# A WEINBERGER ARRAY GENERATOR

William W. Lin
Susan S. Yeh
Andrea S. LaPaugh

CS-TR-023-86

January, 1986

# A Weinberger Array Generator*

*William W. Lin*
*Susan S. Yeh*
*Andrea S. LaPaugh*

Department of Computer Science
Princeton University
Princeton, NJ 08544

TR CS-023
January 1986

## Abstract

The Weinberger Array Generator (*WAG*) is a tool for implementing random logic. Boolean equations are input, and a layout description of gates and wires (the circuit) realizing the equations is output. In the above aspects, *WAG* is similar to a PLA generator. The main difference is that the Weinberger array structure allows many levels of logic, with complex gates such as NAND-of-ORs; whereas a PLA structure allows only two levels of logic, with no gates more complex than NORs.

We shall describe our implementation of *WAG*, presenting issues concerning the optimization of logic, placement of gates, track assignment, and layout. Along with this, we shall discuss the trade-off between space requirements and timing delays that must be considered in choosing between a PLA and a Weinberger Array structure. Finally, we shall discuss possible improvements and uses for *WAG*.

* This work supported in part by NSF grant MCS-8004490, DARPA grant N00014-82-K-0549, and an IBM Faculty Development Award. S. Yeh was supported by ARO fellowship grant DAAG29-83-G-0110.

# Contents

# LIST OF FIGURES AND TABLES

# A Weinberger Array Generator*

*William W. Lin*
*Susan S. Yeh*
*Andrea S. LaPaugh*

Department of Computer Science
Princeton University
Princeton, NJ 08544

TR CS-023
January 1986

## I. Introduction

The Weinberger Array (WA) was first proposed by Arnold Weinberger in [1]. As an alternative to PLAs, WAs allow the use of complex gates and multi-level logic; thus, the layout should be more compact if a WA structure is used. Of course, because of the use of complex gates and multi-level logic, the signal propagation delay tends to be longer for a circuit implemented as a WA than for an equivalent circuit as a PLA.

In a general WA, the logic gates are placed side by side in a straight horizontal line. The input signals may come in from any of the four sides. Wires may cross gates to get from one place to another. Intermediate signals from the gates may propagate either to the left or right, and output signals may leave from any side [see Fig. 1].

Our prime motivation for building *WAG* was to have a tool that can build random logic circuits with complex gates. LaPaugh and Lipton introduced a production testing strategy based on bipartite circuits [2]. In order to test the strategy, we wanted to modify some random circuits using LaPaugh and Lipton's idea. A PLA generator was available, but it would generate circuits with only NOR gates. So, having no appropriate tools, we decided to build *WAG*.

* This work supported in part by NSF grant MCS-8004490, DARPA grant N00014-82-K-0549, and an IBM Faculty Development Award. S. Yeh was supported by ARO fellowship grant DAAG29-83-G-0110.

## I.A. Previous Work

Some projects have already employed the structure of Weinberger arrays for automatically laying out logic circuits. The MacPitts project at MIT Lincoln Laboratory [3] used WA to construct the combinatorial logic portion of the control unit, all in NOR gates in nMOS technology. The Lincoln Boolean Synthesizer [4], developed at MIT Lincoln Laboratory, takes in boolean expressions and generates WA consisting of NOR gates in CMOS technology. The Non-Von project at Columbia University [5] used a WA layout for the OR-plane of the PLA. AT&T Bell Labs [6] also developed a WAG that generates nMOS Weinberger arrays in NAND gates.

## I.B. Design Decisions

In the early stage of design of *WAG*, we were faced with the following decision: Within the framework of Weinberger arrays, what is our overall layout strategy (i.e. what type of gates should we use)? On the one hand, we would like to have flexibility which calls for variety of complex gates. On the other hand, we would like to keep down the complexity of the layout algorithm as well as the layout itself. Performance was also an important factor that we kept in mind. After evaluating the trade-off issues, we decided to use four types of gates - NOR, NAND, NOR-of-ANDs, and NAND-of-ORs, in the Weinberger arrays. We feel that the four types of gates offer us enough freedom in laying out different types of gates, and yet they are simple enough to enable the layout to follow a uniform structure.

The next design decision that we faced was the structure of the gates. We first simplified the structure of the Weinberger arrays by applying some restrictions. In *WAG*, inputs must come in from the left or the bottom, and outputs must leave from the right or the bottom. Also, in order to make the task of layout easier, we don't allow intermediate signals to propagate to the left. The considerations we had in designing the structure of the gates were: speed, area, power, regularity, and ease of implementation. After experimenting with various structures, we decided on the ones shown in Fig. 2.

# II. Design of WAG

*WAG* consists of two major modules. The first module takes in a boolean

description and performs logic and gate manipulations on the description. Three tasks comprise the first module. First, the input is translated into a logic parse tree. Second, the actual gates are created. Third, the number of gates used is minimized.

The second module of *WAG* takes the gates created by the first module and produces an actual circuit description. This second module also consists of three separate subtasks. First, the gates are placed linearly, in some specific order. Second, the wire segments are assigned to specific horizontal tracks in the channel. Finally, the actual gate structures (in nMOS technology) are created.

## II.A. Logic and Gate Manipulation

### II.A.1. The Parser

The purpose of the parser is to translate the input into a logic parse tree. The input consists of a set of boolean logic equations, comprised of intermediate macro and output definitions. We allow the use of the following operators: NOT, OR, AND, NOR, and NAND. With these five operators, there is considerable flexibility for the user, who can specify any logic functions he needs; yet the user is not restricted to, say, a sum-of-products formulation. If there is an ambiguity (i.e. missing parentheses) in the input, the Parser assumes a STRICT left-to-right ordering of the operations, where all the operators have the SAME priority except for NOT, which takes precedence over the rest. Below is an example. ['$' = 'NAND,' '*' = 'AND,' '#' = 'NOR,' '+' = 'OR,' '~' = 'NOT']

" x $ ~y + k * ~e $ (f # g)"

Interpreted as

"((((x $ (~y)) + k) * (~e)) $ (f # g))"

See Fig. C1 in Appendix C for an example of a parse tree.

### II.A.2. The Creation of Gates

The purpose of this phase is to translate the logic parse tree into a tree which only contains operators that can be realized in the target technology. Since our target

technology is nMOS, our operators must be of the inverting-logic type, i.e. NOTs, NORs, NANDs, NOR-of-ANDs, and NAND-of-ORs. These operators may thus be thought of as gates. [see Fig. C2 in Appendix C]

Aside from the obvious translation of noninverting-logic operators to inverting-logic operators by adding an extra NOT operator (eg. OR becomes NOT of a NOR), there are some special considerations. First, perhaps depending on the target technology, NANDs with a large number of inputs may be undesirable because they would exhibit long switching delays. Specifically, for our purposes, we do not want any NANDs with more than three inputs. If one exists, it is transformed into an equivalent NOT of a NOR of NANDs [see Fig. 3]. A second consideration is the possibility that the user wants to keep a specified input phrase together, unchanged, if possible. This feature, while not currently implemented, can be accommodated by setting a flag that indicates that an operator is to be kept intact, as an independent gate. [see Fig. C5 in Appendix C]

### II.A.3. Logic Minimization

The purpose of this phase is to minimize the number of gates used to implement the input boolean equations. This should minimize the area needed for laying out the circuit, even though the complexity of the gates will most likely increase. In general, complex gates require more area than simpler gates; but since there are fewer gates, the total area required for gate separation is less. Further, combining gates into more complex gates decreases the amount of space needed for routing between gates.

Having transformed to inverting-logic gates in the preceding phase, one obvious step is to remove double NOTs. That is, if a NOT operator has an operand which is also a NOT, both operators may be deleted. Two other possibilities for minimization readily come to mind. First, one might try to locate equivalent subexpressions. The difficulty with this is that the subexpressions may be represented in different ways; and to tell whether two different-looking expressions are equivalent may be time-consuming, especially if we must check many pairs. And even the problem of determining the equivalence of a single pair of boolean expressions seems to be tough; in fact , it is Co-NP Complete (reduction from Non-tautology [7]). The second strategy is to collapse gates into more complex structures. For example, a NOR of 2 ANDs could be re-made into a single NOR-of-ANDs. This would certainly reduce the number of gates, but it's not clear whether a greedy algorithm for collapsing would ensure a minimal number of gates. [see Figs. C3 and C4 in Appendix C]

### II.A.4. Alternatives for the Creation of Gates

The program structure we used was adopted mainly for the sake of ease of implementation, so we may well ask if any improvements can be made. We may not want to specify gates too soon, since minimization issues may be important to determining what gates to build. One possibility is first to translate the parse tree into a simplified form, like sum-of-products, for which detecting common subexpressions is all that is required to find equivalent subexpressions, and minimization techniques have already been developed [8] and used in an actual implementation of a Weinberger array generator [9]. After the minimization, we could transform to inverting-logic. Of course, this method may still not give an optimum result, since the step of changing to inverting-logic may significantly increase the number of gates. Research is necessary to determine how the minimization of logic should proceed.

## II.B. Layout

### II.B.1. Linear Placement of Gates

Having defined the gates needed to implement the boolean logic equations, we now order them in a linear array so as to minimize the number of tracks needed for laying out the circuit. For this phase, we may think of each gate as a node in a graph, with a directed edge between two nodes if the corresponding gates have a wire connecting them. The goal is to place the nodes horizontally, in a straight line, so as to minimize the maximum cut, where a cut is the number of edges that bisect a single vertical line placed on the graph [see Fig. 4]. The general form of this problem is called the MINIMUM CUT Linear Arrangement Problem, and it is known to be NP-complete [7]. It is also known that an optimal solution can be found in polynomial time if the class of graphs is restricted to trees [10].

Our actual task does not fall easily into either the general case or the "tree" case. It might seem that our graph must be a directed acyclic graph, since no feedback paths are allowed, but that is not exactly the case. In certain cases, we might have to consider a *hyper* or *multiple* graph. To see why this is so, look at Fig. [5]. A given signal may be input to more than one node (eg. 'A'), but because we are using a linear placement of the nodes, such a signal will seem to enter at one node and then proceed

on from that node. Parts (b) and (c) of the figure show two different linear placements of the nodes. In part (b), 'A' and 'D' enter at node 1; in part (c) they enter at node 2. One cannot know beforehand which of nodes 1 and 2 should be placed farther to the left, so one cannot know in which direction to place a directed arc between the two nodes. There may be more than two nodes that share a common signal, and generally there will be no *a priori* way to tell what the relative ordering of these nodes should be; so we may visualize an undirected hyper-edge connecting these nodes. The graph may be multiple if at least two signals must pass through the same two vertices [see parts (b) and (c), where there are two arcs connecting nodes 1 and 2]. Both edges must be present because the two signals will require two separate tracks in the actual layout.

Finding an optimal placement does not seem to be a simple task. Simulated Annealing may be the most promising try, as indicated by the results obtained by Rowen and Hennessy [11]. We are searching for a specific algorithm to solve our problem.

## II.B.2. Track Assignment

After all the gates have been linearly placed, we want to assign wire segments into horizontal tracks to minimize the number of tracks actually used (hence reducing the height of the circuit). The track assignment problem breaks down into two cases. One case is NOR and NAND gates; the other NOR-of-ANDs and NAND-of-ORs gates.

For the case of NOR and NAND gates, a greedy algorithm will achieve the optimal track assignment [12]. The basic idea behind the greedy algorithm is to pack as many non-overlapping wires into a single track as possible (this is actually an interval graph coloring problem [13]). The method for doing so is for each track (assuming the tracks are vertically divided into columns) choose a wire that fills in the leftmost empty column, if one exists.

For the case of NOR-of-ANDs and NAND-of-ORs, the situation becomes difficult. Due to the structure of the gates, we have to impose adjacency constraints on the wires. For instance, consider a NOR-OF-ANDs gate: NOR of [ a & b ], [ c & d & e ]. The wires a and b are constrained to be adjacent, so are wires c and d and e (the order does not matter though). In some cases routing is necessary in order to satisfy the adjacency constraints (see Fig. 6). In other cases, routing is preferable in order to reduce the number of tracks used (i.e. not to stretch the gate). If we don't allow

routing, then we think the problem of finding a minimum track assignment, if one exists, is NP- complete.

For the above harder case, the track assignment problem becomes: assign wires to tracks so as to minimize the number of tracks used and the amount of routing that is necessary. In our present version of *WAG*, track assignment for NAND-of-ORs and NOR-of-ANDs gates has not been implemented. This is an interesting problem for future work.

## II.B.3. Creation of Gate Structures - Layout

Having defined the order of gates and the order of wires connecting the gates, the last phase is to lay out the Weinberger array in PIF (PIF is an intermediate form that describes the layout [14] ). Working under the PIF framework, we treat each gate as a cell and the wires connecting two gates as another cell. Layout proceeds from left to right, cell by cell. [see Fig. C6 in Appendix C for an example of a final layout]

The information we need in order to lay out each gate is : type of gate, input wire names, left tracks and right tracks with wire names and type of wire for each track. In layout, it is essential to keep track of wire positions. The way we do so is by taking advantage of the regularity of the gate structure and doing case analyses. In the current version of *WAG*, the input and output wire positions are determined by the program. As a future improvement, we would allow the user to specify the sides and order in which the inputs and output(s) appear.

Having completed the final phase we want to know if improvements can be made in layout. We are concerned with the quality of the Weinberger array circuit, and three important quantities we use to measure the quality of a circuit are speed, area, and power. How do we optimize speed, area, and power (we may consider time-area product, time-area-power product, or some weighted combination thereof)? Since we are working in nMOS technology, one possibility is to scale the pullup sizes and pulldown wire widths according to optimization parameters. We are currently considering an integration of *WAG* with a tool that performs speed-area-power optimizations.

A disadvantage of the Weinberger array structure is that circuit tends to be very long. When this happens, we can "fold" the circuit to a more desirable shape [see Fig. 7]. Additional routing of wires may be necessary, though.

## III. Performance Comparison

In this section, we are going to make performance comparisons between a 1-bit full adder implemented as a programmable logic array (PLA) and a 1-bit full adder generated by *WAG*. The three performance measures we used are: speed or time delay $T$, peak or average power dissipation $P_{max}$ or $P_{avg}$, and area $A$.

Figure 8 illustrates the 1-bit adder implemented as a PLA. The equations used are $C_0 = AB + BC + AC$, and $S = ABC + A\bar{B}\bar{C} + \bar{A}B\bar{C} + \bar{A}\bar{B}C$. Iwano and Steiglitz [15] have done some experiments in local optimization of VLSI leaf cells, for example, a 1-bit full adder. The way they perform local optimization is to consider all single or double changes of the circuit parameter vector $\pi$ along the critical path of the leaf cell. We will use their performance results of a 1-bit full adder generated by PLA generators in our comparisons.

Figure 9.a and Figure 9.b are plots of the 1-bit full adder generated by *WAG* with pullup ratios 2 and 1 respectively. We have placed input buffers in the Weinberger arrays since the PLA circuit also has them. In this circuit the inputs enter through the bottom, and the outputs leave from the right. We used different but equivalent equations for $C_0$ and S, as indicated in the figures.

The tools used for estimating the performance of the circuits are ALLENDE[16], MEXTRA[17], CRYSTAL[17], and POWEST[17]. Table 1 shows a comparison of the performance of the PLA circuits and the WA circuits.

| type | $A$ | $P_{AVG}$ | $P_{MAX}$ | $T$ | $APT$ | $PT$ | parameter |
|---|---|---|---|---|---|---|---|
| PLA | 21560 | 6472 | 10183 | 12.8 | 2802 | 1303 | 1) $\pi$ = (4,4,4,4,4,4,3,4,4,8,8,8,4,4,4,8,2) |
| | 21840 | 5678 | 9241 | 15.3 | 3087 | 1413 | 2) $\pi$ = (4,2,3,3,3,3,3,4,3,8,8,8,4,4,4,8,2) |
| | 21762 | 5503 | 8616 | 14.9 | 2794 | 1284 | 3) $\pi$ = (3,3,3,4,4,4,4,3,3,8,8,8,4,4,4,4,3) |
| PLA Berkeley | 22176 | 7314 | 11749 | 12.8 | 3339 | 1504 | 4) $\pi$ = (4,4,4,4,4,4,4,4,4,8,8,8,8,8,8,8,8) |
| WA | 17051 | 1556 | 2349 | 29.3 | 1174 | 688 | 5) Pullup size = 2 |
| | 21535 | 3112 | 4698 | 22.8 | 2307 | 1071 | 6) Pullup size = 1 |

Table 1 Performance Comparison of the 1-bit Full Adder

The first four rows are all local optimal points using time $T$ as criterion. The units of $A$, $P_{avg}$, $P_{max}$, $T$, $APT$, and $PT$ are $\lambda^2$, $(10^{-6} * W)$, $(10^{-6} * W)$, $ns$, $(\lambda^2 * W * ns)$, and $(10^{-4} * W * ns)$ respectively. Each of the PLA circuits has 17 parameters represented by a vector $\Pi = (d_{and_1}, d_{and_2}, \ldots, d_{and_7}, d_{or_1}, d_{or_2}, d_{in_{1,1}}, \ldots, d_{in_{3,2}}, d_{out_1}, d_{out_2})$, where node i has diffusion width $d_i\lambda$. More specifically the parameters are pulldown diffusion widths: 7 nodes in the AND plane, 2 nodes in the OR plane, 6 nodes in the input stage, and 2 nodes in the output stage. In all circuits, the pullup to pulldown ratio of the gates is 4.

The results of the performance comparisons are as expected: the Weinberger arrays circuit is more compact, consumes less power, but is slower than the equivalent PLA circuit. Hence we have to consider the trade-off between space requirements and timing delays in choosing one implementation over the other.

## IV. Conclusion and Future Improvements

We have shown the Weinberger Array Generator to be a viable alternative to PLA generators. Although not recommended if speed is the primary concern, the Weinberger array performs well under the criteria of space and power usage. We have described our implementation of *WAG*, and we note that many improvements may be made. We have already mentioned four possible optimizations: minimizing the amount of logic used by collapsing gates and recognizing equivalent subexpressions, minimizing the number of tracks needed and the lengths of the wires on these tracks, stacking up long chains to obtain a proper aspect ratio, and optimizing the circuit with respect to area, speed, and power usage by improving the layout structure and sizing the gate components. We might also add input buffers to protect against degraded signals. To make the tool more "friendly," we could allow the user to specify where signals are to be input or output. Finally, we might add flexibility by allowing clock lines and internal feedback paths.

# Sources


[1] Arnold Weinberger , "Large scale integration of MOS complex logic: a layout method," *IEEE Journal of Solid-State Circuits*, Vol. SC-2, No.4, pp. 182-190, Dec. 1967.

[2] Andrea S. LaPaugh and Richard J. Lipton, "Total Stuck-at-Fault Testing by Circuit Transformation," *Proceedings of the 1983 International Test Conference*, IEEE, pp. 428-434, October 1983.

[3] Jeffrey M. Siskind, Jay R. Southard, and Kenneth W. Crouch, "Generating Custom High Performance VLSI Designs From Succinct Algorithmic Descriptions," *1982 Conference on Advanced Research in VLSI*, M.I.T., pp. 28-39, 1982.

[4] Jay R. Southard, Antun Domic, and Kenneth W. Crouch, "Report on the Lincoln Boolean Synthesizer," *Digest of International Conference on Computer-Aided Design*, IEEE, pp. 192-193, September 1983.

[5] Theodore M. Sabety, David E. Shaw, and Brian Mathies, "The Semi-Automatic Generation of Processing Element Control Paths for Highly Parallel Machines," *21st Design Automation Conference*, pp. 441-446, 1984.

[6] S. C. Johnson, "Code Generation for Silicon," *Proc. Tenth ACM Symposium on Principles of Programming Languages*, pp. 14-19, 1983.

[7] Michael R. Garey and David S. Johnson, *Computers and Intractability*, pp. 201 and 261, W. H. Freeman Co., San Francisco, 1979.

[8] Robert K.Brayton and Curt McMullen, "Synthesis and Optimization of Multistage Logic," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, pp. 23-28, 1984.

[9] Christopher Rowen, *Multi-Level Logic Array Synthesis,* Tech. Report #85-279, Computer Systems Lab., Stanford Univ., July 1985.

[10] Mihalis Yannakakis , "A Polynomial Algorithm for the Min Cut Linear Arrangement of Trees," *24th Annual Symposium on Foundations of Computer Science*, pp. 274-281, Computer Science Press, 1983.

[11] Christopher Rowen and John Hennesey, "SWAMI: A Flexible Logic Implementation System," *Proc. 22th Design Automation Conf.*, pp. 169-175, Las Vegas, June 1985.

[12] Akihiro Hashimoto and James Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures," *Proc. 8th Design Automation Workshop*, IEEE, pp. 155-169, 1971.

[13] Fanica Gavril, "Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph," *SIAM J. Comput.* Vol. 1, No. 2, pp. 180-187, June 1972.

[14] Jose Mata, *ALLENDE Layout System User's Manual*, VLSI Memo #9, Computer Science Department, Princeton University, Princeton, NJ, June 1984.

[15] Kazuo Iwano and Kenneth Steiglitz, "Some Experiments in VLSI Leaf-cell Optimization," *1984 IEEE Workshop on VLSI Signal Processing,* University of Southern California, Nov 12-14, 1984.

[16] Jose Mata, "ALLENDE: A Procedural Language for the Hierarchical Specification of VLSI Layouts," *Proc. 22nd Design Automation Conf.*, pp. 183-189, Las Vegas, June 1985.

[17] Robert N. Mayo, John K. Ousterhout, and Walter S. Scott, *1983 VLSI Tools,* Report No. UCB/CSD 83/115, Computer Science Division (EECS), University of California, Berkeley, California, March 1983.

# APPENDIX A: The *WAG* User's Manual

# *WAG* user's manual

## Introduction

The Weinberger Array Generator (*WAG*) is a tool for implementing random logic. Boolean equations are input, and a layout description of gates and wires (the circuit) realizing the equations is output. In the above aspects, *WAG* is similar to a *PLA* generator. The main difference is that the Weinberger array structure allows many levels of logic, with complex gates such as a NAND-of-ORs; whereas a *PLA* structure allows only two levels of logic, with no gates more complex than NORs.

For a description of the general structure of Weinberger arrays and their implementations, consult *IEEE Journal of Solid-State Circuits*, Vol. SC-2, No. 4, pp.182-190, December 1967, A. Weinberger, "Large scale integration of MOS complex logic: a layout method" or *Computational Aspects of VLSI*, Jeffrey D. Ullman, Computer Science Press, pp. 338-352. The actual implemented structure of Weinberger arrays will be apparent from the output.

## Usage:

```
wag [ -options [ -options ] ... ] infile
```

The input file *infile* consists of any number of boolean equations, separated by semi-colons. The format for an equation is that of a normal arithmetic equation; that is, the left hand side consists of a single variable, followed by an '=', then the right hand side is a boolean equation consisting of variables and operators. Variables are alphanumeric strings not beginning with a numeral. The legal operators are NOT ('~'), AND ('*'), OR ('+'), NAND ('$'), and NOR ('#'). Parentheses are used for phrasing. The notation used is infix.

For example, a NAND of 'a', 'b', and 'c', with the result to be placed in 'x' would be written as

"x = a $ b $ c;"

The semi-colon following the equation must be present if any other equation follows.

Here is a legal input:

"what = (x $ a) + (~y * you);
ask = w * x # what;
> ans = ~(((ask * you) + what) $ ~ask $ (x # y # z # w))"

Notice the '>' preceding the final equation. The placement of this symbol in front of an equation notifies *WAG* that the variable that immediately follows is an output signal. If this symbol is not present, then the signal is assumed to be an intermediate macro definition.

If the phrasing of a boolean input string is ambiguous, the program will assume that the phrasing is strictly left-to-right, with the NOT operator having the highest priority and the rest having the SAME priority. For instance,

"a + c * e $ ~f"

would be interpreted as if written

"(((a + c) * e) $ (~f))".

*WAG* produces CIF and PIF files specifying the mask layout geometry for a circuit realizing the input equation as the output. CIF, the Caltech Intermediate Form that describes the layout is defined in Mead and Conway, *Introduction to VLSI Systems*, Addison-Wesley, pp. 115-127. PIF is another intermediate form for describing the layout. One can obtain a CIF file from the PIF file by running ALLENDE. For descriptions of ALLENDE and PIF, consult *ALLENDE Layout System User's Manual,* Jose Mata, VLSI Memo #9, Princeton Univ., June 1984, or "ALLENDE: A Procedural Language for the Hierarchical Specification of VLSI Layouts," Jose Mata, *Proc. 22nd Design Automation Conf.*, pp. 183-189, June 1985.

The options for the Weinberger Array Generator are :

-f <filename> <space>

Write the Weinberger arrays in the file specified. In the present version, both 'pif' and 'cif' files are generated. If the option -C is specified, then only the 'pif' file is generated. Note that ".pif" or ".cif" will automatically be concatenated to the filename if the filename is not already in that format.

-F {default} Use the default filename of "wag.cif"

-s Silent execution.

-S {default} Write out main steps of execution.

-c {default} Produce 'cif' file.

-C Do not produce 'cif' file.

-v {default} Compute width of Vdd and Gnd lines, widening them if necessary (minimum width for these lines is assumed to be 4*lambda).

-V n Do not compute width of Vdd and Gnd lines. Use Vdd and Gnd lines of width n*lambda, but no less than 4*lambda.

-m n Use metal lines of width n*lambda, but no less than 3*lambda. (default n is 3).

-d n Use diffusion lines of width n*lambda, but no less than 2*lambda. (default n is 2).

-p n Use poly lines of width n*lambda, but no less than 2*lambda. (default n is 2).

## Future Improvements

(a) Input buffering:
As implemented, *WAG* assumes the input signals to be non-degraded. However, this may not be the case all the time. So, we would place input buffers in the layout to allow for degraded input signals.

(b) Specification of input/output entry:
As implemented, *WAG* arbitrarily places inputs and output(s) (the inputs and output(s) are labeled in the CIF-plot). We would allow the user to specify the sides and order in which the inputs and output(s) appear.

(c) Specification of gates:
The user will be able to specify the exact gates (limited to NOR, NAND, NOT, NOR-of-AND, NAND-of-NOR) to be laid out by turning on an option. Right now, optimizations are performed on the equations and the output gates are

limited to NOT, NOR, and NAND.

(d) Better performance -- TIME:

The user will be able to turn on an option that will call a routine which tries to minimize the total delay of the circuit (by scaling the pull-up sizes and pull-down wire widths).

# APPENDIX B: Figures

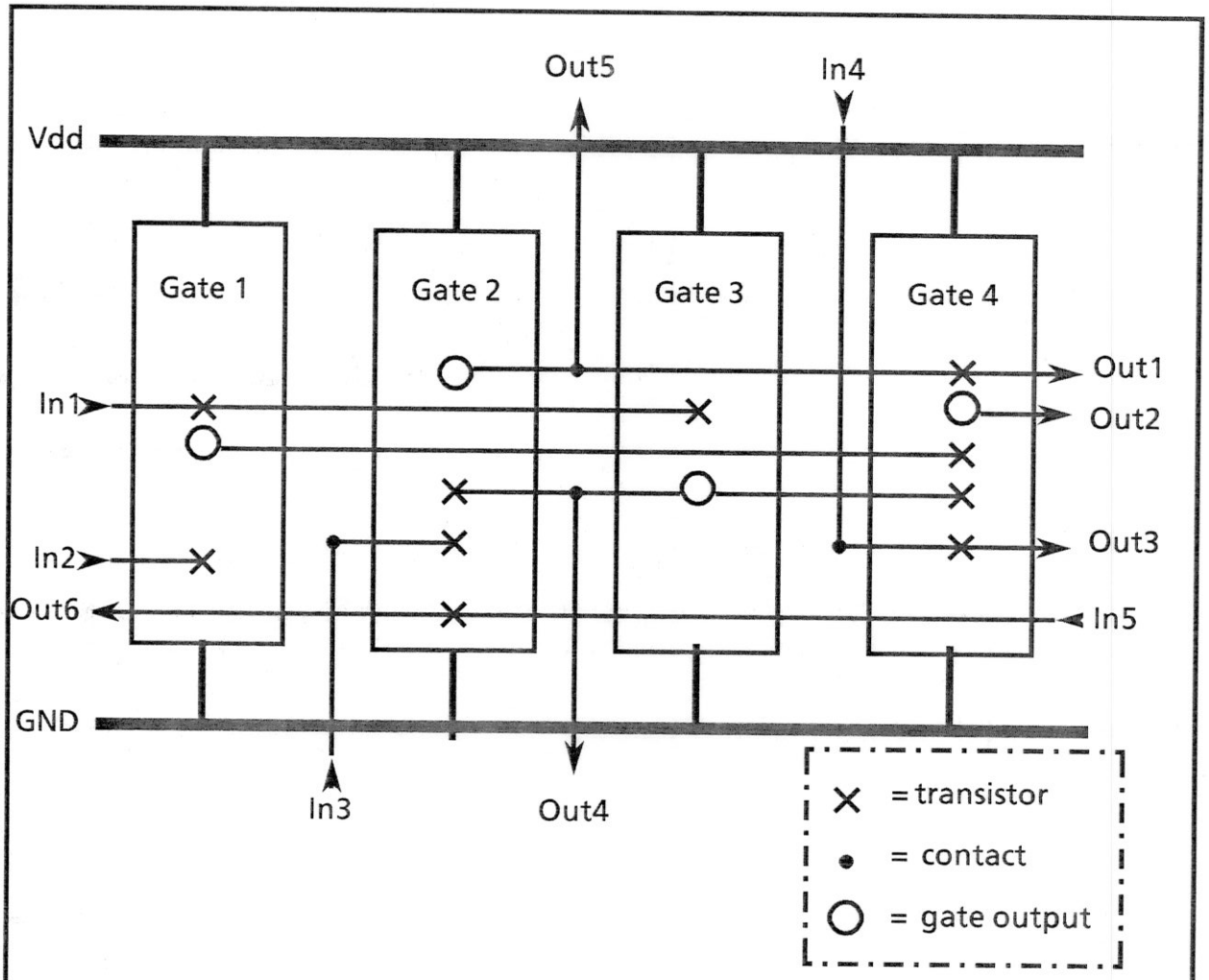



Fig. 1 Generalized example of Weinberger array structure

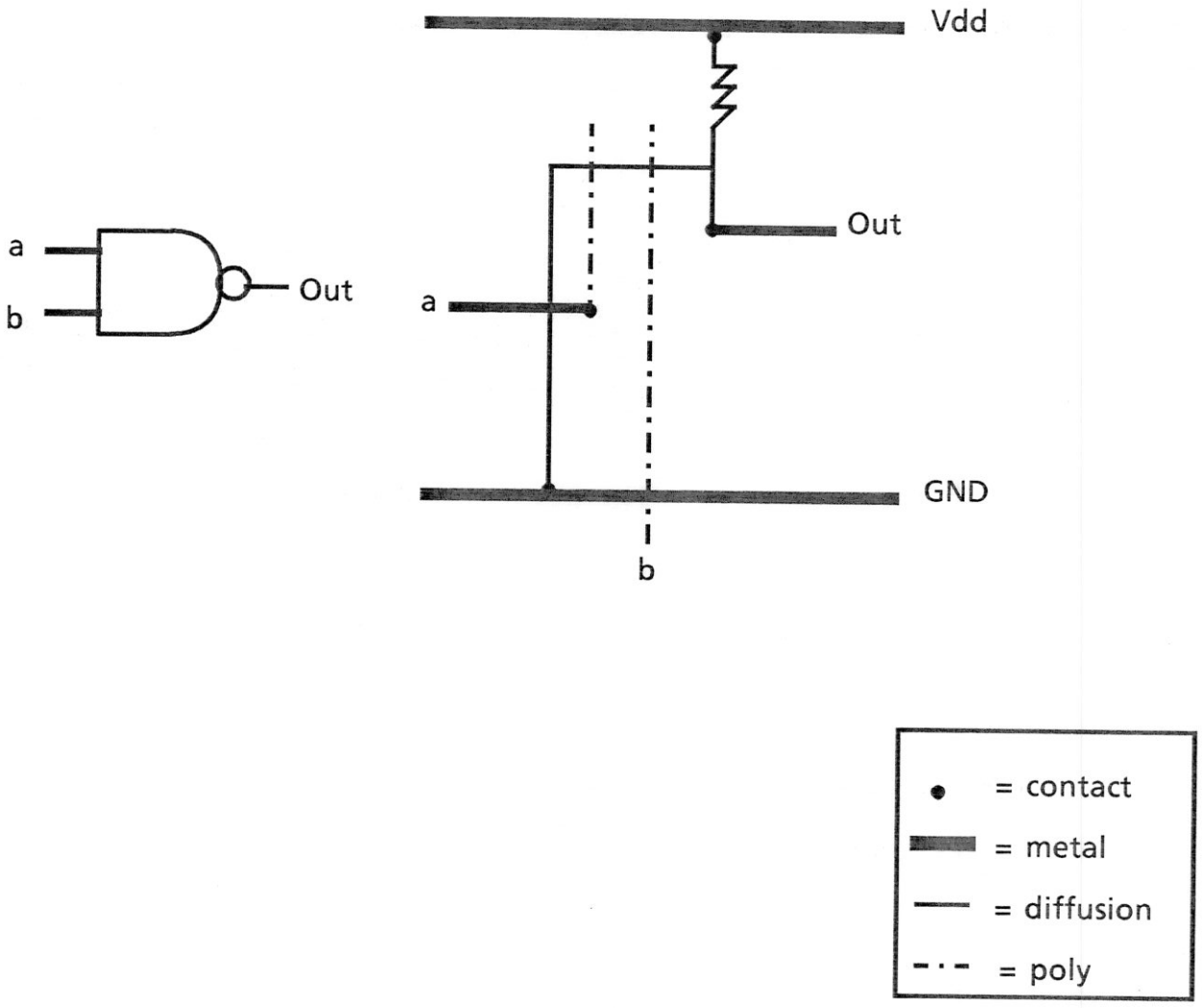



Figure 2.a NAND Gate Structure

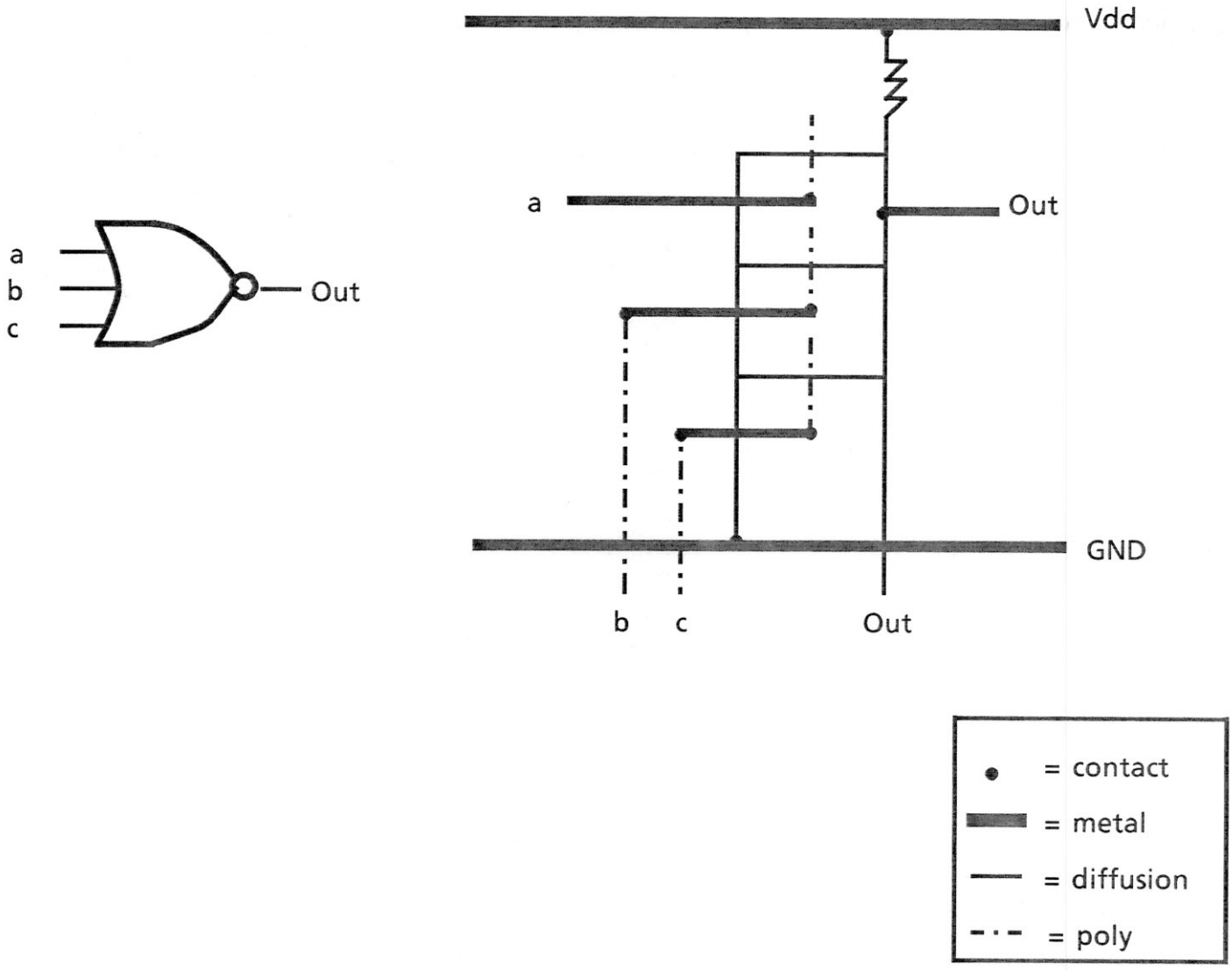



Figure 2.b NOR Gate Structure

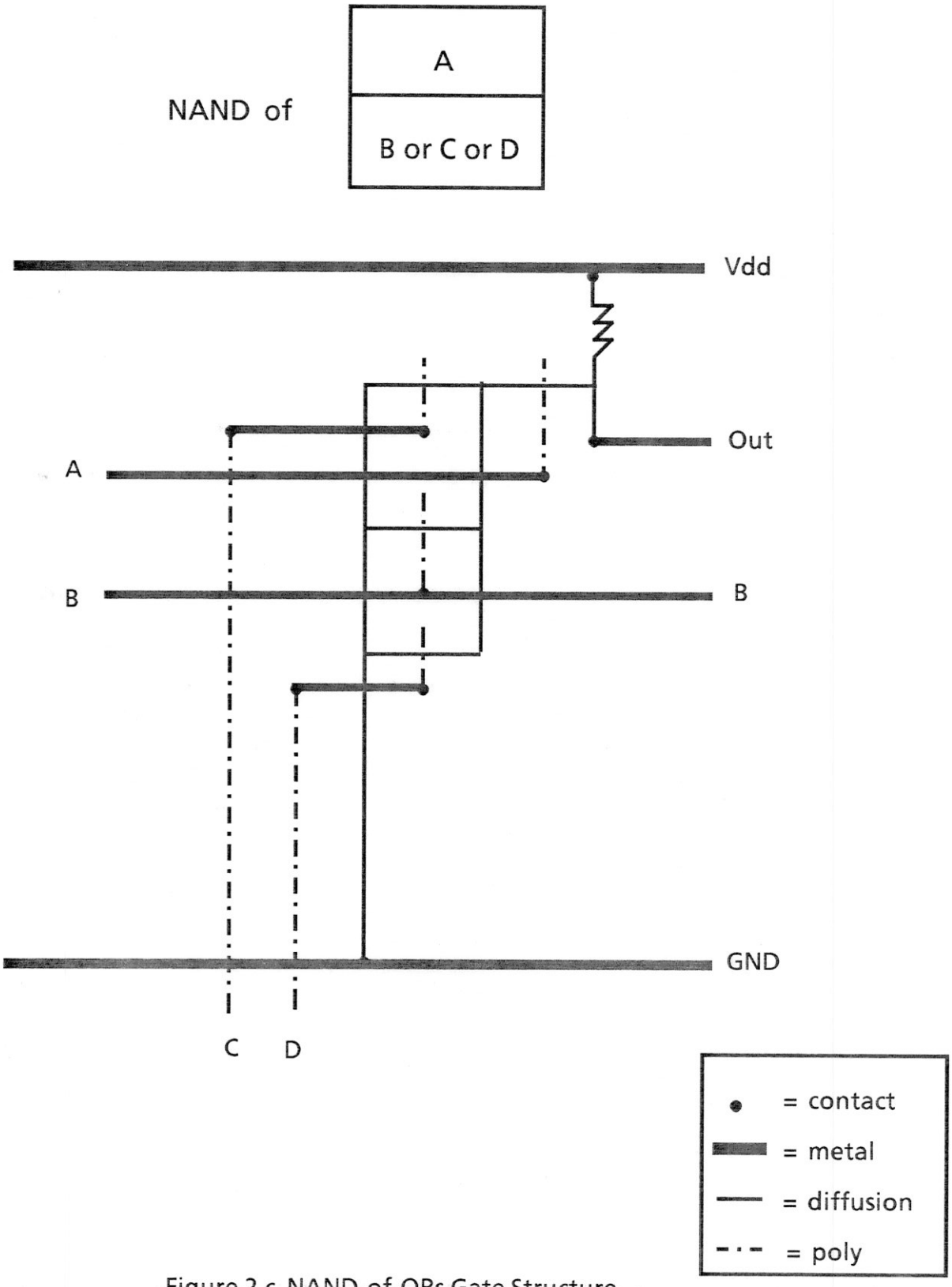



Figure 2.c NAND-of-ORs Gate Structure

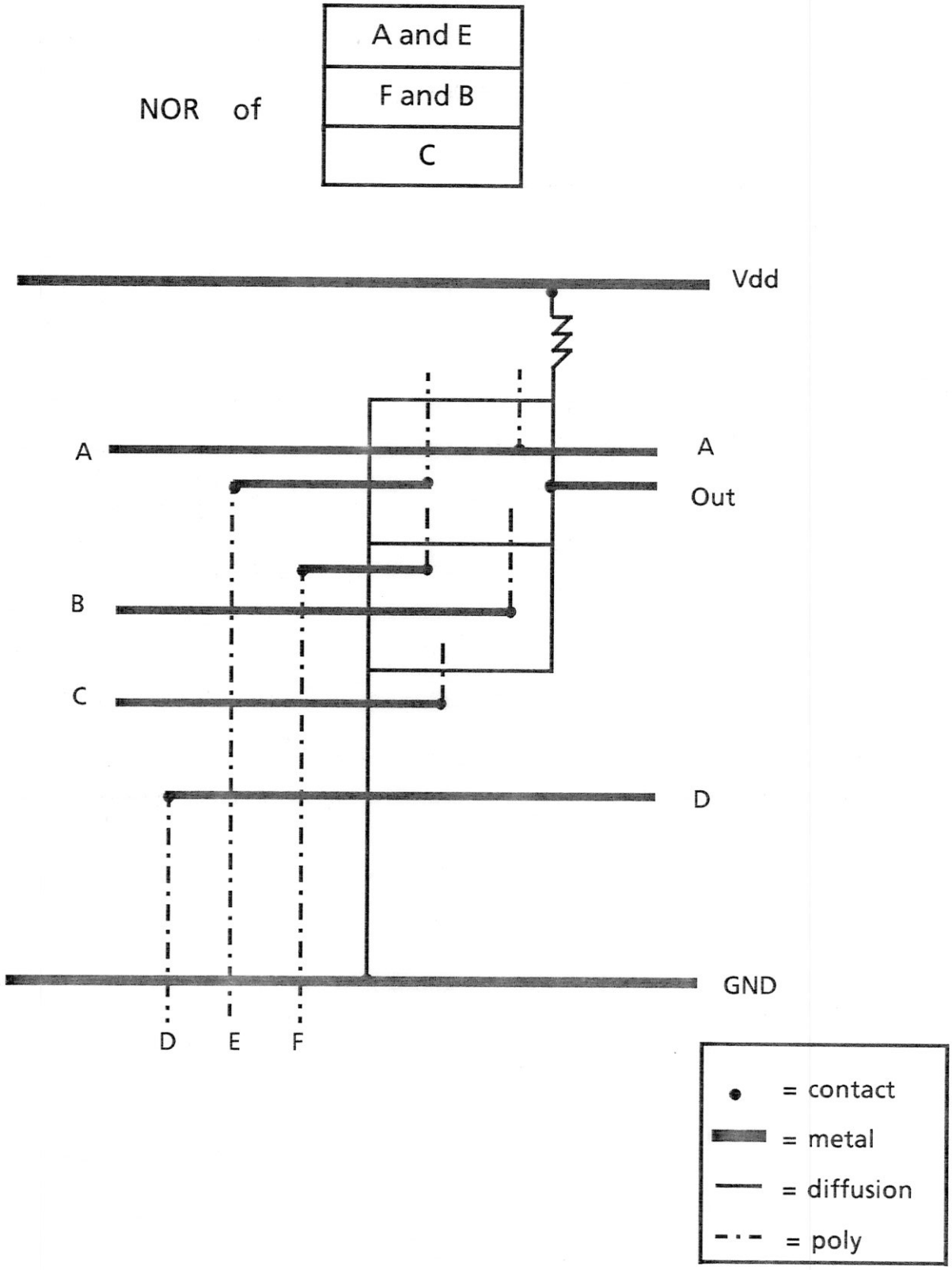



Figure 2.d NOR-of-ANDs Gate Structure

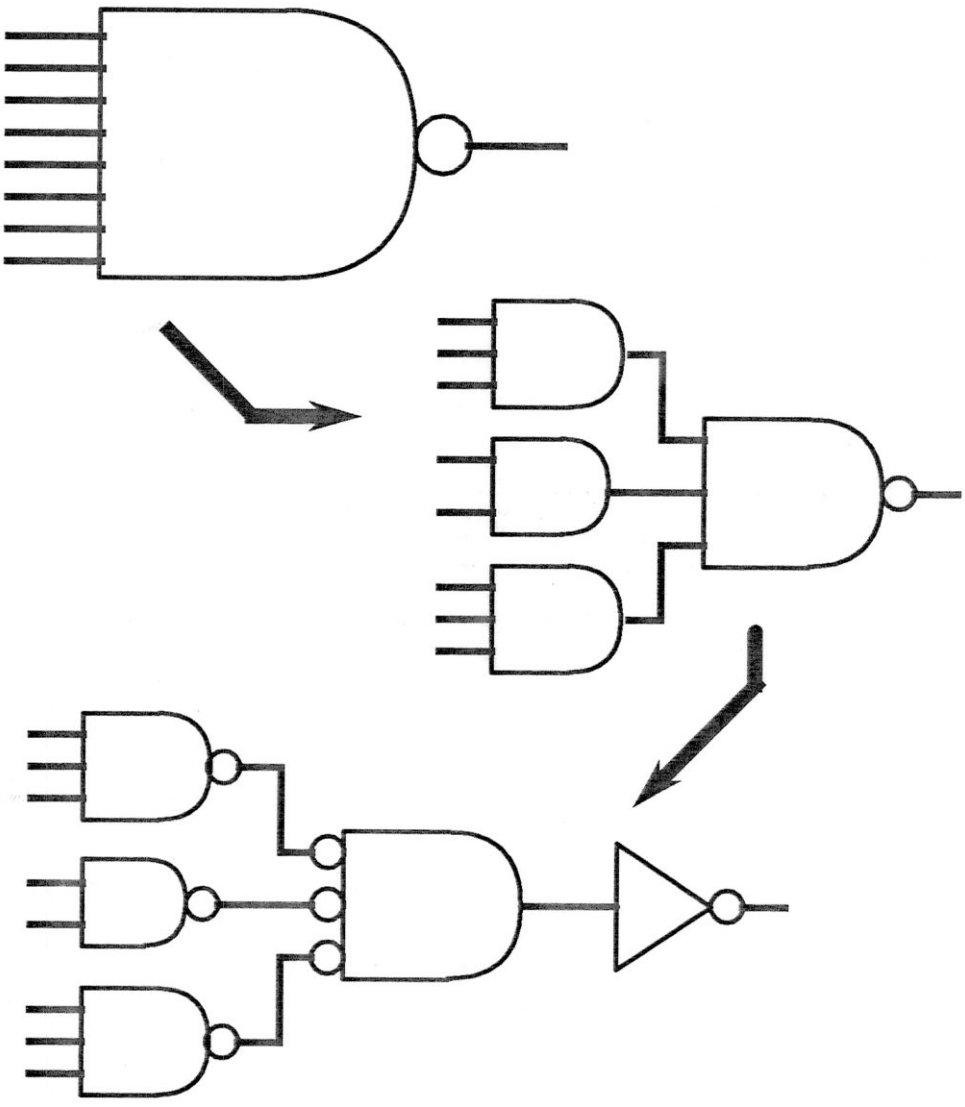


Fig. 3 Equivalence of a NOT of a NOR of NANDs to a large-input NAND

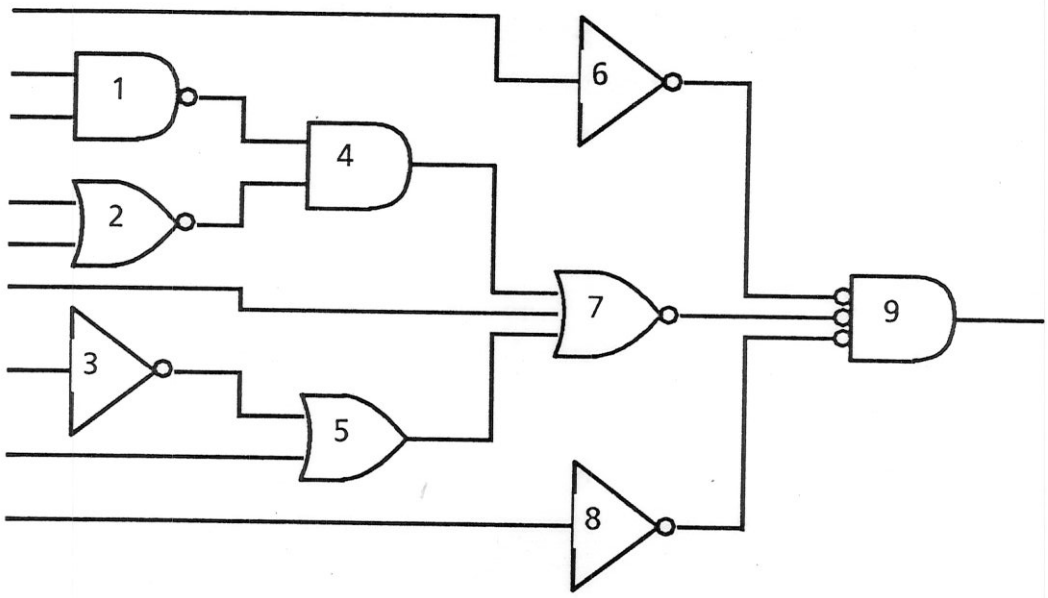



Fig. 4.a Sample boolean circuit

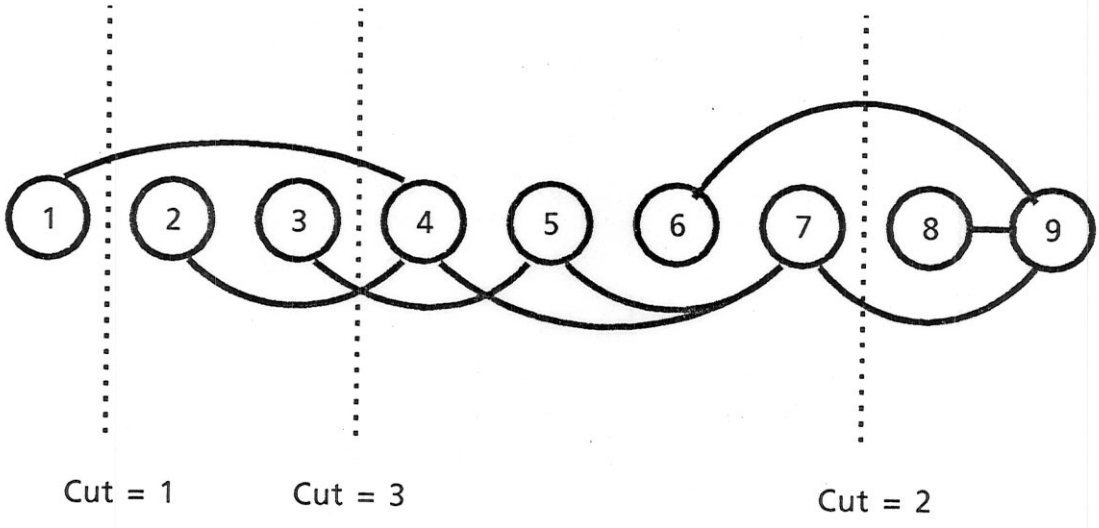



Fig. 4.b Sample linear placement of above circuit with MAX CUT = 3

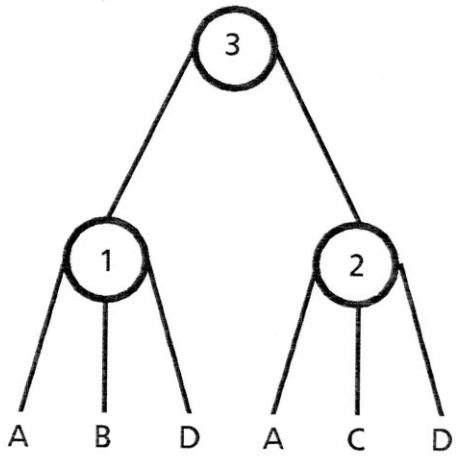
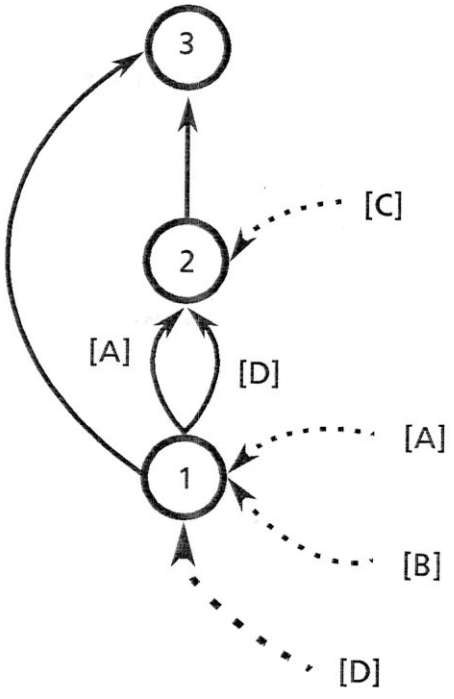



Fig. 5.a An example input to linear placement phase



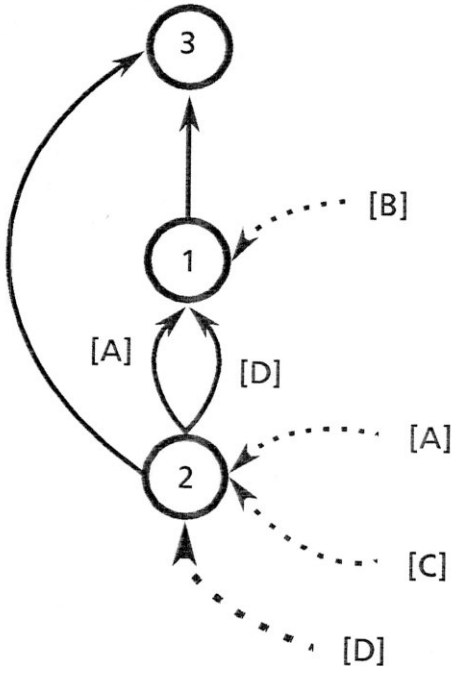




Fig. 5.b, 5.c Two different graphical representations for the above input

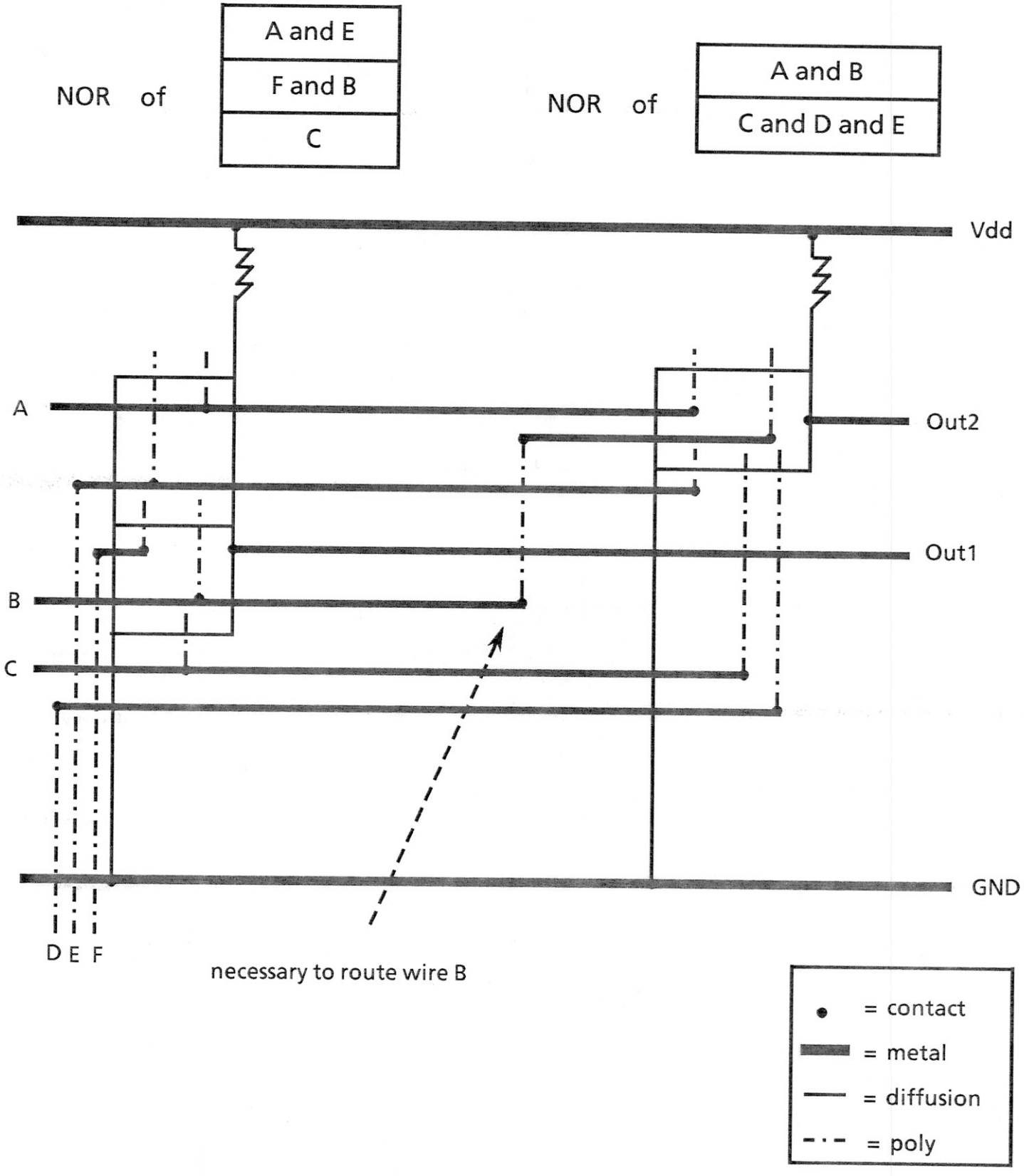



Figure 6 Illustration of Adjacency Constraints and Routing

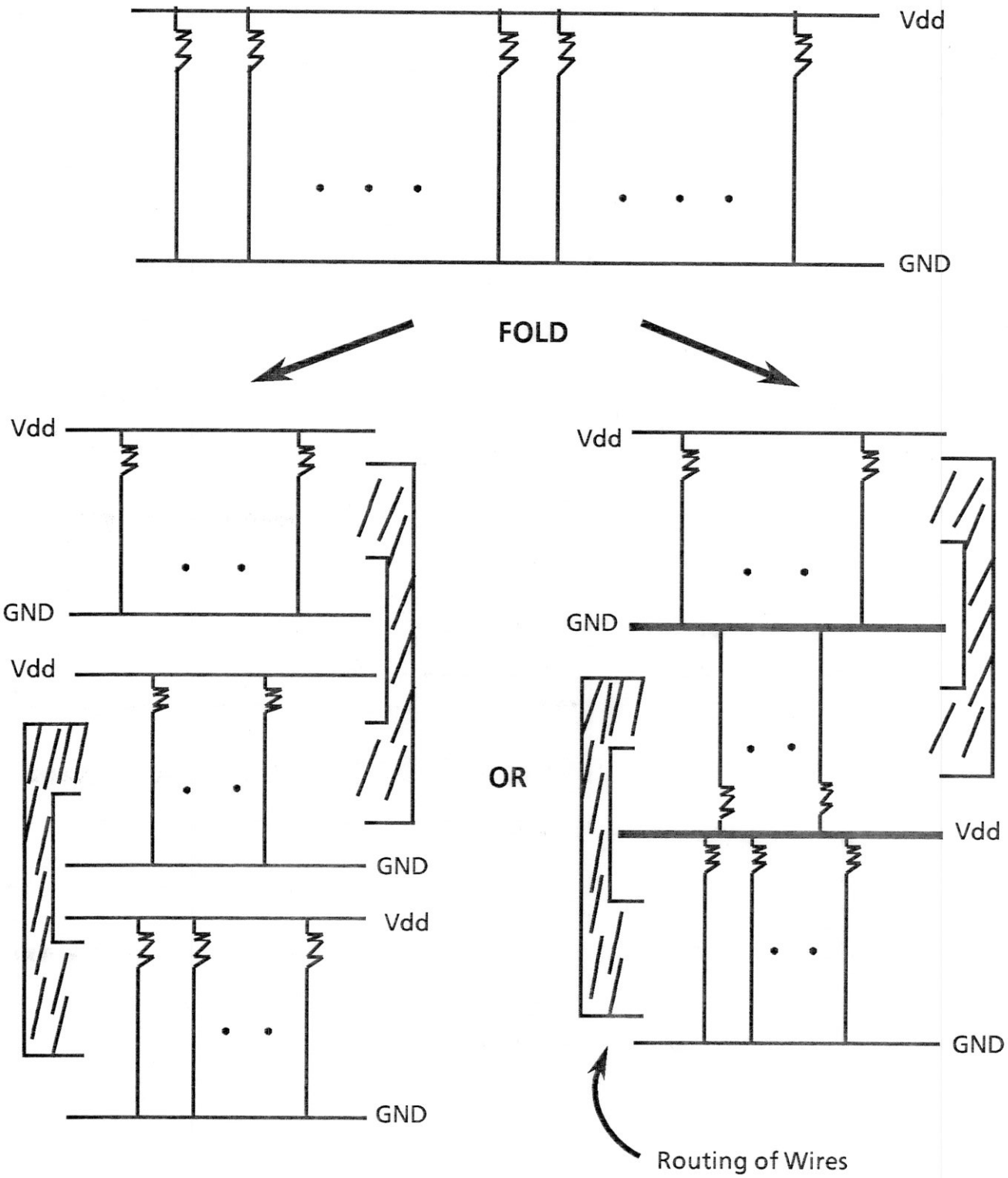



Figure 7 Circuit Folding

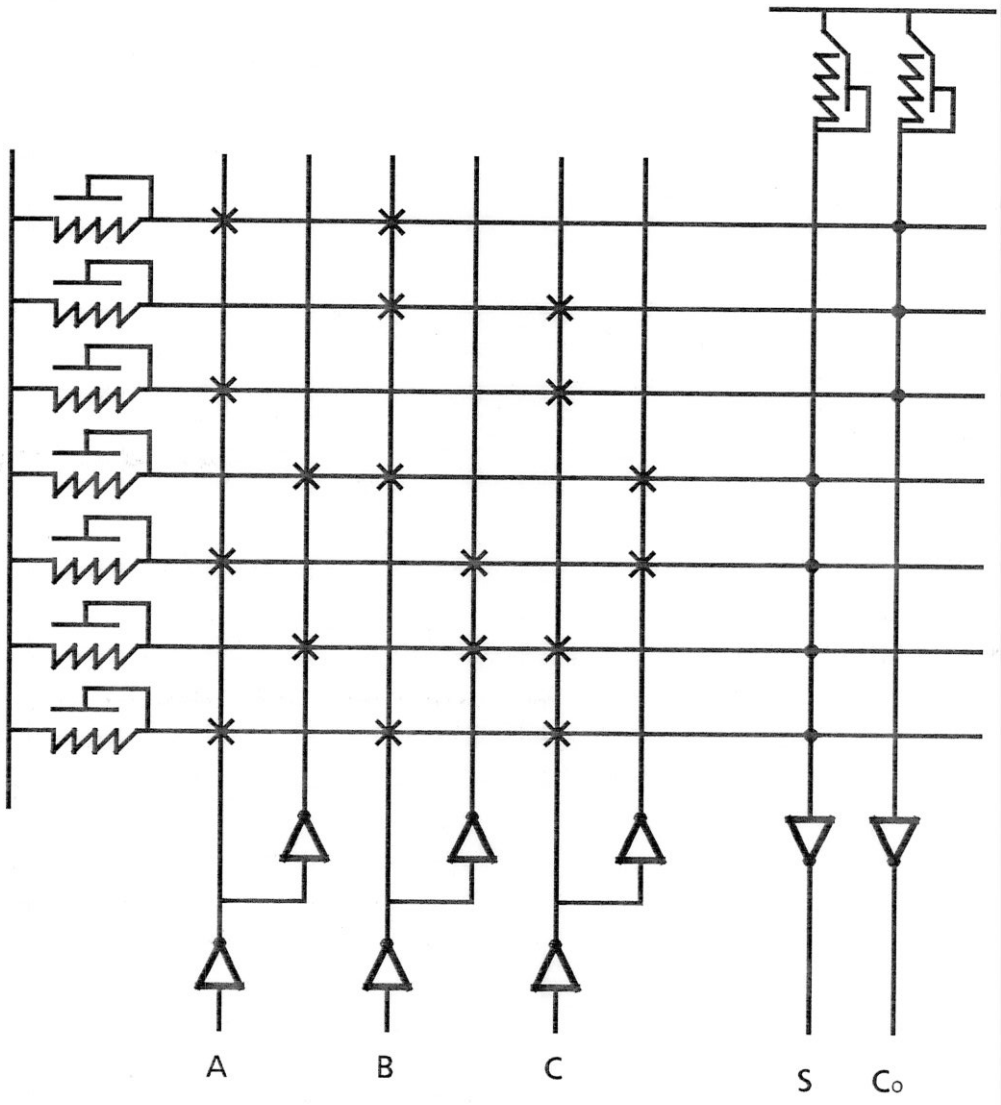
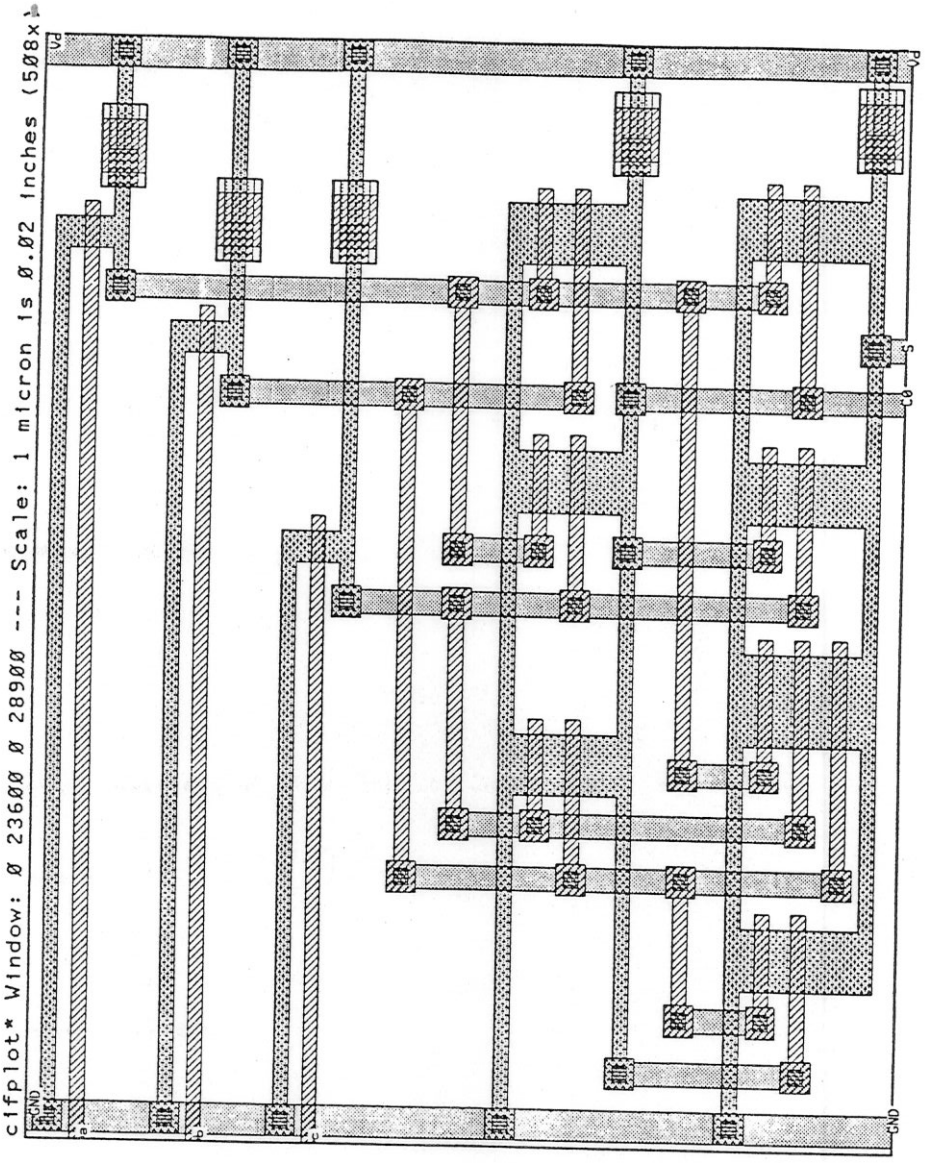


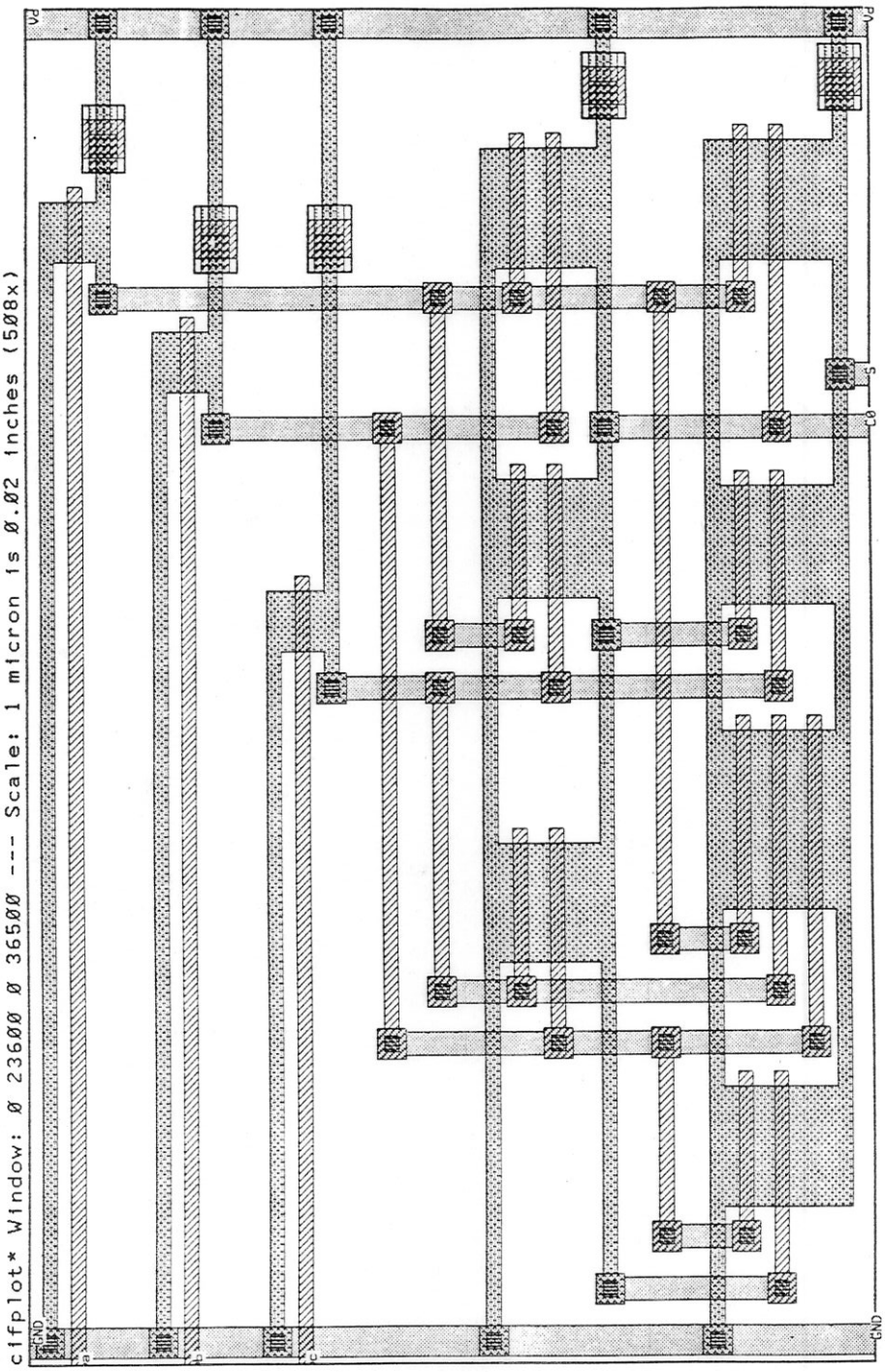

Figure 8 PLA - 1 Bit Full Adder




$$C_0 = \overline{\bar{A}\bar{B} + \bar{A}\bar{C} + \bar{B}\bar{C}}$$

$$S = \overline{C_0\bar{A} + C_0\bar{B} + C_0\bar{C} + \bar{A}\bar{B}\bar{C}}$$

Figure 9.a WA - 1 Bit Full Adder with Pullup Size 2




$$C_0 = \overline{\bar{A}\bar{B} + \bar{A}\bar{C} + \bar{B}\bar{C}}$$

$$S = \overline{C_0\bar{A} + C_0\bar{B} + C_0\bar{C} + \bar{A}\bar{B}\bar{C}}$$

Figure 9.b WA - 1 Bit Full Adder with Pullup Size 1

# APPENDIX C: An Example from Parse to Layout

Input: (A + C) (D + E) + $\underline{(A + C)}$ B

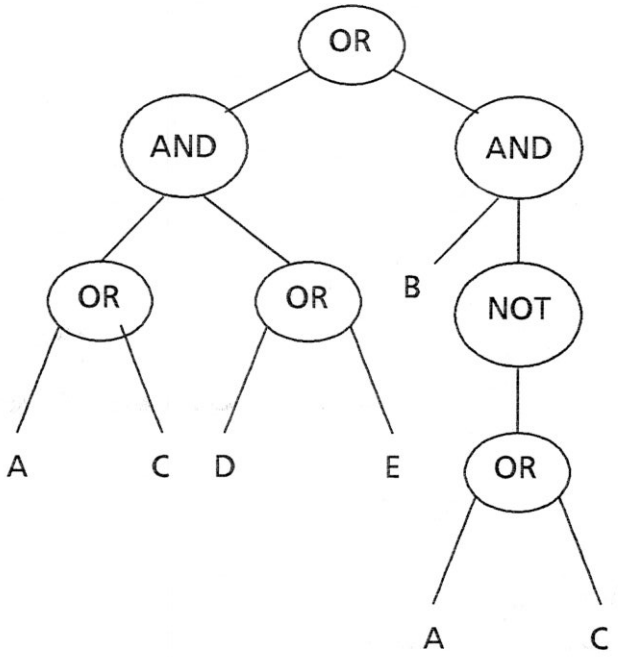



Fig. C1. Original parse

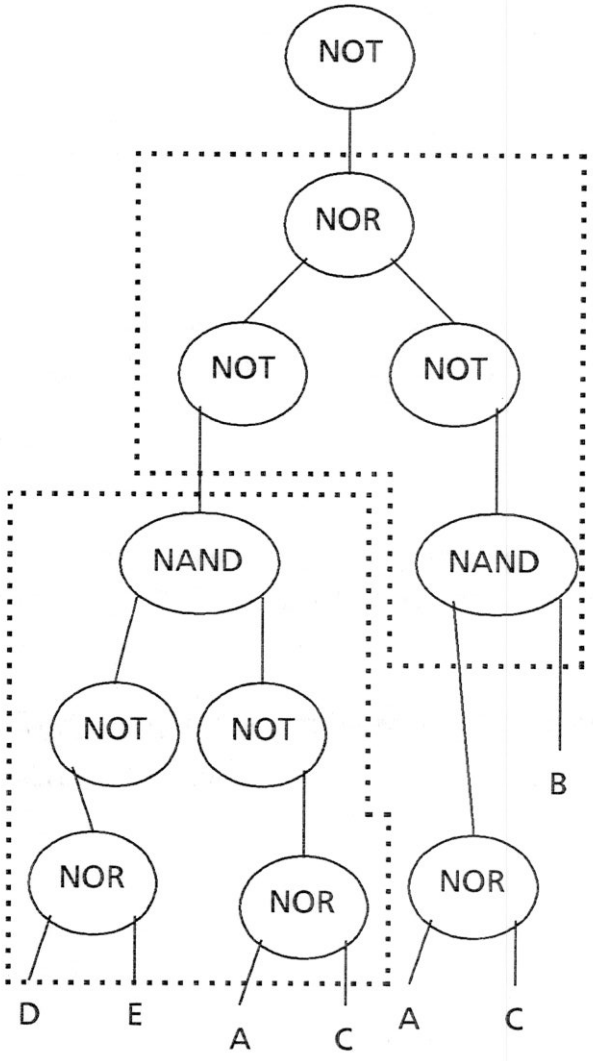



Fig. C2. Transformation to all inverting logic

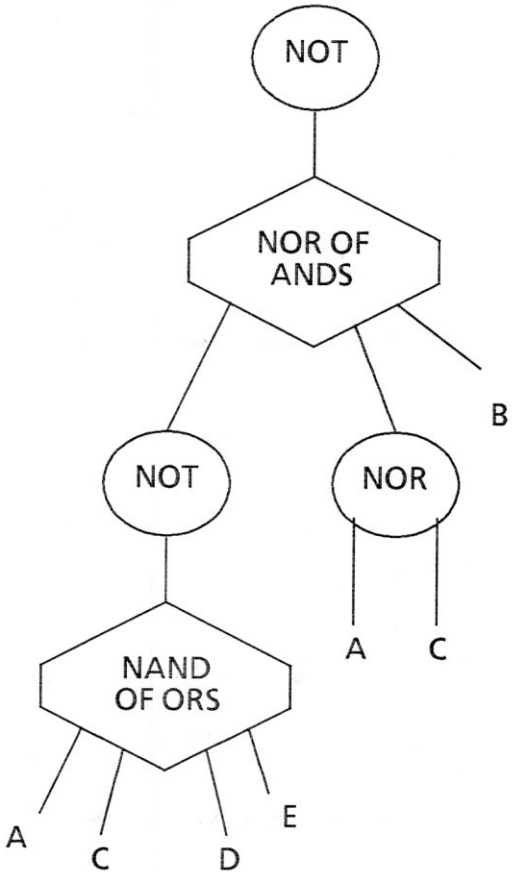



Fig. C3. A possible simplification
(see fig. B2)

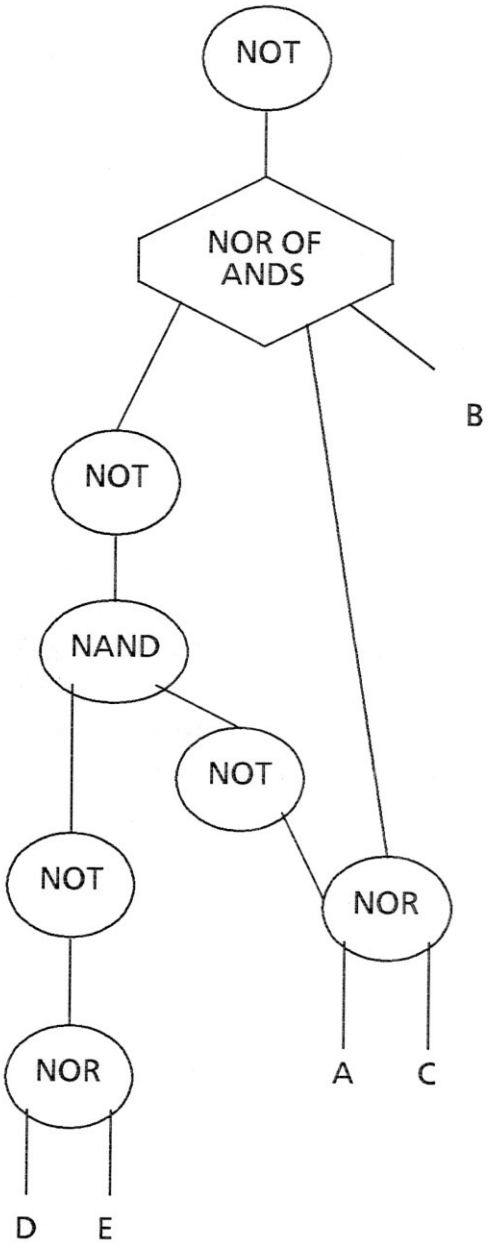



Fig. C4. Another possible simplification:
Input signals only go to one gate

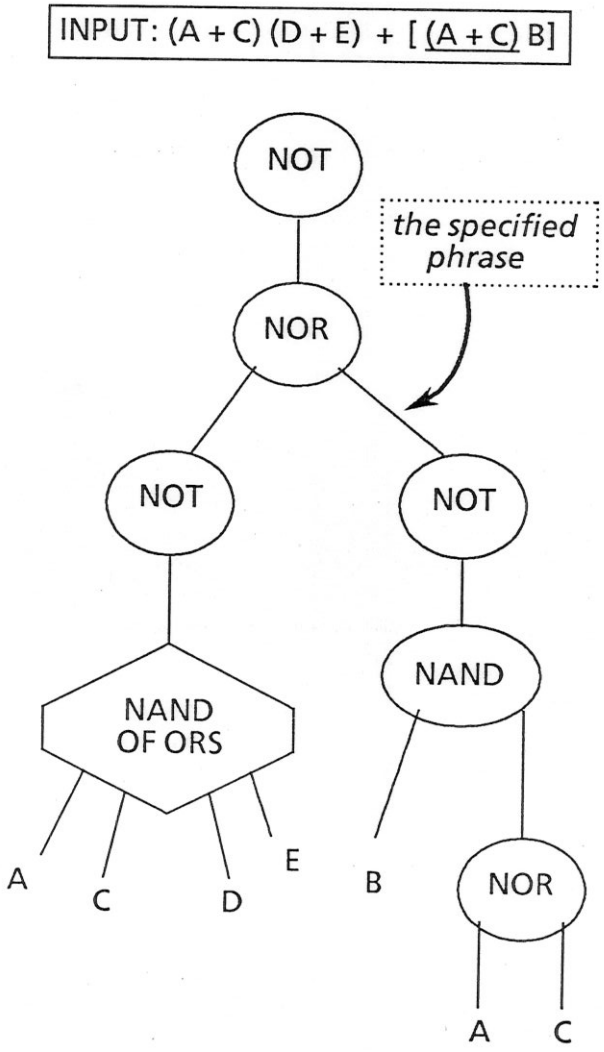



Fig. C5. Alternative if a phrase is specified (in brackets)

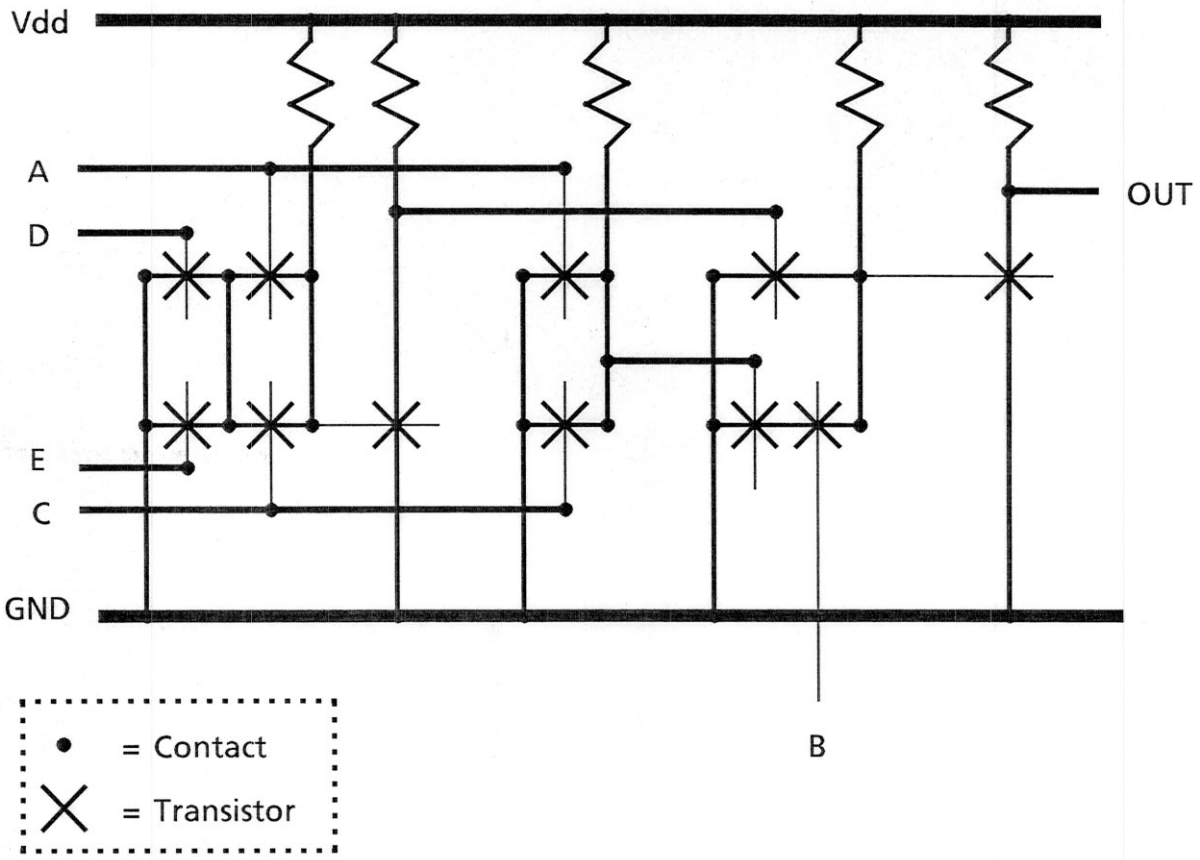



Fig. C6. Possible Weinberger array layout
for logic of Fig. C3.