A DESIGN SYSTEM FOR PLA-BASED DIGITAL CIRCUITS

G. De Micheli, M. Hofmann, A. R. Newton,

and A. L. M. Sangiovanni-Vincentelli

1. INTRODUCTION

The design of a digital system can be viewed as a sequence of transformations of design representations from one level of abstraction to another—from the behavioral, or functional, level to the detailed geometric layout level. At each new level, more *detail* is added to the design while both *functional integrity* and *performance* requirements are maintained.

Nowadays it is considered impractical to perform many of these transformations by hand because they are tedious, error prone, and very time consuming. Rather, one or more computer programs are used at

Advances in Computer-Aided Engineering Design, Volume 1, pages 285–364
Copyright © 1985 by JAI Press Inc.
All rights of reproduction in any form reserved.
ISBN: 0-89232-400-7

each step in the design process. While the inherent *mechanisms*, or algorithms used are the same in many cases, the particular *policy*, or *design method* used will determine the final relationship between each stage of the process and the overall efficiency of the design. For example, cell placement algorithms and channel routers are used in the gate-array, standard-cell, and macrocell design methods [Newt81a]. However, the final layout is quite different in each case and the particular objective functions and optimal algorithms for placement and routing often differ between the design styles [Souk81].

To maintain acceptable design time with ever-increasing integrated-circuit (IC) complexity, techniques for organizing circuit structure are used in almost all design methods. In particular, the use of *hierarchical* design techniques and *regular* structures play an important role here.

In this paper, the use of a particular class of regular structure—the programmable logic array (PLA)—is described. While such regular structures can be implemented in a very straightforward manner, such implementations are often far from optimal. In this paper, we describe a number of techniques for improving the efficiency of combinational logic designs implemented using PLAs as well as the automation of the entire PLA design process.

1.1. Classification of Regular Structures

Regular structures, or cellular arrays [Torn72], include circuits whose mask layout can be represented by a one- or two-dimensional array of possibly dissimilar cells, or *tiles* [Mayo84]. Such structures include both homogeneous arrays, such as RAM, register files, logic slices, and nonhomogeneous arrays, such as ROM, Weinberger array [Wein67], gate matrix [Kang83], storage logic arrays (SLAs) [Pati79], and the programmable logic array (PLA) [Proe76], which is the topic of this paper.

Regular arrays can be characterized by using a one- or two-dimensional array of cells of the form shown in Figure 1.1. To simplify the

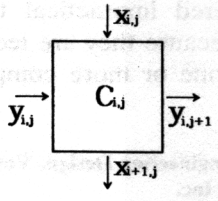


Figure 1.1. Two-input, two-output array cell.

description here, we have restricted the cells to those whose inputs and outputs are aligned with the x and y axes. Note that a particular input or output may also be absent for a specific use of the cell. For example, a one-dimensional array of cells can be implemented by omitting the $x_{i+1,j}$ output for a single row of cells. The cells need not all be of the same size. In that case, pitch alignment of the inputs and outputs is achieved either within the cells, by stretching components until the inputs and outputs of all cells lie on a common grid, or pitch alignment is achieved in the interconnect between adjacent cells, by jogging the interconnect where necessary to achieve a connection [Joha79].

In general, all possible orthogonal transformations of the cell may be used. These eight permutations are illustrated in Figure 1.2 below.

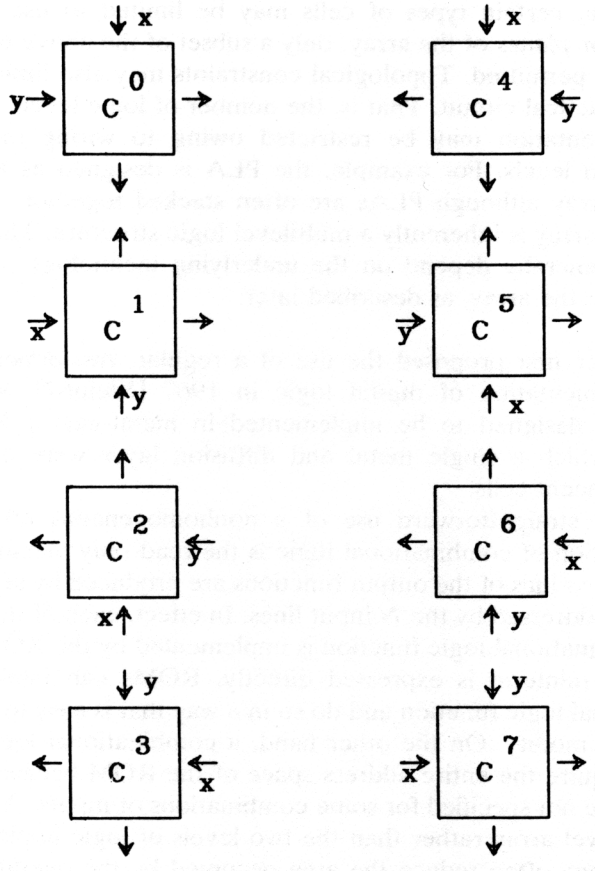


Figure 1.2. The eight possible orthogonal permutations of cell C.

Each of the homogeneous and nonhomogeneous structures listed above can be described in terms of a cellular array of layout titles which implement specific logic functions. It is the set of constraints applied to the use of this general structure that determines the classification of the array. These constraints fall into the following categories:

(i) *Functional constraints*, which limit the type of logic functions a cell may implement. In general these may be simple or complex, combinational or sequential functions. In the PLA, only simple, combinational logic functions are used. The SLA, on the other hand, relaxes the latter constraint by permitting the use of memory elements within the array itself. While adding flexibility, this feature of the SLA also makes synthesis of optimal designs more complex.

(ii) *Topological constraints*, which limit the arrangement of the cells. For example, certain types of cells may be limited to use in specific areas, or *planes* of the array; only a subset of the above orientations may be permitted. Topological constraints may also limit the depth of a practical circuit. That is, the number of logic levels used in the implementation may be restricted owing to wiring inefficiencies between levels. For example, the PLA is designed as a two-level logic array although PLAs are often stacked together. The Weinberger array is inherently a multilevel logic structure. These restrictions generally depend on the underlying technology used to implement the array, as described later.

Weinberger first proposed the use of a regular, two-dimensional tile array implementation of digital logic in 1967 [Wein67]. Weinberger arrays were designed to be implemented in metal-gate *p*-MOS technology in which a single metal and diffusion layer were available to connect adjacent cells.

The most straightforward use of a nonhomogeneous array for the implementation of combinational logic is the read-only memory (ROM), in which the values of the output functions are produced by selecting one of 2^N cells addressed by the N input lines. In effect, each of the minterms of the combinational logic function is implemented by the ROM [Flei75]. Since each minterm is expressed directly, ROMs can implement any combinational logic function and do so in a way that is easy to implement and easy to modify. On the other hand, a combinational logic function may not require the entire address space of the ROM because the logic functions are not specified for some combinations of inputs. Also, the use of a multilevel array rather than the two levels of logic implemented by the ROM may often reduce the area occupied by the overall circuit as well as improve its performance.

These concepts are illustrated in the following example. Consider the carry-out function of a two-bit full adder [Torn72]:

$$F = x_1x_2x_2 + \overline{x_1}x_2x_3 + x_1\overline{x_2}x_3 + x_1x_2\overline{x_3}. \tag{1.1}$$

This function can be implemented in a ROM using cells of six types; the

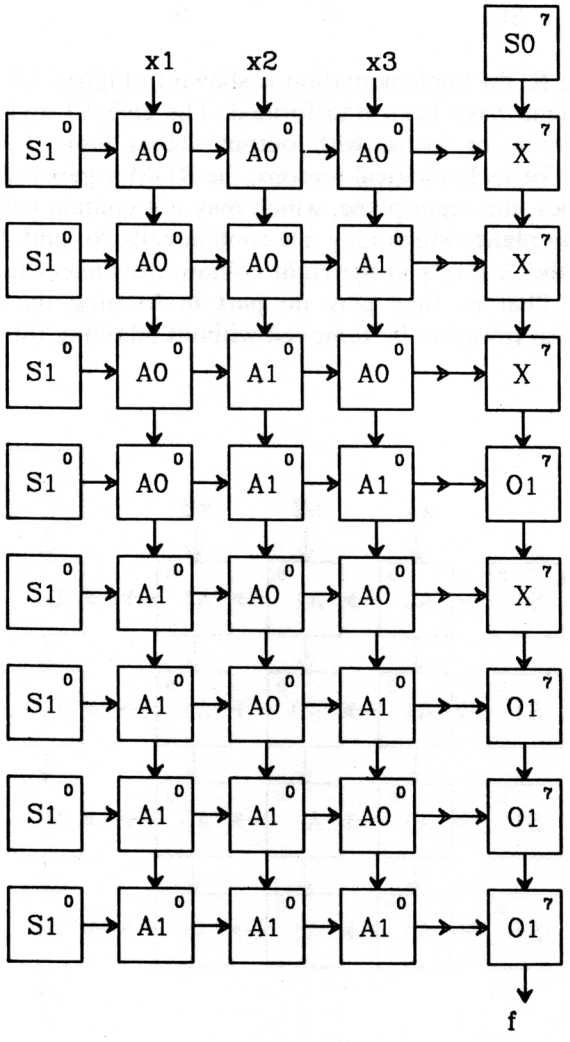


Figure 1.3. ROM implementation of the carry function.

invariant subscripts of Figure 1.1 have been dropped for clarity:

Symbol	y_{j+1}	x_{i+1}
A1	xy	x
A0	$\bar{x}y$	x
O1	$x+y$	x
X	y	x
S1	1	1
S0	0	0

The resulting ROM implementation is shown in Figure 1.3. The cells in the right column have been transformed. The cells S1 and S0 represent constant sources, or loads, with output values logic 1 and logic 0, respectively. For technological reasons, the ROM is generally partitioned into two planes, the *input* plane, which may not contain cells O1 and X, and the *output* plane, which may not contain cells A0 and A1.

Note that the X terms in the right column disconnect their row from the column. That is, they play no part in forming the output logic function. These rows can be removed without affecting the overall logic function.

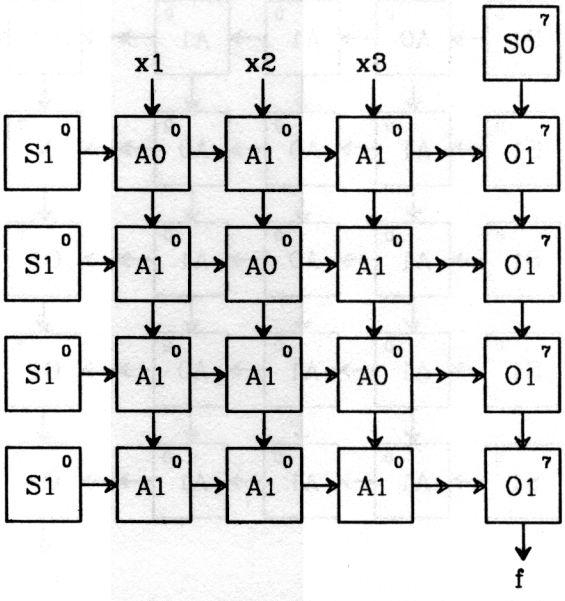


Figure 1.4. Simple PLA implementation of carry function.

A simple PLA implementation of the logic function can be obtained by removing these rows, as illustrated in Figure 1.4. For the PLA, the same restrictions described above for the ROM apply—the structure is partitioned into two planes, an input plane commonly called the "AND" plane, and an output plane commonly called the "OR" plane.

1.2. Automated Synthesis of PLA-Based Combinational Systems

The stages in the design of a PLA-based logic implementation are illustrated in Figure 1.5 [Newt81b]. Each of these steps is introduced below and described in detail in a later section.

1.2A. Logic Optimization

Logic optimization consists of mapping the functional description of the design into an optimal logic representation. For the carry function example, rather than implementing all the minterms of F in the input plane, it is possible to combine rows and implement only the prime implicants of F:

$$F = x_1x_2 + x_2x_3 + x_1x_3. \tag{1.2}$$

Each prime implicant is implemented by using a single row of cells. The resulting implementation is shown in Figure 1.6 and uses one fewer row. Note that cells of type X (crossover) are now permitted in both planes of the array. The process of transforming a logic function into one that requires the minimum number of rows to implement is called *functional optimization* and is generally very complex. It is described in detail in Section 3.

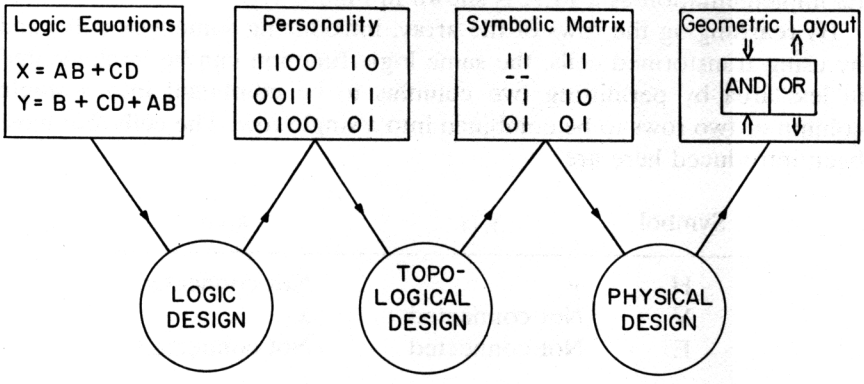


Figure 1.5. Steps in computer-aided PLA synthesis.

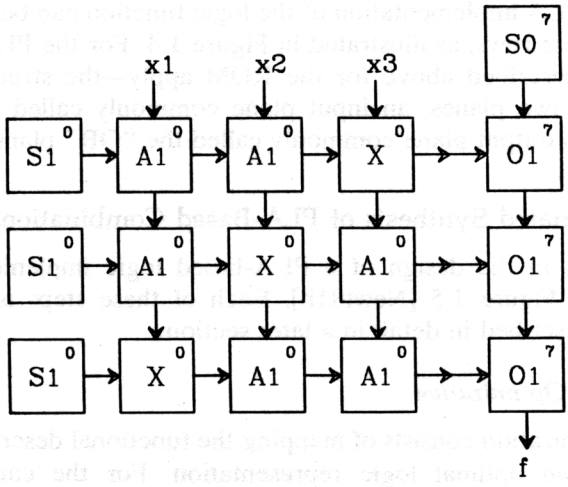


Figure 1.6. Reduced PLA implementation of carry function.

1.2B. Topological Optimization

Until now, a particular location of inputs and outputs to the array has been assumed. Also, specific directions for signal flow have been chosen. In the topological optimization process, these constraints are relaxed. Rather than use the simple example, a more complex function will be used to illustrate the topological optimization process known as *folding*. Consider the following minimal six-input, four-output logic function:

$$F1 = x_3x_6 \qquad F2 = x_2x_4 + x_1x_5 \qquad F3 = x_1 \qquad F4 = x_1x_6 + \overline{x_6}. \quad (1.3)$$

Its implementation as a PLA is shown in Figure 1.7.

By rearranging the rows of the array, introducing some new cells, and by using transformed cells, the same logic function can be implemented in less area by permitting two columns to be combined into a single column or two rows to be combined into a single row. The cells that have been introduced here are

Symbol	y_{j+1}	x_{i+1}
H	y	Not connected
V	Not connected	x
E	Not connected	Not connected

This process is called simple row folding or simple column folding. A

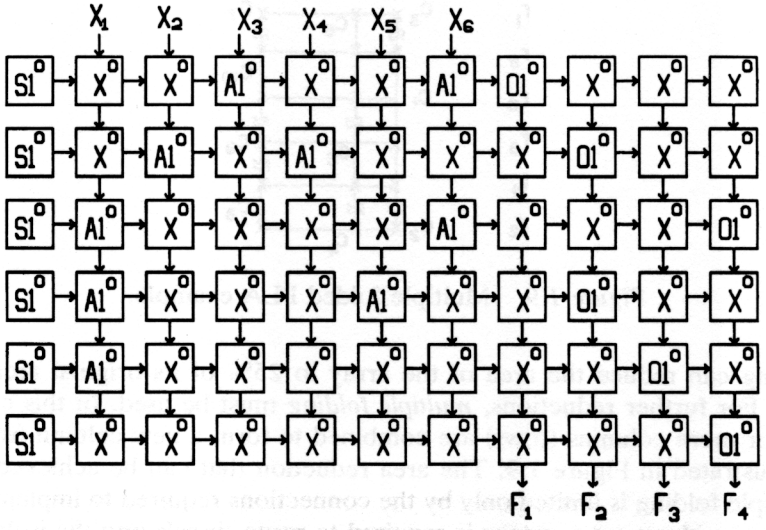


Figure 1.7. Six-input, four-output PLA example.

simple column-folded version of Figure 1.7 is shown in Figure 1.8(a). In Figure 1.8(b) a mixed row- and column-folded version of the logic is shown. Note that the integrity of separate input and output planes has been maintained, although in this case the output plane has been split into two parts forming an OR–AND–OR array. In a similar fashion, the input plane could have been split to form an AND–OR–AND array. It is assumed that S1 (for the AND plane) and S0 (for the OR plane) are available implicitly, which is usually the case for simple folding. The folding process has reduced the area of the array. However, simple

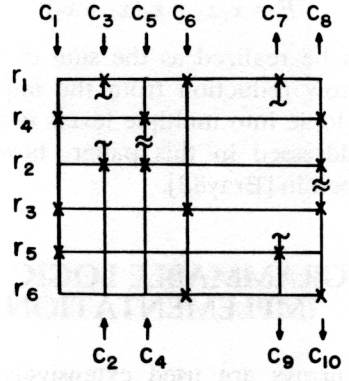


Figure 1.8. Simple column folded PLA.

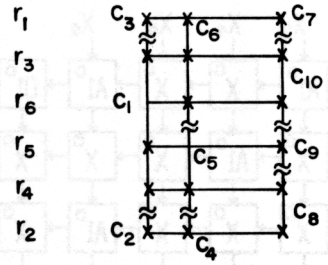


Figure 1.9. Multiple folded PLA example.

folding can reduce the area of the array to 25% of its original area at best. For further reductions, *multiple folding* must be used. In this case, two or more columns (rows) are combined to form a new column (row), as illustrated in Figure 1.9. The area reduction that can be achieved by multiple folding is limited only by the connections required to implement the logic. However, a means is required to route signals into the isolated inputs, such as x_1, and to obtain the isolated outputs, such as $F3$ and $F4$. Either a pitch increase for the additional signal or an additional layer of interconnect is required.

As is evident from this simple example, both simple and multiple folding become quite complex. Computer algorithms for both processes are described in Section 3.

1.2C. Multilevel Decomposition

Further reduction in area can often be achieved by using more than two levels of logic to implement the function. If Eq. (1.1) is rewritten in the form

$$F = x_1x_2 + x_3(x_1 + x_2) \tag{1.4}$$

then the function can be realized as the sum of two prime implicants, resulting in another row reduction from the implementation. Optimal decomposition of the logic into multiple levels is also a tedious process. That topic is not addressed in this paper; however, some powerful techniques are described in [Bray82].

2. PROGRAMMABLE LOGIC ARRAY IMPLEMENTATION

Programmable logic arrays are used extensively in integrated-circuit design. In some cases a PLA, or a PLA-based FSM, occupies an entire

chip [Wood79], e.g., code converters and digital controllers. Recently, programmable logic arrays emerged as a new building block for VLSI circuit design [Mead80]. For example, PLAs are often used to implement the instruction decoder of a microprocessor.

The details of PLA design depend heavily on the particular integrated-circuit design method in use. PLAs that are built in gate-array chips have to be designed so that they fit into a given structure. Therefore topological design is very important in obtaining compact arrays of a given shape. On the other hand, in VLSI custom and macrocell design, the PLA shape is not the major design constraint. However, PLAs must interact with other functional building blocks, and a primary objective of topological design is easing the connection of the PLA to other subcircuits. Other objectives include ease of engineering changes (addition of new logic equations, modification of existing equations) and ease of testing of the final PLA.

Rather than use the general cell array described above, a simplified form will be used for representing a PLA structure, or *personality*. In fact, this character-based symbolic PLA personality format is that used by the computer programs described later. A single character is used to replace the cells used earlier and the connections between cells are assumed. A typical PLA personality matrix is shown in Figure 2.1. For our purposes, the correspondences between the cells used above and the single-character symbols are as follows:

Symbol	Character
$A0^0$	0
$A1^0$	1
X^0	*
$O1^7$	1
X^7	0

Note that in this notation, the separation of input and output planes is necessary to avoid ambiguity. The corresponding PLA implementation is

```
**1**0  1000
*1*0**  0100
1****0  0001
1***1*  0100
0*****  0010
*****1  0001
```

Figure 2.1. A PLA personality matrix.

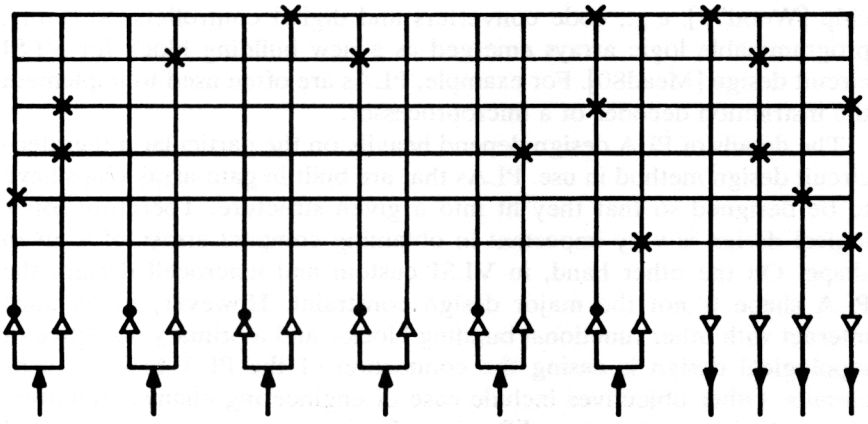


Figure 2.2. PLA schematic implementation.

sketched in Figure 2.2. Every scalar input of the logic function corresponds to a pair of columns in the left part of the physical array. Every implicant, or equivalently every row of the personality matrix corresponds to a row of the physical array. In particular, every implicant input part, or equivalently every input personality row corresponds to a logic product of some inputs. Therefore PLA physical rows are termed *product-term rows* or more simply rows. Every output of the logic function corresponds to a column in the right part of the array. The implementation of a particular switching function is obtained by programming the PLA, i.e., by placing (or connecting) appropriate gates in the array in the input (output) column position specified by 1 or 0 (1). For historical reasons, the arrays are referred to as *AND-plane* and *OR-plane*, although physical implementations different from *sum-of-products* are very common.

It is useful at this point to show the equivalence of the different methods of PLA implementation. The canonical AND–OR form of an example function may be written

$$F = A\bar{B}D + A\bar{C} + A\bar{B}.$$

Using DeMorgan's theorem, we obtain

$$F = \overline{\overline{A\bar{B}D + A\bar{C} + A\bar{B}}}$$

$$= \overline{\overline{A\bar{B}D}\;\overline{A\bar{C}}\;\overline{A\bar{B}}}.$$

This last form could be mapped directly into a NAND–NAND implementation. Restating the example equation with another application

of DeMorgan's theorem, we have

$$F = \overline{\bar{A}+B+\bar{D}\ \bar{A}+C\ \bar{A}+B}$$

$$= \bar{A}+B+\bar{D}+\bar{A}+C+\bar{A}+B$$

which converts to the NOR–NOR form:

$$\bar{F} = \overline{\bar{A}+B+\bar{D}+\bar{A}+C+\bar{A}+B}.$$

Note that the direct implementation of the equation in NOR–NOR form requires inversion of input variables and function output.

The Karnaugh map of the carry function is shown in Figure 2.3 with three schematic implementations. The first is in canonical AND–OR form, the second form is NOR–NOR. Note the inversion at circuit inputs

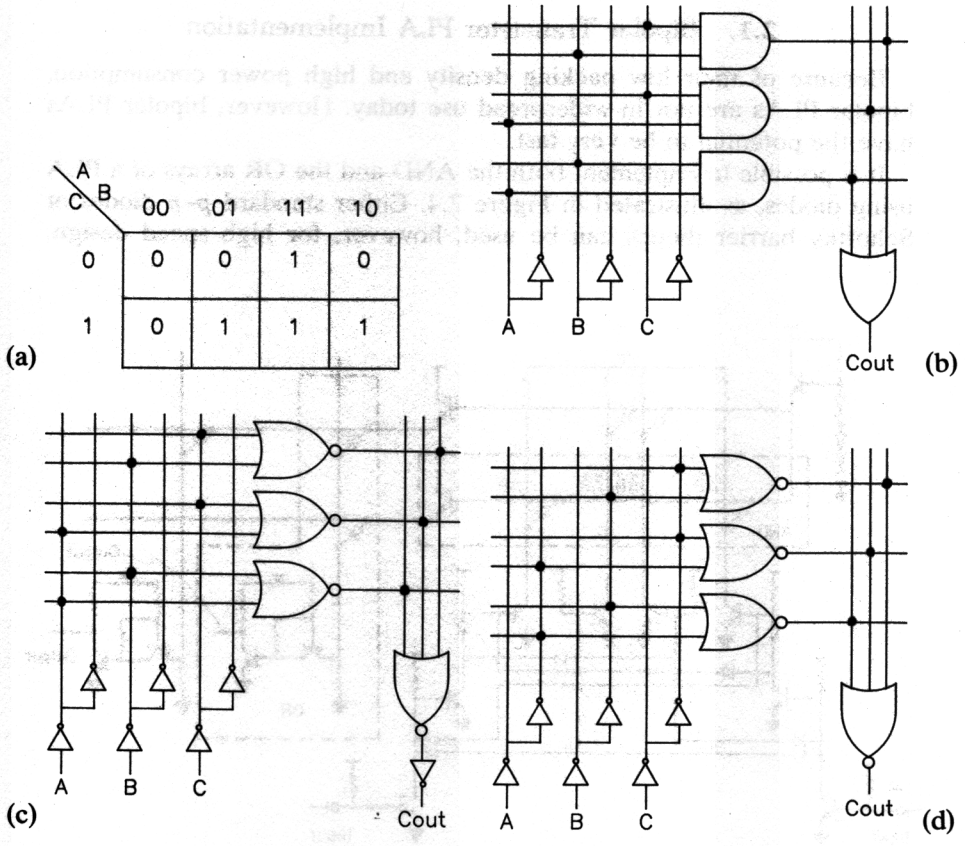


Figure 2.3. Three implementations of the carry function.

and output. The last circuit was produced by implementing the *zeroes* of the Karnaugh map, rather than the *ones*. The NOR–NOR implementation is the same as the second schematic, except that the output inverter has been removed. Buffers are often employed at circuit outputs for added drive capability, so saving the output inverter is not a major consideration. However, minimization of function zeroes may result in less logic and considerable area savings over logic-one minimization for some functions. For the carry function shown here this is not the case.

The key technological advantage of using a PLA in an integrated-circuit technology relies on the straightforward mapping between the symbolic representation (personality) and its physical implementation. Moreover, PLAs are compatible with different technologies and modes of operation, as shown in the following examples.

2.1. Bipolar Transistor PLA Implementation

Because of their low packing density and high power consumption, bipolar PLAs are not in widespread use today. However, bipolar PLAs have the potential to be very fast.

It is possible to implement both the AND and the OR arrays of a PLA using diodes, as illustrated in Figure 2.4. Either standard p–n diodes or Schottky barrier diodes can be used, however, for high speed design,

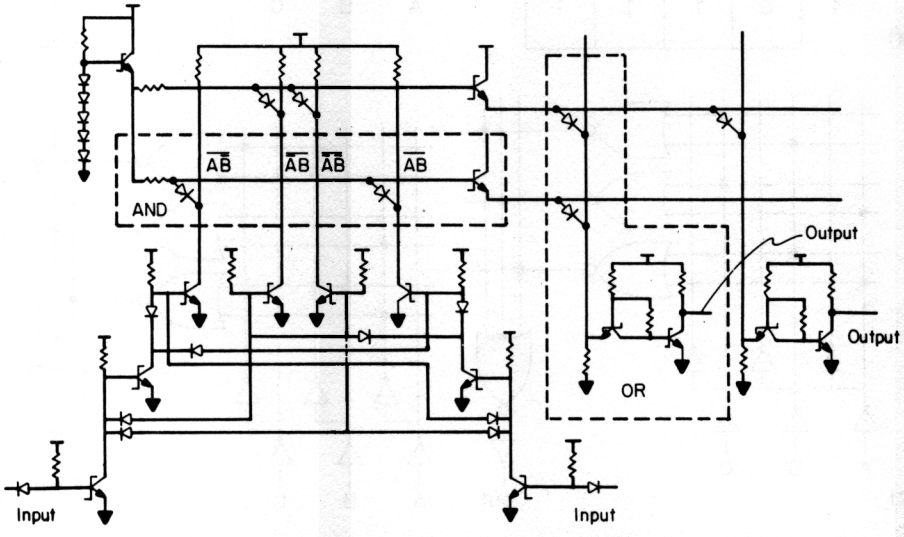


Figure 2.4. Diode array PLA.

Schottky diodes are preferred. By using two layers of metal it is possible to produce very compact diode-based PLAs. Standard diode PLAs do suffer from current loading problems imposed by the previous stage. That is, as the number of product terms or outputs in a PLA grows large, diode methods become impractical. If drivers with high drive capability could be designed (both input drivers and interarray drivers) or if the parasitics that sink drive current could be reduced, large diode PLAs could become practical.

A variety of BJT methods exist for implementing PLAs. Fang [Fang83] has implemented an all-*n*-*p*-*n* PLA which exhibits a 6-ns delay using a derivative of current mode logic (CML) [Coop80]. The logic is nonsaturating and a PLA containing 200 product terms, 30 inputs, and 20 outputs consumes 800 mW of static power. The size of the PLA and associated pads is approximately 5×2.1 mm. This design realizes the PLA in (positive) NOR–NOR form. Figure 2.5 shows the schematic diagram of an all-*n*-*p*-*n* design. Note the presence of interarray drivers, necessary in this case owing to the current loading of the array. Fang employed ECL buffers because of their high drive capability and relative simplicity. He also employed an on-chip compensation circuit which

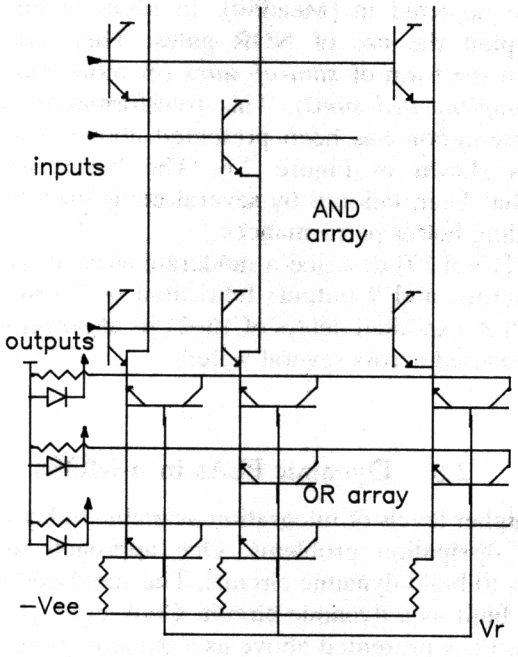


Figure 2.5. All-*n*-*p*-*n* PLA array.

allowed him to adjust input voltage swings on the PLA arrays. Voltage swing for the AND plane (with 30 inputs) was 450 mV, for the OR plane (with 200 inputs) 530 mV. Voltage swing is sensitive to fanin for such a design.

Another design also using emitter follower arrays to build a NOR–NOR PLA yielded a delay of 4 ns [Stil83]. The structure has 22 inputs, 40 product terms, and 12 outputs. Static power dissipation was 1 W in this case.

Another scheme for realizing bipolar PLAs is to use complementary emitter follower logic (CEFL). This approach requires the use of vertical *p-n-p* transistors as well as *n-p-n* devices. Since few process lines are able to fabricate high-quality vertical *p-n-p* bipolar transistors, because of the extra processing steps required, large PLAs have not been fabricated using this style of design. A study [Fang83] has shown, however, that if a PLA similar to the CML-style circuit just described could be built in CEFL it would exhibit similar characteristics.

2.2. NOR–NOR *n*-MOS PLA Implementation

The most common PLA implementation in VLSI MOS circuits has the basic structure depicted in [Mead80]. In MOS technology it is convenient to exploit the use of NOR gates. Therefore the PLAs are implemented in the form of *sum-of-sums* (or more exactly *complemented-sum-of-complemented-sums*). This transformation from a *sum-of-products* representation has been presented above. An *n*-MOS NOR–NOR PLA is shown in Figure 2.6. The basic *n*-MOS PLA implementation has been tailored by several companies to different MOS processes yielding better performances.

Cook et al. [Cook79] describe a moderate-sized PLA with 20 inputs, 105 product terms, and 9 outputs fabricated in 1-μm MOSFET technology. This PLA exhibited delays of 13–21 ns at power consumptions of 15–24 mW measured across several wafers.

2.3. Dynamic PLAs in *n*-MOSs

Often, the higher levels of integration available today in *n*-MOSs have led to power dissipation problems. One approach to reduce power consumption is to build dynamic circuits. The standard NOR–NOR PLA design can be built as a dynamic circuit. Cook et al. [Cook79] describe the same static PLA presented above as a dynamic design. They employ a four-phase clock:

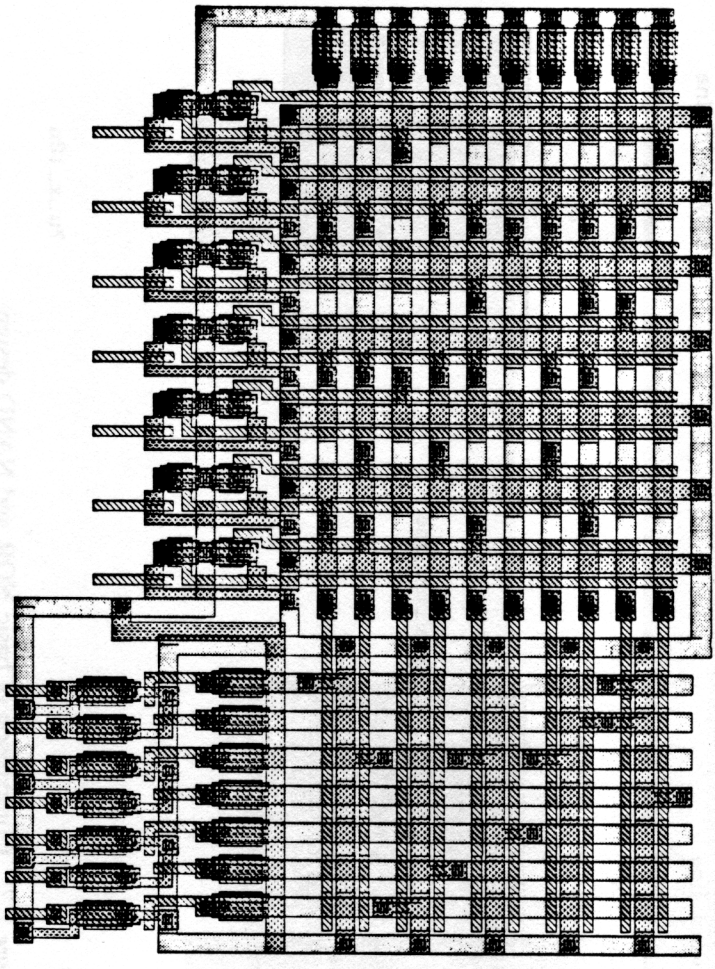


Figure 2.6. NOR–NOR *n*-MOS PLA implementation.

$\phi 1$ precharges the AND array
$\phi 2$ drives the AND array inputs and precharges the OR array
$\phi 3$ drives the inputs to the OR array
$\phi 4$ drives the output of the OR array

The circuitry and the clocking scheme used in the PLA ensure that no data race through the PLA can occur.

In 1-μm MOSFET technology it was possible to run the PLA with a 56-ns cycle time. This meant that each clock phase was 7 ns with rise and

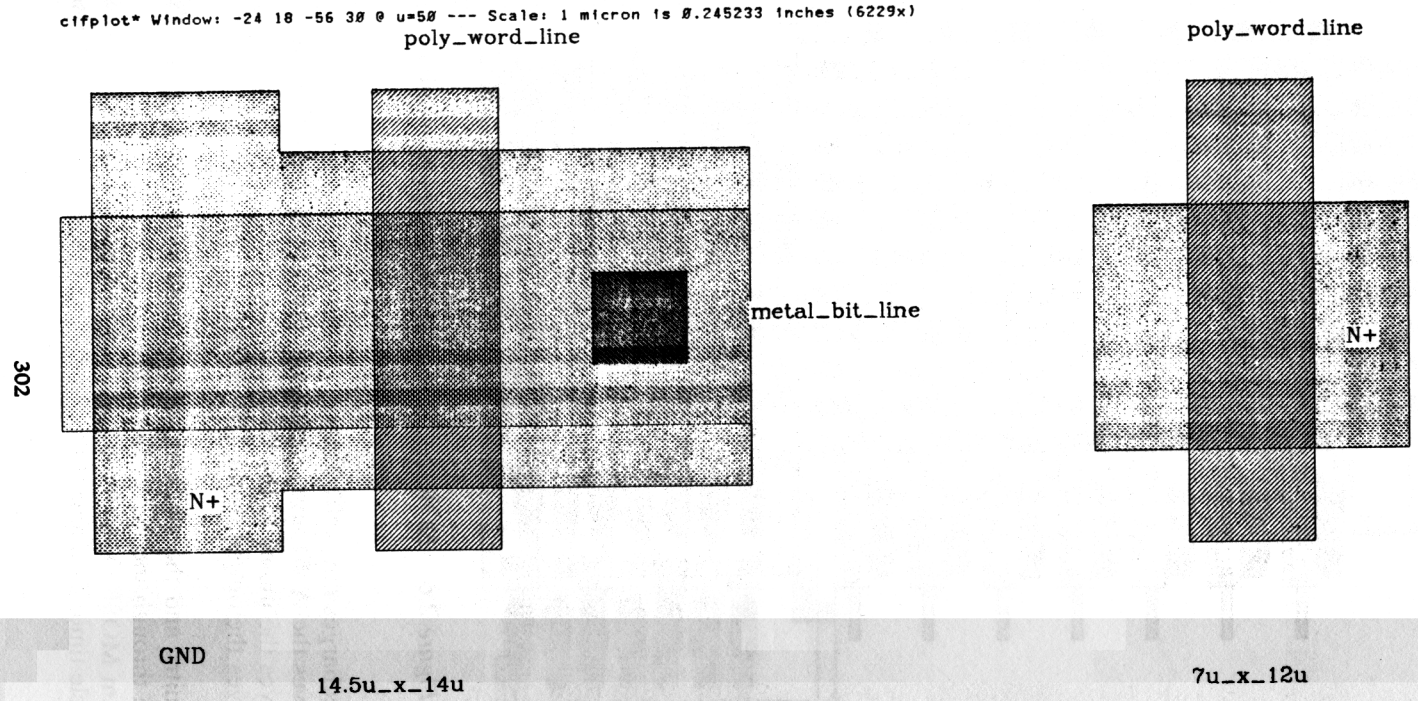


Figure 2.7. Comparison of basic NOR and NAND design.

fall times around 1 ns each. External clock schemes face capacitance problems at these clock rates and ON-chip clock schemes are preferred.

The dynamic design uses three power supplies and in addition to the ON-chip clock circuitry, routing of four independent clocks complicates this approach. Power consumption, however, at an equivalent stage delay of 60 ns is less than one-fourth of the static PLA implementation.

Although the only production MOS PLAs to data are of the NOR–NOR variety, it is possible to build compact NAND–NAND style PLAs. A stacked approach using a contactless cell has been fabricated [Lin81]. The approach is limited to PLAs of moderate size; the PLA described had 20 inputs, 20 product terms, and 20 output terms. By using an implant mask at those FET sites where no connection of input to product term or product term to output is desired, a compact cell can be designed. The FET site takes up less than one-third the area of conventional diffusion or contact-programmed cells. Figure 2.7 shows a comparison of the different cell designs.

The circuit is dynamic and uses five independent clocks, so the timing of the clocks is somewhat involved. The precharge scheme must avoid the charge redistribution problem that can occur in long NAND stacks.

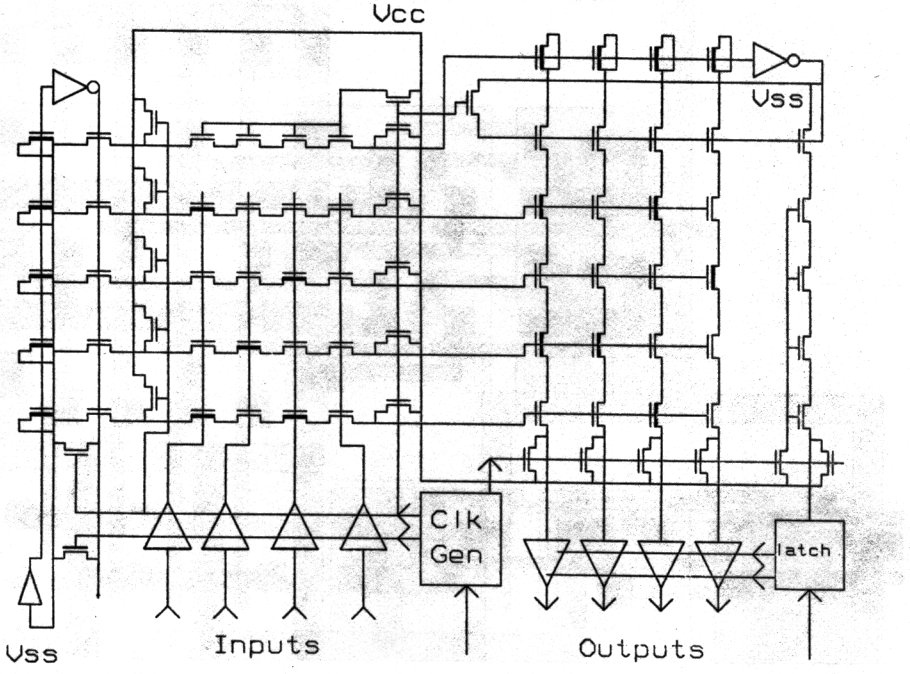


Figure 2.8. Simplified schematic and NAND–NAND PLA.

The test PLA was fabricated in a 4-μm technology and exhibited worst-case delays of around 150 ns with an external clock. Power dissipation for the PLA was 20 mW. Figure 2.8 shows a simplified schematic of such a NAND–NAND circuit.

2.4. CMOS PLAs

As processing expertise increased, there was a natural evolution to complementary technologies. Again, as with *n*-MOS circuits, both static and dynamic PLAs have found application.

A static PLA has the chief advantage that it is easy to design, requires no clocking scheme, and gives good speed. Static CMOS PLAs are often translations of static *n*-MOS designs. As with the standard *n*-MOS schemes, the generation of CMOS PLAs can be automated once a suitable cell library has been designed. Figure 2.9 shows a 3-μm *p*-well CMOS design produced by TPLA [Mayo84] in conjunction with the

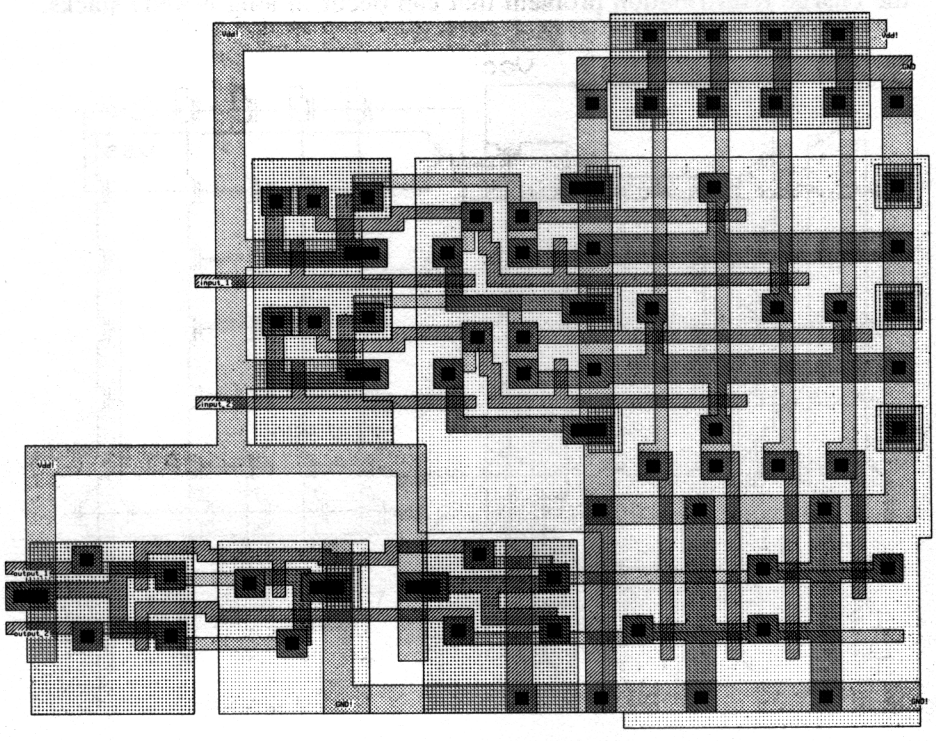


Figure 2.9. Static CMOS PLA from PANDA.

PANDA [Mah84] cell library. Such PLAs find application in medium-speed applications, where multiphase clocks are not available. The disadvantage of this approach is the relatively high power consumption compared to standard CMOS circuits because there is now a DC path to ground. These PLAs consume static power, as do their *n*-MOS counterparts.

In order to take advantage of the low power consumption potential offered by static CMOSs, complementary arrays would be required. This would necessitate not only twice as many active devices, but introduce a major interconnection problem. It is likely that PLA area would more than double. Currently such PLAs are not being used.

Dynamic CMOS PLAs retain the low device count of the static PLAs just mentioned, while at the same time requiring much less power. The Bellmac-32A microprocessor [Law82] utilizes dynamic PLAs in its control section. Figure 2.10 shows a schematic of a Bellmac PLA. Note the use of two clocks in this NOR–NOR implementation: $\phi 1$ controls the input plane, while $\phi 2$ controls the output plane. When $\phi 1$ is low the

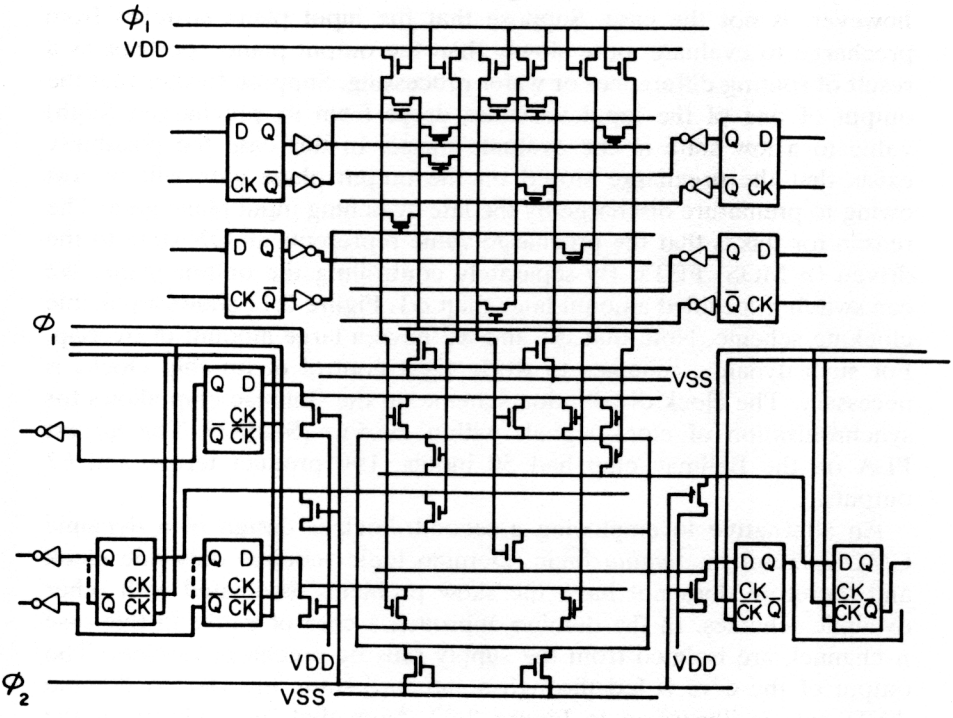


Figure 2.10. Schematic of Bellmac PLA.

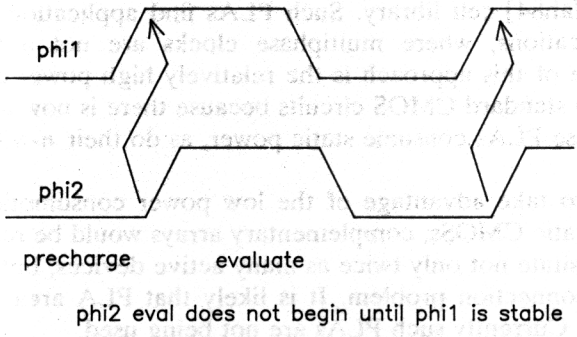


Figure 2.11. **Possible clocking scheme for dynamic PLA.**

input plane devices are in precharge mode; likewise for the output plane when $\phi 2$ is low. When each clock rises, the respective planes are in the evaluating phase.

It appears, at first glance, that only one clock should be required, that both planes could be precharged and evaluated in parallel. This, however, is not the case. Suppose that the input plane switches from precharge to evaluate more slowly than the output plane—perhaps as a result of routing differences or wafer processing. Suppose further that the output of one of the input variables drops from its precharged (high) value to a low state in the evaluate phase. In this case the possibility exists that the precharge stored on the output plane gate will be lost owing to premature discharge by the late-switching input plane gate. The reason for this is that the precharge value represents an ON state to the driven (n-MOS) FETs. By separately controlling the output plane, $\phi 2$ can switch a specified amount later than $\phi 1$. Figure 2.11 shows a possible clocking scheme. Note that $\phi 1$ and $\phi 2$ have a large amount of overlap. For such dynamic schemes to work, tight control of on-chip clocks is necessary. The clock distribution scheme on the Bellmac chip allows for synchronization of clock signals within ± 3.5 ns [Shoj82]. The largest PLA on the Bellmac chip had 50 inputs, 190 product terms, and 67 outputs.

An alternative to employing a second clock is design of a dynamic CMOS circuit in *domino* logic. Domino logic requires only one clock and therefore does not have the skew problems associated with other dynamic schemes. In the domino approach a core of gates, in this case n-channel, are isolated from the supply rails by precharge devices. The output of the core is fed through a standard static inverter. A domino AND gate is illustrated in Figure 2.12. Note that the addition of the

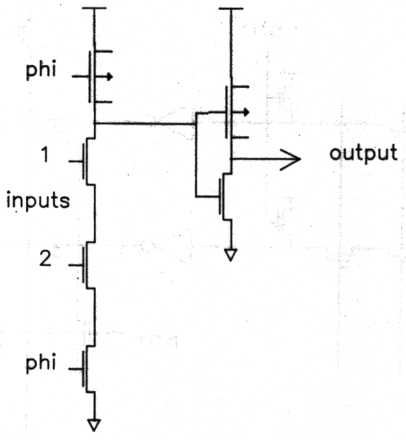


Figure 2.12. Domino AND gate.

inverter, a necessary component in the domino circuit, creates a noninverting logic function. The domino building blocks are AND and OR gates; therefore domino logic is not a logically complete family.

The name *domino* is derived from the way in which the circuits operate: *n*-channel core gates are precharged high; in the evaluate stage the output of each stage may remain high or it may make a single *high* → *low* transition. In the latter case it may cause a succeeding gate to alwo make a transition, which, may in turn, set off another gate, and so on. This process resembles toppling a chain of dominoes.

The addition of the inverter guarantees that the succeeding logic will be in the OFF state, even if there is some internal delay in switching two different sections of the circuitry. This holds if like types of cores are coupled. By alternating core type (e.g., *n*- and *p*-channel) the necessity of an inverter is removed. A pure *n*-channel schematic of a dynamic domino PLA is shown in Figure 2.13. Note that this is an AND–OR implementation. Another rendition of a domino PLA is shown in Figure 2.14. Here, instead of leaving out FETs in the AND plane, an implant step is used to shift their ON-threshold, effectively removing them from the circuit. This is the same approach used in the *n*-MOS NAND–NAND PLA described earlier. It results in a more compact AND plane, because no contacts are needed, but aggravates the stacking problem present in MOS AND gates. In fact, because of the significant ON-resistance of MOSFETs neither AND-plane can be used for large PLAs. Of course, the AND-plane can always be made the smaller of the two planes, since PLAs may be implemented in OR–AND form.

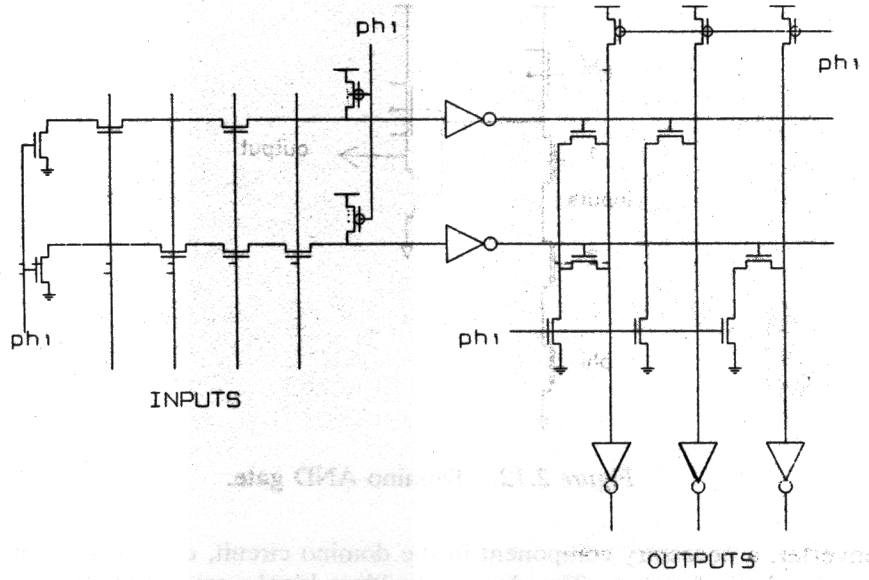


Figure 2.13. **Dynamic domino CMOS PLA.**

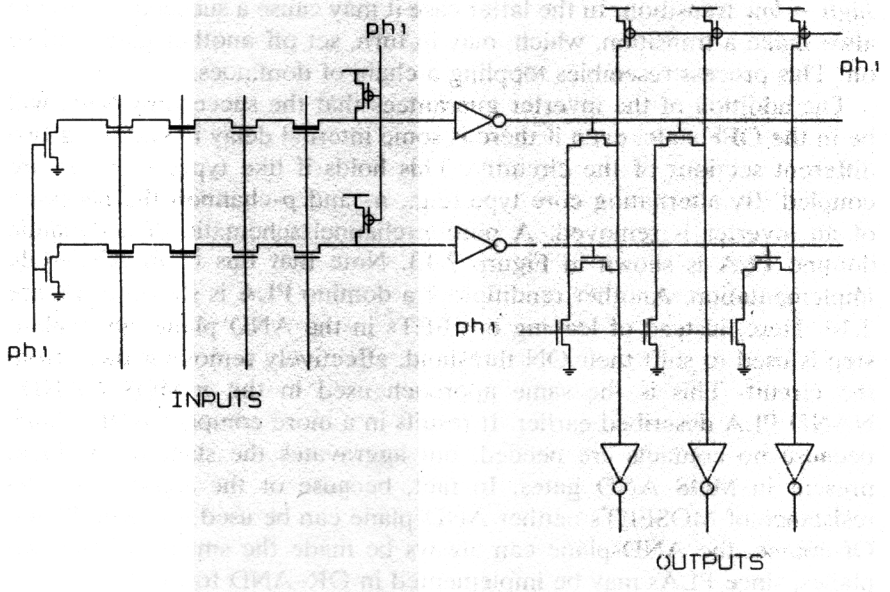


Figure 2.14. **Dynamic domino CMOS PLA with implanted devices.**

2.5. PLA Programming

It is worth mentioning that a choice of a PLA implementation technology depends on the design goals. However, PLA design is independent of the implementation technology used up to the definition of the device and interconnection locations. Therefore methods for PLA automated synthesis at the functional logic and topological level are fairly general and have a wide range of applications.

Integrated-circuit logic arrays come in two different flavors. In particular, PLA personalization can follow or precede fabrication. PLAs used as building blocks in large-scale designs fall into the second category.

In the first case PLAs are manufactured with a programmability feature [Sign79]. Field programmable logic arrays (FPLAs) are user programmed by applying an electric field in appropriate array locations, which induces a short (or open) circuit. Laser programmable logic arrays (LPLAs) are programmed by cutting appropriate connections after manufacturing.

In the second case PLAs are programmed before manufacturing by appropriate patterns on one or more masks. In particular, bipolar PLAs can be programmed by the contact mask. Mead suggested a diffusion mask program for n-MOS PLAs in [Mead80]. However, more complex mask programming techniques are generally used in order to eliminate unnecessary parasitics [Hofm80].

It is therefore inappropriate to refer to the second category of circuits as programmable logic arrays. A better term is programmed logic arrays. Unfortunately it is very common to refer to programmable logic arrays for both cases. In this paper only mask programmed logic arrays for VLSI design are considered. They are referred to as programmable logic arrays for the sake of uniformity.

3. PROGRAMMABLE LOGIC ARRAY MINIMIZATION

3.1. Introduction

In the 1950s, the early years of digital design, logic gates were expensive. It was therefore important to develop techniques that produced an implementation of a given logic function requiring a small number of devices (gates and basic components of gates such as diodes and resistors). At that time, the simplification of logic functions became a very active area of research. Karnaugh and Veitch maps were used to hand-minimize simple combinational two-level (two-stage) logic functions. Later on, more sophisticated techniques were introduced by Quine

and McCluskey [Mccl56] to obtain a two-level implementation of a given combinational logic function with the minimum number of gates. These techniques involved two steps:

1. Generation of all prime implicants.
2. Minimum covering of the set of all prime implicants.

Even though the generation of all prime implicants has been made more and more efficient (in particular, by using the disjoint sharp operation as implemented in the program MINI developed by Hong et al. at IBM [Hong74]), it can be shown that the number of prime implicants of a logic function with N input variables can be as large as $3^N/N$. In addition, the second step, in general implemented with a branch-and-bound technique, involves the solution of a minimum covering problem that has been proven to belong to the class of NP-complete problems, thus leaving almost no hope of finding an efficient exact algorithm, i.e., an algorithm for finding a minimum cover the running time of which is bounded by a polynomial in the number of elements in the covering problem. Note also that the number of elements in the covering problem may be proportional to the exponential of the number of input variables of the logic function. This implies that the use of these techniques is totally impractical even for medium-sized problems (10–15 variables).

In the late 1960s and early 1970s, the cost of logic gates was reduced so that logic minimization was not as essential as before and hand simplification was sufficient to produce satisfactory designs. Two-level logic minimization has again become an important tool in the design of PLAs. In fact, the minimization of product terms obtained by logic minimization has a direct impact on the area used by a PLA since each product term is implemented in a row of the PLA. Now, logic designs often involve logic functions with more than 30 inputs and outputs and with more than 100 product terms, thus ruling out the use of the classical approach to logic minimization. Several attempts have been made to minimize large logic functions sacrificing the optimality of the results.

Two basic approaches have been followed. The first one follows the structure of the classical logic minimization techniques in that the generation of all prime implicants is done first; however, instead of generating a minimum cover, a near-minimum cover is selected heuristically. It is obvious that this procedure still suffers from the possibility of generating a very large number of prime implicants.

The second approach combines the identification and the selection of (not necessarily prime) implicants. Several algorithms have been proposed in this group (e.g., [Roth58, Hong74, Rhyn77, Arev78, Brow81, Sima83]). We review briefly some of the most significant results.

Some of these authors (Rhyne et al., Arevalo and Bredeson) select a base minterm of the care-set of the logic function to be minimized and expand it until it is a prime, removing all minterms that are covered by this prime. The procedure is iterated until all the minterms of the care-set are covered. Rhyne et al., generate all prime implicants of the base minterms and select the ones that either are essential (i.e., they must be in any prime cover of the care-set of the logic function) or satisfy some heuristic criterion. Arevalo et al., do not generate all prime implicants covering the base minterms (in the worst case there are as many as $\alpha 2^n$, where α is a constant and n is the number of inputs). The algorithm of Rhyne generated better covers and the algorithm of Arevalo is much faster. Both these approaches have been coded, but tested only on small logic function (number of inputs less than 10).

A heuristic approach that has found wide application in the design of practical PLAs is based on the developments of Roth [Roth80], Hong et al. [Hong74], and Brown–Swoboda [Brow81]. The earliest and most successful of these led to MINI, developed at IBM in the middle 1970s. Later a heuristic minimization program called PRESTO was introduced by Brown [Brow81] following the ideas of Swoboda. Also a later version of MINI, called SPAM, containing partitioning to allow minimization of large PLAs was implemented by Kang [Kang81]. These approaches are centered around the following procedure: expand each implicant of the logic function to its maximum size and remove other implicants covered by the expanded implicant.

Note that here *implicants* are expanded and covered, thus eliminating a basic problem with the techniques previously mentioned: the generation and storage of all minterms. SHRINK, due to Roth, is a heuristic minimizer contained in the logic minimization program MIN370. It also contains an exact algorithm for generating all primes, but it is based on the disjoint sharp operation for generating all prime implicants efficiently. This followed by the generation of an irredundant cover (i.e., a cover such that no proper subset is also a cover of the given care-set of a logic function). In general, the procedures of expansion and elimination of implicants covered by the expanded implicant produce a cover made of prime implicants but not necessarily an irredundant one.

All of these basic strategies may lead to local minima that are far away from the global minimum. To avoid this problem, after the first expansion and removal of covered implicants, MINI reduces the remaining implicants to their smallest size while still maintaining the coverage of the logic function. Then it examines the implicants in pairs to "reshape" them by enlarging one and reducing the other by the same set of minterms. Then the expansion process is repeated and the entire procedure is iterated until no further reduction is obtained in the number of

product terms. This strategy makes the exploration of larger regions in the optimization space possible.

Our first choice for a logic minimization program was an implementation of the PRESTO algorithms, modified to include some features of MINI and some new heuristics for cube ordering. This program was called POP [Sima83]. The algorithms used in POP were then extensively modified following the theoretical results of Brayton et al. first implemented in ESPRESSO I and II [Bray82,84a,b]. An updated version of POP, written by P. Simanyi [Sima83], was the result of these changes. Recently a C version of ESPRESSO II has been coded by R. Rudell [Rude84]. In this section, we review the basic ideas of the PRESTO logic minimization algorithms, present the POP program, and mention briefly the current work on ESPRESSO II C.

3.2. Basic Concepts and Definitions

A few formal definitions are introduced here. We refer the reader to [Bray84b] for further details. A switching function f in n input variables and m output variables is a map

$$f:\{0,1\}^n \rightarrow Y \subseteq \{0,1,*\}^m$$

where $*$ is the don't care value.

A switching function is called *single output* (*multiple output*) if $m=1$ ($m \geq 1$). For each component of f, f_i, $i=1,2,\ldots,m$, we define the *ON-set* $X_i^{ON} \subseteq \{0,1\}^n$ *OFF-set* $X_i^{OFF} \subseteq \{0,1\}^n$, *don't care set* $X_i^{DC} \subseteq \{0,1\}^n$) as the set of input values such than $f_i(X_i^{ON})=1$ [$f_i(X_i^{OFF})=0$, $f_i(X_i^{DC})=*$].

A logical *implicant* is a pair of row vectors in $\{0,1,*\}^n$ and $\{0,1,*\}^m$. The former is called input part and the latter output part of the implicant. A *don't care* condition in the jth position of an input part means that the jth input signal may be either 1 or 0. A *don't care* condition in the ith position of an output part means that the ith output signal f_i is not considered for the corresponding input and can be either 1 or 0.

A *minterm* is an implicant with no don't cares (*) and only one 1 in the output part. The geometrical representation of the input part of a single-output implicant (minterm) is a cube (vertex) in the n-dimensional space. Hence an implicant is equivalent to the set of minterms corresponding to the vertices of the cube. This result can be generalized to the multiple-output case [Hong74, Kang81].

Two implicants with the same output part and differing only in one position, say j, of the input part can be *merged* into an implicant with the same output part and a don't care in position j of the input part. Similarly, two implicants with the same input part can be merged into an implicant with the same input part and output part entries which are the

logical sum of the entries in the corresponding positions. The inverse procedure is called *split*. An implicant can be transformed into a set of minterms by recursive splitting.

Let A and B be two implicants with the same dimensions. We say that A *covers* (*contains*) B if the set of minterms equivalent to A contains the set of minterms equivalent to B. We say that A *intersects* B if the set of minterms equivalent to A has a nonempty intersection with the set of minterms equivalent to B. Two implicants having an empty intersection are termed *disjoint*. The concepts of covering and intersection can easily be generalized to sets of implicants. Note that algebraic operations corresponding to covering and intersection can be defined for logical implicants. Therefore computing minterm sets is not needed in order to verify covering and/or intersection, with an effective saving of memory and computing time.

A set of implicants is said to be a *cover of a switching function* f if the set of the implicant input parts that have a 1 in the ith position of the output part contains the ON-set of f_i, X_i^{ON} and is disjoint from the OFF-set, X_i^{OFF}, for any output variable $i = 1, 2, \ldots, m$. A cover is said to be *irredundant* if no proper subset is a cover of the function f. A *minimal* cover is an irredundant cover with a minimal number of literals.

A PLA implementation of a switching function requires as many rows as the cardinality of the cover and as many active devices as literals. Therefore it is important ot find a minimum-cardinality cover with a minimum number of literals.

3.3. The Minimization Algorithm of PRESTO

The algorithm on which PRESTO is based minimizes the cardinality of the cover heuristically by means of an iterative improvement of an initial cover. The conceptual algorithm is described below in Pidgin C.

PRESTO Minimization Algorithm

```
presto
{
  squish;
  repeat {
    expand;
    reduce;
    squish;
  }
  until ( no change in the cover of f );
  merge;
  squish;
}
```

The *squish* routine deletes all the implicants having only 0's in the output part. These implicants do not contribute to the cover of the function.

The *expand* routine processes each implicant of the cover and replaces the cares of the input part with *don't cares*, provided that the expanded implicant is still covered by $X_i^{ON} \cup X_i^{DC}$ for every appropriate output variable $i = 1, 2, \ldots, m$. A key role in the expansion process is played by the *cover* routine, which returns 1 if an expanded implicant is covered by $X_i^{ON} \cup X_i^{DC}$ and 0 otherwise. The details of *cover* will be described later.

The *reduce* routine tries to reduce as many output literals as possible to zero. It takes each output literal in each implicant in order, forms a reduced implicant with only the output literal being considered, and chicks if the reduced implicant is covered by $X_i^{ON} \cup X_i^{DC}$. If the test is satisfied, then the literal can be eliminated from the original implicant since the vertices covered by this implicant in that output space are covered by other implicants. If all the output literals of an implicant are removed, the *squish* routine will remove the implicant and decrease the cardinality of the cover. This mechanism ensures that when PRESTO stops the cover obtained is irredundant, i.e., the no subset of the cover is also a cover of the given logic function.

The *merge* routine merges two implicants with the same input part, and hence reduces the cardinality of the cover.

The *cover* routine uses a tree search similar to the algorithm proposed by Morreale [Morr70] to check if implicant P is contained by the cover C [Brow81]. This procedure uses a dummy implicant Q, which is set equal to P at the beginning. A stack is used and initialized empty. At each step of the recursion, it is checked whether there exists a cube in the cover C which contains the cube Q. If this is the case, a new cube is popped from the stack. If the stack is empty, P is covered by C. If Q has empty intersection with the cubes of C, then P is obviously not covered by C. If Q is not contained in any cube of C but has nonempty intersection with the cubes of C, then an input variable which is a *don't care* is selected and Q is split into two cubes which are placed on top of the stack. The steps of the coverage tests are described below; splitting corresponds to partitioning the minterm set into two equal groups. The worst case of the tree method is to split P down to its individual minterms. In this case, the complexity of the procedure will be exponential in the number of *don't cares*.

A heuristic splitting strategy, called smart splitting, is reported in [Brow81]. This strategy tries to answer the covering question as soon as possible in the tree search and hence to speed up the process.

The quality of the final result, i.e., the number of product terms and literals, is obviously dependent on the order in which the cubes and variables are taken into account. The sequence in which the cubes are

```
Cover Procedure
cover
{
  Q = P;
  stack = empty;

  repeat {
    while( Q is covered by a cube in C ) {
      if( the stack is empty ) return( COVERED );
      else pop a new Q off the stack;
    }

    if( Q ∩ C = ∅ ) return( NOT_COVERED );
    else {
      select a don't care of Q;
      split Q into Q1 and Q2;
      Q = Q1;
      push Q2 onto the stack;
    }
  }
}
```

processed as well as the sequence in which the logic variables are expanded is determined in PRESTO in a very simple way: the cubes are ordered according to the input given by the user and the variables are expanded from left to right. The outputs are also reduced from left to right.

As pointed out above, it is possible to prove that the logic cover obtained by PRESTO is an irredundant cover. However, it may not be a prime cover, since the outputs are never raised but always lowered. The lowering strategy is effective in reducing the number of output literals. Note also that PRESTO, contrary to MINI, does not use any technique to try to move away from local minima.

3.4. The POP Program

PRESTO obtained very good results in terms of final cover cardinality and speed of execution. The initial version of the POP program was developed based on the PRESTO cover check but using a modified set of cube ordering heuristics. The largest cubes are expanded first since they have a better chance of covering other cubes and thus allowing the *reduce* and *squish* procedures to remove them. The literals are expanded either from right to left (as in PRESTO) or according to the number of

cubes that the expanded cube, if successfully expanded in that coordinate, covers. However, the comparisons with other existing techniques were not extensive and there were questions concerning the efficiency of the implementations used for the comparison as well as the class of PLAs minimized with the various techniques ESPRESSO I [Bray82] was written to make a fair comparison of the algorithms by having a single program with many switches for controlling the sequence of actions. At the same time, experimentation with logic manipulation allowed us to improve some of the algorithms used in both the PRESTO approach and the MINI approach.

The conclusion we came to was that the basic approach used by MINI was generally superior; i.e., the technique of computing the complement of the PLA allowed one to use more efficient algorithms, and on the average this offset the initial cost involved, which was reduced by our discovery of better complementation algorithms.

The present version of POP has two options for the expansion step: the PRESTO strategy, on the MINI strategy with the improved complementation algorithm used in ESPRESSO I. The basic structure is similar to that used in PRESTO, i.e., the outputs are only lowered and no technique is used to try to escape from local minima.

In many of the PLAs minimized by POP, the complementation strategy allows a faster execution of the program.

3.5. The ESPRESSO II Program

Recent research results have led to the development of a new logic minimizer, ESPRESSO II. The algorithms of ESPRESSO II are reported in [Bray84b]. In this section we can only present a brief overview of the operations and results of ESPRESSO II. ESPRESSO II follows the top level structure used originally in MINI, but departs from it partially, and uses improved algorithms for all of the basis computations. The global sequence used by ESPRESSO II is outlined below:

1. *Complement* (computes the complement of the PLA and the *don't care* set, i.e., computes the care OFF-set).
2. *Expand* (expands each implicant into a prime and removes covered implicants).
3. *Essential primes* (extracts the essential primes and puts them in the *don't care* set).
4. *Irredundant cover* [finds minimal (optionally minimum irredundant) cover].
5. *Reduce* (reduces each implicant to a minimum essential implicant).

6. Iterate steps 2, 4, and 5 until no improvement.
7. *Lastgasp* (try Reduce, Expand, and Irredundant Cover one last time using a different strategy).
8. *Makesparse* (include the essential primes back into the cover and make the PLA structure as sparse as possible).

Although we use different algorithms, procedures 1, 2, 5, and 8 are found in MINI. Since the result of Reduce is an irredundant cover, MINI omits step 4, bue after reduction it uses a procedure called Reshape. Thus in step 6, Expand, Reduce, and Reshape are repeated in MINI until no improvement occurs; in ESPRESSO II, Expand, Irredundant cover, Reduce is the basic iteration. Lastgasp uses one iteration of Modified Reduce, Modified Expand, Irredundant Cover. Makesparse uses a form of Irredundant Cover on the outputs and a form of Expand on the inputs. An improvement found in the APL version of MINI is an option called FAT/SLIM for making the final results more sparse. Essential Primes is a routine which returns the primes that are present in *any* cover of the given logic function. Hence there is no need to process these primes any more—they cannot be covered and they need not be expanded and reduced. Thus they are temporarily placed in the *don't care* set. At the end of the minimization they are added again to the ON-set cover.

It is interesting to note that only Expand uses the complement, and there only to create a "blocking" matrix which indicates how far an implicant can be expanded before it begins to intersect with the complement.

There is an algorithmic theme for ESPRESSO II. The algorithms Complement, Reduce, Tautology, and Irredundant Cover employ the "unate recursive paradigm." Roughly, this paradigm divides a logic function recursively until each part becomes unate. The operation (e.g., complement, reduce, etc.) is then performed on the unate function, usually in a highly efficient way. Finally, the results are merged appropriately to obtain the required result. Tautology, although not one of the top-level procedures, is the major component of Essential Primes, Irredundant Cover, and thus Lastgasp and Makesparse. As pointed out in Section 3.3, the complementation algorithm of POP is essentially the same as the one used in ESPRESSO II and makes use of the unate recursive paradigm as well.

MINI allows bit pairing and, more generally, multiple-valued inputs. A method has been discovered for using ESPRESSO II (or any other two-valued logic minimizer) to minimize logic functions with multiple-valued inputs [Bray84b]. The method is very simple: it involves a particular encoding of the multiple-valued inputs and the creation of a *don't care* set before the use of the two-value logic minimizer. The

Table 3.1. **Comparison between ESPRESSO II and POP.**

PLA	in	out	Cubes			Literals			CPU		Mem	
			Init	Esp	Pop	Init	Esp	Pop	Esp	Pop	Esp	Pop
5xp1	7	10	128	74	76	1472	587	593	273.2s	42.5s	586K	518K
9sym	9	1	420	87	148	4200	609	1036	80.6s	212.3s	638K	2593K
add6	12	7	1092	355	355	9840	2551	2551	441.4s	946.3s	818K	2429K
adr4	8	5	255	75	75	2672	415	415	57.9s	114.9s	507K	853K
alu2	10	8	87	70	74	593	347	357	68.7s	19.2s	485K	454K
alu3	10	8	68	66	68	352	347	352	32.4s	19.6s	526K	431K
boc0	26	11	419	179	183	6673	2138	2165	340.9s	383.4s	916K	1462K
bocc	26	45	245	137	138	4903	2529	2549	413.2s	184.6s	1463K	910K
bocd	26	38	243	117	118	4483	2022	2036	243.4s	158.3s	741K	936K
chkn	29	7	153	140	145	1865	1739	1799	195.3s	225.7s	688K	732K
clpl	11	5	20	20	20	75	75	75	2.6s	2.9s	288K	332K
co14	14	1	14	14	14	210	210	210	1.5s	3.1s	286K	312K
cops	24	109	654	161	400	7810	2884	5205	1316.7s	559.1s	1713K	1350K
ddc2	8	7	58	39	42	455	265	271	9.7s	8.0s	351K	353K
dist	8	5	255	120	128	2631	869	940	304.4s	97.1s	742K	1009K
dk17	10	11	57	18	37	631	135	162	9.6s	30.1s	366K	394K
dk27	9	9	20	10	16	200	46	55	8.2s	12.5s	335K	366K
dk48	15	17	42	22	32	672	132	114	58.1s	168.2s	472K	483K
exep	30	63	149	109	112	1944	1284	1303	304.7s	134.4s	1177K	764K
f51m	8	8	255	76	76	3064	395	399	113.5s	141.2s	555K	840K
gary	15	11	214	107	107	2240	1115	1120	57.7s	66.0s	570K	633K
int0	15	11	135	107	108	1825	1115	1125	70.7s	63.4s	580K	559K
int1	16	17	110	104	105	2100	1970	1982	98.7s	80.9s	775K	536K
int2	19	10	137	135	135	1527	1465	1466	159.0s	61.9s	599K	533K
int3	35	29	75	74	75	852	771	767	58.3s	37.7s	521K	465K
int4	32	20	234	212	212	3291	2557	2544	277.0s	258.8s	781K	977K
int5	24	14	62	62	62	752	741	738	26.4s	25.3s	485K	422K
int6	33	23	54	54	54	552	552	547	23.7s	16.2s	444K	426K
int7	26	10	84	54	55	563	427	417	33.8s	43.9s	452K	457K
intb	15	7	664	629	639	6258	5876	5955	1752.9s	853.8s	581K	3151K
jibp	36	57	166	122	166	1252	1053	1252	305.4s	59.2s	812K	472K
misc	56	23	75	69	69	255	247	247	39.4s	62.5s	440K	474K
mish	94	43	91	82	82	255	238	238	66.0s	30.4s	379K	415K
misj	35	14	48	35	35	125	102	102	9.3s	9.7s	329K	366K
mlp4	8	8	225	124	130	2478	884	923	557.0s	121.0s	620K	956K
rckl	32	7	96	32	96	1238	657	1238	82.6s	116.1s	598K	494K
rd53	5	3	31	31	31	197	175	175	3.6s	3.9s	296K	361K
rd73	7	3	147	127	147	1023	903	1023	37.0s	38.2s	469K	767K
risc	8	31	74	28	30	407	187	184	9.6s	8.2s	364K	376K
root	8	5	255	57	58	2655	374	400	90.4s	53.4s	658K	713K
ryy6	16	1	112	112	112	736	736	736	28.3s	261.8s	577K	956K
seqn	7	3	84	29	39	732	143	235	16.1s	14.0s	403K	491K
sqr6	6	12	63	49	53	637	285	292	76.2s	20.5s	388K	482K

PLA	in	out	Cubes			Literals			CPU		Mem	
			Init	Esp	Pop	Init	Esp	Pop	Esp	Pop	Esp	Pop
ti	47	72	241	215	222	3171	2591	2584	2451.9s	253.8s	908K	753K
tial	14	8	640	581	640	5627	5174	5627	2375.6s	748.9s	1334K	3811K
ts10	22	16	128	128	128	1024	1024	1024	26.3s	22.1s	394K	398K
vg2	25	8	110	110	110	914	924	914	53.3s	50.9s	516K	474K
vtx1	27	6	110	110	110	1074	1084	1074	36.6s	58.2s	469K	490K
wim	4	7	10	9	10	91	58	64	1.7s	3.1s	248K	321K
x1dn	27	6	112	110	110	1090	1084	1074	36.1s	60.6s	471K	490K
x2dn	82	56	112	104	110	578	564	569	127.9s	44.9s	505K	427K
x6dn	39	5	121	81	87	1400	817	862	50.1s	75.3s	556K	627K
x7dn	66	15	622	538	622	5642	4600	5356	1607.5s	912.5s	1107K	2160K
x9dn	27	7	120	120	120	1258	1268	1258	43.4s	63.8s	515K	491K
z4	7	4	127	59	59	1145	311	311	20.5s	35.7s	391K	626K

Key:

in = # variables in Boolean function (inputs to PLA)

out = # Boolean functions (outputs from PLA)

cubes = # cubes in the function (product terms in PLA)

literals = # bound cube coordinates (transistors in PLA)

CPU = Processor seconds used

Mem = Peak memory use in kilobytes

55 PLAs counted in the totals

Initial	10323 cubes,	109709 literals
POP	7185 cubes,	67010 literals
Espresso-II (C)	6489 cubes,	61651 literals

POP	8100.0 sec,	801.3K bytes average
Espresso-II (C)	14986.0 sec,	603.4K bytes average

Running on a VAX 11/780, Berkeley UNIX 4.2 BSD

Table 3.2. Comparison between PRESTO-Covering and Tautology Covering

PLA	Pop					New_pop				
	C	Bl	BO	CPU	Mem	C	Bl	BO	CPU	Mem
5xp1	76	401	192	42.5s	518K	67	277	91	46.7s	405K
9sym	148	888	148	212.3s	2593K	148	888	148	82.8s	471K
add6	355	2196	355	946.3s	2429K	355	2196	355	274.4s	1032K
adr4	75	340	75	114.9s	853K	75	340	75	76.9s	452K
alu1	19	41	19	1.2s	250K	19	41	19		
alu2	74	282	75	19.2s	454K	73	279	73	9.0s	328K
alu3	68	284	68	19.6s	431K	68	284	68	3.9s	282K
boc0	183	1469	696	383.4s	1462K	182	1448	665	254.9s	566K
bocc	138	1931	618	184.6s	910K	141	1965	615	112.8s	423K
bocd	118	1631	405	158.3s	936K	121	1671	403	73.9s	439K
chkn	145	1654	145	225.7s	732K	145	1654	145	47.6s	384K
clpl	20	55	20	2.9s	332K	20	55	20		
co14	14	196	14	3.1s	312K	14	196	14		
cops	400	4599	606	559.1s	1350K	606	6729	606	605.3s	615K
ddc1	14	39	25	15	42	25				
ddc2	42	222	49	8.0s	353K	42	222	49	3.7s	291K
dist	128	760	180	97.1s	1009K	136	804	180	54.8s	522K
dk17	37	123	39	30.1s	394K	37	123	39	14.7s	359K
dk27	16	38	17	12.5s	366K	17	40	17	5.6s	323K
dk48	32	80	34	168.2s	483K	34	84	34	66.5s	520K
exep	112	1191	112	134.4s	764K	112	1190	112	40.2s	411K
f51m	76	323	76	141.2s	840K	76	326	76	95.1s	475K
gary	107	899	221	66.0s	633K	214	1798	442	43.5s	524K
int0	108	903	222	63.4s	559K	110	917	221	39.0s	434K
int1	105	979	1003	80.9s	536K	108	990	1003	40.9s	352K
int2	135	1193	273	61.9s	533K	135	1193	273	43.9s	372K
int3	75	512	255	37.7s	465K	75	512	255	18.9s	327K
int4	212	2133	411	258.8s	977K	212	2133	411	114.6s	481K
int5	62	532	206	25.3s	422K	62	533	219	6.6s	306K
int6	54	437	110	16.2s	426K	54	439	113	4.5s	301K
int7	55	338	79	43.9s	457K	55	338	79	12.9s	386K
intb	639	5316	639	853.8s	3151K	639	5316	639	339.4s	653K
jibp	166	1063	189	59.2s	472K	166	1063	189	23.0s	364K
misc	69	172	75	62.5s	474K	75	180	75	5.8s	305K
mish	82	147	91	30.4s	415K	91	164	91	4.9s	311K
misj	35	54	48	9.7s	366K	48	77	48		
mlp4	130	757	166	121.0s	956K	132	760	167	82.5s	443K
rckl	96	1141	97	116.1s	494K	96	1141	97	18.7s	318K
rd53	31	140	35	3.9s	361K	31	140	35	1.5s	248K
rd73	147	876	147	38.2s	767K	147	876	147	13.0s	335K
risc	30	129	55	8.2s	376K	30	129	55	3.5s	298K
root	58	305	95	53.4s	713K	66	342	95	25.4s	518K
ryy6	112	624	112	261.8s	956K	112	624	112	41.2s	343K

PLA	Pop					New_pop				
	C	BI	BO	CPU	Mem	C	BI	BO	CPU	Mem
seqn	39	189	46	14.0s	491K	39	189	46	7.1s	317K
sqr6	53	219	73	20.5s	482K	53	216	71	13.5s	320K
ti	222	1905	679	253.8s	753K	223	1913	697	211.1s	500K
tial	640	4983	644	748.9s	3811K	640	4983	644	260.9s	393K
ts10	128	896	128	22.1s	398K	128	896	128	10.1s	329K
vg2	110	804	110	50.9s	474K	110	804	110	12.3s	332K
vtx1	110	964	110	58.2s	490K	110	964	110	13.8s	334K
wim	10	24	40	3.1s	321K	9	22	37	1.2s	251K
x1dn	110	964	110	60.6s	490K	112	978	112	14.8s	332K
x2dn	110	452	117	44.9s	427K	110	452	117	12.5s	342K
x6dn	87	677	185	75.3s	627K	86	663	180	28.7s	392K
x7dn	622	4734	622	912.5s	2160K	622	4734	622	313.3s	363K
x9dn	120	1138	120	63.8s	491K	120	1138	120	16.4s	338K
z4	59	252	59	35.7s	626K	59	252	59	21.0s	357K

Average solution (for 52 PLAs)

Pop 136.8 cubes, 1281.2 literals

New_pop 143.6 cubes, 1343.1 literals

Execution statistics (for 52 PLAs)

Pop 155.5 cpu seconds and 828.1K memory

New_pop 70.9 cpu seconds and 400.3K memory

Key:

C = # product terms in PLA

BI = # input-plane transistors in PLA

BO = # output-plane transistors in PLA

Mem = Memory use in K-bytes

CPU = Processor seconds used

Running on VAX 11/780, Berkeley UNIX 4.2 BSD

results obtained to date seem to be better than other minimizers created especially for multiple-valued logic minimization.

There are two basic versions of the ESPRESSO II program: the original one implemented at the T. J. Watson Research Center of IBM by R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, written in APL and running on IBM computers; the other implemented at Berkeley by R. Rudell, written in C for the VAX 11/780 and running on the Berkeley 4.2 BDS UNIX operating system [Rude84]. Both versions have been thoroughly tested and compared against MINI, PRESTO, and POP. The final size of the PLA, its sparsity, and the relative compute time required for each were the objects of the experiment. The PLAs tested include control logic as well as data flow logic. Throughout the development of the program we use these PLAs for deciding on a particular version of an algorithm or on the handling of special cases.

The size of the PLAs minimized with the APL version of ESPRESSO II and their sparsity is equal to or better than the results obtained by MINI. In addition, these results required significantly less computer time than a version of MINI coded in APL. Results on these comparisons are reported in [Bray84a,b].

The comparisons between the C version of ESPRESSO II and POP are shown in Table 3.1. It is interesting to note that in all cases, ESPRESSO obtained results better than or equal to the ones obtained by POP. In some cases, the difference is quite large. The difference in results is due to the particular expansion strategy followed by POP, which does not expand outputs, for this reason, multiple-output logic functions, which can have significant output sharing but are given to the minimization program as almost disjoint functions, are not minimized well by POP. On the other hand, ESPRESSO uses often more CPU time than POP since the organization of the program is much more complex.

Note that ESPRESSO II C can also be run in PRESTO mode by selecting a particular option. The PRESTO mode is based on the Tautology routine and is faster than the corresponding POP implementation, as seen in Table 3.2.

4. PROGRAMMABLE LOGIC ARRAY FOLDING

4.1. Introduction

As seen in Section 1, the straightforward translation of the personality matrix into a physical layout leads in general to a nonoptimal design of the array. In fact the personality matrices representing minimal covers of

switching functions contain a large number of *don't cares*. A straightforward implementation results in a significant waste of silicon area, i.e., area occupied only by interconnect and not directly contributing to the implementation of the logic function. Wasted area reduces circuit yield and degrades the time performance of the PLA by introducing unnecessary parasitics.

Topological design aims to reduce the wasted area. Several authors have proposed algorithms and techniques to achieve an optimal topological design [Gras82, Hach80, Pail81, Suwa81]. These methods fall into two major categories: *array folding*, dealt with in this section, and *array partitioning*, introduced in Section 5.

4.2. Folding

Different techniques have been presented recently which achieve a silicon area savings by *folding* the rows and/or the columns of a PLA. Though folded arrays have in principle the same basic structure, the most sophisticated techniques require the PLA to be organized with particular architectures.

We consider first programmable logic arrays implementing sum-of-products switching functions with a basic simple structure. The PLA consists of two adjacent arrays, the input array or AND plane and the output array or OR plane. Input signals and their complements run vertically in the AND plane, product terms run horizontally in both planes, and outputs run vertically in the OR plane. Both arrays are personalized by the presence of active devices in positions corresponding to the "cares" of the switching function (Figure 4.1).

The objective of folding is to determine a permutation of the rows (and/or columns) of the array, which permits a maximal set of column pairs (and/or row pairs) to be implemented in the same column (row) of

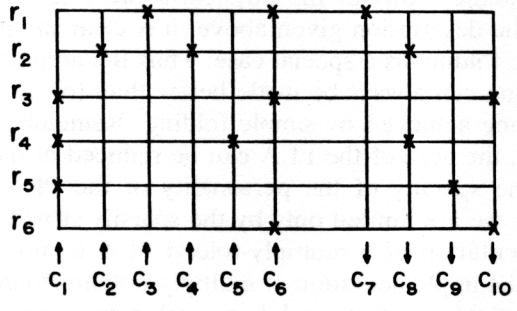


Figure 4.1. Symbolic representation of a programmable logic array.

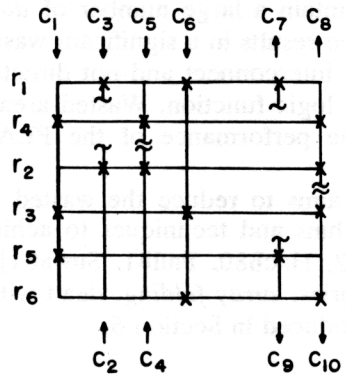


Figure 4.2. Simple folded array.

the logic array. Figure 4.2 gives a pictorial representation of the folding technique. Note that the physical columns (and/or rows) implementing the array which share the same physical location are *split*. Inputs and outputs to the split columns are provided from the top (bottom) of the array.

Wood presented for the first time a folded PLA implementation in [Wood79], and Hachtel et al. an algorithm for PLA folding in [Hach80]. Hu investigated the graph folding problem in [Hu83]. The technique reported in [Hach80] and [Hach82a] is referred here to as *simple folding*. Folding comes in two forms, *column folding* and *row folding*. A simply column-folded array is shown in Figure 4.2. Since large arrays are usually very sparse, a considerable area reduction can be achieved by folding rows and columns.

A generalization of simple folding is *multiple folding*. The objective of multiple column (and/or row) folding is to determine a permutation of the rows (and/or columns) of the PLA, which allows one to implement in each column (and/or row) of the physical array a set of logic columns (rows). From the description given above, it is clear the multiple folding contains simple folding as a special case. Thus the area savings achieved by this technique can always be made better than (or, in the worst case, equal to) the one achieved by simple folding. Remember that if simple folding is used, the area of the PLA can be reduced at most to 25%, no matter what the sparsity of the personality of the PLA is. If multiple folding is used, we are limited only by the sparsity structure of the PLA.

The implementation of a multiply-folded PLA is more complex. We deal first with the implementation of multiply-column-folded logic arrays.

The implementation of several logic columns in the same physical location requires that the physical (mental, poly, or diffusion) columns be

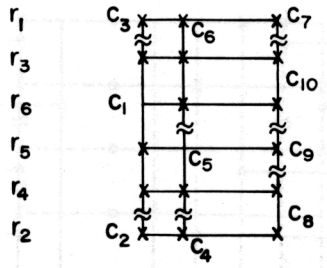


Figure 4.3. Multiple folded array.

split into segments (Figure 4.3). Therefore a path must be provided to route input and output signals to/from the split physical columns inside the array. Thus standard PLA architectures cannot be used to implement multiply-column-folded PLAs. Several authors [Chuq82, Demi81, Gree76], have proposed different architectures for multiply-folded arrays. We consider the following two structures, which can be implemented in *n*-MOS or CMOS technology.

The first architecture is shown in Figure 4.4. It requires two levels of metal (polysilicon), in addition to the usual levels of poly (metal) and diffusion. The PLA is implemented by using two arrays (the AND plane and the OR plane) personalized by MOS transistors. Input signals run vertically in the input columns of the AND plane, product terms run horizontally in the rows of both planes, and output columns run vertically in the OR plane. Two levels of interconnect are used for these rows and columns, in addition to ground diffusion rows and columns. The third level of interconnect (second metal or second poly level) is used to run horizontal *connection–rows* above the product-term rows to route the input and output signals to/from the input and output columns segments to the outside circuitry.

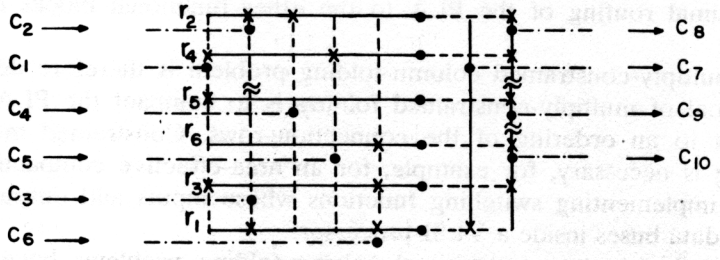


Figure 4.4. Folded PLA mixed diagram: (—) metal 1; (– –) Poly; (–·–) metal 2; (×) active device; (≈) cut; (●) contact (diffusion ground lines not shown).

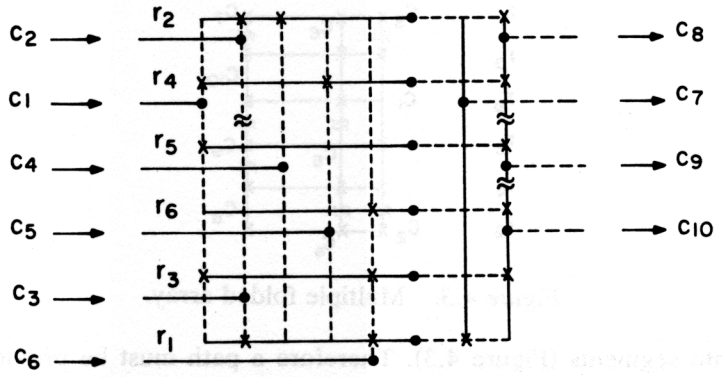


Figure 4.4. Folded PLA mixed diagram: (—) metal 1; (– –) Poly; (–·–) metal 2; (×) active device; (≈) cut; (●) contact (diffusion ground lines not shown).

An alternative architecture supports multiple folding with only one level of metal, poly, and diffusion. Input and output signals are routed inside/outside the array by connection–rows parallel and alternated with the product-term rows and implemented on the same level. This structure is simpler than the previous one but the area used by a multiply-folded PLA is larger (Figure 4.5).

It is important to note that PLAs implemented with either structure are essentially circuit blocks through which input and output buses run straight in the connection–rows. They are therefore suitable building blocks of a regular and structured VLSI design methodology.

Furthermore, it is important to point out that column folding induces a permutation of product terms and connection–rows. While product-term rows provide connection internal to the PLA only, connection–rows connect the array to the outside circuitry and their ordering is essential to an optimal routing of the PLA to the other functional blocks of the circuit.

A multiply-constrained column-folding problem is therefore defined. The goal of multiply-constrained folding is to compact the PLA area subject to an ordering of the connection–rows. Constrained multiply folding is necessary, for example, for an area-effective compaction of PLAs implementing switching functions whose inputs and outputs are signal data buses inside a VLSI processor.

In particular, two constrained column-folding problems have been addressed in [Demi83c]: *column folding with ordered connection–row assignment* and *column folding with bounded connection–row assignment.* In the former problem, each PLA input (and/or output) column is given a

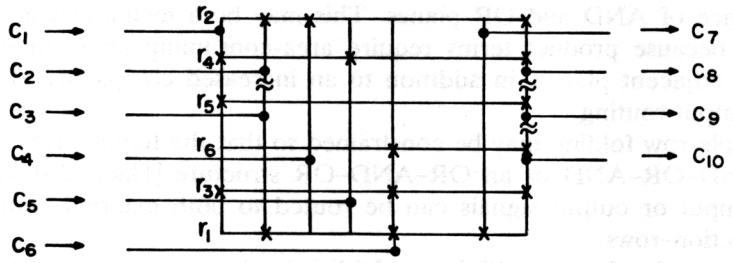


Figure 4.6. Multiple folded array with ordered connection–row assignment.

position index. Folding is constrained so that connection–rows can be positioned according to the sequence of indexes of the connected columns, as shown in Figure 4.6. In the latter, each input (and/or output) is given an upper and a lower bound on the position of the contacted connection–row. Folding is constrained so that each connection–row can be assigned to a position with an index satisfying the given bounds (Figure 4.7).

Unconstrained multiply-row-folded PLAs do not require a particular architecture and can be implemented with a single-poly, single-metal technology [Mead80]. Row folding induces a permutation of input and output columns, which leads to a segmented array consisting of a

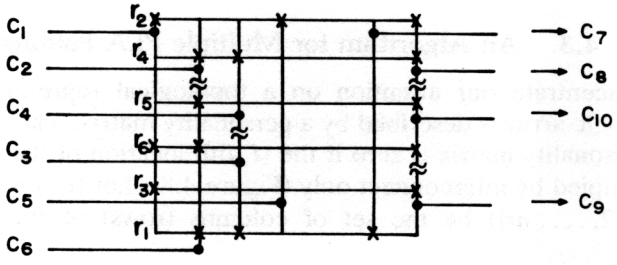


Figure 4.7. Multiple-folded array with bounded connection–row assignment:

Connection Row	Lower Bound	Upper Bound
1	1	3
2	1	3
5	4	6
6	4	6
7	1	1
9	4	6

sequence of AND and OR planes. This may be a technological drawback, because product terms require area-consuming connections between adjacent planes, in addition to an increased complexity of input and output routing.

Simple row folding may be constrained so that the folded array shows an AND–OR–AND or an OR–AND–OR structure [Hach82b]. In this case input or output signals can be routed to both external planes by connection–rows.

On the other hand, multiple row folding leads to a segmentation of the array into more than three planes [Gree76, Suwa81]. Since routing of the columns of the internal planes may be difficult, a new constrained folding problem is defined: *row folding with bounded column assignment*. Each column is given a left and right bound and row folding is constrained so that each column can be assigned to a position within the bounds.

Multiply-row- and column-folded arrays can be implemented with the described architectures, provided that only columns in the external planes are multiply folded. To connect a multiply-row- and column-folded array effectively, it is important to be able to determine which signals are routed to the external planes through connection–rows and which are routed from the top and the bottom of the array.

The related constrained multiple row and column folding problem consists of constraining the folding so that input and output signal can be routed from the desired (left, right, top, bottom) direction.

4.3. An Algorithm for Multiple PLA Folding

We concentrate our attention on a topological representation of a PLA. A logic array is described by a personality matrix. The (i, j)th entry of the personality matrix is zero if the (i, j)th location of the physical array is occupied by interconnect only (Figure 4.8). Let $\{c_i, i = 1, 2, \ldots, nc\}$ ($\{r_i, i = 1, 2, \ldots, nr\}$) be the set of columns (rows) of the personality

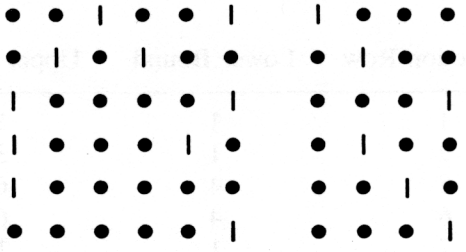


Figure 4.8. Personality matrix.

matrix. Each column is labeled *input* (*output*), if it carries an input (output) signal in the physical array. Two columns c_i, c_j (rows r_i, r_j) are *disjoint* if they have "cares" in different rows (columns). An *ordered column* (*row*) *folding list* is an ordered set of either input or output disjoint columns $o_i = (c_{i,1}, c_{i,2}, \ldots, c_{i,n})$ [rows $o_i = (r_{i,1}, r_{i,2}, \ldots, r_{i,n})$]. An *ordered column* (*row*) *folding set* $O = \{o_1, o_2, \ldots, o_k\}$ is a set of disjoint column- (row) ordered folding lists. Let U be the set of unfolded columns (rows). The column (row) cardinality of a folded PLA is the number of physical columns (rows) that are implemented in a folded array, i.e., $C(O) = |O| + |U|$ [$R(O) = |O| + |U|$]. An ordered column (row) folding list induces a set of ordering relations among the rows (columns). As an example, if column $c_{i,1}$ is folded on top of $c_{i,2}$ ($c_{i,1}, c_{i,2} \in o_i$), then all the rows with a "care" in $c_{i,2}$ must follow all the rows connected to $c_{i,1}$ in the folded array.

A column- (row-) ordered folding set is *implementable* if the transitive closure of the ordering relations induced on the rows (columns) is a partial order of the set of the natural numbers Z^+. The optimal unconstrained column (row) folding problem can be stated as follows:

Find an implementable ordered folding set that minimizes the column (row) cardinality of the PLA.

The optimal multiple PLA folding problem was shown to be NP-complete in [Luby82]. A heuristic algorithm is therefore used to construct an ordered folding set such that the column (row) cardinality is minimal.

At each step the algorithm tries to increase the cardinality of the folded column set and verifies the implementability of the folded array. A conceptual description of the algorithm is the following:

```
main {
  top:
    if ( the set of columns/rows which haven't been processed is empty )
        stop;
    else {
      select a pair of unfolded disjoint columns/rows or an unfolded
      column/row and a column/row folding list as folding candidates };
    if ( the induced folding set is not implementable ) goto top;
    if ( folding is constrained and constraints are not satisfied ) goto top;
    Fold the candidates ;
    goto top ;
}
```

A detailed description of the algorithm for simple column folding is given in [Hach82a]. The extension to multiple folding is presented in [Demi83a]. A graph-theoretic interpretation of the folding problem is used to define a criterion to verify the folded-array implementability and to study heuristics for the multiple folding candidate selection [Demi83b, Demi83c].

4.4. Multiple Constrained Folding

Constraints on PLA folding are classified into two major categories:

(i) Architectural or primary constraints.
(ii) Secondary constraints.

Architectural constraints are related to the structure of the array and to the positions of input/output buses relative to the array. Secondary constraints are related to the positions of input and output lines inside the buses. Examples of architecture-constrained folding problems are

(1a) Simple column folding with a subset of inputs and/or outputs connected to the top (bottom) of the array.
(1b) Simple row folding with AND–OR–AND or OR–AND–OR architecture.
(1c) Segmented arrays: the column set is partitioned into subsets, each forming a segment of the array. Columns are folded with columns in the same segment only and the sequence of segments is preserved.

The following folding problems involve secondary constraints:

(2a) Column folding with bounded product–row assignment.
(2b) Row folding with bounded column assignment.
(2c) Column folding with bounded connection–row assignment.
(2d) Column folding with ordered connection–row assignment.

The algorithm presented in the previous section can handle both architectural and secondary constraints. Different strategies are used in the two cases. To satisfy architectural constraints it is sufficient that folding candidates satisfy the following requirements for the related problems:

(1a) *Columns connected to I/O buses on the top (bottom) of the array are folded either on top (bottom) of an unfolded column or folding list or not folded at all.*

(1b) AND–OR–AND (OR–AND–OR) architecture: *Rows connected to input (output) columns that are connected to rows folded on the left or on the right are selected as candidates to be split on the left or on the right of the array, respectively.*

(1c) Selected candidates for column folding are constrained to be in the same segment. In the case of no more than three segments and simple row folding, the selection of candidates for row folding is as follows: *rows connected to columns in the leftmost (rightmost) segment are folded on the left (right) only or not folded at all.*

Unfortunately we cannot be sure that secondary constraints are satisfied only on the basis of an appropriate selection of folding candidates. The reason is that secondary constraints are related to the row (column) positions induced by a column (row) folding. Therefore an assignment procedure is called by the algorithm which attempts a product– and connection–row ordering compatible with the ordered folding list and the given bounds.

4.4A. Column Folding with Bounded Product–Row Assignment

In this section we consider the problem of constraining product-term row positions only. We therefore refer to product-term rows as rows throughout this section.

We define *lower (upper) row bound map*: a map

$$L_R : \{r_i, ; i = 1, 2, \ldots, nr\} \rightarrow \{1, 2, \ldots, nr\}$$
$$(U_R : \{r_i ; i = 1, 2, \ldots, nr\} \rightarrow \{1, 2, \ldots, nr\}) \tag{4.1}$$

relating each row to a lower (upper) position bound.

We define *row assignment* $P : \{r_i ; i = 1, 2, \ldots, nr\} \rightarrow \{1, 2, \ldots, nr\}$ a permutation of the rows and *implementable row assignment* a permutation compatible with an ordered column-folding set O.

An *implementable bounded-row assignment* is an implementable row assignment such that

$$L_R(r_j) \leq P(r_j) \leq U_R(r_j) \quad \forall\, j = 1, 2, \ldots, nr. \tag{4.2}$$

Example: For the logic array shown in Figure 4.1, the following lower and upper bounds are given:

$$L_R = 1,\ 1,\ 1,\ 4,\ 4,\ 6 \qquad U_R = 1,\ 3,\ 3,\ 6,\ 6,\ 6.$$

This means that r_1 is constrained to the first position, r_2 and r_3 are constrained between positions 1 and 3, and so on. The implementable

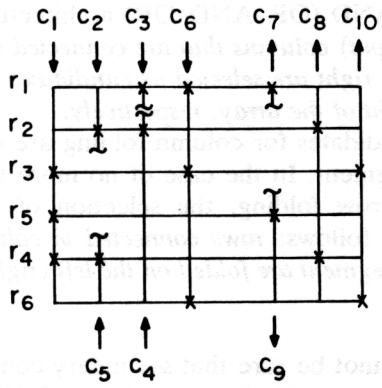


Figure 4.9. Folded logic array with bounded row assignment.

row assignment (r_1, r_4, r_2, r_3, r_5, r_6) induced by the column folding shown in Figure 4.2 does not satisfy the given bound maps. On the contrary, the folded PLA shown in Figure 4.9 has the following implementable row assignment: (r_1, r_2, r_3, r_5, r_4, r_6). Note that rows are numbered from the top to the bottom of the array in the figures. □

The optimal bounded-row column folding problem can be stated as follows:

Find an implementable ordered column-folding set and a related implementable bounded-row assignment that minimizes the column cardinality of the folded PLA.

Let us consider the following subproblem:

Given an ordered column-folding set and lower and upper row bound maps, find an implementable bounded row assignment, if it exists.

The problem consists in matching each product row to a position. The following algorithm will either construct a matching such that the required bounds are satisfied or return the flag FALSE if no possible matching exists.

Assignment Algorithm

```
for ( each position i = 1 to nr ) {
    if ( ∃ an unassigned row which must be assigned to a position
         lower than i ) return ( FALSE );
```

if ($\nexists$ an unassigned row which must be assigned to a position greater or equal to i) **return** (FALSE);

assign to position i the unassigned row whose position upper bound is minimal

}

return (TRUE);

The algorithm runs in linear time since it cycles at most nr times through the main loop. The algorithm uses a greedy strategy: at each iteration it matches the available position with lowest index to the most constrained row (i.e., selects the product–row with lowest upper bound). It is proven in [Demi83c] that the algorithm finds an implementable bounded-row assignment, if one exists.

Example: Consider the column-folded logic array shown in Figure 4.1. The implementable bounded-row assignments given by the algorithm is $(r_1, r_2, r_3, r_5, r_4, r_6)$. □

The row folding with bounded-column assignment problem can be derived *mutatis mutandis* from this problem, and solved by a similar algorithm.

4.4B. Column Folding with Bounded Connection–Row Assignment.

We refer in this subsection to a logic array implemented with connection–rows for routing input and output signals as described before. According to these architectures, there are two sets of connection rows contacting the columns of the left and right array, respectively. For the sake of simplicity, we will consider constrained folding of one array only.

Both proposed architectures support at most as many connection–rows as product–rows. Since each column is contacted to a connection row, we require throughout the subsection that the number of columns in the considered array be at most equal to the number of rows. Most PLA satisfy this assumption.

We define *connection–row assignment* as a one-to-one map: $T:\{c_i, i = 1, 2, \ldots, nc\} \rightarrow M \subseteq \{1, 2, \ldots, nr\}$ such that $j = T(c_i)$ if column c_i is contacted to the connection row in the jth position.

Example: Consider the OR plane of the PLA shown in Figure 4.1. Figure 4.10 shows the unfolded array with the connection–row assignment

$$T(c_7) = 1 \qquad T(c_8) = 2 \qquad T(c_9) = 5 \qquad T(c_{10}) = 6.$$ □

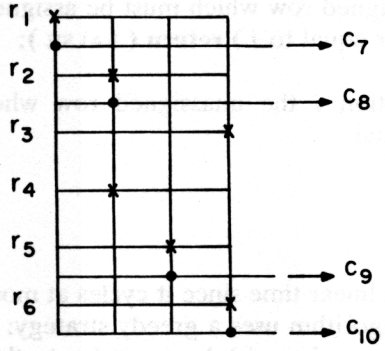


Figure 4.10. Unfolded OR array.

We define *physical connection–row* set M as the image of T. Its elements are the position of the connection–rows which are physically implemented. Note that there are $\Delta = nr - nc$ slack connection–rows which are not implemented and whose positions are irrelevant to the problem.

We define the *lower* (*upper*) *connection–row bound map* as a map

$$L_C : \{c_i, i = 1, 2, \ldots, nc\} \rightarrow 1, 2, \ldots, nr$$

$$(U_R : \{c_i, i = 1, 2, \ldots, nc\} \rightarrow 1, 2, \ldots, nr)$$

relating each column to a lower (upper) position bound on the position of the connected connection–row.

Example: For the OR plane of the PLA shown in Figure 4.1, the following bounds are given:

$$L_C = 1, 1, 4, 6 \qquad U_C = 1, 3, 6, 6.$$

This means that the first column of the OR plane (c_7) must be connected to a connection–row in position 1, the second one (c_8) to a connection–row whose position is bounded between 1 and 3, and so on. □

An *implementable connection–row assignment* is an assignment compatible with a column-ordered folding set, i.e., is an assignment such that

$$\max[P(R(c_{i,j-1}))] < T(c_{i,j}) < \min[P(R(c_{i,j+1}))] \qquad j = 1, 2, \ldots, n$$

for any column $c_{i,j}$ in folding list o_i with cardinality n, where by definition

$$\max[P(R(c_{i,0}))] = 0 \qquad \min[P(R(c_{i,n+1}))] = \infty.$$

Example: Consider the folded OR plane shown in Figure 4.2 with the ordered folding set $O = \{(c_7, c_9), (c_8, c_{10})\}$. An implementable connection–row assignment is

$$T(c_7) = 1 \qquad T(c_8) = 2 \qquad T(c_9) = 3 \qquad T(c_{10}) = 6.$$

The connection–row contacted to c_8 is in position 2, and therefore is above (has lower index than) the product rows connected to c_{10} (in positions 4 and 6). The connection–row contracted to c_{10} is in position 6 and is below (follows) the product–rows connected to c_8 (in positions 2 and 3). □

An *implementable bounded connection–row assignment* is an implementable connection–row assignment such that

$$L_C(c_j) \leq T(c_j) \leq U_C(c_j) \qquad j = 1, 2, \ldots, nc.$$

Example: An implementable bounded connection–row assignment is

$$T(c_7) = 1 \qquad T(c_8) = 2 \qquad T(c_9) = 4 \qquad T(c_{10}) = 6.$$

Figure 4.11 shows a folded implementation of the OR plane compatible with the bounded connection–row assignment. □

We can now state the column folding with bounded connection–row assignment problem as follows:

Find an implementable ordered column-folding set and a related implementable bounded connection–row assignment which minimizes the column cardinality of the folded PLA.

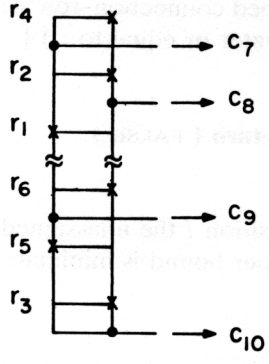


Figure 4.11. Folded OR array with bounded connection row assignment.

In particular, we concentrate on the following subproblem:

Given an ordered column-folding set and lower and upper connection–row bound maps, find an implementable bounded connection–row assignment, if it exists.

Note that an implementable bounded connection–row assignment requires, by definition, a product–row assignment, because the positions of rows in the two sets influence each other. Hence the problem consists in finding the two row assignments compatible with the ordered column-folding set, if they exist.

The problem consists in matching each product and each connection–row to a position. The following algorithm will either construct a matching such that the required bounds are satisfied or return the flag FALSE if no possible matching exists.

Double Assignment Algorithm

$\Delta = nr - nc$
for (each position $i = 1$ to nr) {
    if ($\exists$ an unassigned row or connection–row which must be assigned to a position lower than i) **return** (FALSE);

    if ($\nexists$ an unassigned row which must be assigned to a position greater or equal to i) **return** (FALSE);

    assign to position i the unassigned row whose position upper bound is minimal;

    if ($\nexists$ an unassigned connection–row which must be assigned to a position greater or equal to i) {

        $\Delta = \Delta - 1$;
        if ($\Delta < 0$) **return** (FALSE);

    }
    else assign to position i the unassigned connection–row whose position upper bound is minimal;

return (TRUE);

The double assignment algorithm runs in linear time and uses a greedy

strategy. At each iteration, it tries to match the available position with lowest index with the most constrained product and connection–rows. Note that a connection–row need not be assigned at each iteration, but the total number of slack positions must be lower than or at least equal to Δ. It is proven in [Demi83c] that the assignment algorithm returns TRUE if and only if there exists a row and a connection–row assignment compatible with the ordered folding set and the problem bounds.

4.4C. Column Folding with Ordered Connection–Row Assignment

We extend to this subsection of the considerations on multiply-column-folded PLA implementation and the basic definitions presented in the previous subsection.

We define *order map* $S:\{c_i\,;i=1,2,\ldots,nc\}\rightarrow\{1,2,\ldots,nc\}$ as a one-to-one map relating each column to the required relative position of the contacted connection–row. We define *implementable ordered connection–row assignment* an implementable connection–row assignment such that:

$$T(c_i)<T(c_j) \quad \text{if} \quad S(c_i)<S(c_j) \quad \forall\, i,j=1,2,\ldots,nc\,.$$

Example: Consider the OR plane of the PLA shown in Figure 4.1 and the following order map:

$$S(c_7)=2 \qquad S(c_8)=1 \qquad S(c_9)=3 \qquad S(c_{10})=4\,.$$

This means that column folding is constrained so that the connection–row to c_8 is in the topmost position, followed by those connecting c_7, c_9, and c_{10} in that order. □

We state the column folding with ordered connection–row assignment problem as follows:

> *Find an implementable ordered column-folding set and a related implementable ordered connection–row assignment which minimizes the column cardinality of the folded PLA.*

This problem is equivalent to column folding with the following bounds on connection–row positions,

$$L_C(c_i)=S(c_i) \qquad \forall\, i=1,2,\ldots,nc$$
$$U_C(c_i)=S(c_i)+\Delta \qquad \forall\, i=1,2,\ldots,nc$$

with the additional constraint on the order of the connection–rows, and can be solved by the double assignment algorithm described above.

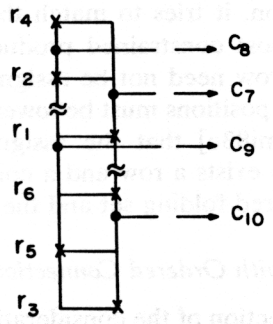


Figure 4.12. Folded OR plane implementation.

Example: Figure 4.12 shows a folded implementation with the implementable ordered connection–row assignment

$$T(c_7) = 2 \qquad T(c_8) = 1 \qquad T(c_9) = 3 \qquad T(c_{10}) = 4. \qquad \square$$

4.5. PLEASURE

PLEASURE is an interactive program for the simple/multiple constrained/unconstrained row and/or column folding of programmable logic arrays.

The PLA description is given as input to the program is the form of two-level sum-of-products logical implicants.

The output of the program is a symbolic table representing the folded array with the positions of the active devices corresponding to the cares of the logic function, the locations of the cuts, and the contacts between columns and connection–rows. The symbolic table can be processed by a *silicon assembler* program which generates the mask layout of the array according to a given technology. Note that the symbolic table generated by PLEASURE is technology independent.

The program is a command interpreter: input files can be read and edited; logic arrays can be folded in a single run or one fold at a time. This allows the user to monitor PLA folding step by step, by displaying the partially folded array. The user can enter column and/or folding candidates of his choice and verify the implementability of his selection. When PLAs are folded in a single run, a soft interrupt capability allows the user to halt the compaction at any point to see the partially compacted array before resuming folding execution. The program can be run in a silent mode (i.e., with no interaction with the user) so that it can be interfaced with other programs in a system for automated synthesis of PLAs.

Table 4.1. Comparison of PLAs Folded by PLEASURE with Different Constraints

PLA	*Size* $nr*(ni+no)$	*Constraints*	*Folding Lists*	*Folded Area (%)*	*Time (s)*
PLA 1	30*(8+31)	None	7	29	8
	30*(8+31)	Row positions	14	51	14
	30*(8+31)	Conn–row positions	15	53	23
	30*(8+31)	Ordered conn–rows	15	53	16
PLA 2	52*(23+20)	None	7	37	15
	52*(23+20)	Row positions	12	60	34
	52*(23+20)	Conn–row positions	13	46	62
	52*(23+20)	Ordered conn–rows	13	58	53
PLA 3	86*(8+63)	None	9	56	112
	86*(8+63)	Row positions	15	67	257
	86*(8+63)	Conn–row positions	12	63	305
	86*(8+63)	Ordered conn–rows	15	73	328
PLA 4	62*(24+14)	None	11	58	23
	62*(24+14)	Row positions	10	73	36
	62*(24+14)	Conn–row positions	9	68	45
	62*(24+14)	Ordered conn–rows	8	76	75
PLA 5	85*(27+10)	None	14	54	30
	85*(27+10)	Row positions	10	67	58
	85*(27+10)	Conn–row positions	9	72	87
	85*(27+10)	Ordered conn–rows	6	70	59
PLA 6	75*(35+29)	None	17	53	50
	75*(35+29)	Row positions	19	62	119
	75*(35+29)	Conn–row positions	18	64	199
	75*(35+29)	Ordered conn–rows	10	73	202
PLA 7	53*(35+29)	None	10	49	26
	53*(35+29)	Row positions	13	67	65
	53*(35+29)	Conn–row positions	17	58	110
	53*(35+29)	Ordered conn–rows	10	80	147
PLA 8	223*(47+62)	None	15	38	1262
	223*(47+62)	Row positions	39	55	3933
	223*(47+62)	Conn–row positions	39	57	4722
	223*(47+62)	Ordered conn–rows	33	60	4769

Folding instructions are entered to the program along with the PLA description in the input file. PLEASURE allows column (row) folding only and row and column folding.

Column folding can be simple or multiple, constrained or unconstrained in either or in both planes. Architectural constraints can be set of column positions. Columns can be required to be folded on the top

(bottom) of the array or not folded at all. Column-folded arrays can be segmented into three adjacent planes, so that columns in the external planes can be multiply folded and contacted by connection rows. Secondary constraints can be put on product– and connection–row positions. In particular, column folding with bounded or order connection–row assignment can be achieved.

Row folding can be simple or multiple. Simple row-folded arrays can be constrained to have an AND–OR–AND or OR–AND–OR architecture. Secondary constraints can be put on the column positions.

Row (column) folding can follow column (row) folding. Row folds can be alternated with column folds, by allowing the program to choose the local "best" fold at each step. This procedure achieves the best results as far as compaction is concerned. Multiple row- and column-folded PLA can be constrained by input/output position. An input (output) can be required to be connected to the top, bottom, left or right of the array.

PLEASURE has been tested on a large set of industrial arrays. To compare results obtained with arrays of different sizes, the following foldings have been tried: (i) unconstrained folding; (ii) column folding with constrained row positions: $L(r_i) = \max(i - \alpha, 0)$, $U(r_i) = \min(i + \alpha, nr)$, $\alpha = \frac{1}{10}nr$; (iii) column folding with constrained connection–row positions $L_C(c_i) = \max(i - \alpha, 0)$, $U_C(c_i) = \min(i + \alpha, nr)$, $\alpha = \frac{1}{10}nr$; (iv) column folding with ordered connection–row assignment: $S(c_i) = i$, $i = 1, 2, \ldots, nc$. The folding results and execution time on a VAX 11/780 computer are reported in Table 4.1.

5. PROGRAMMABLE LOGIC ARRAY PARTITIONING

Another approach to topological design of large arrays is based on *partitioning*. PLA partitioning exploits the structure of an array and breaks it up into subarrays.

Kang [Kang81] proposed various algorithms for PLA partitioning at the logical and topological levels. In particular, he proposed a top-down recursive division of a PLA into smaller PLAs and a bottom-up composition of small PLAs into larger arrays. The partitioning algorithm is based on the search of independent or loosely coupled sets of rows (columns) of the personality matrix. Repetition of rows (columns) is used to decouple loosely coupled sets.

The partitioning algorithm described in the sequel uses a graph interpretation of the PLA connectivity and a cluster search technique to devise loosely coupled sets of rows (columns). Decoupling is achieved by transforming the logic array into an equivalent one by means of rules which take advantage of the logic structure of the array.

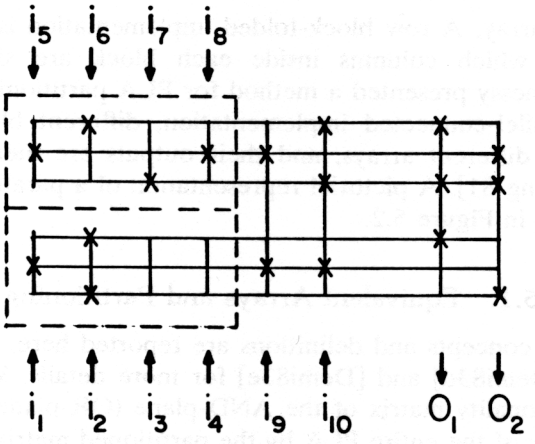


Figure 5.1. Block-folded array.

Partitioned arrays can be implemented as block-folded arrays or parallel-connected arrays. In the block-folded implementation the columns (rows) are partitioned into n sets and any column in different sets can be implemented in the same column (row) of the physical logic array. Block folding can be simple ($n = 2$) or multiple ($n > 2$), and comes in two flavors, column block folding and row block folding. A pictorial representation of a block-folded array is shown in Figure 5.1. The implementation of the multiply-column-block-folded arrays requires a structure with connection–rows as described in the previous sections.

Simple block folding has been also referred to as bipartite folding in [Egan82] in which a heuristic algorithm based on a search tree is presented. Suwa [Suwa81] refers to a partitioned array implementation as

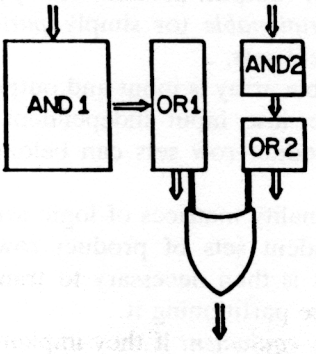


Figure 5.2. Parallel-connected implementation.

a segmented array. A row block-folded implementation is presented in [Suwa81], in which columns inside each block are simply folded. Recently Hennessy presented a method for PLA partitioning [Henn83].

In the parallel-connected implementation, different blocks are implemented as different arrays, and their outputs are merged together [Demi83e, Kang 81]. A pictorial representation of a parallel partitioned array is shown in Figure 5.2.

5.1. Equivalent Arrays and Partitioning

Some basic concepts and definitions are reported here. The reader is referred to [Demi83d] and [Demi83e] for more details. We denote by $\mathbf{A}(\mathbf{B})$ the personality matrix of the AND-plane (OR-plane). We denote the personality of the entire PLA by the partitioned matrix $\mathbf{A}\,|\,\mathbf{B}$. Let us partition the column set of the personality matrix into *input column set* $\mathbf{I}=\{i_1, i_2, \ldots, i_N\}$ and *output column set* $\mathbf{O}=\{o_1, o_2, \ldots, o_M\}$. Let us denote the *product–row set* of the personality matrix by $\mathbf{P}=\{p_1, p_2, \ldots, p_P\}$. A product–row p_j is divided into two parts: p_j^A (input product–row) and p_j^B (output product–row), where p_j^A contains the first N entries of p_j and p_j^B the last ones. Two vectors $\mathbf{x}, \mathbf{y}$ are *independent* (*orthogonal*) if $\mathbf{x} \wedge \mathbf{y} = \mathbf{0}$, where $\mathbf{0}$ is the null vector and $\wedge$ denotes the logical product operator. We denote by $\mathbf{x} \perp \mathbf{y}$ two independent vectors. Two sets of vectors $\mathbf{X}$, $\mathbf{Y}$ are *independent* if $\forall \mathbf{x} \in \mathbf{X}$ and $\forall \mathbf{y} \in \mathbf{Y}$, $\mathbf{x} \perp \mathbf{y}$. The input (output) signals corresponding to two orthogonal set of columns are independent because they are related to disjoint subsets of product terms.

Logic array partitioning relies on determining independent sets of vectors in the personality matrix of the PLA. A logic array is said to be *input* (*output*) *partitionable* if there exist input (output) column independent sets. An input (output) partitionable array has also independent sets of input (output) product–row p_j^A (p_j^B). A logic array is said to be *parallel partitionable* (or simply *partitionable*) if there exist product–row independent sets.

A parallel partitionable array is input and output partitionable, but the inverse is not true because input independent product–row sets and output independent product–row sets can belong to different product–row sets.

In general the personality matrices of logic arrays do not have input- and/or output-independent sets of product rows and cannot be partitioned as they are. It is then necessary to transform an array into an "equivalent" one before partitioning it.

Two logic arrays are *equivalent* if they implement the same switching

function. Equivalent arrays can have different size and can be obtained by introducing redundant rows [Chuq82] and/or columns [Kang81], Suwa81], or by rearranging the personality matrix of the array by a reshape of the logic function.

A general equivalence transformation based on row (column) *augmentation* is considered here. The augmentation of an input, output, or product is the replacement of the input, output column, or product–row with a set of input, output columns, or product–rows which gives an equivalent logic array. We present now rules to obtain equivalent arrays by augmentation:

Rule 1. *Input column augmentation*:
The logic arrays defined by **A**, **B** *and* **A**′, **B** *are equivalent if*

(*i*) *A*′ *is obtained from A by replacing an input column* i_j *with a column set* $I_j = \{i_{j1}, i_{j2}, \ldots, i_{js}\}$ *such that*

$$V_{k=1}^{s} i_{jk} = i_j.$$

(*ii*) *Input signals to columns in* I_j *correspond to the input signal to column* i_j.

An input partitionable array can be obtained by a sequence of input column augmentations.

Rule 2. *Output column augmentation*:
The logic arrays defined by **A**, **B** *and* **A**, **B**′ *are equivalent if*

(*i*) **B**′ *is obtained from* **B** *by replacing an output column* o_j *with a column set* $O_j = \{o_{j1}, o_{j2}, \ldots, o_{js}\}$ *such that*

$$V_{k=1}^{s} o_{jk} = o_j.$$

(*ii*) *The output signal from column* o_j *corresponds to the logic sum of the output signals from the column in* O_j.

An output partitionable array can be obtained by a sequence of output column augmentations, and a partitionable logic array by a sequence of input and output augmentations.

Rule 3. *Product–row augmentation*:
The logic arrays defined by **A**, **B** *and* **A**′, **B**′ *are equivalent if*

(*i*) [**A**′|**B**′] *is obtained from* [**A**|**B**] *by replacing a product–row* p_j *with a row set* $P_j = \{p_{j1}, p_{j2}, \ldots, p_{js}\}$ *such that*

$$V_{k=1}^{s} p_{jk}^{B'} = p_j^B \qquad p_{jk}^{A'} = p_j^A \quad \forall\, k = 1, 2, \ldots, s.$$

An output partitionable array can be obtained by a sequence of product–row augmentations and a partitionable array by a sequence of product augmentations followed by a sequence of input augmentations.

It is clear that there are many different possible augmentations for a row or a column according to rules 1, 2, and 3. For optimal topological design it is convenient that augmented rows and columns keep the array as sparse as possible to ease partitioning. Hence we require the augmented columns and the output part of the augmented product rows to be independent. Moreover, optimal topological design based on array partitioning requires the determination of an optimal sequence of augmentations.

5.2. A Clustering Algorithm for PLA Partitioning

Three different partitioning problems are reported here, namely *input partitioning*, *output partitioning*, and *parallel partitioning*. A graph representation of the problem is useful in understanding the underlying structure.

The AND-plane (OR-plane) of a PLA is represented by a bipartite graph $G^A(I, P, E^A)$ $[G^B(P, O, E^B)]$ whose adjacency matrix is

$$\begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix} \left(\begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix} \right),$$

where A and B are the personality submatrices of the AND- and OR-planes, respectively. The entire logic array is therefore represented by the union of such graphs, i.e., the tripartite graph $G(I, P, O, E)$, where $E = E^A \cup E^B$. The node sets I, P, and O are in one-to-one correspondence with the PLA input column, product–row, and output column sets, respectively. Since the search of loosely coupled sets is performed in a similar fashion for the three problems, for the sake of simplicity the graph will be referred to as $G(V, E)$.

The optimization problems arising from PLA partitioning are stated formally in [Demi83d,e] and require minimization of a nonlinear function with integer constraints. Since this problem is hard to solve, a heuristic algorithm based on a cluster search [Spat80] and on array transformations is proposed.

The rationale of the algorithm is as follows. The algorithm attempts first to find a node cluster inside graph $G(V, E)$ and then partitions V into two subsets V_1 and V_2. The former contains the cluster nodes and the latter the remaining ones. If the node partition induces a graph partition into two disjoint subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ the partition corresponds to the array partition. Else, the array is transformed by augmenting rows or columns according to the rules stated above (i.e.,

by adding appropriate nodes to the graph) until the edge set E is partitionable into E_1 and E_2 and $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are disjoint. Subgraph $G_1(V_1, E_1)$ is stored and the algorithm reattempts a cluster search on the updated graph $G(V, E) = G_2(V_2, E_2)$.

The cluster search in graph $G(V, E)$ is based on the *contour tableau* approach described in [Ogbu70] and in [Sang77]. The contour tableau is an array of three columns. The first one is called *iterating set* (IS); its entries are nodes of the graph. The second one is the *adjacency set* (AS); its entries are sets of nodes of the graph. The third column is the *objective function* (OF) vector; in this particular case its entries are the values of the area estimates. An area estimate can be easily obtained from the knowledge of subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ as shown in [Demi83d].

The tableau is built iteratively until a cluster is found and convenient conditions are met to separate it from the rest of the graph. At this point the tableau is cleared and the algorithm restarts on the rest of the graph. The algorithm is described in Pidgin C.

Partitioning Algorithm

```
while ( V ≠ ∅ ) {
        IS = ∅; AS = ∅; OF = ∅;
        i = 1;
        IS (i ) = inselect( V );
        AS (i ) = adj( IS (i ) );
        while ( cluster criterion not satisfied ) {
                IS (i + 1) = nextselect( AS (i ) );
                AS (i + 1) = nextadj( IS, AS (i ) );
                i = i + 1;
        };
        G (V, E ) = update( G (V, E ) );
        };
}.
```

The algorithm constructs step by step a cluster set $X = \bigcup_{j=1}^{i} \mathrm{IS}(j)$. Procedure *inselect*[$V$] selects the initial cluster node. The cluster set X is increased by adding to it adjacent nodes. Procedure *adj*[i] returns the nodes adjacent to node i. In particular, *adj*[IS(1)] returns the nodes adjacent to the initial node. Procedure *nextadj*[IS, AS(i)] returns all the nodes adjacent to node IS($i+1$) not contained in X. Procedure *nextselect*[AS(i)] selects the next iterating node in AS(i) according to a heuristic criterion described below. Procedure *update*[$G(V, E)$] stores the subgraph $G_1(V_1, E_1)$ and returns the subgraph $G_2(V_2, E_2)$.

Graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are defined according to the partitioning problem and the augmentation strategy required. The strategy follows the augmentation rules presented in the previous section and is reported in detail in [Demi83d].

The cluster criterion is satisfied when at least one of the following conditions is met:

$$|\mathrm{AS}(i)| = 0$$

$$\text{block area} \geq \text{maximum block area}$$

$$\mathrm{OF}(i) \text{ is a local minimum.}$$

The first condition guarantees that a cluster if found in graph $G(V, E)$ is not connected. The second condition allows the user to define the maximum size of each block according to the technological constraints of the implementation of the partitioned array. The third condition is a heuristic rule to determine a cluster. It can be also required that $\mathrm{OF}(i)$ is smaller than a proper fraction of the initial area $\mathrm{OF}(0)$ to ensure that partitioning is performed only if it gives a considerable savings in the total area. Since the objective function vector may have several local minima close to one another, the cluster decision can be taken a few steps after the minimum is detected.

Procedure *nextselect* uses a greedy strategy to select the next iterating node among the nodes in $\mathrm{AS}(i)$. When any node in $\mathrm{AS}(i)$ is added to the cluster node set X, graph $G(V, E)$ can be partitioned according to the augmentation rules and the corresponding value of the objective function can be computed. The selected node is the one that minimizes the objective function at that step of the algorithm. This means that the selected node is the "local best" node.

Procedure *inselect* returns the initial iterating node. As pointed out in [Demi83d], a node connecting two clusters is a bad selection for an initial node. Nodes with degree 1 cannot join two clusters and hopefully the lower the degree of the node, the lower is the probability of choosing a "bad" node. Hence procedure *inselect* returns the min-degree node in the actual implementation of the algorithm.

It is shown in [Demi83d] that the time complexity of the algorithm is polynomially bounded, even if the total number of nodes may increase at each iteration.

5.3. SMILE

SMILE is an interactive program for programmable logic array partitioning. The PLA description is given as input to the program in the form of a personality matrix. The output file of logic minimizer POP,

by adding appropriate nodes to the graph) until the edge set E is partitionable into E_1 and E_2 and $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are disjoint. Subgraph $G_1(V_1, E_1)$ is stored and the algorithm reattempts a cluster search on the updated graph $G(V, E) = G_2(V_2, E_2)$.

The cluster search in graph $G(V, E)$ is based on the *contour tableau* approach described in [Ogbu70] and in [Sang77]. The contour tableau is an array of three columns. The first one is called *iterating set* (IS); its entries are nodes of the graph. The second one is the *adjacency set* (AS); its entries are sets of nodes of the graph. The third column is the *objective function* (OF) vector; in this particular case its entries are the values of the area estimates. An area estimate can be easily obtained from the knowledge of subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ as shown in [Demi83d].

The tableau is built iteratively until a cluster is found and convenient conditions are met to separate it from the rest of the graph. At this point the tableau is cleared and the algorithm restarts on the rest of the graph. The algorithm is described in Pidgin C.

Partitioning Algorithm

```
while ( V ≠ ∅ ) {
        IS = ∅; AS = ∅; OF = ∅;
        i = 1;
        IS (i ) = inselect( V );
        AS (i ) = adj( IS (i ) );
        while ( cluster criterion not satisfied ) {
                IS (i + 1) = nextselect( AS (i ) );
                AS (i + 1) = nextadj( IS, AS (i ) );
                i = i + 1;
        };
        G (V, E ) = update( G (V, E ) );
        };
}.
```

The algorithm constructs step by step a cluster set $X = \bigcup_{j=1}^{i} \mathrm{IS}(j)$. Procedure *inselect*[$V$] selects the initial cluster node. The cluster set X is increased by adding to it adjacent nodes. Procedure *adj*[i] returns the nodes adjacent to node i. In particular, *adj* [IS(1)] returns the nodes adjacent to the initial node. Procedure *nextadj* [IS, AS(i)] returns all the nodes adjacent to node IS($i+1$) not contained in X. Procedure *nextselect*[AS(i)] selects the next iterating node in AS(i) according to a heuristic criterion described below. Procedure *update*[$G(V, E)$] stores the subgraph $G_1(V_1, E_1)$ and returns the subgraph $G_2(V_2, E_2)$.

Graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are defined according to the partitioning problem and the augmentation strategy required. The strategy follows the augmentation rules presented in the previous section and is reported in detail in [Demi83d].

The cluster criterion is satisfied when at least one of the following conditions is met:

$$|\mathrm{AS}(i)| = 0$$

$$\text{block area} \geq \text{maximum block area}$$

$$\mathrm{OF}(i) \text{ is a local minimum.}$$

The first condition guarantees that a cluster if found in graph $G(V, E)$ is not connected. The second condition allows the user to define the maximum size of each block according to the technological constraints of the implementation of the partitioned array. The third condition is a heuristic rule to determine a cluster. It can be also required that $\mathrm{OF}(i)$ is smaller than a proper fraction of the initial area $\mathrm{OF}(0)$ to ensure that partitioning is performed only if it gives a considerable savings in the total area. Since the objective function vector may have several local minima close to one another, the cluster decision can be taken a few steps after the minimum is detected.

Procedure *nextselect* uses a greedy strategy to select the next iterating node among the nodes in $\mathrm{AS}(i)$. When any node in $\mathrm{AS}(i)$ is added to the cluster node set X, graph $G(V, E)$ can be partitioned according to the augmentation rules and the corresponding value of the objective function can be computed. The selected node is the one that minimizes the objective function at that step of the algorithm. This means that the selected node is the "local best" node.

Procedure *inselect* returns the initial iterating node. As pointed out in [Demi83d], a node connecting two clusters is a bad selection for an initial node. Nodes with degree 1 cannot join two clusters and hopefully the lower the degree of the node, the lower is the probability of choosing a "bad" node. Hence procedure *inselect* returns the min-degree node in the actual implementation of the algorithm.

It is shown in [Demi83d] that the time complexity of the algorithm is polynomially bounded, even if the total number of nodes may increase at each iteration.

5.3. SMILE

SMILE is an interactive program for programmable logic array partitioning. The PLA description is given as input to the program in the form of a personality matrix. The output file of logic minimizer POP,

Table 5.1. **PLAs Partitioned by SMILE**

Normalized partitioned array areas. Initial area = 100. Maximum number of clusters allowed: 5.				
PLA	size P*(N+M)	Input Partitioning	Output Partitioning	Parallel Partitioning
PLA 1	6*(6+4)	71	64	61
PLA 2	16*(4+16)	100	71	65
PLA 3	30*(19+10)	78	81	67
PLA 4	75*(35+29)	75	70	46
PLA 5	62*(24+14)	75	80	60
PLA 6	84*(27+10)	71	81	59
PLA 7	84*(27+10)	69	81	57

described in Section 3, can be used as input to SMILE. Partitioning instructions are entered in the program along with the personality in the input file. Input, output, or parallel partitioning can be requested. The program performs input and output augmentations by default. In the case of output partitioning, product augmentations can be allowed.

The user can stipulate a limit on the number of clusters, i.e., the number of subarrays in which a plane (or both planes) is partitioned as well as the maximum size of the subarrays.

SMILE generates an output file containing a symbolic matrix, representing the personality of the partitioned array. This matrix is suitable to be processed by a *silicon assembler* program, which generates the mask layout of the array according to a given technology. Note that the symbolic array is technology independent.

Some results of partioning large industrial arrays are reported in Table 5.1.

6. AUTOMATED PLA LAYOUT

PLA layout generation programs have been designed using a number of paradigms [Glas80, Hofm80, Land81, Mayo84]. The most common

approach is termed *tiling*, in which a one-for-one conversion is performed from a simple, symbolic input format, like the character format used in earlier sections, to a layout mask set of the PLA [Mah84, Mayo84]. This approach has the advantages of speed, simplicity, and locality. However, if the PLA templates are invariant, simple tiling approaches can lead to poor PLA performance or, in fact, PLAs which do not work. This is because if a tile is designed to work correctly for a small PLA, it may in fact not perform adequately when used in a larger PLA. A common problem for these programs is caused by voltage drops along diffusion ground lines and often a very conservative approach must be used to avoid such problems. On the other hand, if the PLA template cells are implemented by using a procedural approach, and a number of service routines are available for the cells to enquire about their environment, then they can "self-adapt" and provide a more effective design.

The basic idea of a tiling program is to parse an ASCII input file and produce mask geometry tile on a character-by-character basis. Figure 6.1 shows sample invariant tiles for AND- and OR-planes in *n*-MOS technology. Another advantage of a tile-based approach is that the technology-related information is encapsulated in the tiles, not in the program that assembles the tiles. As design rules change only a distinct set of small cells must be modified.

Another approach is to use a *prototype* PLA and adapt it via a set of rules to other arrays [Glas80]. In this approach, very efficient designs can be implemented provided a large number of rules are specified. These

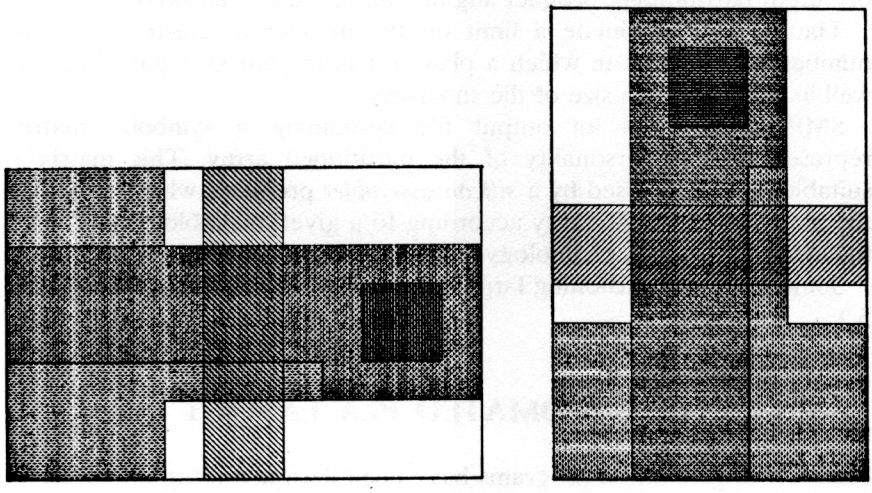


Figure 6.1. Sample invariant tiles.

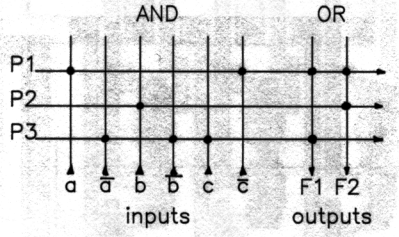
rules are, by nature, global since they refer to the entire PLA prototype. Therefore, at this stage, a large amount of code must be rewritten to move to a different PLA design style. However, the prototype-based approach holds a great deal of potential for future use of rule-based expert systems.

The third approach is similar to the prototype-based approach in that it takes a global view of the circuit [Hofm80, Land81]. The construction of input, output, and product terms is carried out globally, while the actual placement of transistors is a local process. This latter approach has the advantages of an electrically better design due to global considerations and generates much smaller output files even for moderate PLAs. It is also quite fast. The PLAID program uses this latter approach and is described below. PANDA, a tile-based program for constructing CMOS PLAs, is then presented.

6.1 The Generation of Static *n*-MOS PLAs by Using PLAID

PLAID [Hofm80] is capable of generating singly-folded or split PLAs as well as conventional arrays. It should be emphasized that the symbolic

(a)



(b)

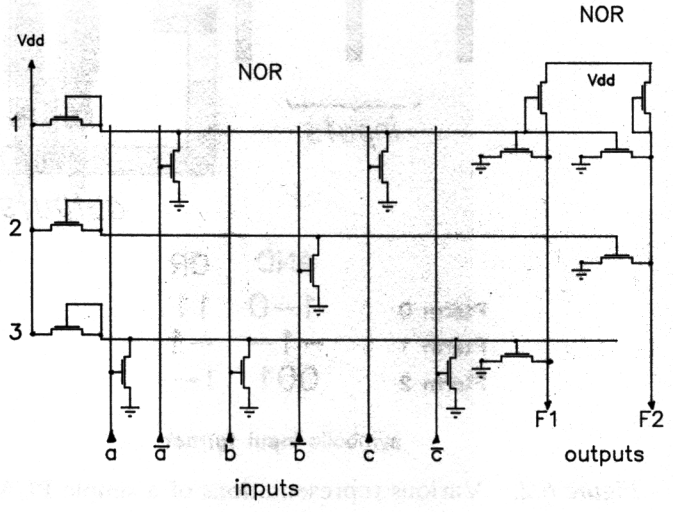
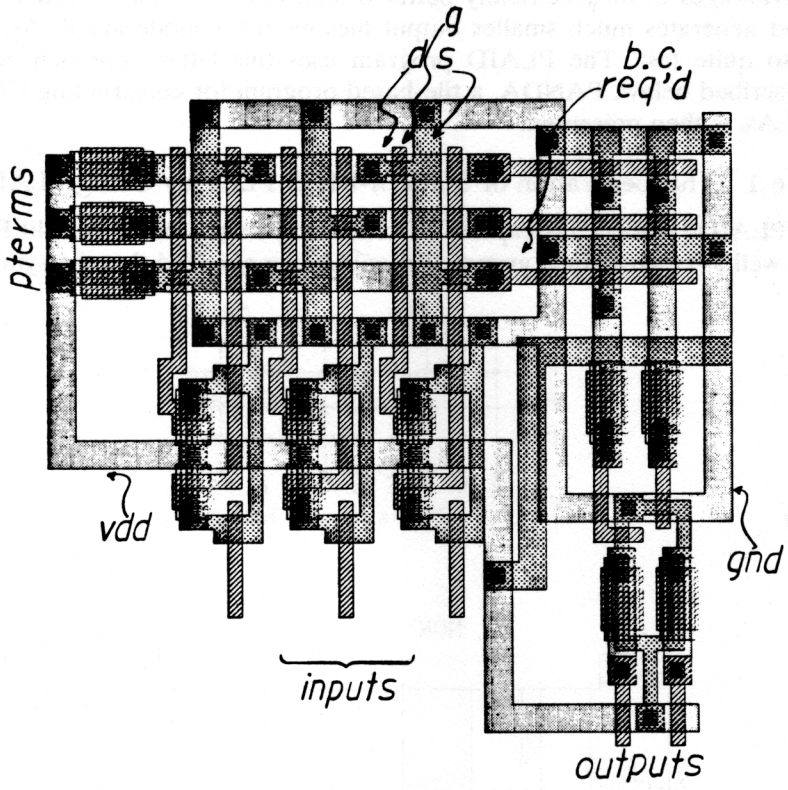


Figure 6.2(a) and (b).

interchange format used by topological compaction programs and read as input by generators such as PLAID must be able to convey folding information as well as the PLA's personality. For simplicity the generation of simple two-plane PLAs will be described. Figure 6.2(c) shows a plot of a PLA generated by the PLAID program which corresponds to the mixed notation schematic of Figure 6.2(b). The left plane is the input array, the right plane is the output array. Again note that both planes are made up of NOR gates in this case. The horizontal connecting runs are the sum terms. The operation of each plane is explained separately.



(c)

	AND	OR
Pterm 0	1−0	11
Pterm 1	−1−	−1
Pterm 2	001	1−

(d) symbolic input format

Figure 6.2. Various representations of a simple PLA.

Input plane structure. An input signal, entering the bottom of the plane [Figure 6.2(c)], feeds through a pair of inverter/buffers. This block serves the dual purpose of proving a complementary signal to the first NOR plane and of isolating the PLA from surrounding circuitry also using the signals. The output of these buffers drives the input plane. For each input there are three vertical runs: polysilicon, diffusion, and polysilicon. The left polysilicon carries the signal while the righty run carries the complement of the signal. The center diffusion line is anchored to a skirt of metal, to be connected to ground. The horizontal metal runs traversing the two planes are the sum terms, one end of which is connected through a depletion-load pullup to a metal bus, to be connected to V_{dd}.

To program this plane (i.e., define its personality) a diffusion run is used to connect the center diffusion band with the metal sum term via a diffusion–metal contact. Either the left or the right polysilicon run interrupts this path, forming the gate of a pulldown device. The diffusion–metal contact forms a drain button with the sum term; the center band acts as common source. If the left (noninverting) polysilicon controls the transistor a positive logic 0 is formed (this provides the signal inversion necessary in the NOR–NOR representation, where the sense of the input signal must be inverted). Right-side polysilicon as gate provides the complementary function. A sum term may connect to an input, or its complement, or it may not depend on the input at all.[1]

Output plane structures. As the sum terms cross into the output array they form contacts with polysilicon runs. A butting contact is formed to save area since it is sometimes necessary to connect diffusion and metal in programming the input plane. A butting or buried connection essentially serves the purpose of two separate (one poly–metal, one diffusion–metal) contacts. Figure 6.2(c) shows a situation in which such a connection is required.

The sum terms, now in polysilicon, are used to control pulldown transistors in the output plane. Diffusion runs at ground potential are interspersed between every other sum term. A transistor is formed when a sum term gate interrupts a diffusion run from ground potential to output term. The output terms are formed on the metal line and the pulldown transistor is again formed with the aid of a diffusion–metal contact. The output terms are driven via a depletion-load transistor situated at the bottom of the output plane.

In the output plane, since a complementary sum term is not available, only connection or no connection to the sum terms is possible. The output from this plane is inverted in accordance with the AND–OR to NOR–NOR transform. The inverter also buffers the PLA output.

PLAID reads a symbolic description of a possibly singly-folded or split

personality matrix and produces mask geometries in CIF of an *n*-MOS, single-metal, single-poly, NOR–NOR PLA. It makes a single pass over the input. Each symbol that PLAID reads is translated directly into a call on a CIF symbol. PLAID understands symbols not only for the core of the PLA but also for placement of input and output buffers and depletion-load pullups. In addition, PLAID supports the translation of label information. Inputs, outputs and product terms are automatically labeled.

The program begins by placing the core devices in the AND and OR planes. During this process it builds up a set of housekeeping vectors which describe the extent of signal runs from the top and bottom of the PLA core. These vectors are interrogated for use in creation of the AND and OR grid, which is the next generation step. Input lines, run in polysilicon, and truncated after the most distant gate to reduce capacitance. Output runs are similarly truncated. A separate procedure generates the product-term lines, which run in metal over AND planes and in poly over the OR plane. Once again the extent of the poly run may be truncated on the basis of information in the housekeeping vectors.

Input and output drivers are placed as they are encountered on read-in. Interarray buffers are not employed. By editing the input format the user may have an input or output buffer placed either at the top or bottom side or both of the PLA. The placement of drivers is controllable on a *per signal* basis. This flexibility greatly eases routing, especially from a folded PLA, to external modules.

After the grid and transistor matrix has been created the next step is generation of the power and ground skirt. Prior to actual placement of mask geometries a calculation is done to determine the width of the power and ground lines and the spacing of additional ground lines in the core, if required. Extra ground lines are provided to reduce the *IR* drop inside the PLA. The last step in generation is encapsulation of the geometries into a root cell, translation of this cell to the origin and calculation of the cell's bounding box.

PLAID produces the PLAs core and buffers by tiling together leaf cells. Each leaf cell is actually a collection of small procedures (written in '*C*') that imitate the CIF constructs of *Box*, *Call*, and so forth. The advantage of embedding the CIF constructs in a programming language is that one can add additional features, notably conditional statements, to the low-level interchange format. Error checking, for example of negative box widths, can also be implemented.

PLAID supports a large number of options. Among these options are separate low-power flags for the AND and OR planes and the buffers. These flags take advantage of conditional CIF generation to resize FET

channel widths and adjust the power and ground rails to the buffer cells. Other options to PLAID allow the user to force the spacing of additional ground bars and to increase the size of power and ground skirts. Inputs and outputs to the PLA may be on the metal or polysilicon layers, at the user's request. Generated subcells are numbered according to the CIF2.0 convention, but may be given more descriptive alphabetic names following Berkeley CIF extensions. PLAID will generate routing frames, if requested. Routing frames, on the metal and polysilicon layers, tell routing tools where they may not route.

6.2. Multiply Folded CMOS PLA Generation by Using PANDA

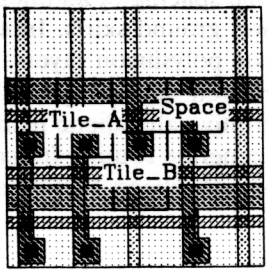
PANDA [Mah84] takes a symbolic description of a possibly folded PLA and produces CIF mask geometries as output. It uses the TPACK [Mayo84] package of tiling subroutines for generating PLAs. The TPACK subroutines manipulate rectangles and tiles defined in the PANDA template procedurally. The TPACK routines can add, subtract, label, and align corners of tiles in the template to other tiles. Other TPACK routines are capable of stretching tiles, adding and subtracting absolute points and rectangles, and creating and deleting tiles.

An example of PANDA's code constructing a set of tiles in the OR plane is shown in Figure 6.3. The rectangles that are drawn and the tiles that are placed by this code are shown next to the code defining their placement. Dark squares in the corners of the tiles show points of alignment. The subroutines *TPdisp* and *TPpaint* displace and draw tiles which are passed as parameters. The *TPdisp* subroutine uses only the boundary of a tile to control overlapping of tiles and placement of tiles to be drawn. *TPpaint* draws mask information contained within a tile.

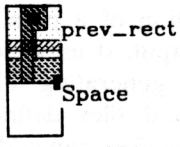
PANDA tiles a PLA according to an array that PANDA generates from a vertical and horizontal pass over the output personality matrix produced by the PLEASURE program. PANDA's array is then translated directly into the physical PLA. The size of PANDA's array is determined from control statements that specify the number of columns and rows for the core of the PLA. Because of the constraints imposed by the output of PLEASURE, the number of contact rows for multiple folding can also be calculated. When PANDA processes the input array, a new set of symbols are used, internal to PANDA. These symbols are described in Figure 6.4.

In PANDA's vertical pass over the input personality matrix, the personality matrix is filled with symbols that reflect the shortening of vertical lines in both the AND- and OR-planes to minimize capacitance. The first minimization is from the top, going down in each column.

Template Tiles

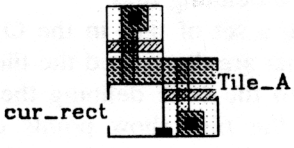



```
RECTANGLE cur_rect, prev_rect;
  /* drawn rectangles */
TILE Tile_A, Tile_B, Space;
  /* tiles to be drawn */
```



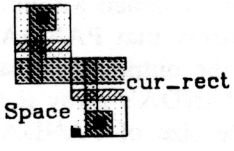

```
cur_rect = TPdisp_tile(Space,
  align(rLR(prev_rect), tUR(Space)))

  /* aligns lower right of prev_rect
       to upper right of Space */
```



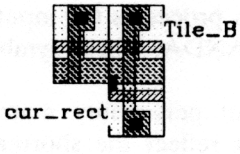

```
cur_rect = TPpaint_tile(Tile_A, out_tile,
  align(rLR(cur_rect), tLL(Tile_A)));

  /* aligns lower right of cur_rect
       to lower left of Tile_A */
```




```
cur_rect = TPdisp_tile(Space,
  align(rLL(cur_rect), tLL(Space)))

  /* aligns lower left of cur_rect
       to lower left of Space */
```




```
cur_rect = TPpaint_tile(Tile_B, out_tile,
  align(rUL(cur_rect), tLL(Tile_B)));

  /* aligns upper left of cur_rect
       to lower left of Tile_B */
```

Figure 6.3. Example of PANDA code.

Symbols for *AND* Plane		
Symbol in Left Column	Symbol in Right Column	Explanation
1	I	Normal contact
0	O	No contact, pass column signal
!	o	Split below
;	@	Fold to the right
:	#	Split below and fold to the right
e	f	Don't pass row signal (optimizing tiles)
.	,	Don't pass column signal (optimizing tiles)
z	y	Don't pass row and column (optimizing tiles)

Symbols for *OR* Plane	
Symbol	Explanation
J	Normal contact
Q	No contact, pass column signal
i	Split below
l	Fold to the right
j	Split below and fold to the right
E	Don't pass row signal (optimizing tiles)
~	Don't pass column signal (optimizing tiles)
Z	Don't pass row and column signal (optimizing tiles)

Additional Symbols	
Symbol	Explanation
*	Input buffer
+	Output buffer
X	No buffer
c	Contact within AND or OR plane
> <	Routing lines for contacts

Figure 6.4. **PANDA array symbols.**

Figure 6.5 shows an example of a PLEASURE file on the left and PANDA's first array on the right.

The resulting personality matrix is then processed a second time horizontally from the left to the right sides (and vice versa) to shorten horizontal lines, thus reducing capacitances in the PLA in the horizontal direction. This is shown in Figure 6.6 for the same example. The final personality matrix, after the horizontal pass, is then translated to layout geometry with each symbol representing a specific tile. Edge tiles and connecting middle tiles between planes are added for each row and the final output from PANDA for this example is shown in Figure 6.7.

Extra spacing between rows and columns is occasionally required for anomalies during folding. These anomalies can exist in a row within the AND-plane of a personality matrix when two logic rows are folded between two transistors that would normally share a contact. In the PLA

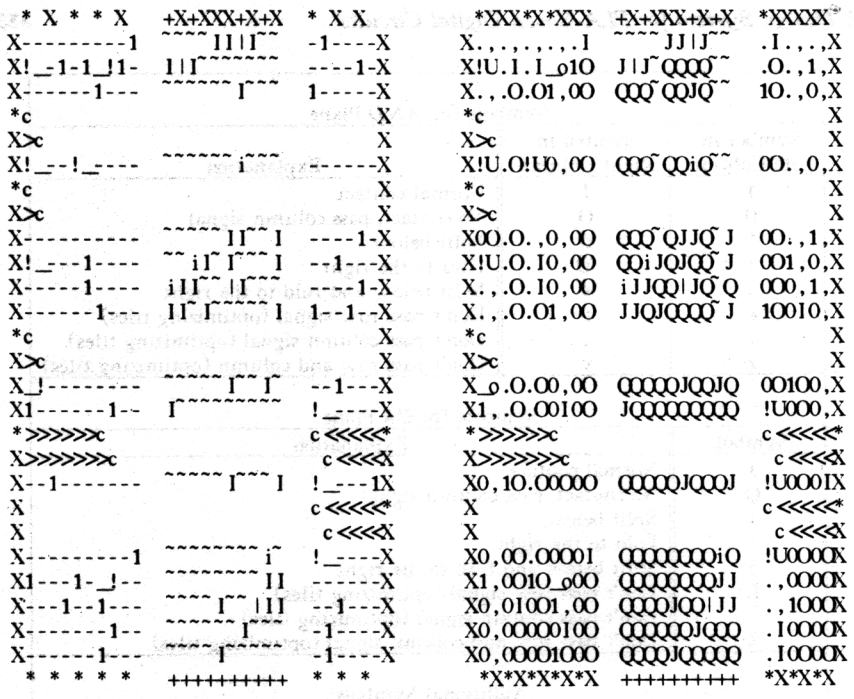


Figure 6.5. PLEASURE file and PANDA's first pass.

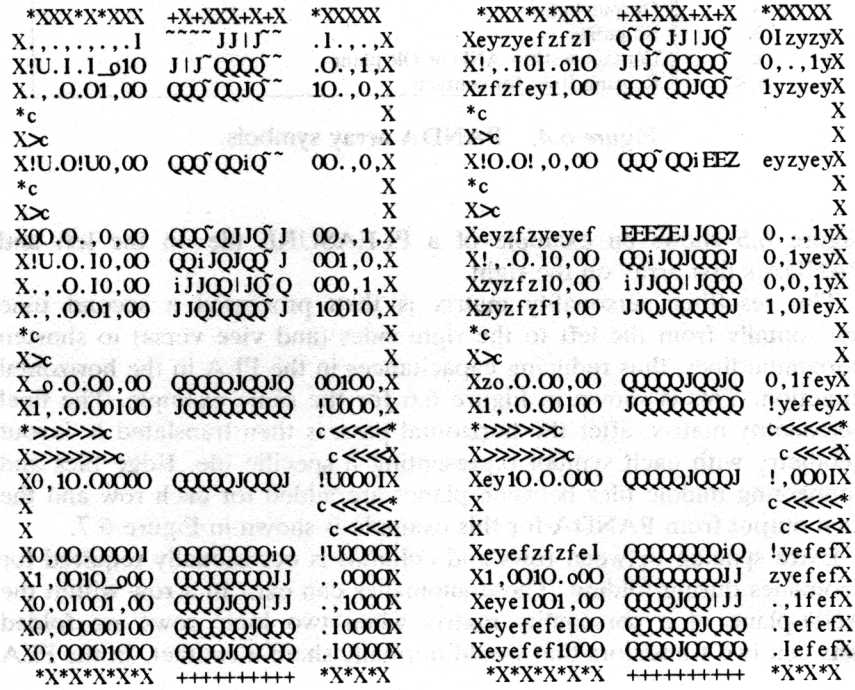


Figure 6.6. PANDA's first and second passes.

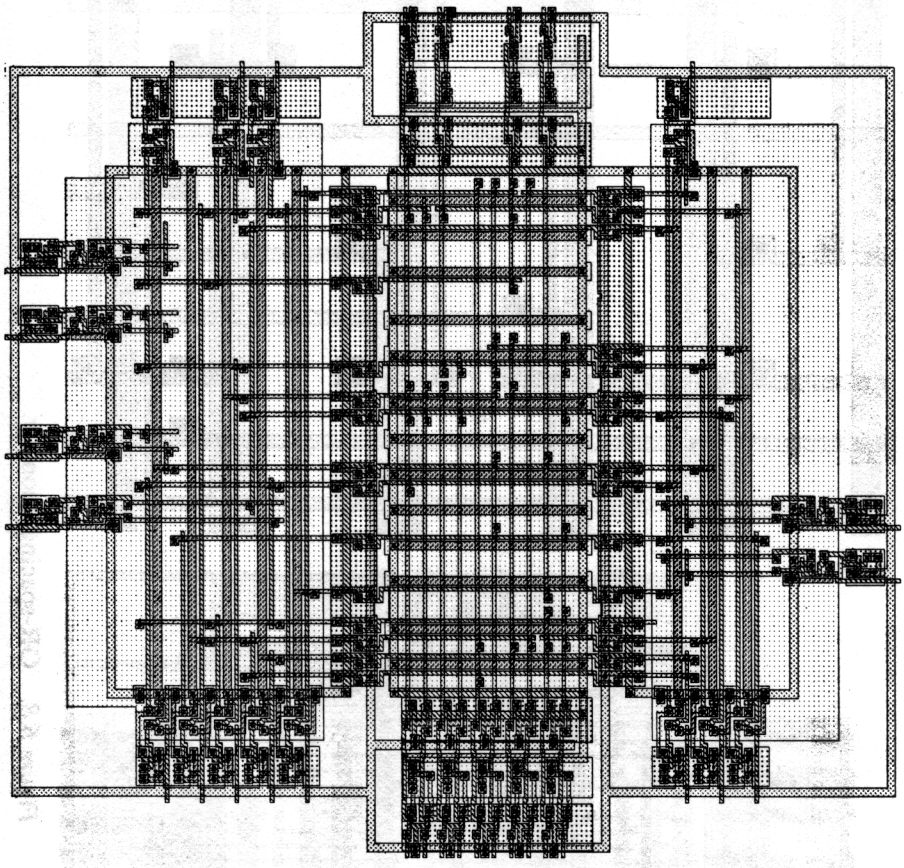


Figure 6.7. PANDA output.

in Figure 6.7, two logic rows were combined into one physical row, but the logic fold between these rows required the duplication of a contact plus some spacing between the contacts. This additional space is provided automatically by PANDA. This type of anomaly can also occur in the OR-plane along columns that are folded between adjacent transistors that would usually share a contact. This can also be seen in the OR-plane in the PLA of Figure 6.7 where PANDA automatically puts in the extra row of spacing.

Another situation in which extra spacing is required is in multiply-folding the columns of the output plane. The output buffers from the side of the PLA must contact columns in the core of the OR-plane by creating a contact–row. This contact–row extends from the side output buffer into the PLA until it makes contact with the appropriate column. This *output contact* connects the output buffer with the folded column

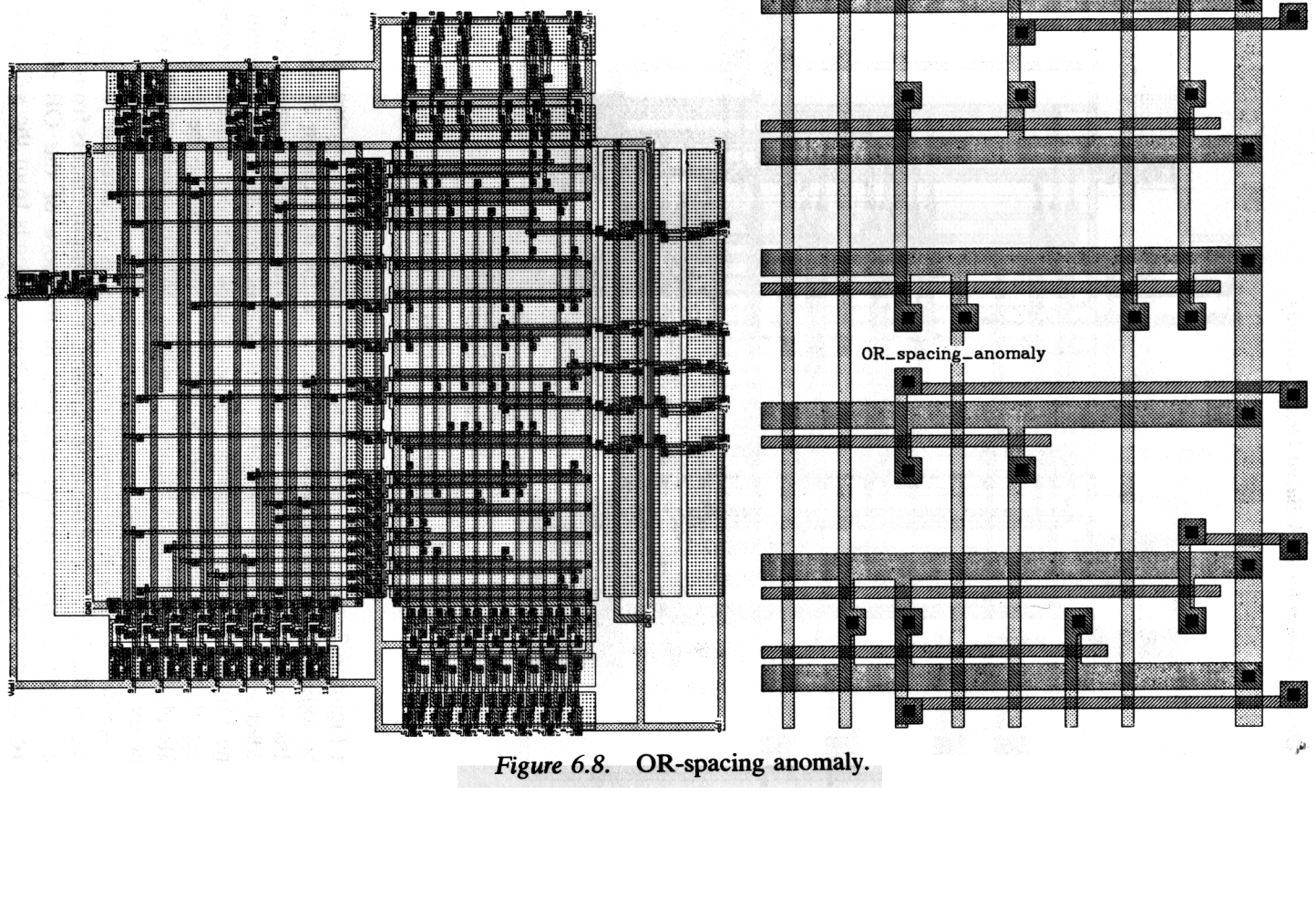


Figure 6.8. OR-spacing anomaly.

within the core of the OR plane. A product–row above this contact–row might have a transistor directly above the *output contact* in the contact–row. Without the extra spacing, the product–row transistor would be connected to the output contact below. This spacing anomaly is shown in Figure 6.8.

7. SUMMARY

Computer-aided synthesis of efficient combinational and sequential logic structures is a key requirement for VLSI design if both high layout productivity and high circuit performance are needed. The use of structured design techniques permits this synthesis problem to be formulated in such a way that powerful computer algorithms can be used to improve the layout efficiency and the performance of the final design. In particular, the use of possibly irregular, two-dimensional logic arrays is well suited to the integrated-circuit environment.

In this chapter, we have presented a number of techniques for implementing combinational logic using a particular array-based approach—the programmable logic array—and we have described a number of computer programs which implement the algorithms described. These programs have been used in the design of a number of industrial integrated circuits. In particular, techniques for two-level logic minimization, multiply-constrained folding, PLA partitioning, and automated layout of electrically corrected PLAs have been described.

Other important research areas that have not been described here include the relationships among these steps (e.g., how does a particular logic minimization affect the folding of the array?); the use of PLAs for sequential circuit design, including such issues as optimal state assignment for a PLA-based implementation; and the application of this synthesis process to other array-based design techniques, such as gate-matrix SLA and Weinberger arrays. These research topics are presently pursued in the CAD of the Integrated Circuit Group of the University of California at Berkeley.

APPENDIX 1: BERKELEY STANDARD PLA INTERCHANGE FORMAT

This format is used by programs which manipulate PLAs to describe the physical implementation. Lines beginning with a # are components and are ignored. Lines beginning with a . contain control information about

the PLA. Currently, the control information is given in the following order:

.i ⟨number of inputs⟩
.o ⟨number of outputs⟩
.p ⟨number of product terms⟩

What follows then is a description of the AND and OR planes of the PLA with one line per product term. Connections in the AND plane are represented with a 1 for connection to the noninverted input line and a 0 for connection to the inverted input line. No connection to an input line is indicated with x, X, or −, with − being preferred. Connections in the OR plane are indicated by a 1, with no connection being indicated with x, X, −, or 0, with − being preferred. Spaces or tabs may be used freely and are ignored.

The end of the PLA description is indicated by

.e

Programs capable of handling split and folded arrays employ the following format:

AND-plane

(1) Contact to input	(2) No contact to input	
(1)	(2)	
1	−	Normal contacts, no splits or folds
!	−	Split below
;	,	Fold to right
:	.	Split below and fold to right

OR-plane

(1) Contact to output	(2) No contact to output	
(1)	(2)	
1	~	Normal contacts, no splits or folds
i	=	Split below
|	,	Fold to right
j	"	Split below and fold to right

Additional elements

*	Input buffer
+	Output buffer
D	Depletion load associated with product term
N	No depletion load associated with product term
c	Contact to a multiply-folded split
v, ^, ⟨, ⟩	Routing lines to contact multiple folds

Note that the decoding function of the AND-plane is separate from the specification of its connectivity. This makes the AND- and OR-plane specifications identical.

These programs handle the following more general set of parameters:

.i	⟨number of inputs⟩
.o	⟨number of outputs⟩
.p	⟨number of product terms⟩
.top	⟨labels of inputs and outputs from the top
.left	⟨labels of inputs and outputs from the left
.right	⟨labels of inputs and outputs from the right
.bottom	⟨labels of inputs and outputs from the bottom

The first group of parameters must precede the second group. If there is only one AND- or OR-plane it is assumed to be the left one.

APPENDIX 2: PLAMAKER COMMAND FILE

A command file or *script* has been created which links together several PLA tools in an automated pipeline. Input to this script is a set of Boolean equations in EQNTOTT format. The output is a mask layout plot of the generated PLA as well as a set of intermediate files describing the various stages of the PLA's creation and compaction.

The script is invoked by:

plamaker boolfile

where **boolfile** contains two-level logic expression in AND–OR form as accepted by EQNTOTT. As each step in the PLA generation process completes the user is notified. Abnormal termination at any stage in the production causes the script to abort.

PLAMAKER Script Listing

```
#!/bin/csh -f
EQNTOTT ⟨$1⟩e.out
echo "plamaker: Equation conversion done."
POP ⟨e.out⟩p.out
echo "plamaker: Boolean minimization done."
PLEASURE p.out⟩b.out
echo "plamaker: Topological minimization done."
PLAID ⟨b.out⟩c.out
echo "plamaker: CIF layout generation done."
CIFPLOT c.out
```

NOTE

1. Making both connections is not permitted. It would have the effect of always asserting the sum term low, effectively removing the sum term from consideration in the output plane.

REFERENCES

[Arev78] Z. Arevalo and J. G. Bredeson, A method to simplify a Boolean function into a near-minimal sum-of-products for programmable logic arrays. *IEEE Trans. Comput.* **C-27**, 1028–1039, 1978.

[Bray82] R. Brayton, G. D. Hachtel, L. Hemachanandra, A. R. Newton, and A. L. Sangiovanni-Vincentelli, A comparison of logic minimization strategies using espresso. An APL program package for partitioned logic minimalization. In Proceedings, International Symposium on Circuits and Systems, Rome, 1982, pp. 43–49.

[Bray84a] R. K. Brayton et al., ESPRESSO-II: A new logic minimizer for programmable logic arrays. In Proceedings, Custom Integrated Circuits Conference, Rochester, 1984.

[Bray84b] R. K. Brayton et al., *ESPRESSO-II: Logic Minimization Algorithms for VLSI Synthesis*. Kluwer, The Netherlands, 1984.

[Brow81] D. W. Brown, A state-machine synthesizer—SMS. In Proceedings, Design Automation Conference, Nashville, 1981, pp. 301–304.

[Chuq82] S. Chuquillanqui and T. Perez Segovia, PAOLA: A tool for topological optimization of large PLAs. In Proceedings, Design Automation Conference, Las Vegas, 1982, pp. 300–306.

[Cook79] P. W. Cook, S. E. Shuster, J. T. Parrish, V. Di Lonardo, and D. R. Freedman, 1 μm MOSFET VLSI technology: Part III—Logic circuit design methodology and applications. *IEEE Trans. Electron Devices* **ED-26**, 333–345, 1979.

[Coop80] M. Cooperman, High speed current mode logic for LSI. *IEEE Trans. Circuits Syst.* **CAS-27**, 626–635, 1980.

[Demi81] G. De Micheli, Pleasure: A program for topological compaction of PLAs. Internal Report, Harris Corporation, 1981.

[Demi83a] G. De Micheli and A. L. Sangiovanni-Vincentelli, Multiple folding of programmable logic arrays. In Proceedings, International Symposium on Circuits and Systems, 1983, pp. 1026–1029.

[Demi83b] G. De Micheli and A. L. Sangiovanni-Vincentelli, PLEASURE: A computer program for simple/multiple constrained/unconstrained folding of programmable logic arrays. In Proceedings, Design Automation Conference, 1983, pp. 530–537.

[Demi83c] G. De Micheli and A. L. Sangiovanni-Vincentelli, Multiple constrained folding of programmable logic arrays: Theory and applications. *IEEE Trans. Comput. Aided Design* **CAD-2**, 167–180, 1983.

[Demi83d] G. De Micheli and M. Santomauro, SMILE: A computer program for partitioning of programmed logic arrays. *Comp. Aided Design* **2**, 89–97, 1983; UCB/ERL Memorandum No. 82/74.

[Demi83e] G. De Micheli and M. Santomauro, Topological partitioning of programmable logic arrays. In Proceedings, International Conference on Computer-Aided Design, Santa Clara, 1983, pp. 182–184.

[Egan82] J. R. Egan and C. L. Liu, Optimal bipartite folding of PLAs. In Proceedings, Design Automation Conference, Las Vegas, 1982, pp. 141–146.

[Fang83] S. Fang, High speed bipolar PLA design techniques. Ph.D. Thesis, University of California, Berkeley, 1983.

[Flei75] H. Fleisher and L. I. Maissel, An introduction to array logic. *IBM J. Res. Dev.*, **19**, 98–109, 1975.

[Glas80] L. A. Glasser and P. Penfield, Jr., An interactive PLA generator as an archetype for a new VLSI design methodology. In Proceedings, International Symposium on Circuits and Systems, 1980, pp. 608–611.

[Gras82] W. Grass, A depth-first branch-and-bound algorithm for optimal PLA folding. In Proceedings, Design Automation Conference, Las Vegas, 1982, pp. 133–140.

[Gree76] D. L. Greer, An associative logic matrix. *J. Solid State Circuitry* **SC-11**, 679–691, 1976.

[Hach80] G. D. Hachtel, A. L. Sangiovanni-Vincentelli, and A. R. Newton, An algorithm for optimal PLA folding. In Proceedings, International Conference on Circuits and Computers, New York, 1980, pp. 1023–1028.

[Hach82a] G. D. Hachtel, A. R. Newton, and A. L. Sangiovanni-Vincentelli, An algorithm for optimal PLA folding. *IEEE Trans. Comput. Aided Design* **CAD-1**, 63–77, 1982.

[Hach82b] G. D. Hachtel, A. R. Newton, and A. L. Sangiovanni-Vincentelli, Techniques for programmable logic array folding. In Proceedings, Design Automation Conference, Las Vegas, 1982, pp. 147–152.

[Henn83] J. Hennessy, Partitioning programmable logic arrays summary. In Proceedings, International Conference on Computer-Aided Design, Santa Clara, 1983, pp. 180–181.

[Hofm80] M. Hofmann, A method for topological compaction of programmed logic arrays. M.S. Report, University of California, Berkeley, 1980.

[Hong74] S. J. Hong, R. G. Cain, and D. L. Ostapko, MINI: A heuristic approach for logic minimization. *IBM J. Res. Dev.*, **18**, 443–458, 1974.

[Hu83] T. C. Hu and Y. S. Kuor, Graph folding and programmable logic arrays. University of California, San Diego, Computer Science Technical Report No. CS-71, 1983.

[Joha79] D. Johannsen, Bristle blocks: A silicon compiler. In Proceedings, Design Automation Conference, 1979, pp. 310–313.

[Kang81] S. Kang, Automated synthesis of PLA-based systems. Ph.D. Thesis, Stanford University, 1981.

[Kang83] S. M. Kang, R. H. Krambeck, H-F. S. Law, and A. D. Lopez, Gate matrix layout of random control logic in a 32-bit CMOS CPU adaptable to evolving logic design. *IEEE Trans. Comput. Aided Design* **CAD-2**, 18–29, 1983.

[Land81] H. Landman, Automatic layout of optimized PLA structures. M.S. Report, University of California, Berkeley, 1981.

[Law82] H-F. S. Law and M. Shoji, PLA design for the BELLMAC-32A microprocessor. In Proceedings, International Conference on Circuits and Computers, 1982, pp. 161–164.

[Lin81] C. Lin, A 4μm NMOS NAND structure PLA. *J. Solid State Circuitry* **SC-16**, 103–107, 1981.

[Luby82] M. Luby, U. Vazirani, V. Vazirani, and A. L. Sangiovanni-Vincentelli, Some theoretical results on the optimal PLA folding problem. In Proceedings, International Conference on Circuits and Computers, New York, 1982, pp. 165–170.

[Mah84] G. Mah, PANDA—A PLA generator for multiply folded arrays. M.S. Report, University of California, Berkeley, 1984.

[Mayo84] R. N. Mayo, Combining graphics and procedures in a VLSI layout tool: The tpack system. M.S. Report, University of California, Berkeley, 1984.

[Mccl56] E. J. McCluskey, Minimization of Boolean functions. *BELL Syst. Tech. J.* **35**, 1417–1444, 1956.

[Mead80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

[Morr70] E. Morreale, Recursive operators for prime implicant and irredundant normal form determination. *IEEE Trans. Comput.* **C-19**, 504, 1970.

[Newt81a] A. R. Newton, Computer-aided design of VLSI circuits. *Proc. IEEE*, **69**, 1189–1199, 1981.

[Newt81b] A. R. Newton, D. O. Pederson, A. L. Sangiovanni-Vincentelli, and C. H. Sequin, Design aids for VLSI: The Berkeley perspective. *IEEE Trans. Circuits Syst.* **CAS-28**, 666–680, 1981.

[Ogbu70] E. C. Ogbuobiri, W. F. Tinney, and J. W. Walker, Sparsity-directed decomposition for Gaussian elimination on matrices. *IEEE Trans. Power Appl. Syst.* **PAS-89**, 141–150, 1970.

[Pail81] J. F. Paillotin, Optimization of the PLA area. In Proceedings, Design Automation Conference, Nashville, 1981, pp. 406–410.

[Pati79] S. Patil and T. Welch, A programmable logic approach for VLSI. *IEEE Trans. Comput.* **C-28**, 594–601, 1979.

[Proe76] R. Proebsting, *Electronics*, 82, 1976.

[Rhyn77] V. T. Rhyne et al., A new technique for the fast minimization of switching functions. *IEEE Trans. Comput.* **C-26**, 757–764, 1977.

[Roth58] J. P. Roth, Algebraic topological methods for the synthesis of switching functions. *Trans. Am. Math. Soc.* **88**, 301–326, 1958.

[Roth80] J. P. Roth, *Computer Logic, Testing and Verification*, Computer Science, 1980.

[Rude84] R. Rudell, ESPRESSO-IIC User's Manual. University of California, Berkeley, 1984.

[Sang77] A. L. Sangiovanni-Vincentelli, L-K. Chen, and L. O. Chua, An efficient cluster algorithm for tearing large-scale networks. *IEEE Trans. Circuits Syst.* **CAS-24**, 709–717, 1977.

[Shoj82] M. Shoji, Electrical design of the BELLMAC-32A microprocessor. In Proceedings, International Design Conference on Circuits and Computers, 1982, pp. 112–115.

[Sign79] Signetics Bipolar and MOS Memory Data Manual, pp. 156–188, 1979.

[Sima83] P. Simanyi, A. R. Newton, and A. L. Sangiovanni-Vincentelli, The POP PLA optimizer. University of California, Berkeley, CAD Group User's Guide, 1983.

[Souk81] J. Soukup, Circuit layout. *Proc. IEEE* **69**, 1281–1304, 1981.

[Spat80] H. Spath, *Cluster Analysis Algorithms*, Ellis Horwood, 1980.

[Stil83] D. W. Still, A 4.0 ns laser customized PLA with pre-program test capability. In ISSCC Digest of Technical Papers, 1983, p. 154.

[Suwa81] I. Suwa and W. J. Kubitz, A computer aided design system for segment-folded PLA macro cells. In Proceedings, Design Automation Conference, Nashville, 1981, pp. 398–405.

[Torn72] H. C. Torng, *Switching Circuits—Theory and Logic Design*, Addison-Wesley, Reading, Massachusetts, 1972.

[Wein67] A. Weinberger, Large scale integration of MOS complex logic: A layout method. *J. Solid State Circuitry* **SC-2**, 182–190, 1967.

[Wood79] R. A. Wood, A high density programmable logic array chip. *IEEE Trans. Comput.* **C-28**, 602–608, 1979.