i) a and b are explicitly stated to be in contact by some primitive operation, or

(ii) 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 which 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: 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 instance, 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. Although the case of cells with branches and iteration is not as simple, we have failed to write a single cell of which more than three different instances need to be checked in order to guarantee its completeness.

The end result is that completeness has the flavor of a static, almost syntactic, property for all non malicious examples, and is much easier to check in a well structured layout than design rule freedom 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) seems to be solvable in $O(n^2)$ steps. 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.

5. Experience with ALI

The current implementation of ALI has shown the soundness of most of our original ideas. The layouts produced are relatively dense, the whole process efficient and the language easy to learn. Unfortunately, this evidence has been gathered mostly from people who had a hand in designing or implementing ALI. Perhaps a more reliable response to the utility of ALI, ought to 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.

The fact that very little effort was invested in error recovery for the sake of expediency in getting a prototype running, and that no mechanism for integrating separately produced layout pieces was provided make the current system 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. We expect to implement a new system over the summer which can be distributed and will be able to handle full sized layouts.

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

(1) Memory requirements. The solving process requires very large memory. We will use a different algorithm for solving the linear inequalities which 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...) 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. A must for large layouts.

Acknowledgements

We would like to thank Jose Mata, Vijaya Ramachandran and Jerry Spinrad for their help in the implementation of ALI and the simulator, and Jean Vuillemin and Scot Drysdale for their comments. We also want to thank Bruce Arden for his advice and support.

The work of Richard Lipton has been partially supported by grants MCS8023-806 from NSF and N00014-81-K-0681 from ONR. Stephen C. North is being supported by Bell Laboratories. Robert Sedgewick's work was partially supported by NSF grant MCS80-17579. The work of Jacobo Valdes has been supported by ONR grant N00014-81-K-0681.

6. References

[1] Ackland, B., Weste, N., "A pragmatic approach to topological symbolic IC design. design," *VLSI'81*, pp 117-129, John P. Gray ed., Academic Press.

[2] Apsvall, B. and Shiloach Y., "A Polynomial Time Algorithm for Solving Systems of Linear Inequalities with Two variables per Inequality", pp 205-217, *Proc. of the twentieth IEEE Symp. on Foundations of Computer Science*, 1979.

[3] Baker, C. M., "Artwork Analysis Tools for VLSI Circuits," M. S. Thesis, MIT, EECS Department, June, 1980.

[4] Batali, J., Mayle, N., Shrobe, H., Sussman, G., Weise, D., "The DPL/Daedalus Design Environment," *VLSI'81*, pp 183-192, John P. Gray ed., Academic Press.

[5] Davis, T., Clark, J., "SILT: A VLSI Design Language (Preliminary Draft)", unpublished manuscript, Digital Systems Laboratory, Stanford University.

[6] Eichemberger, P., "Lava: an IC layout language", unpublished manuscript, Electronics Research Laboratory, Stanford University.

[7] Fairbairn, D., Rowson, "Icarus: an Interactive Integrated Circuit Layout Program", pp 188-192, *15th Design Automation Conference Proceedings*, 1978.

[8] Franco, D., Reed, L., "The Cell Design System", pp 240-247, *18th Design Automation Conference Proceedings*, 1981.

[9] Holt, D., Shapiro, S., "BOLT -- A Block Oriented Design Specification Language", pp 276-279, *18th Design Automation Conference Proceedings*, 1981.

[10] Johannsen, D., "Bristle-Blocks", pp 310-313, *16th Design Automation Conference Proceedings*, 1979.

[11] Johnson, S. C., "The LSI Design Language i", unpublished manuscript.

[12] Knuth, D. E., *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms*, Addison-Wesley, 1971.

[13] Mead, C., Conway, L., *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[14] Mosleller, R.C., "REST: A leaf cell design system," *VLSI'81*, pp 163-172, John P. Gray ed., Academic Press.

[15] Sastry, S., Klein, S., "PLATES: A Metric Free VLSI Layout Language", pp 165-169, *Proceedings of the 1982 Conference on Advanced Research in VLSI*, 1982.

[16] Trimberger, S., "Combining Graphics and a Layout Language in a Simple Interactive System," *18th Design Automation Conference Proceedings*, 1981.

[17] Vijayan, G., "Completeness of VLSI Layouts", Princeton University, Department of Electrical Engineering and Computer Science Technical Report (in preparation).

[18] Williams, J., "STICKS, A Graphical Compiler for High Level LSI design", pp 289-295, *Proceedings of the 1978 NCC*, 1978.

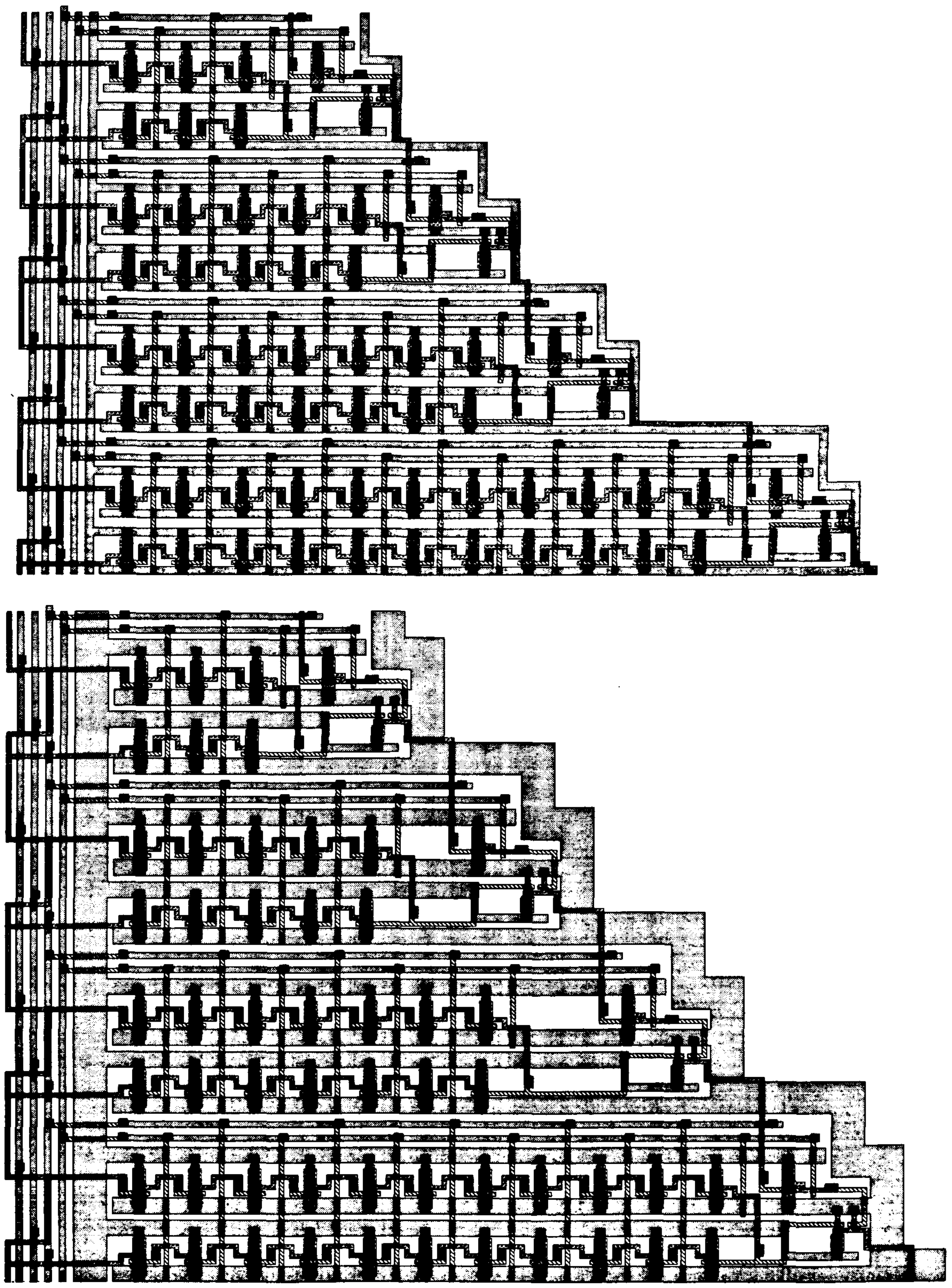

Fig. 5
Two ALI layouts generated by programs
differing only in the values of four constants