Programming Aspects of VLSI (Preliminary Version)

Richard J. Lipton
Department of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ

Robert Sedgewick
Computer Science Department
Brown University
Providence, RI

Jacobo Valdes
Department of Electrical Engineering and Computer Science
Princeton University
Princeton, NJ

Abstract: Two components of a VLSI design environment being built at Princeton are described. The general theme of this effort is to make the design of VLSI circuits as similar to programming as possible. A conscious attempt is being made to apply experience in the design of large software systems to the creation of an appropriate environment for VLSI circuits. The two components described are a procedural language to specify circuit layouts and a switch-level circuit simulator for layout produced with this language. They have been chosen for presentation because many issues in their design are very similar to the issues that arise in the design of programming languages and software environments.

1. Introduction

In this paper we describe two of the most important components of a VLSI design environment: a layout language and a switch level circuit simulator. Both systems grew out 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.

Our main thesis is that the VLSI design 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 new activity. Thus we have given much weight to principles learned during the travel from absolute machine language programming to the use of high level languages, such as working in as high level a language as the efficiency of the final product permits, creating tools in which the division of tasks among designers can be easily done, and making sure that the tools are extensible.

We feel that this approach has helped us create better tools for some of the central tasks of VLSI design. We also know that there is much room for improvement and would like to help convince the community of people interested in programming language design that there are fresh and important challenges in this relatively new direction.

The remainder of this paper is divided into two main sections. The section that follows is devoted to describe ALI, our layout specification language. This is the one tool in our system where most of the familiar programming issues arise so more space is devoted to its description. A shorter section describes a switch level simulator for layouts generated with the aid of ALI.

2. ALI: an overview

ALI is a procedural language to describe VLSI layouts. It differs radically in its basic philosophy from most current systems for this task in that it 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 truly flexible and general purpose libraries of frequently used layout pieces and (iv) provides the designer with the mechanisms to describe a layout hierarchically so that objects at one level of the hierarchy are truly hidden from other levels.

Many of these advantages are gained through a *delayed binding* approach: the user may not specify the absolute location or the size for any element of the layout. These positions and sizes are determined after the whole layout has been completely specified at a conceptual level.

In the sections that follow we review the principles on which the language is based, describe its operation and provide a short discussion of why we feel that our approach to layout design has advantages over those of other *procedural* layout languages and over the more popular *graphics-based systems.*

2.1. General principles

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 of which it is composed, and stating the relations that hold between them.

To declare a rectangle the ALI user specifies its *name* and its *type*. The types of ALI have the same structure as the Pascal types. A rectangle can be of a *simple type* (having no internal structure) or of a *structured type*. There are a small number of *standard types* for rectangles, all of them simple. The structured types are the *array* (a collection of rectangles of the same type) or a *bus* (a collection of rectangles 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. Also like Pascal, it is a *strongly typed* language.

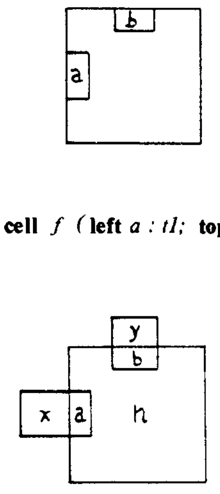
The relations between the rectangles that make up a layout are specified in ALI through calls to a small set of primitive operations. All such operations take as arguments rectangles of simple types. As an example, one such operation is *touches*, which indicates that its two arguments (which must be of the same simple type) have a certain amount of area in common. There are other operations (such as *above, below, left,* and *right*) which allow the user to state that two rectangles (of any simple type) are separated in the direction of the cartesian axes by a minimum distance (supplied by ALI) that depends on their types. Other primitive operations allow the user to connect rectangles of different types, to create pass transistors or implanted transistors, specify minimum sizes for rectangles and so on.

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 our current prototype, which is based on NMOS, has about twenty primitive relations. Most of these primitives are dictated by peculiarities of NMOS. A much smaller set of primitive operations (maybe six) are truly needed; all the others can be written in terms of the more basic ones. It is important however to know that ALI expects any two rectangles in a layout to be related in some way, and will make no inferences beyond those implied by the transitivity of some primitive operations (i.e., if a is above b and b above c it need not be explicitly stated that a is above c).

Notice that ALI *does not allow its user to specify absolute positions or dimensions for any rectangle.* All the rectangles of a layout can be stretched and compressed (up to a minimum size which depends on their type) and all can float in any direction. If one single characteristic is to used to separate ALI from all other layout systems we know of, this must be it. Most of the power of ALI and most of the problems one faces while implementing ALI are consequences of this fact.

Perhaps the most powerful feature of ALI is its procedure-like mechanism for the definition and creation of *cells*. By a cell we mean a rectangle which has some internal structure more complex than that of an array or bus. 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 between 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 *glued* (another primitive operation) to the actual parameters. A pictorial oversimplification of the relationship between a cell definition and an instantiation is shown in Fig. 1.



Definition: **cell** *f* (**left** *a : t1;* **top** *b : t2);* ... **end;**

Creation: **create** *f (x, y)* **named** *n;*

Fig. 1
A simple representation of a cell definition and of an instance of the cell.

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 powerful mechanism since in many cases (i.e., *above,* *below...*) specifying 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. Other ways in which cell instantiation helps define implicit relations between layout elements is explained in the next section.

The cell mechanism gives the ALI user the ability to describe layouts in a truly hierarchical manner. A proper 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.

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. 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.

2.2. An ALI program

In an attempt to make the description of the previous section a little more concrete we will now exhibit a simple ALI program and discuss some of its features. Our current prototype system has been implemented as a superset of Pascal† and therefore ALI programs look very much like Pascal programs as exhibited by the sample program of fig. 2.

The ways in which this program differs from a Pascal program and some details of the semantics of ALI not described earlier are explained briefly in the following paragraphs.

(1) The header is trivially different from a Pascal program header.

(2) The *boxtype* section is an addition to Pascal. In this section user defined types for rectangles are described. In our example *metal* and *poly* are standard ALI types (corresponding to the NMOS layers of metal and polysilicon) and *power, ground, vtp* and *htp* are user defined types.

(3) The *boxvar* section is an addition to Pascal. In it the symbolic names of the rectangles that make up the layout are defined and given a type. Thus, in our example *hc, vc* and *squares* are created, each one a structured rectangle having several component rectangles which can be accessed through the standard array indexing and record field extraction mechanisms of Pascal. The type *virtual* is a standard ALI type used to indicate that those variables will become cell instances.

(4) The *cell* definitions (see the definitions of *white* and *black*) are also additions to Pascal. We have not given bodies for the cells of our example to avoid making it too long, but cell bodies have the same general form as ALI programs. Note that the header of a cell definition not only specifies what parameters and of what type are required to instantiate the cell but also specifies *where* (on which side of the rectangle that encloses the cell) the actual parameters are to be located (i.e., *left, right, top* or *bottom*). When more than one parameter is specified for a cell side, the convention is that the order in which the parameters are given determines their left-right (for the top and bottom sides) or top-bottom (for the left and right sides) order. Although not used in the example, this is a very powerful convention which can be used to implicitly define relations between many rectangles by simply using them as arguments in a cell instantiation. Another convention that allows the implicit specification of many relations among rectangles in an intuitive way is the following. When a rectangle of a structured type is passed as a parameter, as in our example, a left-to-right or top-to-bottom order for its component rectangles is implicit. This order is the one generated by applying the following two rules recursively: array elements are ordered from lower bound to upper bound and record fields in the order in which they are listed.

(5) The use of *create* in the executable part of the ALI program is another addition to Pascal. The effect of *create* is to instantiate a cell definition, possibly giving a name to the instance being created. An important effect of instantiating a cell is that the actual parameters given are related to the resulting rectangle by the fact of being passed as parameters. Thus the instantiation

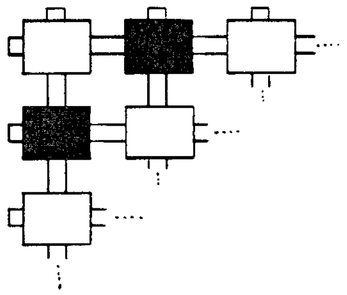
create *white (nt, nb, nl, nr)* **named** *silly*

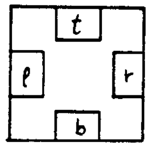
would immediately imply that nt is above *silly.* It is quite common that most of the relations between rectangles in a well structured layout are specified in this implicit way.

(6) Finally, our example contains calls to the procedures *above* and *left* which are not defined. These are ALI primitive operations.

†This was done to speed up implementation. No lasting commitment to Pascal as a base language has been made.

```
{ This program lays out a checkerboard-like pattern made up of two different types of cells like the one pictured below }
chip checkerboard;
  const
    nrows = ...;     { number of rows }
    ncols = ...;     { number of columns }
  boxtype
    power = metal;
    ground = metal;
    vtp = array [1..2] of poly;     { vertical links }
    htp = bus     { horizontal links }
            vdd : power;
            gnd : ground;
            data : metal;
          end;
  boxvar
    hc : array [0..nrows, 0..ncols] of htp;   { vertical connectors }
    vc : array [0..nrows, 0..ncols] of vtp;   { horizontal connectors }
    squares : array [1..nrows, 1..ncols] of virtual;   { the squares of the board }
  var
    row, col : integer;

  function sameparity ( x, y : integer ) : boolean;
    begin sameparity := not odd ( x + y ) end;

  cell white ( top t : vtp; bottom b : vtp; left l : htp; right r : htp );
    { ...Here goes the definition of the body of a white cell... }
    end;

  cell black ( top t : vtp; bottom b : vtp; left l : htp; right r : htp );
    { ...Here goes the definition of the body of a black cell... }
    end;

  begin
  { lay out the checkerboard pattern }
    for row := 1 to nrows do
      for col := 1 to ncols do
        if sameparity (row, col) then
          create white ( vc[row-1, col-1], vc[row, col-1], hc[row-1, col-1], hc[row-1, col]) named squares[row, col]
        else
          create black ( vc[row-1, col-1], vc[row, col-1], hc[row-1, col-1], hc[row-1, col]) named squares[row, col];
  end.
```





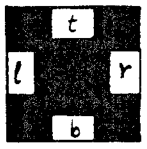


Fig. 2
An ALI program and a pictorial representation of the layout it defines.

2.3. From ALI program to layout

The semantics of an ALI program are the following. Every time a call to a primitive operation is encountered during the execution of an ALI program, a few *linear inequalities* that involve the coordinates of the corners of the rectangles are generated. These inequalities *constrain* the sizes and relative positions of the rectangles that make up the layout. The solution of the set of linear inequalities generated during the execution of an ALI program constitutes the layout described by the program. The linear inequalities are generated and solved so that the resulting layout will have the bounding rectangle of smallest dimensions among all those that satisfy the relations stated in the program.

The inequalities generated by the primitive relations between rectangles are selected to guarantee that the resulting layout will satisfy some set of *design rules* [9]. These rules are, in principle, sufficient to guarantee that the circuit represented by the layout can be fabricated. The fact that the layout process is *design rule driven* means that there is no need for a costly design rule check step to verify the validity of a layout generated by ALI.

Not every possible ALI program specifies a realizable layout. A trivial way in which an ALI program may fail to do so is by describing an impossible situation such as "*a* is above *b*, *b* above *c* and *c* above *a*". An ALI program which is free from errors of this type will be said to be *consistent*. Another way in which an ALI program may fail to specify a valid layout will be to have two objects in the ALI program that are not related in any way (i.e., they do not "touch", neither one is "above" or "below" the other, nor are they related by any other primitive operation). An ALI program is *complete* if it does not contain any errors of this kind. A consistent and complete ALI program specifies a layout.

We clearly would like to be able to determine whether an ALI program is consistent and complete efficiently. We also need to solve efficiently the large number of linear inequalities that ALI programs will generate. A brief description of the computational complexity issues that these problems pose and our solutions for them can be found in the next section.

2.4. Complexity issues in the implementation of ALI

We have just described three non trivial computational problems that had to be solved in any ALI implementation: determining whether an ALI program is consistent and complete, and solving the possibly very large system of linear inequalities generated by an ALI program. We will now describe briefly how these problems are handled.

Before discussing the details however, it is important to realize that VLSI circuit layout is one area where the asymptotic behavior of an algorithm is truly important. The number of objects that have to be manipulated is quite large (10^6 or more rectangles may be part of a large commercial layout today) and is growing rapidly. Perhaps less obvious is the fact that the space requirements of an algorithm can be more critical than its running time. Linear time algorithms may not be acceptable if they require also linear space: storing 10^6 objects approaches the limits of the address capabilities on most of today's medium size machines and solutions that use secondary storage in a substantial way will in all likelihood be too slow. It is therefore essential for the tool designer to keep in mind the time and space complexity his or her tools.

We will discuss first the problem of testing consistency and completeness and then describe how to solve the systems of linear inequalities produced by ALI programs.

Conceptually, the solutions to completeness and consistency checking are simple: testing consistency translates into checking whether a directed graph is acyclic, and testing completeness involves finding the transitive closure of two graphs, taking their union and testing a simple predicate on the resulting graph.

There are however some practical problems. The most severe is that transitive closure is too expensive an operation to consider in this case since it requires roughly $O(n^3)$ time and $O(n^2)$ space, and n can be very large. Even the test for a cycle which may appear to be as simple a solution as one could ask for (requires linear time) may require too much space (linear space is needed in the worst case).

Fortunately these problems can be solved efficiently in most practical cases by observing that they can be tested on a cell by cell basis. This observation leads to a solution that checks the consistency and completeness of a layout generated by an ALI program by checking it level by level along the hierarchy of cell instances that defines the layout. Thus the n in the time and space complexities of the process turn out to be the maximum number of rectangles in a cell, a distinct improvement over the total number of rectangles in the whole layout for "well structured" layouts. Hierarchical designs are therefored favored in ALI for reasons other than readability or aesthetics.

This idea of "hierarchical solutions" appears to us a promising paradigm for dealing with problems of this type in which the total number of objects to be manipulated is enormous but they are -- in most practical cases -- reasonably well organized in a tree-like hierarchy with small depth and a reasonably small branching factor.

We turn now to the question of how to solve the very large systems of linear equations that ALI programs will produce. At the moment we solve this problem by avoiding it: our set of primitive relations between rectangles is rich enough to specify a layout and restricted enough so it will only generate linear equations of the form

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

or

$$x_i = x_j$$

A set of equations of this type can be solved in linear time using techniques similar to those employed in solving PERT problems (variations of topological sort [8]). Note that the solution to this problem requires again linear space making a hierarchical solution highly desirable.

We are far from having solved all our problems in this area. The set of primitive relations that we currently use to guarantee that the linear inequalities generated can be solved efficiently is sufficient to specify a layout, but can hardly be called convenient. Quite a few important properties of a layout (such as current densities on wires or making the dimensions of certain rectangles be fixed ratios) can be captured in linear equations that do not permit as simple and efficient a solution as those produced by our reduced set of primitive relations.

We are quite far from a general solution to the question of what are the useful properties of a layout that can be captured in systems of linear inequalities allowing a hierarchical solution. Most of the work published on the solution of special systems of linear equations (such as those with only two variables per inequality [2]) is of no use to us because the memory requirements of the algorithms described are far too big given the size of our problems and the possibility of hierarchical solutions is not considered. Progress in this area would make ALI a better tool that it is now.

2.5. An apprisal of ALI

ALI was conceived as a system which would lack the more obvious shortcomings of other layout systems we were familiar with ([1], [4], [10], [12]), namely (i) the need for rule checking and compaction, (ii) lack of true modularity and (iii) lack of extensibility. In this section we quickly review these problems and point out what we believe to be the contributions of ALI towards their solution.

2.5.1. Design rule checking and compaction

All layout systems that we are aware of, let the user specify layouts that violate the design rules of the technology being used. This implies that some effort has to be made later on to validate layouts produced with their help.

Most systems (graphics based systems in particular), force the user to worry about minimizing the area of a layout as it is being generated. To ease this burden, the design process in many of these systems includes a *compaction* step in which an attempt is made to automatically reduce the size of the layout generated without violating the design rules.

Compactors and checkers are invariably large programs ([3], [6]) requiring hours of CPU time in a large machine to compact or verify medium sized layouts. The asymptotic time complexity of these programs is $O(n^{3/2})$ or worse in the "average case" and they require linear space (or use secondary storage heavily). Clearly, this approach will not scale well as the number of components per circuit continues to increase.

The possibility of performing these two operations hierarchically has not been considered until fairly recently. This may be due in part to the sheer complexity of the checkers and compactors, but undoubtedly is also related to the lack of a flexible cell mechanism in most of these systems. Note that since the checking and compaction process occur after the layout has been produced, information about hierarchical relationships between elements of the layout would have to be given to these programs separately.

A complete ALI program will never produce a layout with a design rule violation so no rule checking is necessary. The process of solving the linear inequalities generated by an ALI program is the equivalent of compaction. We have however two advantages in this process: (i) that of having the hierarchical information used to generate the layout available and (ii) that of having designed the constraints so they can be solved hierarchically.

Design rule checkers and automatic compactors spend most of their running time trying to extract semantic information from the layout. This information is something which the designer must have had very clear in his or her mind when producing the layout but that either was never communicated to the system or has been discarded. It seems more reasonable to ask the designer to make this information explicit and saving it than spending much time and effort trying to recreate it at a later time.

In contrast, ALI requires all semantic relations in a layout to either be given explicitly or to be easily obtainable through transitivity. This information is then used to guarantee that the layout produced will satisfy the design rules and be of minimum overall size among those which satisfy it.

2.5.2. Modularity

How easy is it to divide cleanly a design task among several designers? This seems to be a key question when considering how modular a layout system really is, and for just about all the systems we know about the answer seems to be "not easy at all".

The reason is their lack of a flexible cell mechanism and their requiring *absolute sizes and locations* for most of the layout elements. In these systems, it is necessary for individuals in a design group to know the absolute location and size of those elements that are common to two or more of the pieces into which the layout has been decomposed. In many cases any change in the sizes and positions of these common elements, no matter how small, will force several designers into redoing a substantial part of their work.

The same characteristics that makes cooperation between several designers hard also precludes the creation of libraries of frequently used cells in most of these systems. Because of the reliance on absolute sizes and positions, different instances of the same cell definition can differ only in whichever manner the person that defined the cell allowed. Such cell definitions would most likely be either too particular to be of any use or so parameterized so as to be too hard to use.

In contrast ALI offers the possibility of dividing a layout much as one divides a large software project. Individual workers need only agree on the header of the cells they have to define much as they would only have to agree on the header of the procedures in a software effort.

To pursue the analogy to programming, we feel that most layout systems available today are like absolute assemblers without a general subroutine mechanism. ALI is, at least, a relocatable assembler with a subroutine call instruction.

2.5.3. Layout systems as parts of larger systems

If one envisions an environment where VLSI design is akin to programming, tools more sophisticated than those available today will have to be built. It is therefore relevant to ask how current systems stack up when considered as possible stepping stones for future tools.

A tremendous inherent disadvantage of graphics-based systems in this respect is that they are an evolutionary dead end: no way to further automate the layout process is considered. Most procedural systems are open tools to a certain extent: one can conceive of adding a front end to them which would translate functional specifications into layouts, for instance, farfetched as it may sound today. Such a front end for a graphics based system is hard to envision since it will need to process visual information like a human.

The original work on ALI dates back to the time when the authors were looking for an intermediate language into which high level descriptions of VLSI algorithms could be compiled. Although we will not yet claim success on this endeavor, the fact that ALI had to be usable as part of a larger system has been kept in mind throughout its design.

2.5.4. Some final comments

Another aspect of ALI in which we believe it to be superior to most other procedural layout languages is in its type structures and strongly typed nature. It is not common for procedural systems to allow the definition of arrays or records of standard types and even less common to have user defined types and to provide type checking for them. Most of them simply deal with standard types. We feel that type checking at the layout specification time will prove to be as useful as it is in the specification of programs.

On the other side of the coin, several procedural languages provide their users with facilities that are not available in ALI. The most common one is automatic routing, where pairs of points to be connected (absolute positions) are given to the system which automatically connects them. It is not at all clear how such facilities fit with the overall philosophy of ALI and therefore not clear that they could easily be added to it.

Another weakness of ALI is its lack of a graphics interface. At the moment, a graphical representation of the layout produced by an ALI program is used much as one would use the listing produced by a program to verify its correct operation. Clearly there is much room for improvement in this respect, but we are not yet certain which approach fits better with the overall philosophy of ALI.

3. A simulator for ALI

The ALI language allows the VLSI designer to create layouts. While those layouts are guaranteed to satisfy the NMOS design rules there still is the issue of the "correctness" of the layout. By this we mean whether or not the layout correctly implements the desired circuit. One of the key complications in VLSI is that a layout may be incorrect for a wide range of reasons. These range from purely digital errors to layouts that are digitally correct but misbehave due to analog errors.

In order to check that such a wide range of errors, VLSI designers use a number of circuit simulators. A classic kind of simulator is the *logic-level* simulator. This simulator models the circuit as a connection of gates and memoryless one-way wires. These gates, of course, compute one of a number of standard boolean functions: typical gates include and's, or's, nand's, and so on. While such simulators are simple and useful, they do not capture the bidirectional nature of NMOS.

Another important class of simulator is the *analog-level* simulator such as SPICE [7]. This kind of simulator accurately captures the detailed nature of NMOS circuits. It does this by operating at a very low level; hence, such simulators require great amounts of computing resources. Therefore, it is rare to use such simulators on entire VLSI circuits.

A third type of simulator studied here is the so called *switch-level* simulator. These simulators such as MOSSIM [5] lie mid-way between the above two kinds. They attempt to correctly model the bidirectional nature of NMOS circuits, and yet they retain the conceptual simplicity of logic-level simulators. Such simulators appear to be quite useful to VLSI designers in catching a variety of errors.

The simulator for ALI is a switch-level simulator. It differs from previous simulators in a number of critical ways.

First, and most important, it does *not* operate on all possible NMOS circuits. It will reject any circuit that does not satisfy what we call the *clocking axiom.* Roughly, this axiom forces the VLSI circuit to use its clock phases so that on each clock phase the circuit divides into "acyclic" pieces. It appears to be the case that most well "structured" circuits will satisfy it. Most common VLSI circuits, such as PLA's and shift registers, do indeed satisfy it.

Second, our simulator is guaranteed to run in *essentially linear* time. By this we mean that it runs in time $O(T\alpha(T,T))$ where T is the number of transistors in the layout and as usual $\alpha(n,n)$ is the Ackermann Inverse function. (Recall $\alpha(n,n)$ is less than 5 for all "possible" values of n [11].) We have an initial version of the simulator written in Pascal running on a VAX 11/750 which processes circuits having several thousand transistors in only a few seconds of cpu time.

Third, our simulator also guarantees that for the circuits that satisfy the clocking axiom, the final state exists and is unique. By this we mean that for such circuits not only will the simulation eventually stop, but moreover, the final answer is the unique final answer. Since NMOS circuits can in general, contain hazards and race conditions this seems to be a critical advantage to our approach.

3.1. The model

We represent as in [5] each NMOS circuit as a set of nodes that are connected by pass transistors. There are *input* nodes, *pull-up* nodes, and *normal* nodes. Input nodes provided a strong externally generated value, while normal nodes can only store a charge but cannot generate a value. Pull-up nodes are connected via a pull-up resistor to a voltage source. They are at a high value unless they are connected to ground.

Pass transistor act as switches that connect such nodes. A pass transistor is described by a triple:

(gate, source, drain).

Source and drain are not distinguished. Such a device acts as a switch: if the gate is high, then the source and drain are connected; otherwise, the source and drain are not so connected.

3.2. The clocking axiom

Let C be an NMOS circuit. Also, let Φ be a set of input nodes to C. Then Φ is a *clock set for* C provided that at most one node in Φ is ever high at a time. Note, this property is *not* a function of the circuit C, but instead reflects how we *use* the circuit. In many circuits Φ is simply the two input nodes ϕ_1 and ϕ_2 that control the two clock phases.

We now will state precisely the clocking axiom that is central to our approach to switch-level simulation. In order to do this, we need a few simple definitions.

Let ϕ be in the set Φ. Let T_ϕ be the set of transistors in C with their gates either equal to ϕ or not in Φ. The importance of this class of transistors is that when ϕ is high only these transistors can potentially be on. We say that two nodes x and y are *ϕ-connected* provided there are nodes $x_1,\ldots,x_k$ so that $x = x_1$, $y = x_k$ and for each $1 \leq i \leq k-1$, there is a transistor (g_i,x_i,x_{i+1}) in T_ϕ. Clearly, *ϕ-connectivity* is an equivalence relationship, and so it partitions the nodes of C into equivalence classes, in a natural way. We further define a directed graph on these equivalence classes as follows: if (g,x,y) is a transistor in T_ϕ, then place an arc from the equivalence class of g to that of x. (Note, this is well-defined since x and y lie in the same such class.) Let us call this graph the *control graph* of ϕ.

We are now in position to state the clocking axiom. A circuit C with clock set Φ satisfies the *clocking axiom* provided for each ϕ in Φ, the control graph is acyclic. Recall this means that as a directed graph there is no directed cycle. Roughly, this axiom means that all "feedback" is controlled by "clocks" from Φ. The consequences of this axiom are discussed in the next section.

3.3. The simulation method

The *state* of a circuit is the array of values associated with its nodes. Given a circuit and a state S we say that S' is a *next* state provided the circuit can eventually reach S' simply by the actions of its pass transistors. Such a state S' is a *final* state provided further that no additional change can occur without external changes to the nodes. Clearly, not all states ever reach a final state: it is quite possible for a circuit to oscillate forever. However, we can prove the following:

Theorem: *Any circuit that satisfies the clocking axiom always has a unique final state.*

This then is the primary motivation for the clocking axiom. Such circuits are guaranteed to be "well behaved". Not only can they never oscillate, but in addition they always reach the same final state. Clearly, this is critically important to a VLSI designer. Without such a basic theorem it would be possible for a circuit to work some times but not others.

The proof of the theorem is actually quite constructive and it supplies a very fast way of finding final states. Let us fix one of the nodes from Φ as being on, say ϕ. Since Φ is a clocking set there always is at most one such node. (If no such node exists then we can trivially modify the method that follows.) Now recall that the control graph of ϕ is acyclic; therefore, we can by topological sort, order its vertices into a list so arcs only go from earlier vertices to those later in the list. Now since each of these vertices represents a set of nodes from the circuit we can in a natural way order the nodes of the circuit. Our simulator then proceeds as follows: it "fires" each transistor once in the order induced by the above ordering of the nodes. More exactly, if (g,s,d) is a transistor and (g',s',d') is also a transistor, then fire (g,s,d) first provided g occurs before g' in the node ordering. If g equals g' then the order of the transistors is immaterial. An induction shows that this always yields the unique final state of the circuit.

Since each transistor fires at most once the algorithm runs in T times the cost of each such firing where T is the number of transistors. But it is easy to see that the firing of a transistor involves merging two equivalence classes. This operation can be performed using the operations described in [9], yielding the claimed time bound of $O(T\alpha(T,T))$. On an example with about 3,000 transistors the current simulator takes about 3 seconds of cpu time on a VAX 11/750. Note, the topological sort and hence the transistor ordering can also be done as a preprocessor step.

The application of our simulation method depends of course on the clocking axiom. We have observed that many classes of circuits satisfy it. These include shift registers, PLA's, and more generally combinatorial logic. However, many simple circuits will not satisfy our clocking axiom. A typical such circuit is one in which the clock is "anded" together with another control line. While it is true that this will lead to circuits that do not satisfy the clocking axiom, it is easy to extend our ideas to such circuits. The key to this extension is to observe that as long as the interaction of the clock lines and the control lines is without feedback, then our methods will generalize.

Acknowledgements

We would like to thank Jose Mata, Steve North, Vijaya Ramachandran and Jerry Spinrad for their help in the implementation of ALI and the simulator, Jean Vuillemin and Scot Drysdale for their comments. We also want to thank Bruce Arden for his advice and support. Special thanks are due to G. Vijayan for his probing questions, ingenious solutions and tireless work in the implementation of the prototype of ALI.

The work of Richard Lipton has been partially supported by grants MCS8023-806 from NSF and N00014-81-K-0681 from ONR. Robert Sedgewick's work was partially supported by NSF grant MCS80-17579. The work of Jacobo Valdes has been supported by NSF grant MCS79-05524 and ONR grant N00014-81-K-0681.

4. References

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

[5] Bryant, R. E., "MOSSIM: A switch-Level Simulator for MOS LSI," pp 786-790, *18th Design Automation Conference*, 1981.

[6] Corbin, L. V., "Custom VLSI Electrical Rule Checking in an Intelligent," pp 696-701, *18th Design Automaton Conference Proceedings*, 1981.

[7] Fan, S. P., Hsueh, M. Y., Newton, A. R., Peterson, D. O., "MOTISC: A new circuit simulator for MOSLSI circutis," *IEEE Proc. Int. Symp. Circuits and System*, pp 700-703, 1977.

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

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

[10] Mosleller, R.C., "REST: A leaf cell design system," *VLSI '81* pp 163-172.

[11] Tarjan, R. E., "Efficiency of a Good but Not Linear Set Union Algorithm", *JACM*, vol. 22, no.2, pp 215-225, 1975.

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