CELL DESIGN IN PROLOG

Ashar A Butt

Master's Report Plan II
Professor Alvin Despain
Computer Science Division
University of California, Berkeley

ABSTRACT

Automatic synthesis of CMOS cells is considered in the programming idiom of Prolog. Cells are specified at the boolean level using Prolog structures and a layout of the cell is generated at the CIF (Caltech Intermediate Format) level.

A set of Prolog programs has been developed for this purpose. A three pass approach has been taken. The first pass takes boolean equations and transforms it into a net-list of transistors corresponding to static CMOS design style. The second pass (implemented by Rick Mcgeer) takes the transistor net-list and lays it out on a virtual grid in the style of a Gate Matrix. The third pass takes the sticks layout on the virtual grid and compacts it onto a lambda grid to produce CIF code.

A few test cells have been passed through this cell synthesizer. Example cases include an exclusive-nor gate, a cross-coupled nand gate pair, a hand laid out exclusive-nor gate passed through the compactor only.

Part of this research was sponsored by Defense Research Projects Agency (DoD) Arpa Order No. 4871 Monitored by Naval Electronic Systems under Contract No. N00039-84-C-0089

February 25, 1986

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
25 FEB 1986		**00-00-1986 to 00-00-1986**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Cell Design in Prolog	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Automatic synthesis of CMOS cells is considered in the programming idiom of Prolog. Cells are specified at the boolean level using Prolog structures and a layout of the cell is generated at the CIF (Caltech Intermediate Format) level. A set of Prolog programs has been developed for this purpose. A three pass approach has been taken. The first pass takes boolean equations and transforms it into a net-list of transistors corresponding to static CMOS design style. The second pass (implemented by Rick Mcgeer) takes the transistor net-list and lays it out on a virtual grid in the style of a Gate Matrix. The third pass takes the sticks layout on the virtual grid and compacts it onto a lambda grid to produce CIF code. A few test cells have been passed through this cell synthesizer. Example cases include an exclusive-nor gate, a cross-coupled nand gate pair, a hand laid out exclusive-nor gate passed through the compactor only.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**52**	

CONTENTS

1. Introduction.

2. Cell Specification & Transistor Net-list Generation.

3. Layout Methodologies.

4. Sticks In Prolog (SIP).

5. Compilation of SIP.

6. Conclusions.

References.

Appendix A

Appendix B

CELL DESIGN IN PROLOG

Ashar A Butt

Master's Report Plan II
Professor Alvin Despain
Computer Science Division
University of California, Berkeley

1. INTRODUCTION

Recently, many research efforts have been directed towards allowing higher and higher levels of chip specifications. From computer based circuit-capture tools to standard cell libraries (such as NCR's standard cell library for CMOS) , onto functional specification languages [1], the search is on to find efficient ways of generating Integrated Circuit (IC) layouts, without paying too great a penalty in the area and performance of the IC. Currently the systems dealing with the translation of functional specification to mask layouts, (loosely grouped as "silicon compilers") generate IC's that are inferior in both performance and area to layouts obtained by manual methods [2].

Several different approaches have been taken to the silicon compilation problem [3]. The MacPitts silicon compiler [1] generates silicon layouts from algorithmic descriptions of the circuit. The approach essentially assumes a general target architecture for all ICs, and then attempts to connect pre-defined modules ("organelles") within that target architecture. All functional descriptions are realized on the same target architecture. The compiler does not attempt to take advantage of redefining the organelle layout for a particular local environment.

The problem with existing silicon compilers is the typical trade-off between general and specific solutions to a problem. By gearing the layout generator to a specific IC architecture, and tuning it until it approaches the density obtainable by manual layout tech-

niques the automatic layout of a particular class of IC's can be optimized. The key issue in such a system would then be the ease with which the tuning (or incremental addition of intelligence) could be performed. A rule based system is then the best choice for such a system, because of the separation of the control and knowledge aspects, as opposed to a purely algorithmic system. Prolog [4] is a good choice for implementing rule based systems, because the inherent nature of Prolog is suited for the implementation of rules embodying knowledge.

Other researchers have taken this approach to silicon compilers, though not necessarily in Prolog. The Design Automation Assistant [12] tried to encapsulate expert design knowledge in a rule based system, implemented in OPS5. This design knowledge was geared towards the design of a specific processor chip, the IBM 370 processor. The designs produced by the system were rated to be better than a novice designer, and as good as the designs of some of the "better" IC designers.

As part of a special purpose Prolog silicon compiler, an automatic module generator has been developed. A high level description of a functional module, possibly containing feedback, is provided to the module generator. The module generator then takes the input boolean structures, and outputs a net list of transistors using static CMOS design style. This netlist of transistor is then laid out on a virtual grid in the form of Gate Matrix [5]. This virtual grid representation (sticks format) is then compacted onto a lambda grid [2] , obeying all necessary design rules for a particular process, and finally translated into Caltech Intermediate Format (CIF) [6] code.

Each of the above steps has been divided into a pass. The first pass to generate the transistor net list. The second pass to lay out the electrical net list in a sticks form on a virtual grid. The third and final pass to convert the sticks language into CIF code describing the cell in terms of process layers. With the exception of the final pass (which produces CIF), the other passes output files of Prolog structures. These Prolog structures provide a language for describing the intermediate phases of the cell design. Using the consult facil-

ity of Prolog these structures (consisting primarily of Prolog factual assertions) are read into the next pass. This simplified communication between the passes.

The first pass takes as input a file of type <filename>.bln (where 'bln' stands for boolean). The output of the first pass is a file <filename>.net (where 'net' stands for net-list). This <filename>.net is given to the second pass which produces a file <filename>.sip (where 'sip' stands for Sticks In Prolog). The <filename>.sip file is given as input to the third pass, which produces a geometry description of the cell in <filename>.cif file, which can be given as input to a CIFPLOT program, or an IC fabrication facility.

Breaking up the cell design process in this way allows entry into the cell design process at any of the intermediate levels; e.g, a cell could be specified as a netlist of N and P type transistors to the second pass, as opposed to a boolean specification to the first pass, or alternatively a description of the cell at the sticks level could be provided directly to the third pass. Several such examples have been tried, and are illustrated in the section on examples.

Another advantage of this multi-pass approach is to reduce the domain of a particular problem, so that optimization can be performed independently at every step of the problem. This approach is consistent with the approach to the silicon compiler problem in general, in which translation of an Instruction Set Architecture specification to an IC layout is the goal. There also, the problem has been partitioned into many steps, with each step performing optimization at that level. Furthermore, it allows for experimentation at different levels, e.g. different layout methodologies can be tried at the layout pass like Weinberger arrays or PLA's instead of the gate matrix style. The same sticks interface can then be used to obtain a CIF description of the cell.

This system has similarities to the Silicon Conveter [7] project at Bell Labs in the way the tasks have been partitioned. However the Silicon Converter was meant primarily to complement the Ida toolset [8], and is based on the i-language, a C-like language. The

system considered here is based entirely on Prolog, developing the expressiveness of a language like the i-language through Prolog structures, representing different levels of the design abstraction.

2. CELL SPECIFICATION & TRANSISTOR NET-LIST GENERATION

The first pass of the module generation system deals with the conversion of a boolean equation to a net-list of connected N and P-type transistors. The input structures are specified in a file <filename>.bln and are 'consulted' by the first pass into the program database. After processing the boolean structures (described below), the program generates output structures representing the transistor net-list and writes them out to file <filename>.net. That file can then be consulted by other programs for laying out the transistors as crossing poly and diffusion wires connected by either poly or different layers of metal.

2.1. Input Structures

Static CMOS, which is based on the duality of N and P type circuits [9], is best described as the NOT of an AND-OR expression. Any arbitrary network of N transistors (with it's dual network of P transistors) can be mapped onto the boolean form of Not(And-Or), as described in Figure 1. Translation of such a boolean specification is particularly straight forward, as each AND sub-expression maps to a series sub-tree of N transistors, and each OR sub-expression maps to a parallel sub-tree of N transistors. The transistor net-list specification is completed by implementing the dual of the boolean expression in P transistors; where the dual of AND is OR, and the dual of OR is AND. Note that this is far more straight-forward than a Nand-Nor description, as the series network of N transistors (for e.g) represents the Nand of the inputs (the gate signals) only when that is the only sub-tree between the output and the ground. If, for e.g., there is another parallel network in between the output and the series sub-tree, then the circuit would not have a convenient nand expression, but rather a not(and-or) expression.

Another reason for choosing the not(and-or) formulation is the ease of a recursive solution to translating the expression to a net-list. The top-level of the not(and(List)) can be

expressed as nand(List), but then the list would contain "and"s internally, and only the top level would be a "nand". By leaving the top-level as an "and", a concise recursive solution can be formulated, so that the boolean compiler can traverse the nested "List" mapping "and"s to series networks, and "or"s to parallel networks. For example, the following rules handle the entire translation of boolean equations to a net-list of transistors.

```
%Rules for generating transistors.
convert(and([H]),From,To,Polarity) :-        %Single gate left.
      convert(H,From,To,Polarity), !.

convert(and([H|T]),From,To,Polarity) :-      %Serial transistors.
      convert(H,From,New,Polarity),
      convert(and(T),New,To,Polarity),!.

convert(or([H|T]),From,To,Polarity) :-       %Parallel transistors.
      convert(H,From,To,Polarity),
      convert(or(T),To,From,Polarity),!.     %Maximize Source-Drain conn.

convert(and([]),From,To,Polarity).           %Do nothing.

convert(or([]),From,To,Polarity).            %Do nothing.

convert(Gate,Source,Drain,Polarity) :-           %Form a transistor.
      atomic(Gate),
      bindNet(Drain),                  %Gen. node if new.
      write(xstor(Polarity,Source,Gate,Drain)),
      write('.'), nl.
```

So, every time an "and" structure is encountered a new intermediate node is created in the uninstantiated variable "New". When the last "convert" rule is invoked with the "Drain" variable uninstantiated, a new unique integer is created to represent that node. This way the node numbers are created by traversing the N-network from the ground node on up. Encountering an "or" structure does not entail creating new network nodes, but rather a parallel network with the Source-Drain connections reversed. The reason for the reversal of the Source-Drain connections on one of the parallel transistors is to maximize the number of Source-Drain connections. This anticipates the fact that the transistors will be laid out in rows, and the more the number of Source-Drain connections, the more the number of transistors that will be laid out in a single row. The Source and Drain are reversible in that it really doesn't matter which node of the transistor is called Source or Drain. They are

merely helpful names for indicating the intended direction of the current flow. Doing the reversal thus permits a more compact layout.

The Prolog structures used to implement boolean expressions are 'and', 'or' and 'compl' (for complement). All boolean expressions are specified within the compl structure. The form of the compl structure is

```
compl(Out = and(List)).
compl(Out = or(List)).
```

The 'List' variable should instantiate to a list of atoms corresponding to input signal names or further embedded 'and' 'or' structures. The 'Out' variable will be the signal name for the output of the boolean function. The following are a few examples;

```
compl(z = and([x,y]).        {Corresponds to z = NOT(AND(x,y)).}
compl(out = and([z,or([x,y])])). {Corresponds to out = NOT(AND(z,OR(x,y))).}
compl(sig = or([and([a,b]),or([x,y])])). {sig = NOT(OR(AND(a,b),OR(x,y))).}
```

Incorporating feedback is as simple as using the same signal names in specifying the input-output relations of the feedback circuit. For example,

```
compl(z = and([set,y])).
compl(y = and([reset,z])).
```

specifies a cross-coupled nand-gate pair, as shown in Figure 2. Each pure combinational path is specified in a compl statement, and multiple compl statements allow feedback paths.

Another design element commonly used in CMOS circuit design is the transmission gate. A separate Prolog structure 'pass' has been allowed for the specification of transmission gates. The same compl statement is used to specify the transmission (or pass) gate. The form of the statement is,

```
compl(Out = pass(In, Control)).
```

where 'Out' should instantiate to the signal name of the output node, 'In' should instantiate to the input node, and 'Control' should instantiate to the signal controlling the passage of the input signal to the output node. The logic symbol corresponding to the structure is shown in Figure 3. The output of the pass gate is the same as the input signal, with no

implicit inversion, even though it is inside a compl statement. Further embedded structures are not allowed within the pass structure.

Although arbitrary nesting is allowed within the "and" and "or" structures, there is a practical limit to the amount of transistors that can form a functional CMOS circuit. This limit is imposed by the resistivity of the the pulldown net in CMOS. If increased ad-infitum with minimum sized transistors, the outputs of certain transistors may become over-loaded in supplying the appropriate voltage levels at the right nodes. To guarantee functionality, the resulting layout should be simulated, and then the transistor dimensions should be adjusted to supply the right amount of drive and capacitive loadings.

2.2. Output Structures.

The output of the first pass produces a file containing assertions of Prolog structure 'xstor'. Each 'xstor' assertion corresponds to a MOS transistor. The arguments of the 'xstor' structure describe the type of the transistor (e.g P type or N type), and the connectivity of the transistor nodes. The form of the structure is,

```
xstor(Polarity, Source, Gate, Drain).
```

where 'Polarity' instantiates to atoms 'pmos' or 'nmos', depending on the type of transistor being represented. 'Gate', 'Source' and 'Drain' instantiate to atoms representing signal names, or integers representing nodes in the electrical network not connected to any external signals. These node numbers are generated by the first pass, that transforms the boolean equations into a transistor netlist. Any of these terminal names with the same atom or integer represent an electrical connection, i.e all terminals with the same name (atom or integer) are on the same electrical node.

For example,

```
xstor(nmos,out,in,gnd).
xstor(pmos,vdd,in,out).
```

represents a static CMOS inverter with input signal 'in' and output signal 'out'. This

electrical circuit is shown in Figure 4. Power lines are represented by atoms 'vdd' and 'gnd'. There is one P-type transistor and one N-type transistor. As another example, the following net-list represents the net-list of transistors corresponding to a cross-coupled nand-gate pair, as specified in Sec 2.1 above, and illustrated in Figure 5.

```
xstor(nmos,gnd,set,0).
xstor(nmos,0,y,z).
xstor(pmos,vdd,set,z).
xstor(pmos,z,y,vdd).
xstor(nmos,gnd,reset,1).
xstor(nmos,1,z,y).
xstor(pmos,vdd,reset,y).
xstor(pmos,y,z,vdd).
```

Note that the same names were used for wires connected to externally specified signals, and where intermediate transistor connections needed to be specified, unique integers were generated by the program.

2.3. Example Session of the First Pass

As an example of how the first pass is invoked by the interpreter the following script is reproduced below. The operating system used is UNIX 4.2 running the C-shell, and the Prolog is interpreted C-Prolog. The machine name is "dali". The input structures specifying the cross-coupled nand-gate are stored in file "crossnand.bln". The source code of the first pass is in file "genxstor.pl". In giving the name of the input file to the first pass the ".bln" extension is omitted, and the program automatically generates file "crossnand.net" which is displayed after the session.

Here is a run of the session,

```
dali% cat crossnand.bln
%input set of equations.
compl(z = and([set,y])).
compl(y = and([reset,z])).
dali% cprolog
CProlog version 1.2
[ restoring file /b/hprg/Prolog/Cprolog/environment ]

yes
| ?- ['genxstor.pl'].
```

```
genxstor.pl consulted 2988 bytes 1.283333 sec.

yes
| ?- genxstor(crossnand).
crossnand.bln reconsulted -8 bytes 0.083335 sec.

yes
| ?- halt.

[ Prolog execution halted ]
dali% cat crossnand.net
xstor(nmos,gnd,set,0).
xstor(nmos,0,y,z).
xstor(pmos,vdd,set,z).
xstor(pmos,z,y,vdd).
xstor(nmos,gnd,reset,1).
xstor(nmos,1,z,y).
xstor(pmos,vdd,reset,y).
xstor(pmos,y,z,vdd).
dali%
```

The above translation of boolean expressions on a VAX 750 took about the same amount of time that it took to consult the input file, i.e instantaneous under normal load conditions.

2.4. Implementation and Possible Enhancements

The implementation of the transistor net-list generator (Appendix A) has been geared towards easy addition of new input structure types. The control has been separated from the rules for mapping boolean structures onto transistor networks. As an example the following rules handles the general case of mapping an "and" structure onto a transistor network;

```
convert(and([H|T]), From ,To, Polarity) :-   %Series transistor
      convert(H,From, New, Polarity),
      convert(and(T),New,To,Polarity),!.

convert(Gate,Source,Drain,Polarity) :-              %Form a transistor
      atomic(Gate),
      bindNet(Drain),              %Generate unique integer, if required.
      write(xstor(Polarity,Source,Gate,Drain)),
      write('.'), nl.
```

The addition of other data types can be handled by adding new rules for mapping the structure onto a transistor network.

By adding rules for structures other than "and" and "or", further input data types can

be added to the repertoire of the net-list generator. For example, a data type can be added for a two-input exclusive-nor gate, thereby resulting in a more compact layout for such functional modules as compared to realizing them as the NOT of an AND-OR expression. Any functional module that has a more compact layout directly, in terms of transistors required, when compared to the number of transistors required by realizing that function with the existing input data-types can be more efficiently realized by adding rules for that particular functional module. This may be especially useful if that functional module is used extensively in the design.

3. LAYOUT METHODOLOGIES

Given a net-list of transistors, there are a multitude of ways to lay them out on a two dimensional rectangular grid. There are several different methodologies that limit the topology of the circuit in order to achieve some compromise between area-efficiency and tractability of layout problem. Three of the most common ones are Gate Matrix, Weinberger arrays [2] and PLA's [2]. The one chosen for initial implementation is the Gate Matrix method.

Gate matrices and Weinberger arrays differ from PLA's in that the latter represents two-level logic, whereas the former allow arbitrary levels of logic (i.e arbitrary nesting of boolean equations). Another difference is that, in CMOS, Gate Matrices and Weinberger arrays have static implementations, whereas PLA's conventionally have a dynamic implementation [9].

3.1. Weinberger Arrays

They are similar to Gate matrices in that they also allow arbitrary levels of logic. They differ in that the vertical columns are all diffusion. The horizontal interconnect is all metal. Transistors are formed by dropping a horizontal piece of poly at the right point (called a tap) across the vertical diffusion column. By dropping two taps on the same diffusion column, a pair of series transistors can be formed. Weinberger arrays have an easy implementation in NMOS because the pullup net is simply depletion mode transistors. In static CMOS, the pullup net is formed from the dual of the N transistor circuit. Therefore a parallel transistor pair in the N region would have one diffusion column crossed at two tap points in the P region, implying that the signals, traveling on horizontal metal wires would have to be routed so as to allow an extra horizontal track per parallel structure in the N region. This was the gate layout technique used in the Lincoln Boolean Synthesizer (LBS) [10], by restricting the input type to be NOR only. In that way, all parallel structures

were restricted to the N region, and all the series structures were restricted to the P region. Thus the compactness of the constant pitch of metal wires in NMOS Weinberger arrays is somewhat lost in CMOS. Because of the restriction of input types to NOR only (or NAND only), and the stipulation of the extra horizontal wire, it is not clear if there is an advantage gained from Weinberger arrays in CMOS.

3.2. Gate Matrix

Gate Matrix is implemented by laying all the poly in vertical columns and forming transistors by horizontal diffusion rows, with horizontal interconnect done in metal. All vertical interconnect is done in poly. The inputs and outputs are derived from the vertical poly columns. An example of a Gate Matrix layout of a Nand gate is shown in Figure 6. In CMOS, the poly columns simply straddle both the P and the N region, forming gates to the transistors in both regions. Optimization is performed by choosing the right ordering of columns, and by assigning the rows correctly.

The Gate Matrix style of layout has been chosen for the first implementation. This phase has been implemented by Rick Mcgeer as part of the ASP project. Currently, the user has to specify the ordering of the columns, and the router (Second Pass of program) tries to figure out the best row assignment. It then creates an output file containing a SIP (Chap 4) description of the cell. Work is currently underway to devise a heuristic approach to the optimization problem. The general problem of finding the optimal row and column assignment is NP-complete [11], just as it is for Weinberger array and PLA optimization [2].

4. STICKS IN PROLOG

The sticks specification of an IC or cell is an intermediate level, after the routing between all layers has taken place, but before a final geometry (CIF) description of the IC has been specified. All routing on a sticks diagram of an IC is on a virtual grid, as opposed to a real grid on which the grid separation is related to lambda, the process design parameter. The virtual grid nodes are only relative placements of the different wires in all the layers, and have no real distances associated with them. The distances are then deduced by a sticks language compiler, by converting the sticks to rectangular boxes and imposing the process design rules onto the resulting diagram. This level of specification is the same level as a CIF specification, from which an IC can then be fabricated.

4.1. Structures Of SIP

The format for describing Sticks In Prolog (SIP) is in some respects similar to the description of LAVA, a sticks language oriented to C, in reference 6. The format essentially allows the description of an IC in terms of Prolog structures representing the sticks layout, plus an optional means of associating widths with the sticks. The following is a list of the Prolog structures that are currently understood by the sticks compiler, along with a description of those structures.

4.1.1. node(Row, Column)

This structure indicates a location on the virtual grid, and is used to show locations of objects as wells as conditions, such as bends in wires etc.

4.1.2. tran(Type, Node, Terminal)

This structure is used to represent the transistors of the circuit. These will normally have "type" unified to either "nd" or "pd" (for N and P-type transistors respectively) but

could conceivably be extended to handle depletion mode transistors (as used in NMOS) and the like. The "node" will specify the location of the specified terminal on the virtual grid. The terminal instantiates to atoms "gatel" (gate connected by poly from the left side), "gater" (gate connected by poly from the right), "gatet" (gate connected by poly from the top) and "gateb" (gate connected by poly from the bottom).

4.1.3. via(Type, Node)

This structure is used to show contact cuts in between layers. As typically all contacts involve the first layer of metal, it suffices to name the second layer involved in the contact cut. Typical instantiations of Type variable are atoms "m2" (second layer of metal), p (poly), nd (N-diffusion) and pd (P-diffusion). The Node variable instantiates to the node structure described above, and indicates the location of the via on the virtual grid.

4.1.4. wire(Layer, Obj1, Obj2, Width)

This structure is used for the representation of all the wires or "sticks" on the virtual grid. It is used to connect the other objects described above. Objects "Obj1" and "Obj2" are the objects connected by the wire, and is of types described in the above three sections. All the objects appear within a wire statement, as it does not make sense to have an object on the IC not connected to anything else. The "Layer" variable instantiates to the name of the layer in which the wire is running. Typical values for a CMOS process with two metal layers are "m1" (1st layer of metal), "m2" (2nd layer of metal), "p" (poly), "nd" (N-diffusion) and "pd" (P-diffusion). The "Width" variable instantiates to an integer indicating the desired width of the wire in multiples of process design parameter lambda. If uninstantiated, the sticks compiler defaults to the minimum allowable width for that layer. This option can be used to describe power and ground busses where the minimum width will not suffice, for example. The two objects in the wire always share either the same row or the same column on the virtual grid. Bends in wires are denoted by using the node structure as one of the objects in the wire statement.

4.2. An Example Using SIP

The following is an example of a possible sticks diagram and its representation.

Following is an explanation of the symbols used to represent the different layers and vias:-

```
          metal1 ======
          poly   ------
          ndiff  nnnnnn
          pdiff  pppppp
          via    [X]


row3   ==[X]============  vdd
            n
            n
            n
            n
row2   ======[X]======  gnd
            n       p
            n       p
            n       p
            n       p
row1   ----------------  in
            n       p
            n       p
            n       p
            n       p
row0   ==[X]===[X]======  out

    col0    col1      col2          col3
```

The above is, believe it or not, an area efficient layout of a CMOS inverter cell. The type of via in the above diagram is evident from the context. There are four virtual rows and columns required to represent the above cell.

The SIP (Sticks In Prolog) format of the above cell would be,

```
wire(m1,node(3,0),node(3,3),_).   %vdd wire, minimum width.
wire(m1,node(2,0),node(2,3),_).   %gnd wire, minimum width.
wire(p,node(1,0),tran(pd,node(1,2),gatel),_).  %in  wire, minimum width.
wire(m1,node(0,0),node(0,3),_).   %out wire, minimum width.
wire(nd,via(m1d,node(3,1)),via(m1d,node(0,1)),_).  %ndiff wire, minimum width.
wire(pd,via(m1d,node(2,2)),via(m1d,node(0,2)),_).  %pdiff wire, minimum width.
```

Note that there was no need to specify the existence of a n-type transistor, which is

automatically formed by the ndiff crossing poly. On the other hand there was no need to specify the ending node of the poly which will be suitably extended enough so as to form a p-transistor at (1,2). The terminal specification of 'gatel' means the poly will be coming in from the left to connect the gate. There was also no need to specify the diffusion type in the via specification as there will only be one type of diffusion at a node.

Pass 3, which will compile SIP into CIF, will first consult a design rule library, and then satisfy the given constraints to produce a lambda-grid diagram. Given a value for lambda, it will generate the CIF file, in which all dimensions are in .01 micron multiples.

5. COMPILATION OF SIP

The translation of a sticks description to a geometry description is simply a process of converting the sticks into rectangles of appropriate width and height, and then deducing the correct distances between the rectangles from the layer separation constraints from the design rules. This is essentially what is done at the third and final pass that compiles SIP code to CIF code.

5.1. Implementation of the Compactor

The SIP compiler (or Compactor) (Appendix B) first "consults" the names of all the process layers, and the process design rule separation between layers, into it's own database from the library files. Then, working upwards first on the rows, and then leftwards on the columns, it maps every virtual row and column to a lambda-grid row and column, using assertions of "maprow" and "mapcol" structure. The way it performs the mapping is by expanding every layer wire into the intermediate grid points that comprise that wire, and saving that information in structures describing a particular row (or column). That structure contains information about the column points (or row points) on which that particular layer is present. There is one structure for each layer on each row and each column. The compacter then checks each adjacent row for the worst separation imposed by any overlap between two layers. It then asserts that worst separation (in terms of lambda) in the "maprow" structure and proceeds on to the next row. It then repeats this for the columns. After all the rows and columns have been mapped it retracts the original SIP data-base and asserts a new one with the virtual rows and columns replaced by lambda grid rows and columns. In doing the mapping the widths of the wires are taken into account.

The second phase simply takes the wires, and breaks up each wire into wire statements that that contain only nodes as the first and second objects in it. The other objects (tran's and via's) are decomposed into boxes (or rectangles) separately. The primitive wire

statement is then converted into a box of the appropriate width. The "box" structure is then mapped into the appropriate CIF box (B) statement. Since the width of the wires were originally taken into account, it is simple to just expand the horizontal (or row) wires upwards, and the vertical (or column) wires leftwards. In mapping the wires to boxes, some fractional numbers could be generated, as the CIF box is described by the center point and not the end point. The fractional cases are kept track of, and the whole of the data-base is multiplied by a suitable number such that no fractions occur. This is because CIF files have only integers in them. The whole thing is then appropriately scaled down by the CIF scale (DS) statement.

The mapping of vias currently is a simple square box of the appropriate length (as specified in the design rules). The transistor mapping is such as to comply with the definition of the poly and diffusion overhangs specified in the design rule library. Each transistor is map to poly and diffusion boxes such that the overhangs are automatically created. The direction in which the poly is extended is inferred by the "Terminal" instantiation of the "tran" structure in the data-base. Atom "gatel" implies that the poly is to be extended to the right, and atom "gatet" implies that the poly is to be extended downwards. Similarly for "gater" and "gateb".

The layer names are read in from a separate file, and can be added or deleted without affecting the compacter. The design rule separations are read in from a design rule library file, and can be similarly changed without any change in the structure of the program.

5.2. Sample Session

The following is an example session invoking the compacter. The system is again UNIX 4.2 running interpreted C-Prolog. The output of the second pass for the transistor net-list of the cross-coupled nand gate pair of Sec. 2.2 & 2.3 is in file "crossnand.sip". The source code for the third pass is stored in files "compact.pl" and "expand.pl". The layer names are in file "layers.pl". The design rules for layer separations, contact cut sizes, and transistor formation rules are in file "sep.pl". These last two files are automatically

consulted by the program. Note that the input file also contains information about the maximum row and column number.

```
dali% cat crossnand.sip
wire(pd,via(m2pd,node(2,10)),via(m2pd,node(2,14)),_17862).
wire(pd,via(m2pd,node(2,2)),via(m2pd,node(2,6)),_17967).
wire(pd,via(m2pd,node(3,5)),via(m2pd,node(3,8)),_18081).
wire(pd,via(m2pd,node(3,8)),via(m2pd,node(3,11)),_18186).
wire(m1,via(m1p,node(4,3)),via(m1m2,node(4,11)),_18300).
wire(m1,via(m1m2,node(5,1)),via(m1m2,node(5,13)),_18414).
wire(m1,via(m1m2,node(6,2)),via(m1m2,node(6,14)),_18528).
wire(nd,via(m2nd,node(7,10)),via(m2nd,node(7,13)),_18642).
wire(nd,via(m2nd,node(8,5)),via(m2nd,node(8,10)),_18756).
wire(nd,via(m2nd,node(8,1)),via(m2nd,node(8,4)),_18861).
wire(m2,via(m1m2,node(5,1)),via(m2nd,node(8,1)),_19033).
wire(m2,via(m2pd,node(2,2)),via(m1m2,node(6,2)),_19203).
wire(p,node(1,3),node(9,3),_19357).
wire(m2,via(m1m2,node(6,4)),via(m2nd,node(8,4)),_19467).
wire(m2,via(m2pd,node(3,5)),via(m2nd,node(8,5)),_19648).
wire(m2,via(m2pd,node(2,6)),via(m1m2,node(1,6)),_19842).
wire(p,node(1,7),node(9,7),_19996).
wire(m2,via(m2pd,node(3,8)),via(m1m2,node(1,8)),_20130).
wire(p,node(1,9),node(9,9),_20284).
wire(m2,via(m2nd,node(7,10)),via(m1m2,node(9,10)),_20428).
wire(m2,via(m2pd,node(2,10)),via(m1m2,node(1,10)),_20590).
wire(m2,via(m2pd,node(3,11)),via(m1m2,node(4,11)),_20760).
wire(p,node(1,12),node(9,12),_20914).
wire(m2,via(m1m2,node(5,13)),via(m2nd,node(7,13)),_21024).
wire(m2,via(m2pd,node(2,14)),via(m1m2,node(6,14)),_21194).
wire(p,node(1,15),node(9,15),_21348).
wire(m1,node(1,1),node(1,15),_X).
wire(m1,node(9,1),node(9,15),_Y).
maxrow(9).
maxcol(15).
dali% cprolog
CProlog version 1.2
[ restoring file /b/hprg/Prolog/Cprolog/environment ]

yes
| ?- ['compact.pl'].
compact.pl consulted 15364 bytes 6.299999 sec.

yes
| ?- compact(crossnand).
crossnand.sip reconsulted 2472 bytes 1.200001 sec.
layers.pl reconsulted 260 bytes 0.150005 sec.
sep.pl reconsulted 580 bytes 0.316671 sec.
expand.pl reconsulted 8108 bytes 3.600004 sec.

yes
| ?- halt.

[ Prolog execution halted ]
```

```
122.7u 6.1s 4:32 47% 96+228k 26+9io 29pf+0w
dali% cat crossnand.cif
DS 1 150 2;
L CP;
B 4 84 118 44;
B 4 84 90 44;
B 4 84 72 44;
B 4 84 56 44;
B 4 84 26 44;
L CM;
B 118 4 61 84;
B 118 4 61 4;
B 96 4 62 58;
B 96 4 50 48;
B 58 4 53 38;
L CD;
B 32 4 18 74;
B 38 4 61 74;
B 22 4 87 64;
L CS;
B 22 8 71 30;
B 22 8 53 30;
B 34 8 31 16;
B 34 8 93 16;
L CM2;
B 4 48 108 36;
B 4 20 96 56;
B 4 14 80 33;
B 4 18 78 11;
B 4 24 78 74;
B 4 32 62 18;
B 4 18 46 11;
B 4 50 44 51;
B 4 20 32 66;
B 4 48 16 36;
B 4 30 4 61;
L CC;
B 8 8 110 16;
B 8 8 110 60;
B 8 8 98 50;
B 8 8 98 66;
B 8 8 82 30;
B 8 8 82 40;
B 8 8 80 16;
B 8 8 80 6;
B 8 8 80 66;
B 8 8 80 86;
B 8 8 64 30;
B 8 8 64 6;
B 8 8 48 16;
B 8 8 48 6;
B 8 8 46 30;
B 8 8 46 76;
B 8 8 34 60;
B 8 8 34 76;
```

```
B 8 8 18 16;
B 8 8 18 60;
B 8 8 6 50;
B 8 8 6 76;
B 8 8 6 76;
B 8 8 34 76;
B 8 8 46 76;
B 8 8 80 76;
B 8 8 80 66;
B 8 8 98 66;
B 8 8 18 60;
B 8 8 110 60;
B 8 8 6 50;
B 8 8 98 50;
B 8 8 28 40;
B 8 8 82 40;
B 8 8 64 30;
B 8 8 82 30;
B 8 8 46 30;
B 8 8 64 30;
B 8 8 18 16;
B 8 8 48 16;
B 8 8 80 16;
B 8 8 110 16;
DF;
C 1;
End
dali%
```

The output of the compacter is stored in file "crossnand.cif" and is displayed at the end of the session. This is also shown as a cifplot in Figure 8. It took approximately six minutes interactive time on a VAX 750, with a load average of 2.8, to produce the output file. This includes the time to read in the input file and the source file for the expander (second phase of pass3).

5.3. Example Cells

A few test cases of the compacter are given below.

5.3.1. Hand Designed Inverter Cell

The inverter cell described in Sec. 4.2 above is compacted by the third pass and is shown in Figure 7. Note the length of the cell is considerably longer than the height because of the twelve lambda separation requirement between N and P-type diffusion. Also note the fact that the poly got extended two lambda to the right, because of the specification

of a transistor at that node. All the widths of the layer have defaulted to minimum width, because the "Width" variable is uninstantiated in the wire statements. The vias are specified to be 4 X 4 lambda. The P diffusion has been specified to be twice as big as the N diffusion to make the appropriate difference in the resistivity of the P-pullup and the N-pulldown, because of the mobility difference of the holes and electrons in the P and N regions respectively.

5.3.2. Cross Coupled Nand Gate Pair

The example that has been followed through from the boolean specification level to the CIF layout is shown in Figure 8. The only observation to make here is that currently the row assignment is not optimal, as all the N-transistors can all be laid out in a single row. The second row can simply be folded back.

5.3.3. Exclusive-Nor Gate

The boolean specification given below was transformed into the layout shown in Figure 9. The comments made about row optimization in the previous section apply here also, in that row and column reassignment could reduce the area of the layout.

5.3.4. Hand Laid Exclusive-Nor Gate

The sticks layout of an exclusive-nor gate from reference [9] was written manually in SIP code, and compacted by pass3. The resulting layout is shown in Figure 10. The resulting layout is significantly smaller than the one designed by the prolog system. This is discussed in the section on conclusions.

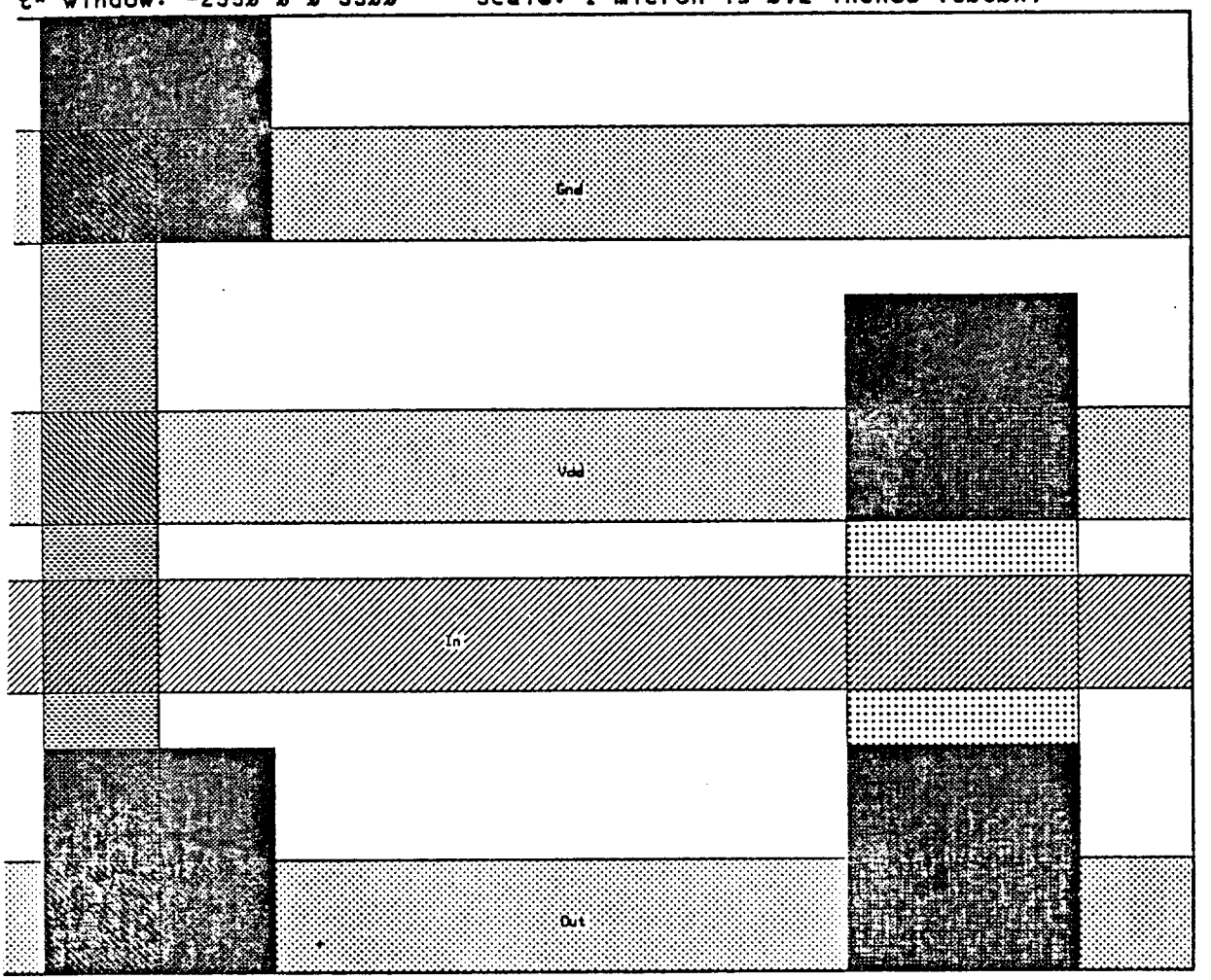

le:butt Job: cifplot Date: Wed Aug 14 15:5#:28 1985
t* Window: -255# # # 33## --- Scale: 1 micron is #.2 inches (5##8x)
Gnd
Vdd
In
Out

FIGURE 7

FIGURE 8

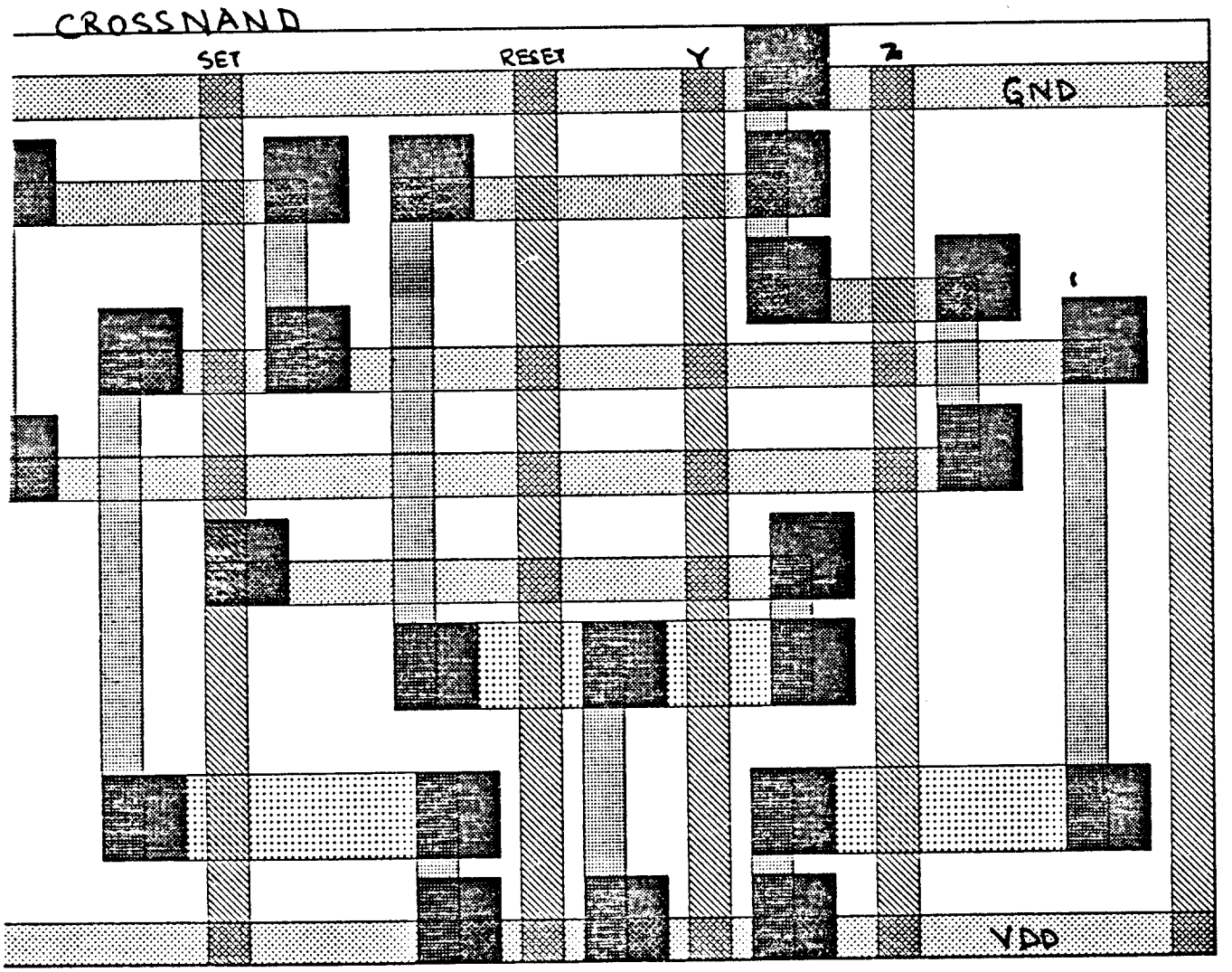


FIGURE 9

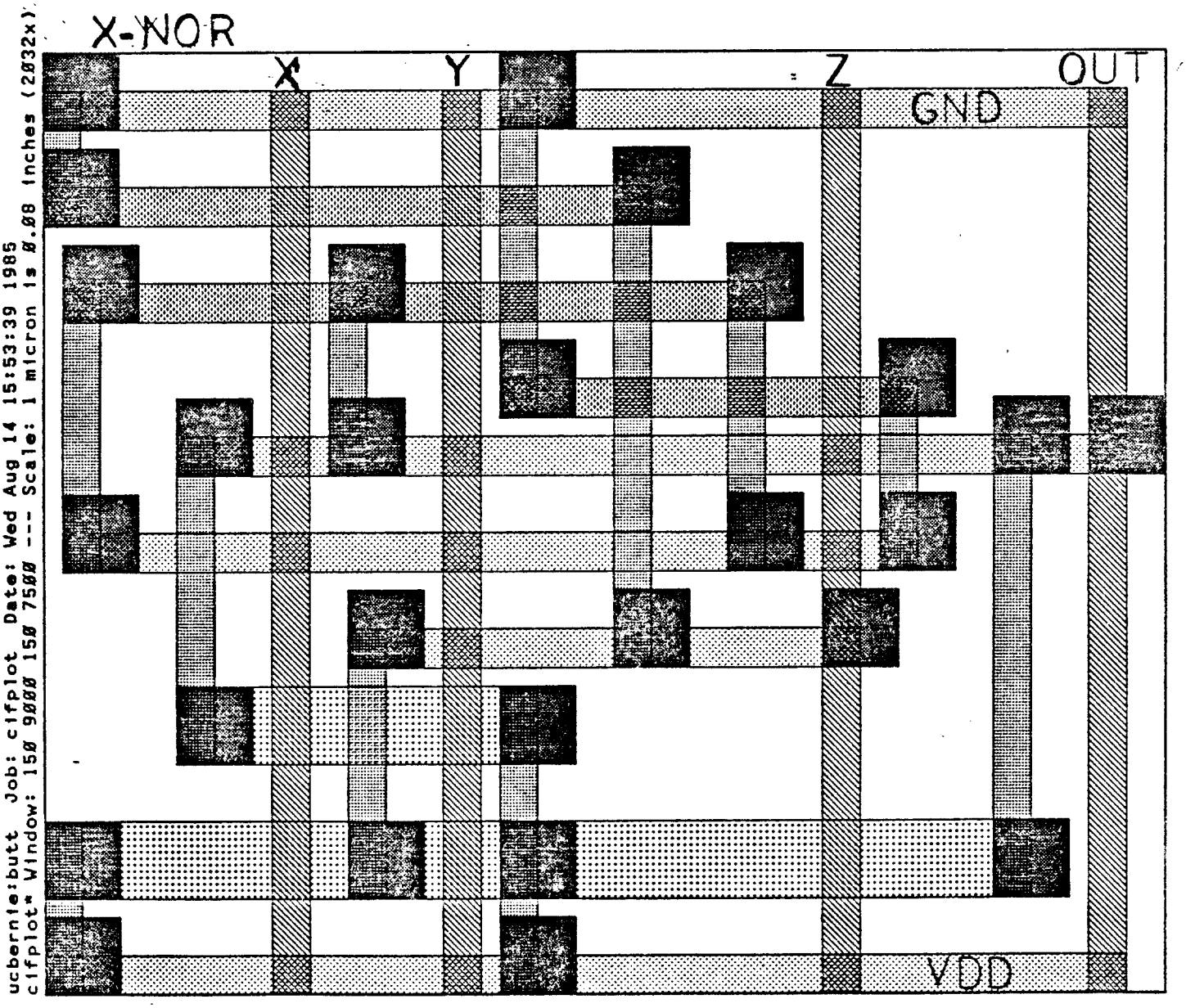
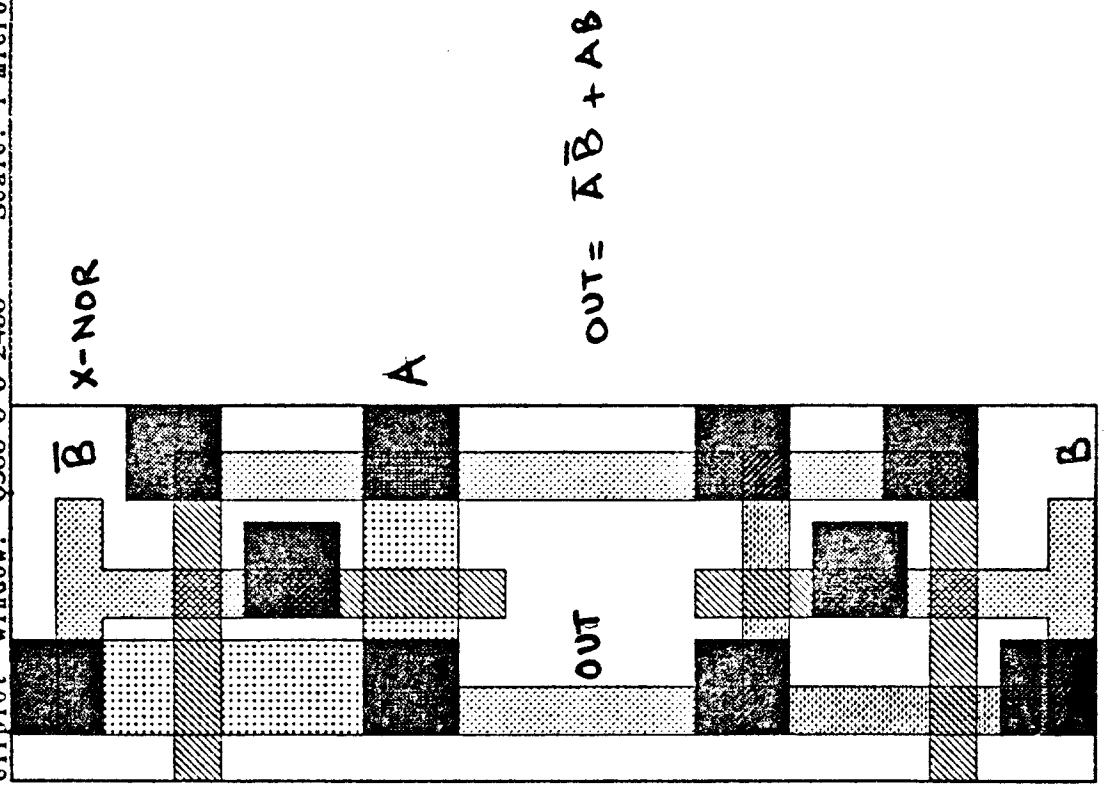

X-NOR
X
Y
Z
OUT
GND
VDD
ucbernie:butt Job: clfplot Date: Wed Aug 14 15:53:39 1985
clfplot* Window: 150 9000 150 7500 --- Scale: 1 micron is 0.08 inches (2032x)

FIGURE 10


cifplot* Window: -6900 0 0 2400 --- Scale: 1 micron is 0.08 inches (2032x)
X-NOR
$\bar{B}$
A
OUT = $\bar{A}\bar{B} + AB$
OUT
B

6. CONCLUSIONS

The current limitations and directions for future work are discussed below.

6.1. Current Limitations

There are a number of areas in which the cell design system could be improved in. First, as observed in Chapter 5, gate matrix optimization techniques such as the Kernighan-Lin [2] method or some other algorithm based on graph theory could be used. An alternative method could be the addition of a large number of heuristics (like the ones used by human designers) to the router. The compacter could be similarly improved by the use of heuristics. One area in which it is currently lacking is the optimal positioning of vias. Because of the way the compaction algorithm works, vias always work upwards and rightwards. Possible Enhancements could take advantage of blank spaces to the right and bottom of the via. By such repositioning of the vias, savings of 2 lambda in height and length of the cell could be achieved per repositioned via.

Another deficiency lies in the fact that current automation methods rely on restricted methodologies such as the Gate Matrix method, and a fixed input format for expressing boolean equations. As the example on the hand-laid exclusive-nor gate shows, an exclusive-nor gate can be implemented by a total of four N and P type transistors. This is not self-evident if the exclusive-nor is first realized with an and-or based boolean equation, which is then converted into transistors. That makes the total number of required transistors be ten. This net-list of ten transistors is then laid out in the Gate Matrix style, losing yet another 20-50% in area. The hand laid technique takes advantage of both vertical and horizontal transistors and CMOS characteristics [9]. By making some effort in capturing this knowledge-base of designers' tricks, a significant improvement can be made in the area of random-logic cells. Larger boolean cells such as the control sections of finite state machines would still have to be done using restrictive methodologies , and it is towards that end that

optimization algorithms and heuristics for those methodologies should be investigated.

6.2. Functional Enhancements

The system described in the preceding chapters presents a simple cell design system. There are several aspects of cell design that could be added to the system for greater functionality and usefulness. First of all different layout methodologies can be experimented with, such as PLA's, Weinberger arrays and Domino logic. For some of them, such as PLA's, some change would have to be done to the three pass approach to cell layout. Specifically, CMOS PLA's are traditionally implemented in dynamic as opposed to static logic. Therefore the net-list generated by the first pass is no longer the net-list of transistors that will be implemented in the corresponding PLA. Therefore a direct mapping of the boolean structures to the SIP layout would be desirable. That pass could also perform traditional optimization such as PLA folding.

Another functional addition could be the addition of more structures in SIP, in order to more efficiently describe a leaf-cell in terms of it's environment. Specifically some means of describing the directionality of signals, as well as some relative ordering in order to comply with global routing considerations. So for example there could be a "defcell" rule for defining the environment and relative ordering of signals of the cell. The body of the rule could contain the SIP statements of the cell. Also included in the "defcell" argument-list could be a number to indicate the number of bits of the cell.

Yet another desirable feature could be the automatic sizing of transistors based on loading and timing considerations. This could utilize the results of circuit simulation, and be added as a post-processor. Some structures to allow independent sizing of transistors would have to be added for this purpose.

REFERENCES

1. J.M. Siskind, J.R. Southard, and K.W. Crouch, 1982, "Generating Custom VLSI designs From Succinct Algorithmic Descriptions", MIT Conference on Advanced VLSI, Cambridge MA.

2. J.D. Ullman, 1984, "Computational Aspects of VLSI", Computer Science Press.

3. J. Werner, Oct. 1983, "Progress Towards the Ideal Silicon Compiler", VLSI Design.

4. W.F. Clocksin and C.S. Mellish, 1984, "Programming in Prolog", 2nd Edition, Springer-Verlag.

5. A.D. Lopez and H.F.S. Law, 1980, "A dense Gate-Matrix Method for MOS VLSI", IEEE Transactions on Electron Devices.

6. C.A. Mead and L.A. Conway,1980, "Introduction to VLSI Systems", Sec 4.5, Addison-Wesley.

7. D.D. Hill, AT&T, Bell Labs, Murray-Hill NJ, Private Communications.

8. D.D. Hill, Nov. 1984, "Icon: A Tool for Design at Schematic, Virtual Grid, and Layout Levels", IEEE Design and Test.

9. CS 250, Spring 1985, Lecture Notes, UC. Berkeley.

10. J.R. Southard, A. Domic, and K.W. Crouch,1982, "Report on the Lincoln Boolean Synthesizer (LBS) ", Design Automation Conference.

11. Ohtsuki et al , Sep 1979, "One Dimensional Logic Gate Assignment and Interval Graphs," IEEE Transactions on Circuits and Systems.

12. T.D. Kowalski and D.E Thomas, Feb 1984, "The VLSI Design Automation Assistant: An IBM System/370 Design," IEEE Design and Test.

APPENDIX A

```
%The following procedures will take an input of boolean eqns
%and output a netlist of transistors of CMOS style, corresponding
%to the circuit described by the boolean equations.

%Types of prolog structures for boolean ops that are implemented.
compl(Out = and(List)).
compl(Out = or(List)).
compl(Out = pass(In, Control)).

%An example structure is
%compl(out = and([and([a,b]),or([x,y])])).
%which corresponds to .b).(x + y)).


%Main loop to convert logic structures into prolog.
genxstor(X) :-
      readfile(X),              %Consult file with .bln extension.
      openfile(X),              %Open file with .net extension.
      compl(Out = Struct),%Loop over all input types.
      compile(Struct,Out),  %Compile the structure into xstors.
      fail;told.          %Close output file.

%Rules for compiling structures.
compile(and(List),Out) :-
      convert(and(List),gnd,Out,nmos),  %Make pulldown net in nmos.
      dual(and(List),Dual),          %Form dual of circuit.
      convert(Dual,vdd,Out,pmos).       %Make pullup net in pmos.

compile(or(List),Out) :-
      convert(or(List),gnd,Out,nmos),   %Make pulldown net in nmos.
      dual(or(List),Dual),           %Form dual of circuit.
      convert(Dual,vdd,Out,pmos).       %Make pullup net in pmos.

compile(pass(In,Control),Out) :-
      convert(Control,gnd,New,nmos),    %Form the 'not' of Control to gate.
      convert(Control,vdd,New,pmos),
      convert(Control,In,Out,nmos),     %Form 'n' transistor of pass gate.
      convert(New,In,Out,pmos).   %Form 'p' transistor of pass gate.

%Rules for generating transistors.
convert(and([H]),From,To,Polarity) :-       %Single gate left.
      convert(H,From,To,Polarity), !.

convert(and([H|T]),From,To,Polarity) :-     %Serial transistors.
      convert(H,From,New,Polarity),
      convert(and(T),New,To,Polarity),!.

convert(or([H|T]),From,To,Polarity) :-      %Parallel transistors.
      convert(H,From,To,Polarity),
      convert(or(T),To,From,Polarity),!.    %Maximize Source-Drain conn.

convert(and([]),From,To,Polarity).          %Do nothing.

convert(or([]),From,To,Polarity).           %Do nothing.
```

```
convert(Gate,Source,Drain,Polarity) :-          %Form a transistor.
      atomic(Gate),
      bindNet(Drain),                  %Gen. node if new.
      write(xstor(Polarity,Source,Gate,Drain)),
      write('.'), nl.

%Generate new node numbers for uninstantiated nodes.
nodeNum(0).
bindNet(X) :-
      var(X),          %Make sure node is uninstantiated.
      !,
      nodeNum(X),
      retract(nodeNum(X)),
      Y is X + 1,
      assert(nodeNum(Y)).

bindNet(X).          %Let instantiated nodes stay as they are.

%Define list predicates.
append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

%Add '.bln' to input file.
readfile(X) :-
      name(X,L),
      append(L,".bln",L1),
      name(Y,L1),
      reconsult(Y).

%Add '.net' to output file.
openfile(X) :-
      name(X,L),
      append(L,".net",L1),
      name(Y,L1),
      tell(Y).

%Rules for computing the dual of an expression.
dual(and([H|T]),or([H1|T1])) :-
      dual(H,H1),
      dual(T,T1),!.

dual(or([H|T]),and([H1|T1])) :-
      dual(H,H1),
      dual(T,T1),!.

dual([H|T],[H1|T1]) :-
      dual(H,H1),
      dual(T,T1),!.

dual(X,X) :- atomic(X), !.
```

APPENDIX B

```
%The following code reads in a sticks representation of a circuit
%on a virtual grid and compacts it to produce a sticks description
%on a lambda grid. Then it transforms the real grid sticks to a CIF
%description.


%Main pass.
compact(X) :-
      init(X),     %Read in all technology plus initial data.
      compacty, %Compact rows.
      compactx, %Compact cols.
      updatevl, %Update wire database from virtual to real.
      expand(X).       %Expand sticks to CIF boxes.


%Generate and satisfy vertical constraints.
compacty :-
      getrownum(RowNum),      %Loop over all the rows.
      findlayersinrow(RowNum),%Find  all ocurrences of all layers in row.
      check_yconstr(RowNum),  %Create lambda value for RowNum, based on
                        %layer constraints.
      check_ytran1(RowNum),   %Check for transistors imposing y constraints.
      compacty; true.

check_yconstr(0) :- !.        %Zeros are mapped to zero.
check_yconstr(RowNum) :-
      LastRow is RowNum - 1,
      maprow(LastRow,LastMap),
      NewMap is LastMap + 1,
      asserta((maprow(RowNum,NewMap):-!)),
      !,                %Default mapping is LastMap + 1.
      layer(Layer),           %Fetch layer.
      check_yconstr1(Layer,RowNum),
      fail;true.
check_yconstr1(Layer1,RowNum) :-
      layer(Layer2),          %Make all possible two layer combinations.
      LastRow is RowNum - 1,
      maprow(LastRow,LastRowMap),%Fetch  mapping of the last row.
      row(LastRow,[Layer1,T1]),
      row(RowNum,[Layer2,T2]),
      overlap(T1,T2),         %If there is an overlap, continue, else loop.
      sep(Layer1,Layer2,S),
      minwidth(Layer1,W),
      maprow(RowNum,LambdaNum),       %Fetch current mapping.
      TestMap is S + W + LastRowMap,
      TestMap > LambdaNum, %If new constraint is stronger, use it.
      asserta((maprow(RowNum,TestMap):-!)),
      fail;true.

findlayersinrow(RowIn) :-
      layer(Layer),           %Loop over all the layers.
      fetchrow(Layer, RowIn, L),%Fetch all occurences of Layer in row.
      assert(row(RowIn,L)),
      fail;true.
```

```
%Fetch all occurences of LayerIn in input file at RowIn.

fetchrow(LayerIn, RowIn, [LayerIn,Listout]) :-
      findall(L,fetchrow1(LayerIn,RowIn,L), L1),
      ap(L1,Listout).

fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
            wire(LayerIn,node(RowIn,Col1),node(RowIn,Col2),_),
            L1),      %Horizontal wires.
      recur_expand(L1,L2),
      ap(L2, L).
fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
            wire(LayerIn,via(_,node(RowIn,Col1)),node(RowIn,Col2),_),
            L4),      %More horizontal wires.
      recur_expand(L4,L5),
      ap(L5,L).
fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
            wire(LayerIn,node(RowIn,Col1),via(_,node(RowIn,Col2)),_),
            L7),      %More horizontal wires.
      recur_expand(L7,L8),
      ap(L8,L).
fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
           wire(LayerIn,via(_,node(RowIn,Col1)),via(_,node(RowIn,Col2)),_),
            L10),          %More horizontal wires.
      recur_expand(L10,L11),
      ap(L11,L).
fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
           wire(LayerIn,node(RowIn,Col1),tran(_,node(RowIn,Col2),_),_),
            L10),          %More horizontal wires.
      recur_expand(L10,L11),
      ap(L11,L).
fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
           wire(LayerIn,tran(_,node(RowIn,Col1),_),node(RowIn,Col2),_),
            L10),          %More horizontal wires.
      recur_expand(L10,L11),
      ap(L11,L).
fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
           wire(LayerIn,tran(_,node(RowIn,Col1),_),tran(_,node(RowIn,Col2),_),_),
            L10),          %More horizontal wires.
      recur_expand(L10,L11),
      ap(L11,L).
fetchrow1(LayerIn, RowIn, L) :-
      findall([Col1, Col2],
           wire(LayerIn,tran(_,node(RowIn,Col1),_),via(_,node(RowIn,Col2)),_),
            L10),          %More horizontal wires.
      recur_expand(L10,L11),
      ap(L11,L).
fetchrow1(LayerIn, RowIn, L) :-
```

```
        findall([Col1, Col2],
            wire(LayerIn,via(_,node(RowIn,Col1)),tran(_,node(RowIn,Col2),_),_),
            L10),          %More horizontal wires.
        recur_expand(L10,L11),
        ap(L11,L).

%Find prescence of Contact Cut layer from vias.
fetchrow1(cc, RowIn, L) :-
        findall( Col,
            wire(Layer,via(_,node(RowIn,Col)),_,_),
            L).
fetchrow1(cc, RowIn, L) :-
        findall( Col,
            wire(Layer,_,via(_,node(RowIn,Col)),_),
            L).

%Generate and satisfy column constraints.
compactx :-
        getcolnum(Colnum),  %Loop over all columns.
        findlayersincol(Colnum),%Find  all occurences of all layers in col.
        check_xconstr(Colnum),    %Create lambda values for Colnum, based on
                    %layer constraints.
        check_xtran1(Colnum),     %Check for transistors imposing x constraints.
        compactx;true.

%Check constraints in x direction.
check_xconstr(0) :- !.        %Zeros are mapped to zero
check_xconstr(Colnum) :-
        LastCol is Colnum - 1,
        mapcol(LastCol,LastMap),%Fetch  last map.
        NewMap is LastMap + 1,
        asserta((mapcol(Colnum,NewMap):-!)),
        !,                %Default map is LastMap + 1.
        layer(Layer),          %Fetch layer.
        check_xconstr1(Layer,Colnum),
        fail;true.

check_xconstr1(Layer1,Colnum) :-
        layer(Layer2),         %Make all possible two layer combinations.
        LastCol is Colnum - 1,
        mapcol(LastCol,LastColMap),%Fetch  mapping of the last col.
        col(LastCol,[Layer1,T1]),
        col(Colnum,[Layer2,T2]),
        overlap(T1,T2),        %If there is an overlap, continue, else loop.
        sep(Layer1,Layer2,S),       %Fetch seperation of layers
        minwidth(Layer1,W), %Fetch min width of layers.
        mapcol(Colnum,LambdaNum),  %Fetch current mapping.
        TestMap is S + W + LastColMap,
        TestMap > LambdaNum, %If new constraint is stronger, use it.
        asserta((mapcol(Colnum,TestMap):-!)),
        fail;true.

%Check to see if last col is consistent.
checklastcol :-
            maxcol(M),       %Make sure last col is consistent.
```

```
        mapcol(M,X),
        M1 is M - 1,
        mapcol(M1,Y),
        X =< Y,
        L is Y + 1,
        asserta((mapcol(M,L):- !));true.

findlayersincol(ColIn) :-
     layer(Layer),              %Loop over all the layers.
     fetchcol(Layer, ColIn, L), %Fetch all occurences of Layer in row.
     assert(col(ColIn,L)),
     fail;true.

%Fetch all occurences of LayerIn in input file at RowIn.

fetchcol(LayerIn, ColIn, [LayerIn,Listout]) :-
     findall(L,fetchcol1(LayerIn,ColIn,L), L1),
     ap(L1,Listout).

fetchcol1(LayerIn, ColIn, L) :-
     findall([Row1, Row2],
          wire(LayerIn,node(Row1,ColIn),node(Row2,ColIn),_),
          L1),      %Vertical wires.
     recur_expand(L1,L2),
     ap(L2, L).
fetchcol1(LayerIn, ColIn, L) :-
     findall([Row1, Row2],
          wire(LayerIn,via(_,node(Row1,ColIn)),node(Row2,ColIn),_),
          L4),      %More vertical wires.
     recur_expand(L4,L5),
     ap(L5,L).
fetchcol1(LayerIn, ColIn, L) :-
     findall([Row1, Row2],
          wire(LayerIn,node(Row1,ColIn),via(_,node(Row1,ColIn)),_),
          L7),      %More horizontal wires.
     recur_expand(L7,L8),
     ap(L8,L).
fetchcol1(LayerIn, ColIn, L) :-
     findall([Row1, Row2],
         wire(LayerIn,via(_,node(Row1,ColIn)),via(_,node(Row2,ColIn)),_),
          L10),          %More vertical wires.
     recur_expand(L10,L11),
     ap(L11,L).
fetchcol1(LayerIn, ColIn, L) :-
     findall([Row1, Row2],
          wire(LayerIn,node(Row1,ColIn),tran(_,node(Row2,ColIn),_),_),
          L10),          %More vertical wires.
     recur_expand(L10,L11),
     ap(L11,L).
fetchcol1(LayerIn, ColIn, L) :-
     findall([Row1, Row2],
          wire(LayerIn,tran(_,node(Row1,ColIn),_),node(Row2,ColIn),_),
          L10),          %More vertical wires.
     recur_expand(L10,L11),
     ap(L11,L).
```

```
fetchcol1(LayerIn, ColIn, L) :-
      findall([Row1, Row2],
           wire(LayerIn,tran(_,node(Row1,ColIn),_),tran(_,node(Row2,ColIn),_),_),
            L10),          %More vertical wires.
      recur_expand(L10,L11),
      ap(L11,L).
fetchcol1(LayerIn, ColIn, L) :-
      findall([Row1, Row2],
           wire(LayerIn,tran(_,node(Row1,ColIn),_),via(_,node(Row1,ColIn)),_),
            L10),          %More vertical wires.
      recur_expand(L10,L11),
      ap(L11,L).
fetchcol1(LayerIn, ColIn, L) :-
      findall([Row1, Row2],
           wire(LayerIn,via(_,node(Row1,ColIn)),tran(_,node(Row2,ColIn),_),_),
            L10),          %More vertical wires.
      recur_expand(L10,L11),
      ap(L11,L).

%Find prescence of Contact Cut layer from vias.
fetchcol1(cc, ColIn, L) :-
     findall( Row,
           wire(Layer,via(_,node(Row,ColIn)),_,_),
           L).
fetchcol1(cc, ColIn, L) :-
     findall( Row,
           wire(Layer,_,via(_,node(Row,ColIn)),_),
           L).

%Append all the elements of a list.
ap([],[]).
ap([[]|T],D) :- ap(T,D).
ap([[Ah|At]|T],[Ah|D]) :-ap([At|T],D).

%Define some list predicates here.
append([], L, L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

%Recursively apply expand to the elements of L.
recur_expand([], []).
recur_expand([H|T], [L|Ls]) :-
      expand(H,L),
      recur_expand(T,Ls).

%Expand two integers to a list of all intermediate and boundary integers.
expand([IntIn1, IntIn1], [IntIn1]).     %End of expansion.
expand([IntIn1, IntIn2], [IntIn1|Y]) :-
      IntIn1 < IntIn2,
      N is IntIn1 + 1,
      expand([N, IntIn2], Y).
expand([IntIn1, IntIn2], [IntIn1|Y]) :-
      IntIn1 > IntIn2,
      N is IntIn1 - 1,
      expand([N, IntIn2], Y).
```

```
%Define a findall. findall(X,G,L) will find the list of all X's
%such that G is provable.

findall(X,G,_) :-
      asserta(found(mark)),
      call(G),
      asserta(found(X)),
      fail.
findall(_,_,L) :- collect_found([], M),!, L = M.

collect_found(S,L) :- getnext(X), !, collect_found([X|S],L).
collect_found(L,L).

getnext(X) :- retract(found(X)), !, X == mark.

%Loop over all rows until maxrow is reached.
rownumber(0).
getrownum(X) :-
      rownumber(X),
      maxrow(M),
      X =< M,
      retract(rownumber(X)),
      Y is X + 1,
      assert(rownumber(Y)).
%Get next column number.
colnumber(0).
getcolnum(X) :-
      colnumber(X),
      maxcol(M),
      X =< M,
      Y is X + 1,
      retract(colnumber(X)),
      assert(colnumber(Y)).

%Overlap of two layers in adjacent row, or column.
overlap([X|_],L) :- member(X,L), !.
overlap([_|T],L) :- overlap(T,L).

%Member predicate.
member(X,[X|_]) :- !.
member(X,[_|T]) :- member(X,T).

%Update the virtual row and col numbers by lambda-grid numbers.
updatevl :-
      object(Obj1),     %Make all possible structure combinations.
      up1(Obj1),
      fail;true.

up1(Obj1) :-
      object(Obj2),
      wire(L,Obj1,Obj2,W),
      mapobj(Obj1,Obj3),
      mapobj(Obj2,Obj4),
      retract(wire(L,Obj1,Obj2,W)),
      asserta(wire(L,Obj3,Obj4,W)),     %Map only once.
```

```
      fail;true.

mapobj(node(Row,Col),node(Row1,Col1)) :-
      maprow(Row,Row1),
      mapcol(Col,Col1).
mapobj(via(T,node(Row,Col)),via(T,node(Row1,Col1))) :-
      maprow(Row,Row1),
      mapcol(Col,Col1).
mapobj(tran(Ty,node(Row,Col),Te),tran(Ty,node(Row1,Col1),Te)):-
      maprow(Row,Row1),
      mapcol(Col,Col1).

%Objects, for letting prolog make all possible wire structures.
object(via(T,node(R,C))).
object(node(R,C)).
object(tran(Ty,node(R,C),Te)).

%Initialize system.
init(X) :-
      name(X,L),
      append(L,".sip",L1),
      name(Y,L1),
      reconsult(Y),          %Read input sticks.
      reconsult('layers.pl'), %Read technology info.
      reconsult('sep.pl'),    %Read in design rules.
      reconsult('expand.pl'), %Read stick compiler.
      assert(mapcol(Z,Z)),   %Initial mapping is identity.
      assert(maprow(Z,Z)).

%Check vertical constraints imposed by horizontal transistors.
check_ytran1(RowNum) :-
      check_ytran(RowNum);true. %Check all possibilites

check_ytran(0) :- !.          %Do not remap zero.

check_ytran(RowNum) :-
      LastRow is RowNum - 1,
      wire(p,tran(D,node(LastRow,C),gateb),_,_),
      minwidth(D,W),
      polyext(P),
      seprowlayer(node(RowNum,C),S),      %Seperation between poly and layer.
      X is W + P + S,
      maprow(LastRow,LambdaNum),
      TestMap is X + LambdaNum,
      maprow(RowNum,CurMap),
      TestMap > CurMap,
      asserta((maprow(RowNum,TestMap):-!)).

check_ytran(RowNum) :-
      LastRow is RowNum - 1,
      wire(p,_,tran(D,node(LastRow,C),gateb),_),
      minwidth(D,W),
      polyext(P),
      seprowlayer(node(RowNum,C),S),
      X is W + P + S,
```

```
      maprow(LastRow,LambdaNum),
      TestMap is X + LambdaNum,
      maprow(RowNum,CurMap),
      TestMap > CurMap,
      asserta((maprow(RowNum,TestMap):-!)).

check_ytran(RowNum) :-
      LastRow is RowNum - 1,
      wire(p,tran(D,node(RowNum,C),gatet),_,_),
      polyext(P),
      seprowlayer(node(LastRow,C),S),%Fetch  sep from poly of layer.
      widthrowlayer(node(LastRow,C),W),%Fetch  width of layer.
      maprow(LastRow,LambdaNum),
      TestMap is P + S + W + LambdaNum,
      maprow(RowNum,CurMap),
      TestMap > CurMap,
      asserta((maprow(RowNum,TestMap):-!)).

check_ytran(RowNum) :-
      LastRow is RowNum - 1,
      wire(_,tran(D,node(RowNum,C),gatet),_,_),
      polyext(P),
      seprowlayer(node(LastRow,C),S),  %Fetch sep of poly from layer.
      widthrowlayer(node(LastRow,C),W),%Fetch  width of layer.
      maprow(LastRow,LambdaNum),
      TestMap is P + S + W + LambdaNum,
      maprow(RowNum,CurMap),
      TestMap > CurMap,
      asserta((maprow(RowNum,TestMap):-!)).

%Fetch separation between poly, and whatever is at node.
seprowlayer(node(R,C),S) :-
      row(R,[L,L1]),
      member(C,L1),
      sep(p,L,S),!.

seprowlayer(N,0).      %If nothing at node,return zero.

%Fetch width of whatever is at node.
widthrowlayer(node(R,C),W) :-
      row(R,[L,L1]),
      member(C,L1),
      minwidth(L,W),!.

widthrowlayer(N,0).          %If nothing at node,return zero.


%Check for horizontal constraints imposed by vertical tran's.
check_xtran1(ColNum) :-
      check_xtran(ColNum);true.       %Check all possibilities.

check_xtran(0) :- !.                  %Do not remap zero.

check_xtran(ColNum) :-
      LastCol is ColNum - 1,
```

```
        wire(p,tran(D,node(R,LastCol),gatel),_,_),
        minwidth(D,W),
        polyext(P),
        sepcollayer(node(R,ColNum),S),
        X is W + P + S,
        mapcol(LastCol,LastMap),
        TestMap is X + LastMap,
        mapcol(ColNum,CurMap),
        TestMap > CurMap,
        asserta((mapcol(ColNum,TestMap):-!)).

check_xtran(ColNum) :-
        LastCol is ColNum - 1,
        wire(p,_,tran(D,node(R,LastCol),gatel),_),
        minwidth(D,W),
        polyext(P),
        sepcollayer(node(R,ColNum),S),
        X is W + P + S,
        mapcol(LastCol,LastMap),
        TestMap is X + LastMap,
        mapcol(ColNum,CurMap),
        TestMap > CurMap,
        asserta((mapcol(ColNum,TestMap):-!)).

check_xtran(ColNum) :-
        LastCol is ColNum - 1,
        wire(p,tran(D,node(R,ColNum),gater),_,_),
        polyext(P),
        sepcollayer(node(R,LastCol),S),   %Fetch sep from poly of layer.
        widthcollayer(node(R,LastCol),W),      %Fetch width of layer
        mapcol(LastCol,LastMap),
        TestMap is P + S + W + LastMap,
        mapcol(ColNum,CurMap),
        TestMap > CurMap,
        asserta((mapcol(ColNum,TestMap):-!)).

check_xtran(ColNum) :-
        LastCol is ColNum - 1,
        wire(p,_,tran(D,node(R,LastCol),gater),_),
        polyext(P),
        sepcollayer(node(R,LastCol),S),
        widthcollayer(node(R,LastCol),W),  %Fetch width of layer.
        mapcol(LastCol,LastMap),
        TestMap is P + S + W + LastMap,
        mapcol(ColNum,CurMap),
        TestMap > CurMap,
        asserta((mapcol(ColNum,TestMap):-!)).

%Fetch sepeartion between poly, and whatever is at node.
sepcollayer(node(R,C),S) :-
        col(C,[L,L1]),
        member(C,L1),
        sep(p,L,S),!.

sepcollayer(N,0).              %If nothing at node, return zero.
```

```
%Fetch width of whatever is at node.
widthcollayer(node(R,C),W) :-
        col(C,[L,L1]),
        member(C,L1),
        minwidth(L,W),!.

widthcollayer(N,0).          %If nothing at node, return zero.
```

```
%The following code will take in a sticks description on a lambda grid
%and produce a CIF description, using a specified value of lambda.
% The name of output file is X, without any . extensions.
%
expand(X) :-
      init,             %Initialize values.
      makeboxes,              %Expand all wires to boxes.
      updateboxes,            %Update boxes to remove fractions.
      open_file(X),           %Start writing to CIF file.
      writescale,       %Write the scale factors.
      writeboxes,             %Write out CIF file of all layers.
      writefincell,           %Finish CIF cell definition.
      told.             %Close file.

%Go through wire database, replacing it with box database.
makeboxes :-
      wire(Layer,Obj1,Obj2,Width),
      wired(Layer,Obj1,Obj2,Width),
      retract(wire(Layer,Obj1,Obj2,Width)),
      makeboxes;true.

%Map all wires to boxes.
wired(Layer,node(Row1,Col1),node(Row1,Col2),Width)  :- %Horizon. wire.
      var(Width),
      !,
      minwidth(Layer,Width),
      box_extend(node(Row1,Col1),node(Row1,Col2),Ext),
      Y is Row1 + (Width/2),
      X is (Col1 + Col2 + Ext)/2,
      setscale(Y),            %Mark the fact that Y is fractional.
      setscale(X),            % "  "        X       ".
      Z is Col2 - Col1,
      abs(Z,L1),
      L is L1 + Ext,
      assert(box(Layer,L,Width,X,Y)).

wired(Layer,node(Row1,Col1),node(Row1,Col2),Width)  :- %Horizon. wire.
      box_extend(node(Row1,Col1),node(Row1,Col2),Ext),
      Y is Row1 + (Width/2),
      X is (Col1 + Col2 + Ext)/2,
      setscale(Y),            %Mark the fact that Y is fractional.
      setscale(X),            % "  "        X       ".
      Z is Col2 - Col1,
      abs(Z,L1),
      L is L1 + Ext,
      assert(box(Layer,L,Width,X,Y)).

wired(Layer,node(Row1,Col1),node(Row2,Col1),Width)  :- %Vertical wire.
      var(Width),
      !,
      minwidth(Layer,Width),
      box_extend(node(Row1,Col1),node(Row2,Col1),Ext),
      Y is (Row1 + Row2 + Ext)/2,
      X is Col1 + (Width/2),
      setscale(Y),            %Mark Y being fractional.
```

```
    setscale(X),            % "  X "  ".
    Z is Row2 - Row1,
    abs(Z, L1),
    L is L1 + Ext,
    assert(box(Layer,Width,L,X,Y)).

wired(Layer,node(Row1,Col1),node(Row2,Col1),Width)  :- %Vertical wire.
    box_extend(node(Row1,Col1),node(Row2,Col1),Ext),
    Y is (Row1 + Row2 + Ext)/2,
    X is Col1 + (Width/2),
    setscale(Y),            %Mark Y being fractional.
    setscale(X),            % "  X "  ".
    Z is Row2 - Row1,
    abs(Z, L1),
    L is L1 + Ext,
    assert(box(Layer,Width,L,X,Y)),!.

wired(Layer,node(Row1,Col1),tran(Type,  node(Row2,Col2),Terminal),Width)  :-
    !,              %Only such rule.
    wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
    tran(Type, node(Row2,Col2),Terminal).

wired(Layer,tran(Type,node(Row1,Col1),Term),node(Row2,Col2),Width)  :-
    !,              %Only such rule.
    wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
    tran(Type,node(Row1,Col1),Term).

wired(Layer,via(Type1,node(Row1,Col1)),via(Type2,node(Row2,Col2)),Width)  :-
    !,              %Only such rule.
    wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
    via(Type1,node(Row1,Col1)),
    via(Type2,node(Row2,Col2)).

wired(Layer,tran(Type1,node(Row1,Col1),Term1),tran(Type2,node(Row2,Col2),Term2),Width)  :-
    !,              %Only such rule.
    wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
    tran(Type1,node(Row1,Col1),Term1),
    tran(Type2,node(Row2,Col2),Term2).

wired(Layer,node(Row1,Col1),via(Type,node(Row2,Col2)),Width)  :-
    !,              %Only such rule.
    wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
    via(Type,node(Row2,Col2)).

wired(Layer,tran(Type1,node(Row1,Col1),Term),via(Type2,node(Row2,Col2)),Width)  :-
    !,              %Only such rule.
    wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
    tran(Type1,node(Row1,Col1),Term),
    via(Type2,node(Row2,Col2)).

wired(Layer,via(Type1,node(Row1,Col1)),node(Row2,Col2),Width)  :-
    !,              %Only such rule.
    wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
    via(Type1,node(Row1,Col1)).
```

```
wired(Layer,via(Type1,node(Row1,Col1)),tran(Type2,node(Row2,Col2),Term),Width)   :-
      !,                  %Only such rule.
      wired(Layer,node(Row1,Col1),node(Row2,Col2),Width),
      via(Type1,node(Row1,Col1)),
      tran(Type2,node(Row2,Col2),Term).


%Expand transistors.
tran(Diff,node(Row,Col),gatel)  :- %Extend poly to right, vertical diff.
      polyext(P),       %Extension of poly beyond diff.
      minwidth(Diff,Wd),   %Minimum width of diff.
      minwidth(p,Wp),            %Minimum width of poly.
      X is Col + Wd + P, %Extend poly to right
      assertz(wire(p,node(Row,Col),node(Row,X),Wp)),!.

%
tran(Diff,node(Row,Col),gater)  :- %Extend poly to right, vertical diff.
      polyext(P),       %Extension of poly beyond diff.
      minwidth(p,Wp),
      X is Col - P,           %Extend poly to right
      assertz(wire(p,node(Row,X),node(Row,Col),Wp)),!.
%
tran(Diff,node(Row,Col),gatet)  :- %Extend poly to bottom.
      polyext(P),       %Extension of poly beyond diff.
      minwidth(p,Wp),
      X is Row - P,
      assertz(wire(p,node(X,Col),node(Row,Col),Wp)),!.
%
tran(Diff,node(Row,Col),gateb)  :- %Extend poly to top
      polyext(P),       %Extension of poly beyond diff.
      minwidth(p,Wp),
      minwidth(Diff,Wd),
      X is Row + P + Wd,
      assertz(wire(p,node(Row,Col),node(X,Col),Wp)),!.

%Dummy tran to fake out chanx for now
%**********************************

tran(D,N,T).


%Expand vias.
via(Type,node(Row,Col))  :-
      minwidth(cc,W),
      Y is Row + W/2,
      X is Col + W/2,
      setscale(X),
      setscale(Y),
      assert(box(cc,W,W,X,Y)).

%Absolute value function.
abs(X,Y) :- X < 0, Y is -X.
abs(X,X) :- X >= 0.

%Take boxes from database and write out CIF file.
```

```
writeboxes :-          %Loop over all the layers.
      layer(Layer),
      writelayer(Layer),
      writeboxes1(Layer),
      fail; true.

writelayer(m1) :-
      write('L CM;'), nl.
writelayer(p) :-
      write('L CP;'), nl.
writelayer(nd) :-
      write('L CD;'), nl.
writelayer(pd) :-
      write('L CS;'), nl.
writelayer(m2) :-
      write('L CM2;'), nl.
writelayer(cc) :-
      write('L CC;'), nl.

writeboxes1(Layer) :-
      box(Layer,L,W,X,Y),
      write('B '),
      write(L), write(' '),
      write(W), write(' '),
      write(X), write(' '),
      write(Y), write(';'),
      nl,
      fail;true.

%Find out if there are any non-integers in boxes.
%If so, update entire data base by scale factor of 2.
setscale(X) :-
      integer(X).       %Do nothing.
setscale(X) :-
      b(2).       %Do nothing.
setscale(X) :-
      not(integer(X)),
      not(b(2)),
      retract(b(1)),          %Remove default value.
      assert(b(2)). %Set scale factor of 2.

%Update all boxes.
updateboxes :-
      b(1).             %Do nothing.
updateboxes :-
      b(2),             %Scaling required?
      assertz(box(0,0,0,0,0)),      %Mark end of database.
      updateboxes1;retract(box(0,0,0,0,0)).
updateboxes1 :-
      box(Layer,L,W,X,Y),
      !,                %No bactracking, let recursion work.
      Layer = = 0,          %End marker.
      L1 is L*2,
      W1 is W*2,
      X1 is X*2,
```

```
        Y1 is Y*2,
        assertz(box(Layer,L1,W1,X1,Y1)),
        retract(box(Layer,L,W,X,Y)),
        updateboxes1.

%Initialize.
init :-
        assert(b(1)).           %Initialize scale factor.

%Write out Definition Start line of CIF cell.
writescale :-
        lambda(L),              %Fetch lambda value.
        A is L*100,             %Find first scale factor.
        b(B),           %Fetch second scale factor.
        write('DS '),
        write('1 '),        %Hard wired cell number for now.
        write(A), write(' '),   %First scale factor.
        write(B), write(';'),   %Second scale factor.
        nl.

%Write end cell definition instruction on CIF file.
writefincell :-
        write('DF;'), nl,
        write('C 1;'), nl,
        write('End'), nl.

%Box Extension routines
%Extend vertical box towards the top, if it is meant to intersect
%Some layer there.
box_extend(node(R1,C1),node(R2,C1),E):-
        R2 > R1,        %Check at top node only.
        rowpresent(node(R2,C1),E),!.    %Check for material width, if any.

box_extend(node(R1,C1),node(R2,C1),E):-
        R1 >= R2,           %Check top node.
        rowpresent(node(R1,C1),E),!.    %If material there,return width.

%Extend horizontal boxes to right.
box_extend(node(R1,C1),node(R1,C2),E):-
        C2 > C1,        %Check right node.
        colpresent(node(R1,C2),E),!.    %If material there, return width.

box_extend(node(R1,C1),node(R1,C2),E):-
        C1 >= C2,           %Check right node.
        colpresent(node(R1,C1),E),!.    %If material there, return width.

%If no material present, do not extend box.
box_extend(N1,N2,0).

%Check for prescence of material on row.
rowpresent(node(R,C),E) :-
        maprow(R1,R),       %Find inverse mapping of lambda row.
        mapcol(C1,C),       %Find inverse mapping of lambda col.
        row(R1,[L,L1]),     %Use virtual row in search.
        member(C1,L1),      %See if there is overlap.
```

```
        minwidth(L,E).          %If so, return width of horizontal layer.

%Check for prescence of material on col.
colpresent(node(R,C),E) :-
        maprow(R1,R),           %Find inverse maping of lambda row.
        mapcol(C1,C),           %"                  "     col.
        col(C1,[L,L1]),         %Use virtual col in search.
        member(R1,L1),          %See if there is overlap.
        minwidth(L,E).          %If so, return width of vertical layer.

%Open file with .cif extension.
open_file(X) :-
        name(X,L),
        append(L,".cif",L1),
        name(Y,L1),
        tell(Y).
```

```
%The following is a list of all technology layers in CMOS.

layer(p).   %Poly
layer(m1). %Metal1
layer(nd).  %N Diffusion
layer(pd).  %P Diffusion.
layer(m2). %Metal 2.
layer(cc).  %Contact Cut.
lambda(1.5).     %Lamda value.
```

```
%List all separations between layers.
sep(m1,m1,3).          %Metal1 spacing.
sep(m2,m2,4).          %Metal2 spacing.
sep(p,p,2).        %Poly spacing.
sep(nd,nd,3).          %Ndiff spacing.
sep(pd,pd,3).          %Pdiff spacing.
sep(cc,cc,1).          %Contact cut spacing.
sep(pd,nd,12):- !.     %Ndiff Pdiff spacing.
sep(nd,pd,12):- !.
sep(X,Y,1) :-
     X == Y.

%Min Width facts.
minwidth(cc,4) :- !.   %Contact Cut width is 4.
minwidth(pd,4) :- !.   %pdiff is 4 lambda.
minwidth(X,2).         %Everything else is 2.

%Rules for forming transistors.
polyext(2).        %Extension of poly beyond diffusion.
diffext(2).        %Extension of diff beyond poly.
```