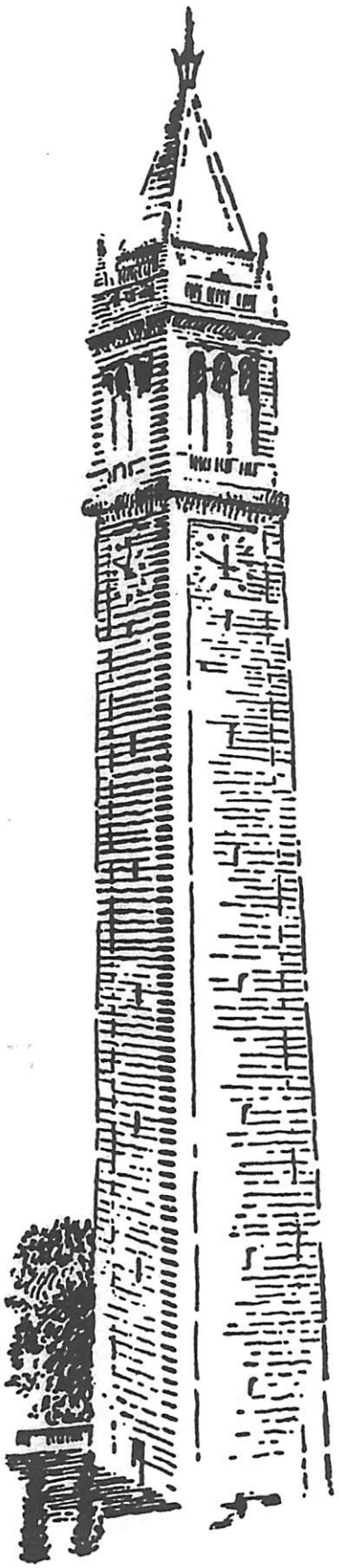


COMPUTER-AIDED SYNTHESIS OF PLA-BASED SYSTEMS

by

Giovanni De Micheli

Memorandum No. UCB/ERL M84/31

11 April 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley, CA 94720

COMPUTER-AIDED SYNTHESIS OF PLA-BASED SYSTEMS

by

G. DeMicheli

Memorandum No. UCB/ERL M84/31

11 April 1984

COMPUTER-AIDED SYNTHESIS OF PLA-BASED SYSTEMS

by

Giovanni De Micheli

Memorandum No. UCB/ERL M84/31

11 April 1984

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Computer-Aided Synthesis of PLA-Based Systems

By

Giovanni De Micheli

Engineer (Polytechnic Institute of Milan, Italy) 1979
M.S. (University of California) 1980

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Engineering

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

Approved: 12/1/1983
Date

.....

.....

.....

Computer-Aided Synthesis of PLA-Based Systems

by Giovanni De Micheli

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Prof. Alberto Sangiovanni Vincentelli, for introducing me to the exciting world of research. His knowledge, enthusiasm and friendship guided me through my graduate studies at U.C. Berkeley. He taught me patiently a rigorous research method and showed me how knowledge is perpetuated and enhanced in the university environment. His continuous availability to discuss scientific issues was valuable in maturing new concepts and ideas.

I would like to thank the faculty of the EECS department at U.C.Berkeley for having taught and advised me in several fields. In particular I wish to thank Prof. Richard Newton for the sharp criticisms and accurate advices that helped me in understanding some critical issues of Computer-Aided Design. I thank Prof. Don Pederson, Prof. Charles Desoer and Prof. Shankar Sastry for many stimulating discussions. I thank Prof. Leo Harrington of the Math department for reading this thesis.

I wish to express my gratitude to Dr. Bob Brayton, of IBM T.J. Watson Research Center. Several discussions on the role of symbolic minimization and graph embedding helped in developing the state assignment technique presented in this dissertation.

I wish to thank Prof. Mauro Santomauro of the EE department of Politecnico di Milano, who advised me during my undergraduate studies. In particular I acknowledge his contribution in formulating the PLA partitioning problem.

I thank Prof. Gary Hachtel of University of Colorado, Boulder for the invaluable discussions on the issues related to PLA folding and partitioning.

I would like to thank the colleague graduate students of the EECS department at U.C.Berkeley. Tiziano Villa contributed to the early efforts of understanding the state assignment problem. Duksoon Kay helped in maintaining and improving the PLEASURE program. Stimulating discussions with Jeffrey Deutsch, Jim Kleckner, Mark Hoffman, Ken Keller, Grace Mah, Dr. William Nye and Tom Quarles are acknowledged.

I express my gratitude for the financial support that made this research possible. In particular I acknowledge the IBM Fellowship for VLSI in the years 1982 and 1983, the Rotary International Fellowship for the academic year 1980/81 and the Fulbright Scholarship for the academic year 1979/80. I thank Harris Semiconductor Corporation for the opportunity of joining the Analog Division during fall 1981, and in particular Dr. John Cornell and Mr. James Spoto.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 VLSI DESIGN METHODS

Very Large Scale Integration (VLSI) circuits play a major role in the development of complex electronic systems. As a result of the progress of semiconductor technology, an increasing number of devices can be realized on a single chip.

A design method for large scale integrated circuits has to satisfy different requirements in order to cope with the increased design complexity. In particular a method based on a structured and hierarchical design supported by the use of computer aids can ensure functional correctness while maintaining a reasonable design time [SEQU83].

Computer-aided design of VLSI circuits addresses several goals such as automated synthesis, circuit optimization, design verification, system simulation and test pattern generation. A design system for VLSI circuits is an integrated set of design tools which allows an engineer to produce a design description ready for manufacturing by using his ingenuity in the creative design phases and where as many transformations among design representations as possible are automated.

Many different philosophies concerning the automated synthesis of VLSI systems have been proposed in recent years, e.g. [TRIM81] [NEWT81] [DIRE81], [DUTT81], [ALLE81] [BOSE83]. Research in the area of VLSI design systems at the University of California, Berkeley aims at creating a software

framework in which modular programs coexist and interact and can be easily updated according to the advances in semiconductor technology [NEWT81]. The design system is accessed by the circuit designer through a graphic-oriented design station [KELL82] [ELLI82] [KELL83] [OUST84]. A strong emphasis is placed on the optimization of the area, performances and power of the designed circuit. Therefore the designer monitors and steers the transformations among the design description by interacting with the design system and by having access to human-understandable representations of the circuit at the different levels of abstraction .

1.2 HIERARCHICAL DESIGN OF DIGITAL CIRCUITS

This dissertation addresses the synthesis of digital circuits. The design of a digital system can be viewed as a sequence of transformations of design representations at different levels of abstraction. A flow-chart showing the steps in the top-down synthesis of a digital circuit is included as Fig. 1.2.1.

The behavioral specifications of the system are described first by a **functional representation**. For example, Hardware Description Languages (HDL) such as ISP [BARB77],[BARB81], DDL [DIET68] and AHPL [HILL78], are commonly used. The functional representation is transformed into a **logic description** , consisting of a net-list of logic gates, including storage gates, that can be visualized by a logic schematic diagram. Subsequently a **topological description** is obtained by determining the mutual positions of the gates and the related interconnections. The next step is to specify an **electrical representation** , according to an implementation technology, which is eventually transformed into a **geometric layout** of the integrated circuit implementing the given functionality.

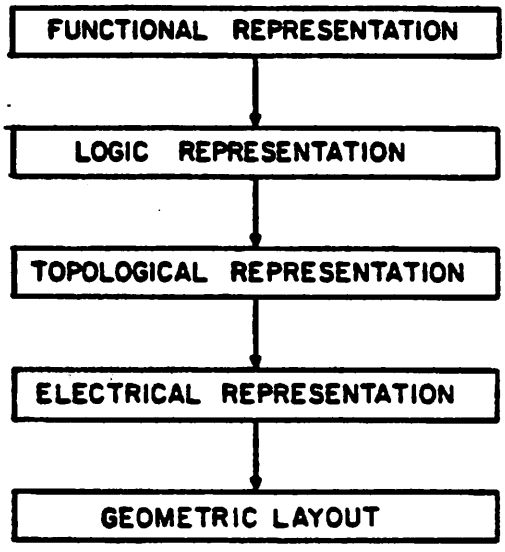


Fig. 1.2.1 Synthesis of a digital circuit

Nowadays it is considered impractical to perform these transformations by hand, because they are tedious, error-prone and very time-consuming. In an automated synthesis environment, the designer just monitors the transformations and verifies the practical feasibility of the design at each step. As a result the length of the design cycle is shortened, the associated development cost is reduced and the circuit reliability is enhanced greatly.

The transformations involved in the synthesis of a VLSI circuit depend on the design method used. Several "styles" are used in industrial VLSI design according to the function and the market of the circuit.

The objective of a fully **custom** design method is an *ad hoc* implementation. Therefore each transformation between design representation is optim-

ized for the particular circuit being designed. High performance implementations can be achieved. To date, computer-aided design techniques support custom design to a limited extent, because customized implementations are so complex that designers' experience cannot be replaced by automated transformations between design representation. As a consequence, custom design has a longer development time compared to other methods. Therefore custom design is profitable to date only for large volume production of complex systems, such as microprocessors or memories, or for circuits where special performance is required.

In a **gate-array** design method, a circuit is implemented in silicon by personalizing a master array of uncommitted gates using a set of interconnections [BLUM79]. Therefore design is constrained by the fixed structure of the master array and is limited to routing the interconnections. Computer-aids support widely gate-array design and complex circuits can be implemented on gate-arrays in short time. Gate-arrays are widely used, in particular for small volume production or for prototyping new designs.

The design of a VLSI circuit in a **standard-cell** (or **poly-cell**) design method requires partitioning the circuit into atomic units that are implemented by pre-committed cells [FELL76]. Therefore design includes placement and routing of the cells, that are supported by computer-aided design tools. The standard-cell and gate-array design methods alone do not support highly optimized circuits. However standard cell designs are more flexible than gate-array designs, but require longer development time.

The design of VLSI circuits, using parametrized modules, or **macro-cells**, bridges the gap between custom and standard-cell design and is compatible with both methods. Parametrized macro-cells can implement functional

units that are specified by design parameters and by their functionality [NEWT81]. Macro-cells are highly regular and structured. Therefore computer programs, called **module generators**, can produce the layout of a macro-cell from its functional description.

The macro-cell approach is very attractive because of its flexibility, that allows to exploit the advantages of both custom and standard-cell methods. Highly optimized and area-efficient modules can be designed in a short time. In particular, Programmable Logic Array macros [FLEI75] have shown to be very effective means of designing both combinational and sequential functions.

1.3 ARRAY LOGIC DESIGN

Weinberger proposed for the first time a regular one-dimensional array implementation of a digital circuit [WEIN67a]. A sketch is shown in Fig. 1.3.1. **Weinberger arrays** were designed to be implemented on a metal-gate PMOS technology, where just one level of interconnect was available (metal) in addition to diffusion. The gates are therefore placed along the position of a linear array, and metal horizontal segments are used to connect gates. Weinberger arrays provided a structured technique for implementing multiple-level logic functions. However the inherent one-dimensional structure of the array limited their applications.

A structured approach for the implementation of combinational logic functions can also be obtained by look-up tables [FLEI75]. In particular Read Only Memories (ROM) can be used to evaluate a logic function whose entries are stored in the memory locations corresponding to appropriate addresses. Read Only Memories are two-dimensional arrays and are implemented in several technologies. The implementation of a combinational logic function

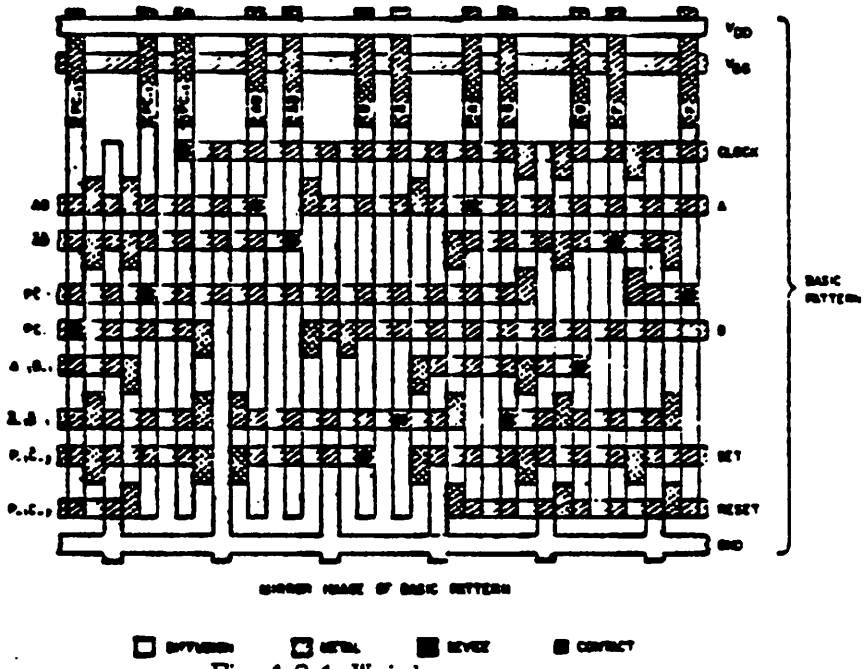


Fig. 1.3.1 Weinberger array

by means of a ROM yields a very structured design, which can be easily modified. Moreover the speed of operation can be fast and easily determined for the set of combinational logic functions requiring the same address space. Unfortunately a ROM implementation of a combinational function is not effective in terms of silicon area. A ROM implementation of a n-input function requires 2^n memory cells. On the other hand, a combinational function may not require the entire address space of a ROM because the function is not specified for some input combination. Look-up techniques that do not associate a single memory cell to every specified input combination, as Programmable Logic Arrays do, are more efficient in terms of silicon area.

Programmable Logic Arrays (PLAs) allow to exploit the advantages of the look-up table approach, while reducing the area penalty [PROE76]. In

particular, only a subset of input combinations, describing completely the switching function, is related to the physical implementation.

A Programmable Logic Array implements a two-level combinational function on a two dimensional array. Each position of the array is programmed by the presence or absence of a gate. Two orthogonal sets of segments provide connection between them (Fig. 1.3.2). The functionality of a PLA can be represented by a 0-1 matrix. Therefore design and optimization of a PLA can be performed on a symbolic array, and the physical layout can be obtained from it in a straight-forward way. Therefore PLAs are attractive building blocks of a structured design methodology [LOGU75].

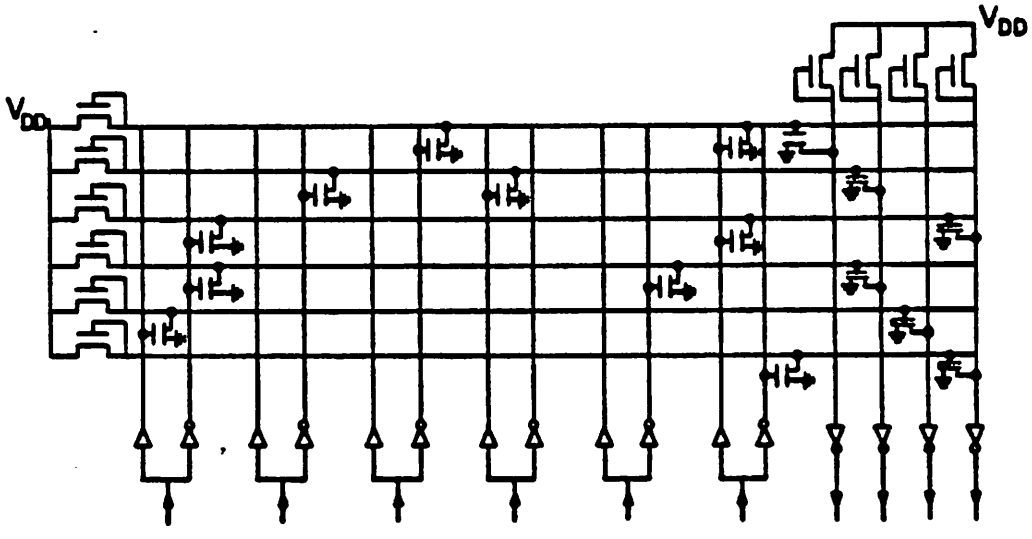


Fig 1.3.2 PLA integrated circuit implementation

Many industrial VLSI circuits, such as the Intel 8086, the Motorola 68000 and the Hewlett-Packard 32-bit micro-processors, use PLAs as building-blocks.

Programmable Logic Array allow arbitrary two-level logic function implementation, where inputs and their complements are available. Any combinational logic function can be represented by a two-level form such as *sum-of-products* or *product-of-sums* . Therefore any combinational logic function can be implemented by a PLA.

The trade-offs of two-level versus multiple-level implementations of combinational functions is a subject of intensive investigation [BRAY82b]. To date, there is no general criterion to decide how many levels of logic correspond to the most effective implementation in terms of silicon area and switching-time performance. However PLAs can also be used to implement multiple-level logic functions by cascading logic arrays [LOGU75].

Recently a more elaborate array architecture has been introduced and referred to as **gate matrix** [KANG83]. Gate matrices allow the implementation of multi-level switching functions on a two-dimensional array in CMOS technology. The gate matrix consists of a set of polysilicon columns intersecting a set of orthogonal diffusion rows. Gates are formed at appropriate intersections, and metal segments provide connections. The design of the *Bellmac* 32 microprocessor was based on the use of gate matrices [KANG83].

Sequential logic functions can be represented as **Finite State Machines** (FSMs) and implemented by a combinational and a storage component [HILL81]. A regular array can implement effectively the FSM combinational component. In particular, PLA-based Finite State Machines can be designed efficiently, because the properties of two-level combinational functions are

well understood. Therefore PLAs and Memory elements can be seen as primitives of a general digital design methodology.

A **Storage Logic Array** (SLA) is an array implementation of a sequential function, where both small PLAs and memory elements are placed along the positions of a two-dimensional array [PATT79]

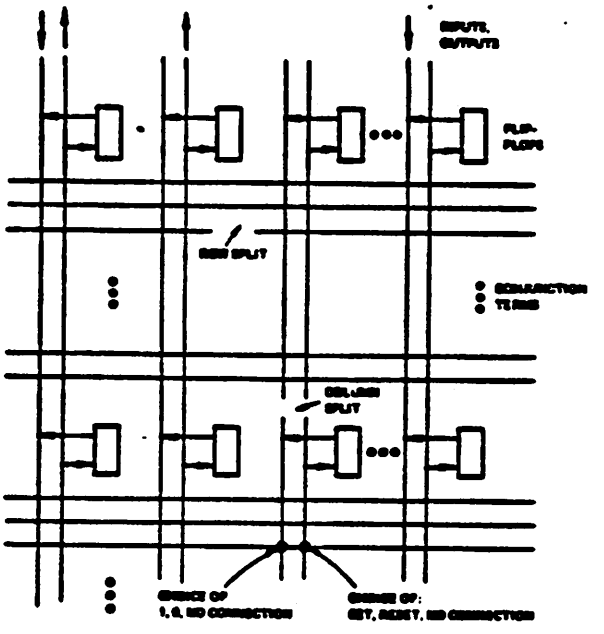


Fig 1.3.3 Storage Logic Array

1.4 AUTOMATED SYNTHESIS OF COMBINATIONAL PLA-BASED SYSTEMS

The automated synthesis of a combinational system as a Programmable Logic Array can be partitioned into several tasks: **functional design, logic design, topological design** and **physical design** (Fig. 1.4.1.).

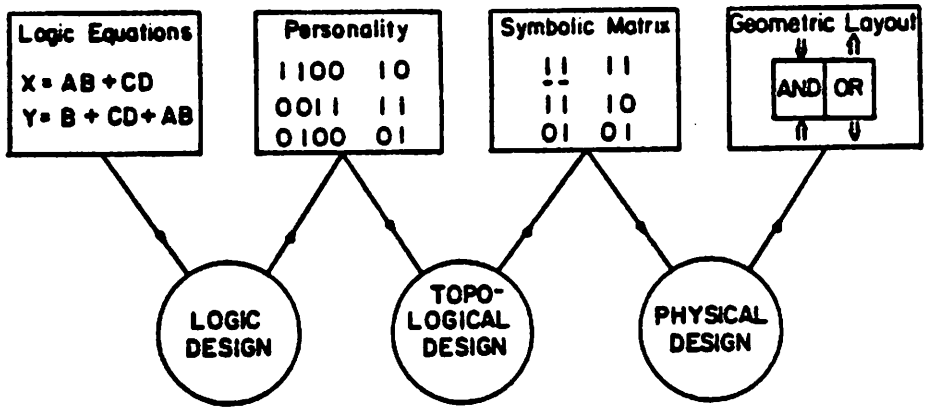


Fig. 1.4.1 Computer-aided PLA synthesis

Functional design consists of defining the functional specification of the system. For example, a designer can specify a combinational circuit in a Hardware Description Language. The functional description is then transformed into a logic representation in terms of Boolean (logic) variables.

Logic design is a manipulation of the logic representation without modifying the circuit functionality. Optimal logic design optimizes the logic representation to obtain a convenient implementation. In particular **logic minimization**, [MCKL56] [HONG74], [BROW81], [BRAY82a],[BRAY84] and **logic partitioning** [BRAY82b] have been explored. Two-level logic minimization tries to obtain a logic representation with a minimal number of implicants and literals. The reduction of the number of implicants allows a PLA implementation in a smaller area, and therefore leads to a faster speed of

operation and higher yield. The reduction of the number of literals corresponds to a reduction of the number of devices and contacts required. This reduction leads to a faster speed of operation and to an enhanced reliability.

Topological design of a PLA involves the definition of the location of the devices in the array. Note that it is possible to define a straight-forward mapping from a two-level logic description to the PLA layout. Unfortunately this strategy leads in general to a non optimal design of the array in terms of silicon area. In fact, most of the array locations are personalized by the absence of active devices, especially in large arrays [WOOD79]. Hence the straight-forward implementation would result 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. Optimal topological design increases the area utilization by rearranging the array structure by means of topological operations. Topological optimization methods fall into two major categories: **array folding** and **array partitioning** and are described extensively in the sequel. Topological design produces a PLA description with detailed information about the position of the PLA devices, interconnections and contacts. This information can be in the form of a symbolic array or a stick diagram, and is independent of the implementation technology to a large extent.

Physical design is the translation of the topological representation into the layout of the masks used to manufacture the circuit. This step depends heavily on the semiconductor process and layout design rules. The final representation is generally in the form of a geometric design language such

as the *Caltech Intermediate Form* (CIF) [MEAD80] or equivalent mask language or format.

1.5 AUTOMATED SYNTHESIS OF SEQUENTIAL PLA-BASED SYSTEMS

The design of a sequential logic systems departs from combinational logic design mainly at the functional and logic level. As in combinational logic design, the automated synthesis of a sequential circuit as a PLA-based Finite State Machines can be partitioned in the following tasks: **functional design, logic design, topological design** and **physical design** . A flow-chart of the transformations involved in the computer aided synthesis of a PLA-based sequential functions is shown in Fig. 1.5.1.

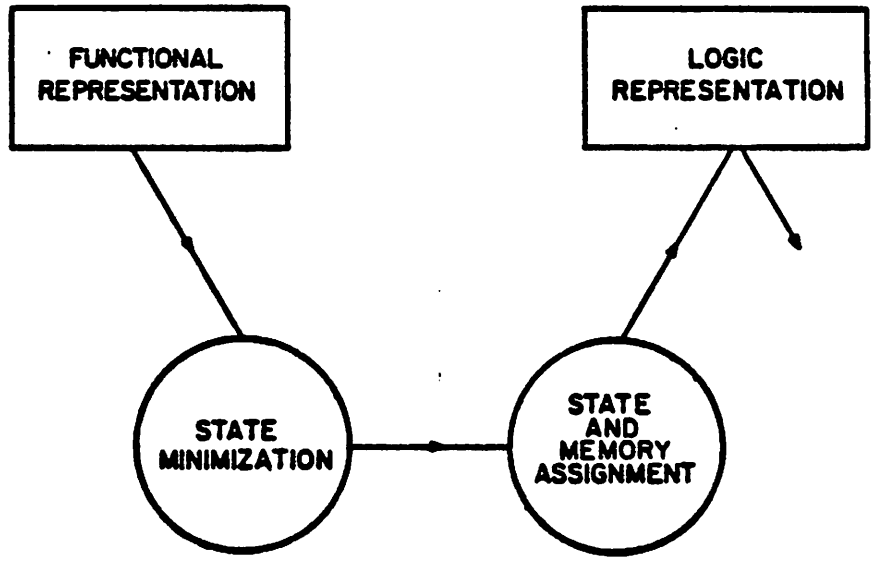


Fig. 1.5.1 Computer aided FSM synthesis

Functional design consists in defining the system behavior by means of a functional description such as a **flow-chart** , a **Hardware Description Language** or a **state table** . Flow chart descriptions, such as the Algorithmic State Machine (ASM) chart [CLAR75] or the Mnemonic Documented State diagram (MDS) [FLET80], allow an algorithmic description of the sequential system and enable the designer to visualize the entire machine functionality. Hardware description languages allow to describe a sequential function as a software program. State tables are tabular descriptions of the system functionality. Optimal functional design includes **state minimization** [HART66]. State minimization reduces the number of states of a sequential system, without modifying its functionality.

Logic design consists of mapping the functional description into a logic representation in terms of logic variables. In particular, a representation of the states in terms of Boolean variables is chosen. This step is referred to as **state assignment** [HART66]. The memory configuration used to store the machine state is selected at the same time. This allows to express the excitation maps of the memory elements by Boolean equations. Note that the complexity of the combinational component of the machine depends heavily on the state assignment and on the excitation maps of the chosen memory elements. Therefore optimal logic design of sequential functions include optimal state assignment and selection of memory elements. This step allows to specify the FSM combinational component in relation to the optimal implementation of the sequential function.

Logic , topological and **physical design** of the FSM combinational component ***per se*** are similar to the design steps of combinational circuits. In addition, memory elements have to be designed and connected to the combi-

national component. Therefore an objective of FSM topological design is to locate the PLA input and output connections as to simplify the routing between the PLA and memory elements.

1.6 PROGRAMMABLE LOGIC ARRAY IMPLEMENTATION

Programmable Logic Arrays are extensively used in integrated circuit design. In some cases a PLA, or a PLA-based FSM, occupies an entire chip [WOOD76]. Examples are code-converters or digital controllers. Recently Programmable Logic Arrays emerged as a new building-block for VLSI circuit design [MEAD80]. For example PLAs can implement the instruction decoder of a microprocessor.

PLA design depends heavily on the integrated circuit design method. 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 with a given shape. On the other hand, in VLSI custom and macro-cell 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.

Basic concepts and definitions of switching theory and representations of combinational logic functions are reported in App. A. A combinational logic function can be described by a logic cover. While designing a PLA implementation of a combinational function, the logic cover is seen as a pair of matrices, called **input** and **output personality** matrices.

Example 1.6.1: Personality matrices:

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

The corresponding PLA implementation is sketched in Fig. 1.6.1.

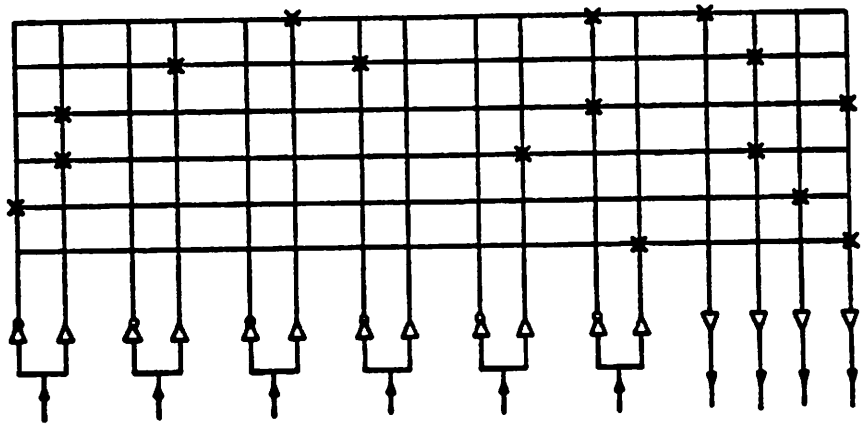


Fig. 1.6.1 PLA Schematic implementation

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 par-

ticular every implicant input part, or equivalently every input personality row corresponds to a logical 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"). Note that a PLA can be ideally partitioned into input and output arrays, corresponding to the sets of input and output columns respectively. 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.

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

Example 1.6.2: **AND-OR bipolar transistor PLA implementation**

In this case the AND and OR gates can be easily implemented by placing a bipolar transistor in the locations of the physical array corresponding to the cares of the personality matrix (Fig. 1.6.2.). The resulting array is very compact and output signal are directly available from the output columns. Note that very fast PLAs can be built by limiting the logical voltage swing. Fang [FANG83] reported the implementation of a 6 nanosecond PLA with 200 product terms.

■

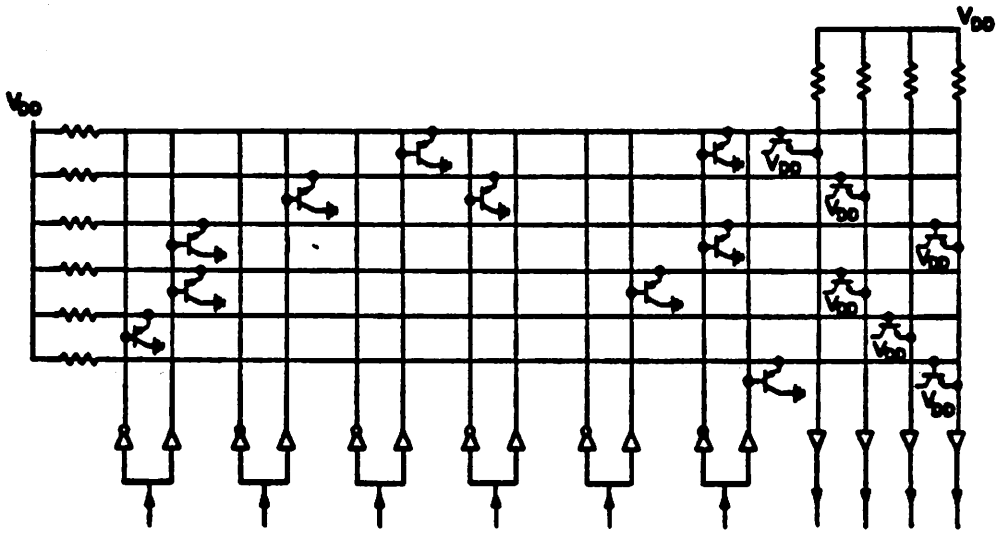


Fig. 1.6.2 AND-OR Bipolar transistor PLA implementation

Remark 1.6.1: Primitive bipolar PLAs were built with bipolar diodes in place of transistors. Fan-out considerations limited severely the use of such arrays for large switching functions.

Example 1.6.3: **NOR-NOR NMOS PLA implementation**

The most common PLA implementation in VLSI MOS circuits has the basic structure as 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*). The transformation

from a *sum-of-products* representation can be easily done as shown in Fig. 1.6.3 . Note that output inverters are required.

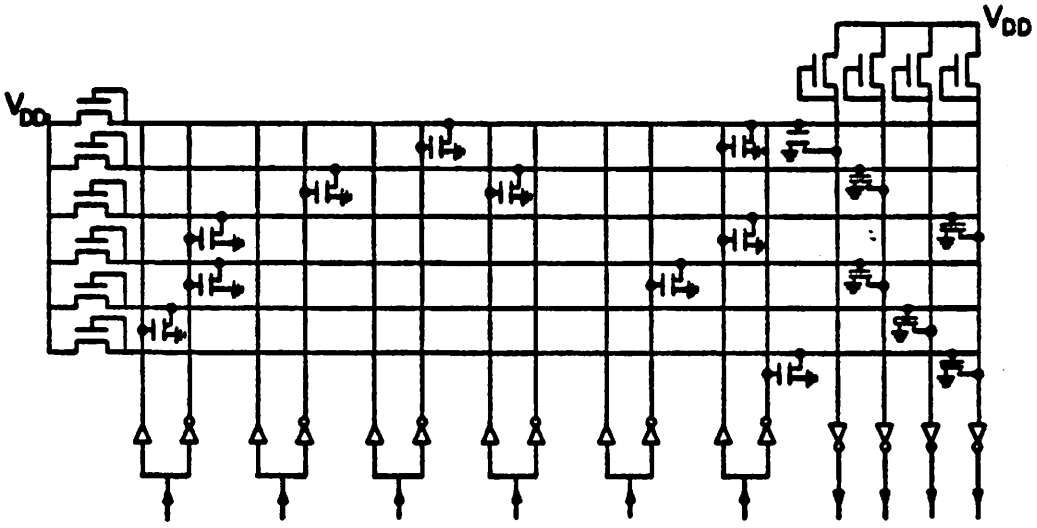


Fig. 1.6.3 NOR-NOR NMOS PLA implementation

Remark 1.6.2: The basic NMOS PLA implementation has been tailored by several companies to different MOS processes yielding better performances. High performance NMOS implementations with static and dynamic operation are presented in [COOK79]. Dynamic CMOS implementations are achieved by using the same structure shown in Fig. 2.4.2. Small-sized PLAs can be implemented in NMOS technology by a *complemented-product-of-complemented-products* form (NAND-

NAND). Such arrays are also referred to as stack-PLAs, and their use is limited by fan-in considerations of NAND ports in MOS technology.

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 an 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.

Remark 1.6.3: Integrated circuit logic arrays come in two different flavors. In particular PLA personalization can follow or precede fabrication. PLA 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 NMOS PLAs in [MEAD80]. However more complex mask-programming techniques are generally used in order to eliminate unnecessary parasitics [HOFF81].

It is therefore unappropriate to refer to the second category of cir-

cuits as Programmable Logic Arrays. A better term should be Programmed Logic Arrays. Unfortunately it is very common to refer to Programmable Logic Arrays for both cases.

In the sequel only Mask Programmed Logic Arrays for VLSI design are considered. They will be referred to as Programmable Logic Arrays for the sake of uniformity.

■

1.7 PREVIOUS WORK

There are several contributions related to the optimal synthesis of PLA-based systems. Here they are grouped in three major categories:

i) papers describing systems using PLAs as a macro building blocks;

ii) papers related to the optimal synthesis of sequential circuits;

iii) papers presenting PLA optimization techniques.

The contributions related to the first and third group are quite recent. Optimal synthesis of sequential functions was a popular research subject in the sixties. However no major result was achieved in recent years in connection with VLSI circuit design. It is a goal of this dissertation to bridge this gap.

PLA-based systems have been used extensively in controller and data path design [FLEI75] [LOGU75]. Jones [JONE75] described the design of a system where PLAs were used as logic macros. Weinberger [WEIN79] first used PLAs to design adders and Schmookler [SCHM80] designed large ALUs by means of PLAs.

Computer-Aided Design tools and methods for PLA-based system design have been studied extensively. However many research problems are still open and under investigation.

Ayres [AYRE79] and Weber [WEBE79] addressed the use of high level language to design PLAs. Dietmeyer et al. [DIET68] used a DDL description to synthesize a digital system using PLAs. The APLAS system [KANG81] allowed the complete design of a FSM controller from a DDL-P description, though several transformations (as the state assignment) were not implemented as to yield an optimal synthesis. Recently, Floyd and Ullmann [FLOY82] addressed the syntesis of sequential functions, described by regular expressions, by means of PLAs.

The state assignment problem has been the object of extensive theoretical research. Hartmanis [HART61], Stearns [STEA61] Karp [KARP64] and Kohavi [KOHA64] developed algebraic methods based on partition theory. Their approach was based on a reduced dependence criterion, which lead to a good assignment. However no theoretical result was presented that related reduced dependencies to optimal FSM implementation. Moreover no systematic procedure was developed that could be used to encode large machines. Armstrong [ARMS62a] [ARMS62b] developed a method capable of coding large machines based on a graph interpretation of the problem. However his approach was still ineffective in obtaining a substantial reduction of the complexity of the FSM combinational component, because i) he did not take into account the techniques of fast heuristic logic minimizers, ii) he transformed the state assignment problem into a graph embedding problem, that represented only partially the state encoding problem, iii) his graph embedding technique was ineffective (see Section 4.3). Dolotta and McKlus-

key [DOLO62] introduced the concept of codable columns, used for finding near-optimal solutions. Their algorithm was able to estimate the complexity of the combinational component of the FSM as a function of partial-length state assignments. However the method was computationally efficient only for small machines. Their method was later improved by Weiner et al. [WEIN67b], Torng [TORN68] and Story et al. [STOR72]. Curtis [CURT69] [CURT70] considered the problem of using different types of storage elements in relation with the state assignment problem and generalized the methods proposed in [DOLO62] and [WEIN67b]. Tracey [TRAC66] and Saucier [SAUC72] addressed the state assignment problem in connection with race-free asynchronous machine design. Despite all these efforts, to the best of my knowledge, no tool for designing FSMs is in use today for a time-effective encoding of industrial digital controllers.

The logic minimization problem has also been object of extensive investigation. Most classical minimization algorithms [MKCL56] [ROTH58] [TISO67] are based essentially on a similar procedure. All the prime implicants of a logic function are generated first, and a minimal cover is selected by determining an appropriate subset. Classical algorithms have shown to be impractical for large combinational functions, because of computing time and memory space requirements. Heuristic logic minimization algorithms are based on an iterative improvement of the cover of a logic function. Hong et al. [HONG74] presented an algorithm that is capable of handling large logic functions and that was implemented in program MINI. Kang [KANG81] addressed the problem of minimizing very large switching functions. He used a *divide et impera* strategy. Logic functions are partitioned first and then minimized using a technique similar to the one reported in [HONG74]. Programs PAPA and SPAM implemented the algorithms. Svoboda described

in an unpublished manuscript a minimization algorithm which was implemented in programs PRESTO[BROW80] and POP [SYMA83],[DEMI84]. Brayton et al. [BRAY84] developed minimization techniques based on Shannon decomposition and the properties of unate covers. The related program, ESPRESSO-II, has shown performances superior to other minimizers.

Topological design has been studied only recently. Greer [GREE76] and Wood [WOOD79] presented for the first time a folded PLA implementation. A theoretical formulation of the optimal PLA folding problem was given for the first time by Hachtel et al. [HACT80] [HACH82a] who presented a heuristic algorithm as well. Hoffman [HOFF81] implemented a branch-and-bound folding algorithm in program BLAM, that could handle small PLAs. Luby et al. [LUBY82] addressed the complexity of the PLA folding problem. Hu [HU83] studied graph theoretic properties of folded PLA structures and a folding algorithm based on grouping. Grass [GRAS82] and Paillotin [PAIL81] explored different folding strategies. A taxonomy of the folding techniques is given in [HACT82b].

PLA compaction by means of topological partitioning was addressed first by Kang [KANG81], who proposed a heuristic algorithm. His technique was perfected lately by Hennessy [HENN83]. Egan [EGAN82] presented a method for PLA partitioning which is referred to as bipartite folding. Suwa [SUWA81] presented a technique to design compacted arrays, called segmented-folded PLAs. In particular, Suwa uses both partitioning (segmentation) and folding techniques. The PLA is segmented first in multiple planes, and then columns of each plane are pairwise folded.

The physical automated layout of PLAs has been developed mainly in the industrial environment for internal purposes. Few results have been pub-

lished. Glasser presented an interactive PLA generator in [GLAS80]. Landman [LAND81] wrote program MKPLA that generates the PLA layout from a logic description. Hoffman wrote program PLAID to lay-out simple-folded arrays [HOFF81]. A program to lay-out multiple-folded PLAs is being designed at U.C.Berkeley [MAH83]. More complex PLA structures can be laid-out by the PAOLA system [CHUQ82]. Patil [PATI79] described a design system for Storage Logic Arrays, where the layout of the combinational and the memory components of a FSM are considered along with the related interconnections.

1.8 DISSERTATION OUTLINE

The design of an automated system for the design and optimization of PLA based integrated circuits involves the concurrent work of several researchers, because of the complexity and the wide span of the problems involved.

This dissertation addresses two major points:

i) the description of a framework for optimal topological design, based on PLA folding and partitioning techniques;

ii) an effective technique for the optimal state assignment of PLA based FSM.

An innovative PLA **folding** method is presented in Chapter 2. First a general folding technique is introduced, called multiple constrained folding. Two PLA architectures are proposed to implement effectively multiply folded arrays [DEMI83a]. The problem of interconnecting a PLA to other building blocks is then considered. In fact, the folding techniques that have been previously proposed by other researchers have a major drawback. The connection of a

folded PLA to the outside circuitry may involve complex and area-consuming routing, because the positions of the inputs and outputs of a folded array are permuted by folding. In order to use effectively PLA folding for VLSI design, it is crucial to allow the positions of inputs ad outputs to be constrained. Therefore a set of constrained multiple folding techniques are introduced to compact the PLA area while ensuring effective routing of the folded array [DEMI83b] [DEMI83c].

A new approach to PLA topological **partitioning** is described in Chapter 3. The PLA partitioning problem is presented for the first time in graph theoretic terms [DEMI83d]. An algorithm for PLA partitioning based on a cluster search is presented. The algorithm uses array transformations based on logical operations to ease partitioning. Partitioned PLAs can then be implemented as block-folded arrays or alternatively as parallel-connected arrays [DEMI83e].

In Chapter 4 the design of PLA-based Finite State machines is addressed, and, in particular, the **optimal state assignment** problem. The optimal state assignment problem is studied in connection with logic minimization of the FSM combinational component, that is implemented here by a PLA [DEMI83f]. In particular a binary encoding (assignment) of the states is optimal when the unfolded/unpartitioned PLA area is minimal. Due to the computational complexity of the problem, a heuristic technique for state assignment is presented. First the class of present-states assignments that minimize the PLA rows is determined. Then a minimal-length assignment (leading to a minimal-column PLA) is selected. This technique has shown to be effective in achieving minimal area PLA implementations of the FSM combinational component on a set of industrial benchmark circuits.

The methodology used to approach the above problems is based on a rigorous mathematical formulation. Graph theory has been used extensively as a vehicle to understand the problem structure and to develop heuristic strategies. The proposed solutions have been always considered in relation with the actual implementation on a VLSI chip. Therefore electrical properties and limitations of the physical implementation have been taken into account. The algorithms have been coded and tested. The resulting programs are a part of the U.C. Berkeley VLSI design system. Experimental results on a set of industrial examples are reported.

CHAPTER 2

ARRAY FOLDING

2.1. TOPOLOGICAL DESIGN OF PROGRAMMABLE LOGIC ARRAYS

The straight-forward translation of the PLA personality matrix into a physical layout, leads in general to a non optimal design of the array. In fact, the input (output) personality matrices representing minimal covers of switching functions contain a large number of ***don't cares*** ("0"s), corresponding to the absence of a personalizing device. A straight-forward implementation would result 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. The 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. A topological compaction technique called **array folding** is described in this Chapter, while an alternative technique based on **array partitioning** is reported in Chapter 3.

For the sake of uniformity, PLAs are assumed to be implemented by one of the architectures shown in Example 2.4.1 and Example 2.4.2, as sketched in Fig. 2.1.1. 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.

In the sequel the folding technique presented by Wood [WOOD76] and studied theoretically by Hachtel et al. [HACH82b] is referred to as **simple**

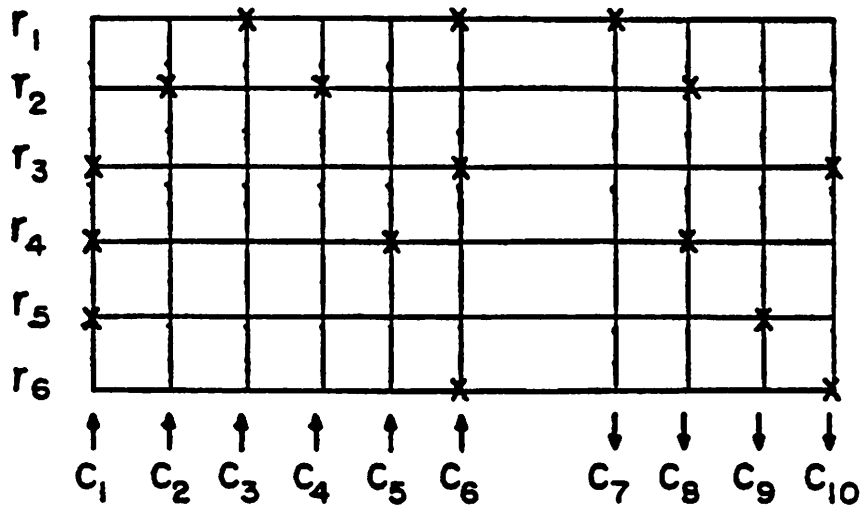


Fig. 2.1.1 Basic PLA structure

folding . Simple folding aims at determining 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 the physical array (Fig. 2.1.2). Folding comes in two flavors : **column folding** and **row folding** . 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 to implement in each column (and/or row) of the physical array a set of logic columns (rows) (Fig. 2.1.3). From the description given above , it is clear that multiple folding contains simple folding as a special case. Thus, the area saving achieved by this

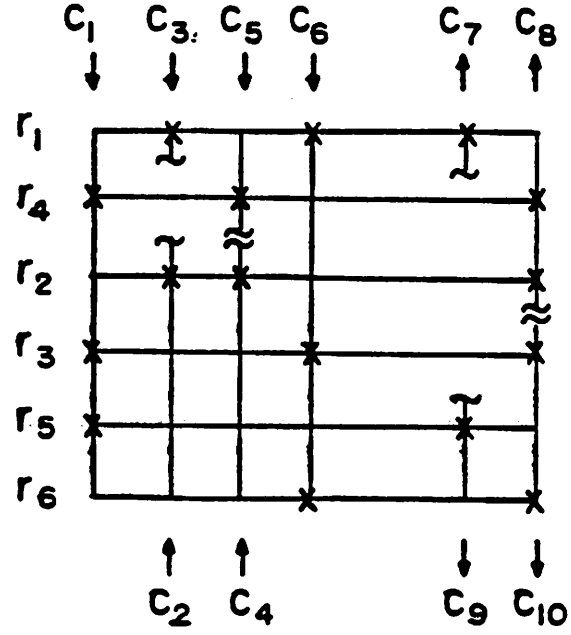


Fig. 2.1.2 Simply folded array

technique can always be made better than (or, in the worst case, equal to) the one achieved by simple folding. Note 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, the PLA sparsity is the ultimate limit.

All existing folding techniques have a major drawback. The connection of a folded PLA to the outside circuitry may involve complex and area-consuming routing , because the positions of the inputs and the outputs of a folded array are permuted by the folding algorithm. In order to use effectively PLA folding for VLSI design , it is crucial to allow the positions of inputs and outputs to be constrained.

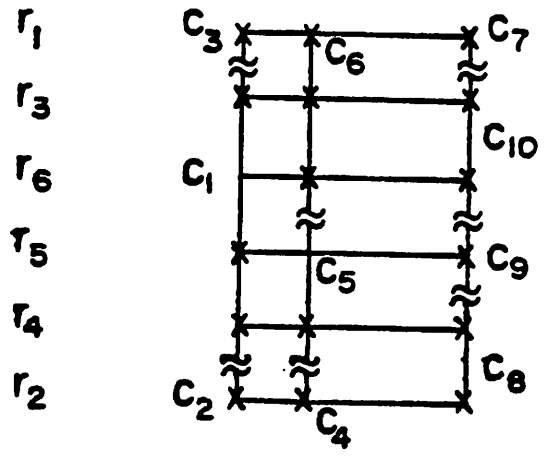


Fig. 2.1.3 Multiply folded array

Constrained folding techniques are strictly related to the folded PLA implementation. Multiply folded PLAs are implemented by structures which are more complex than those shown in Fig 2.2.1 and Fig. 2.2.2. Therefore a more detailed description of the constrained folding problems is postponed to the next Section.

2.2. MULTIPLY FOLDED PLA IMPLEMENTATIONS

The implementation of simple column (and/or) row-folded PLA is straight-forward, since at most two columns (rows) are folded together and connection to the outside circuitry can be done from the top or the bottom of the array. (Fig 2.1.2) [HACH82a] [WOOD79]. The implementation of a multiply-folded PLA is more complex. The implementation of multiply

column-folded logic arrays is considered first.

The implementation of several logic columns in the same physical location requires the physical (metal, poly or diffusion) columns be split into segments (Fig 2.1.3). 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 [GREE76] [CHUQ82] [DEMI81] have proposed different architectures for multiply-folded arrays. The following two structures are considered, which can be implemented in nMOS or cMOS technology. The extension to bipolar technology is straight-forward.

The first architecture is shown in Fig. 2.2.1. It requires two levels of metal (polysilicon), in addition to the usual levels of poly (metal) and diffusion. The PLA is implemented 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 column segments to the outside circuitry.

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 to 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 (Fig. 2.2.2).

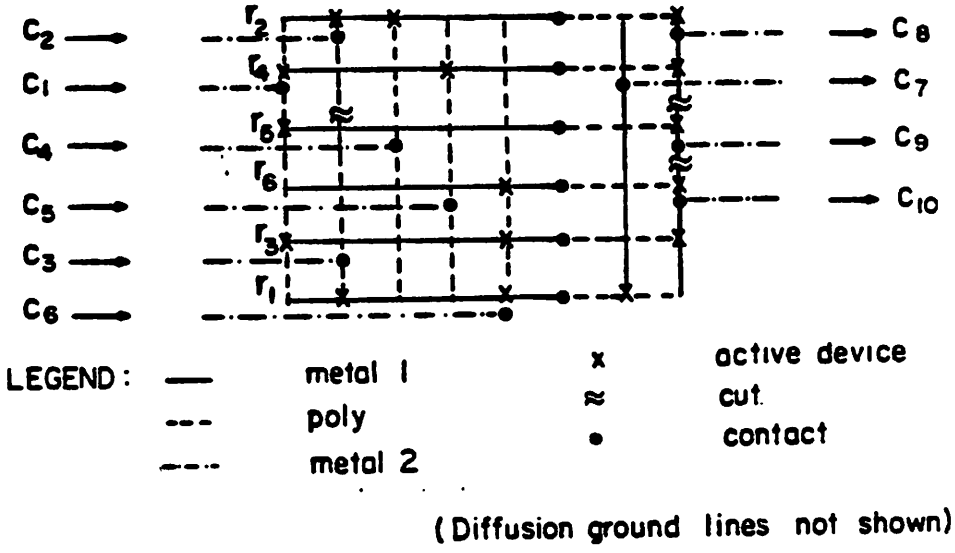


Fig. 2.2.1 Multiply-folded PLA implementation

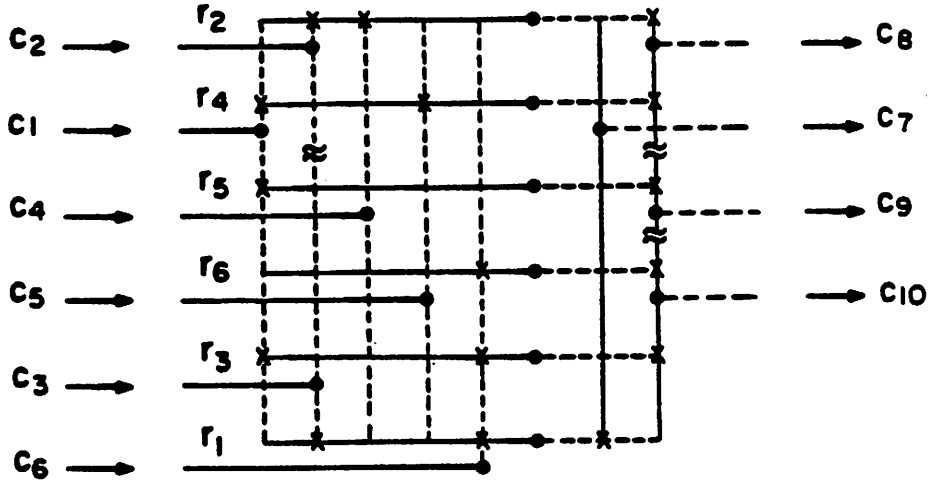


Fig. 2.2.2 Multiply folded PLA implementation

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

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

Two constrained column folding are addressed: **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 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 Fig. 2.2.3.

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 (Fig. 2.2.4).

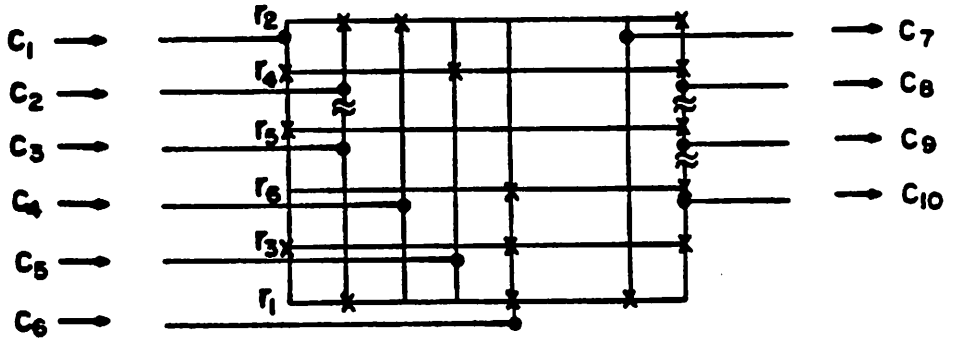
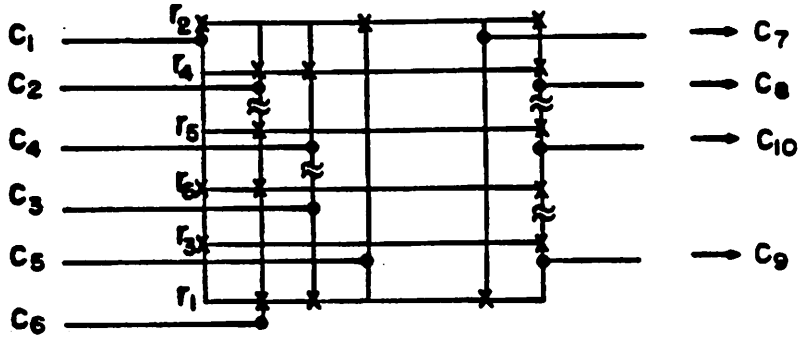


Fig. 2.2.3 Multiply folded PLA with connection-row oder constraint

Unconstrained multiply row-folded PLAs 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 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.



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

Fig. 2.2.4 Multiply folded PLA with bounded connection-row assignment

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, we introduce a new multiple constrained row folding problem : **row folding with bounded column assign-**

ment . 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 fold so that input and output signal can be routed from the desired (left, right, top, bottom) direction.

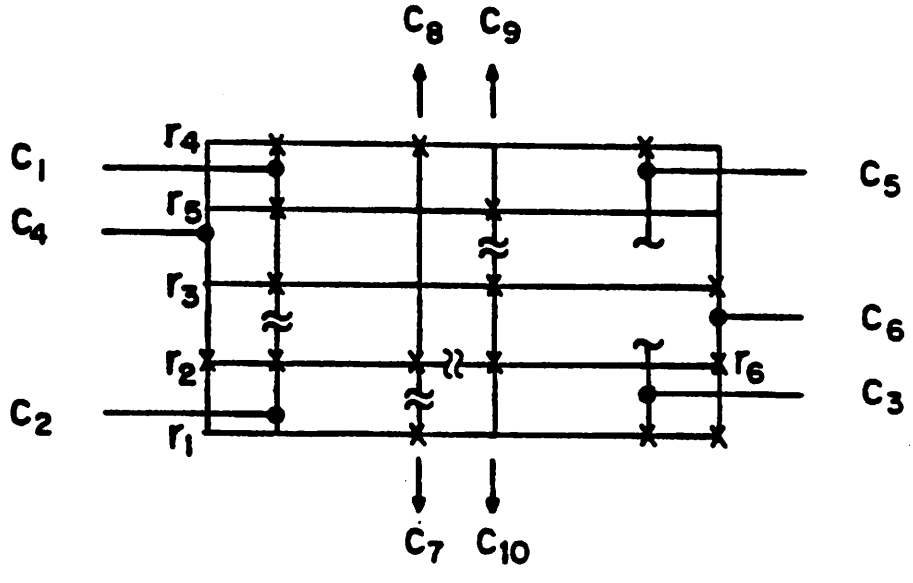


Fig. 2.2.5 Row and column folded array

2.3. GRAPH THEORETIC INTERPRETATION OF THE MULTIPLE FOLDING PROBLEM

Topological design concerns PLA devices location only. A convenient representation is achieved by two 1-0 matrices, called **input** and **output topological personality matrices** (TPM). The PLA topological personality matrix is the partitioned matrix whose components are the input and output personality matrices. The TPMs are obtained from the (logical) personality matrix in a straight-forward way.

In general it is assumed that the physical columns carrying each input signal and its complement are kept adjacent. Therefore both columns are dealt with as a single entity. Then the input TPM is obtained from the (logical) personality matrix by replacing each "0" by a "1" and each "*" by a "0". The output TPM is obtained by replacing each "*" by a "0". Fig. 2.3.1 shows the TPM for the PLA structure of Fig. 2.1.1 .

```
001001 1000
010100 0100
100001 0001
100010 0100
100000 0010
000001 0001
```

Fig. 2.3.1 Topological personality matrix

However the adjacency requirement between each input column and its complement may be relaxed, and every physical input column be dealt with as a single entity. In this case the input TPM is a matrix having twice as many columns as the (logical) personality matrix, and obtained by replacing each "0" by "10", each "1" by "01" and each "*" by "00".

In other terms, for the sake of generality, the topological personality matrix of a PLA can be defined as a 1-0 matrix, whose $(i, j)^{th}$ entry is "0" if

the $(i, j)^{th}$ location of the physical array is occupied by interconnect only.

As far as folding is concerned, PLAs are conveniently represented by means of the set of columns and rows of its topological personality matrix. Let $\{c_i, \quad i = 1, 2, \cdots, nc\}$ ($\{r_i, \quad i = 1, 2, \cdots, nr\}$) be the set of columns (rows) of the personality matrix. Each column is labeled **input (output)**, if it carries an input (output) signal in the physical array. A maximal set of adjacent input (output) columns is called **input array** or **AND plane (output array or OR plane)**. Let $R(c_i)(C(r_i))$ be the set of rows (columns) with a nonzero entry in the i^{th} column (row) of the personality matrix. Two columns c_i, c_j (rows r_i, r_j) are **disjoint** if $R(c_i) \cap R(c_j) = \phi \quad (C(r_i) \cap C(r_j) = \phi)$. A **column-folding list (row-folding list)** is a set of either input or output disjoint columns $f_i = \{c_{i,1}, c_{i,2}, \cdots c_{i,n}\}$ (rows $f_i = \{r_{i,1}, r_{i,2}, \cdots r_{i,n}\}$). An **ordered column-folding list** $o_i = (c_{i,1}, c_{i,2}, \cdots c_{i,n})$ (**ordered row-folding list** $o_i = (r_{i,1}, r_{i,2}, \cdots r_{i,n})$) is a column (row) folding list whose elements are ordered. A **column (row)-folding set** is a set of disjoint column (row)-folding lists $F = \{f_1, f_2, \cdots, f_k\}$ and **ordered column (row)-folding set** is a set of disjoint column (row) ordered folding lists $O = \{o_1, o_2, \cdots, o_k\}$. Let U be the set of unfolded columns (rows), i.e. $U = \{c \mid \nexists k \; s.t. \; c \in o_k\}$ ($U = \{r \mid \nexists k \; s.t. \; r \in o_k\}$). The column (row) cardinality of a folded PLA is $C(O) = |O| + |U|$ ($R(O) = |O| + |U|$). An ordered folding list of columns (rows) induces a set $QR(O)$ ($QC(O)$) of ordering relations among the rows (columns):

$$QR(O) = \{r_x < r_y \mid r_x \in R(c_{i,j}) \; ; \; r_y \in R(c_{i,j+1}) \; ; c_{i,j}, c_{i,j+1} \in o_i ; o_i \in O\}$$

$$(QC(O) = \{c_x < c_y \mid c_x \in C(r_{i,j}) \; ; \; c_y \in C(r_{i,j+1}) \; ; r_{i,j}, r_{i,j+1} \in o_i ; o_i \in O\})$$

Let $QR^+(O)$ ($QC^+(O)$) be the transitive closure of $QR(O)$ ($QC(O)$) [AHO74]. A column (row) ordered folding set is **implementable** if

$QR^+(O)(QC^+(O))$ is a partial order of the set 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.

Remark 2.3.1 : In the simple folding case $|U|$ = (initial column/row set cardinality) $-2|O|$. Hence the optimal unconstrained simple folding problem is to find an implementable ordered folding set with maximum cardinality. ■

A graph theoretic interpretation of the multiple folding problem is introduced, in order to gain a better insight into the problem and to study heuristics for the related algorithm. Column folding is considered first. According to [HACH80], a **column-intersection graph** $G(V, E)$ is defined to be a graph whose nodes $v \in V$ are in one-to-one correspondence with the columns of the logic array and the set E is defined as $E = \{v_i, v_j \mid R(c_i) \cap R(c_j) \neq \phi\}$. Given an ordered column-folding set O, an associated mixed graph $G(O) = G(V, E, A(O))$ is introduced. A mixed graph $G(V, E, A)$ is a graph with two sets of edges, a set of undirected edges E and a set of directed edges A. V and E are defined as in the column-intersection graph. $A(O)$ is defined as:

$$A(O) = \{v_{i,k}, v_{i,k+1} \mid (c_{i,1}, c_{i,2}, \cdots c_{i,k}, c_{i,k+1}, \cdots c_{i,n}) \in O;$$

$$k = 1, 2, \cdots n-1\}$$

A *χ-path* in $G(V, E, A(O))$, is a directed path $\chi = [v_1, v_2, \ldots . v_p]$ such that:

i) the first edge in χ is directed and the last undirected; i.e. $(v_1, v_2) \in A(O)$ and $\{v_{p-1}, v_p\} \in E$

ii) every undirected edge in χ is followed by a directed edge; i.e. $\{v_i, v_{i+1}\} \in E \rightarrow (v_{i+1}, v_{i+2}) \in A(O) \quad \forall i = 1, 2, \cdots, p-3$

Example 2.3.1 : For the PLA sketched in Fig. 2.3.1 and the ordered folding set $O = \{o_1\}$; $o_1 = (c_{10}, c_7, c_9)$, the associated mixed graph is shown in Fig. 2.3.2 and the partially folded array in Fig. 2.3.3. A *χ-path* is $[v_{10}, v_7, v_9, v_1]$.

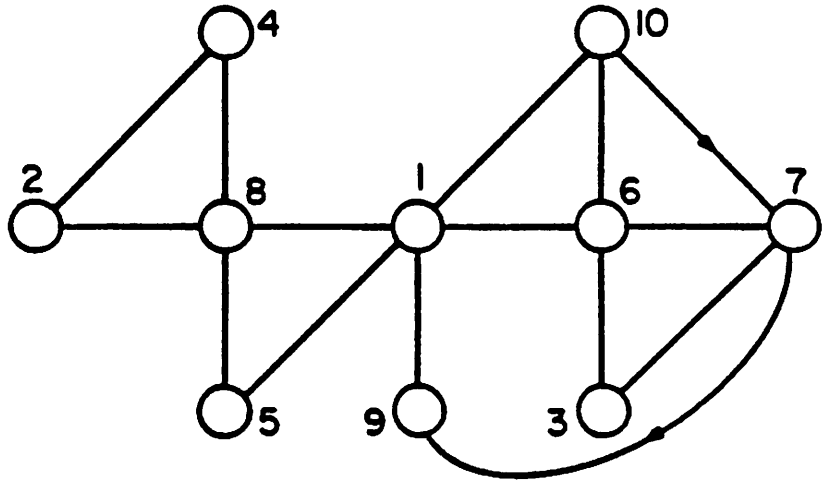


Fig. 2.3.2 χ-path

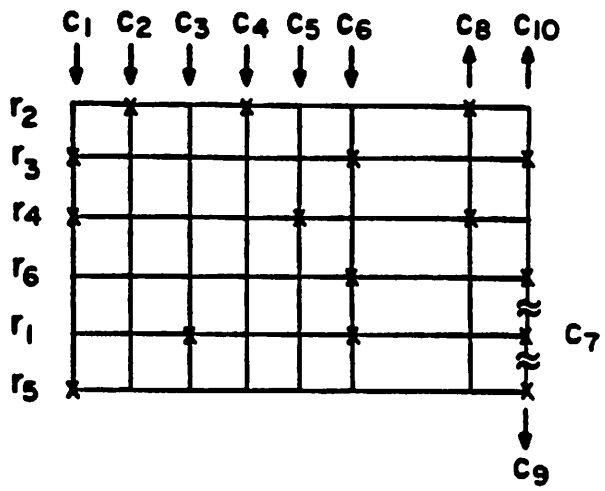


Fig. 2.3.3 Partially folded array

A " *χ–cycle* in *G(V, E, A(O))* is a closed *χ–path* having at least two undirected edges.

Theorem 2.3.1: An ordered column-folding set *O* is implementable if and only if the induced mixed graph *G(V, E, A(O))* has no *χ–cycles*.

Proof:

(if)

Assume that the folding set is implementable. For the sake of contradiction, suppose that ∃ a *χ–cycle* in *G(V, E, A(O))*. Without loss of generality, the vertices of the cycle can be labeled so that :

$$\{v_{p-1}, v_p\} \in E.$$

$$\{v_l, v_{l+1}\} \in E.$$

$$(v_k, v_{k+1}) \in A \quad \forall k \in \{1, 2, \cdots, l-1\} \cup \{l+1, l+2, \cdots, m-1\}$$

It is always possible to achieve such a labeling, because a *χ−cycle* has at least two undirected edges ($\{v_{p-1}, v_p\}$ and $\{v_l, v_{l+1}\}$) and two paths of directed edges (joining v_1 to v_l and v_{l+1} to v_{m-1}). Moreover a path of directed and undirected edges (possibly of zero length) joins v_m to v_{p-1}.

Since paths of directed edges are related to column ordered folding lists, and the column ordered folding lists induce a row order relation, we have :

$$R(c_1) < R(c_l);$$

and :

$$R(c_{l+1}) < R(c_m).$$

Take any row $r \in R(c_l) \cap R(c_{l+1})$:

$$R(c_1) < r < R(c_m)$$

and from the definition of transitive closure :

$$\{R(c_1) < R(c_m)\} \subset QR^+(O)$$

Since there is a finite number of vertices along the *χ−cycle* from vertex v_{l+1} to vertex v_{p-1}, by repeating the same argument and by the transitivity of $QR^+(O)$:

$$\{R(c_{l+1}) < R(c_{p-1})\} \subset QR^+(O).$$

Let row $\hat{r} \in R(c_{p-1}) \cap R(c_p)$. From the transitive closure relation:

$$R(c_{i+1}) < \hat{\tau}$$

and in particular:

$$\tau < \hat{\tau}$$

But since $c_1 = c_p$ and $R(c_1) < \hat{\tau}$, then :

$$\hat{\tau} < \tau$$

Hence we have a contradiction because $QR^+(O)$ is not a partial order on the set Z^+.

(only if)

Assume that $G(V, E, A(O))$ has no χ-cycles. For the sake of contradiction suppose that $QR^+(O)$ is not a partial order. Therefore there exist two rows, τ_1 and $\tilde{\tau}_1$, such that:

$$\tau_1 <^+ \tilde{\tau}_1$$

and:

$$\tilde{\tau}_1 <^+ \tau_1$$

Hence there exists a sequence:

$$[\tau_1, \tau_2, \cdots, \tau_n, \tilde{\tau}_1]$$

such that:

$$\tau_j \in R(c_{1,j}) \cap R(c_{2,j}) \quad j = 1, 2, \cdots, n$$

where: $c_{1,1} = c_{2,1}$, $c_{1,n} = c_{2,n}$, $(c_{2,j-1}, c_{1,j}) \in A$ $j = 2, 3, \cdots, n$ and either $c_{1,j} = c_{2,j}$ or $\{c_{1,j}, c_{2,j}\} \in E$, $j = 2, 3, \cdots, n-1$. Hence there is a directed path from $c_{1,1}$ to $c_{2,n}$ having no more than one consecutive undirected edge. Moreover $\tilde{\tau}_1 \in R(\tilde{c}_1)$ and $(c_{2,n}, \tilde{c}_1) \in A$. Furthermore there exist a sequence:

$$[\tilde{r}_1, \tilde{r}_2, \cdots, \tilde{r}_n, r_1]$$

such that:

$$\tilde{r}_j \in R(\tilde{c}_{1,j}) \cap R(\tilde{c}_{2,j}) \quad j = 1, 2, \cdots, n$$

where $\tilde{c}_{1,1} = \tilde{c}_{2,1}$, $\tilde{c}_{1,n} = \tilde{c}_{2,n}$, $(\tilde{c}_{2,j-1}, \tilde{c}_{1,j}) \in A \quad j = 2, 3, \cdots, n$ and either $\tilde{c}_{1,j} = \tilde{c}_{2,j}$ or $\{\tilde{c}_{1,j}, \tilde{c}_{2,j}\} \in E$, $j = 2, 3, \cdots, n-1$. Hence there is a directed path from $\tilde{c}_{1,1}$ to $\tilde{c}_{2,n}$ having no more than one consecutive undirected edge. Moreover $r_1 \in R(c_1)$and$(\tilde{c}_{2,n}, c_1) \in A$, and either $c_{1,1}$ and c_1 ($\tilde{c}_{1,1}$ and $\tilde{c}_1$) coincide or $\{c_{1,1}, c_1\} \in E$ ($\{\tilde{c}_{1,1}, \tilde{c}_1\} \in E$).

Thus $[c_{1,1}, \cdots, c_{2,n}, \tilde{c}_1, \tilde{c}_{1,1}, \cdots, \tilde{c}_{2,n}, c_1, c_{1,1}]$ is a cycle having at least two undirected edges by definition of $G(V, E, A(O))$. Hence $G(V, E, A(O))$ has a χ-cycle and we have a contradiction. ■

Remark 2.3.2: Theorem 2.3.1 allows to verify the existence of a row ordering compatible with a column ordered folding set by checking relations among columns only. This procedure is much simpler (and therefore much faster to be executed on a digital computer) than to verify directly cyclic relations in $QR^+(O)$. ■

Remark 2.3.3: The graph interpretation and Theorem 2.3.1 applies "mutatis mutandis" to the multiple unconstrained row folding problem. In this case $G(V, E)$ is the row-intersection graph and $G(V, E, A(O))$ is the mixed graph obtained by adding to $G(V, E)$ the set of directed edges:

$$A(O) = \{v_{i,k}, v_{i,k+1} | (\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,k}, \tau_{i,k+1}, \ldots, \tau_n) \in O;$$

$$k = 1, 2, \ldots, n-1\}$$

A graph interpretation of unconstrained row and column folding is more complex, because it involves bookkeeping of the ordering relations among rows and among columns. For this problem the information contained in the column and row intersection graphs is not sufficient.

Example 2.3.2 : Consider the partially column folded array shown in Fig. 2.3.3. Let us question the implementability of the array after folding row τ_5 with row τ_6. The folded array is clearly not implementable, even though it does not introduce any cycle in both intersection graphs.

Therefore the row constraint graph G_R and the column constraint graph G_C are introduced. Directed graphs G_R and G_C correspond to the transitive closure relations $QR^+(O_C)$ and $QR^+(O_R)$ induced by the column and row folding sets O_C and O_R [HACH82b]. By definition, the ordered folding sets O_R and O_C are implementable if graphs G_R and G_C are acyclic.

2.4. AN ALGORITHM FOR MULTIPLE PLA FOLDING

The optimal multiple PLA folding problem was shown to be NP-complete in [LUBY82]. Therefore a heuristic algorithm is proposed. The algorithm can be considered an extension of the simple folding algorithm presented in [HACH80].

Multiple column folding problem is considered first. The ordered column folding set and the mixed graph $G(V, E, A(O))$ are constructed by the algorithm. At each step the algorithm tries to increase the cardinality of the folded column set and verifies the implementability of the folding by checking that the mixed graph has no χ-*cycle*.

A conceptual description of the algorithm is the following:

MASTER ALGORITHM

Step 0: Initialize the folding procedure

Step 1: If the set of columns which have not been processed is empty , stop. Else select a pair of unfolded disjoint columns or an unfolded column and a column folding list as folding candidates.

Step 2: If the fold induces χ-*cycle* in graph $G(V, E, A(O))$, reject it and go to Step 1.

Step 3: If folding has secondary constraints and constraints are not satisfied reject the fold and goto Step 1.
(This step is performed by the algorithms described in Section 2.5.)

Step 4: Fold the candidates, modify the PLA accordingly. Go to Step 1.

■

A detailed description of the algorithm for simple column folding is given in [HACH80]. This Section deals primarily with the generalization to multiple folding and with the procedure for multiple folding candidate selection.

The selection of the candidate columns for multiple folding can be done according to one of the following folding patterns:

1) a new folding list can be formed by folding two unfolded columns.

2) an unfolded column can be folded on top (bottom) of an existing folding list.

3) a folding list can be "opened" and an unfolded column can be folded "by insertion" into an existing folding list.

A selection of the folding pattern and candidate column is done at each step according to a heuristic strategy.

Let us define first the set of descendants $D(v)$ (ancestors $A(v)$) of a vertex V as follows:

a vertex d is **descendant** of v if there is a χ*–path* from v to d.

a vertex a is **ancestor** of v, if v is descendant of a.

The adjacency set $ADJ(v)$ of a vertex v is defined to be the set of vertices connected to v by an undirected edge. By definition, every vertex is considered adjacent to itself.

The pseudo-descendant set $\tilde{D}(v)$ of a vertex v is the union of the adjacency set of v and the descendant sets of each vertex adjacent to v.

$$\tilde{D}(v) = \bigcup_{\tilde{v} \in ADJ(v)} D(\tilde{v}) \cup ADJ(v)$$

Remark 2.4.1 : It follows from Theorem 2.3.1 that for each pair of consecutive columns in an implementable ordered folding list, the corresponding vertices v_1 and v_2 are such that:

$$ADJ(v_2) \cap A(v_1) = \phi$$

■

Let us consider now the selection strategy for folding pattern 1.

Example 2.4.1 : When two columns, say c_1 and c_2, are folded, a directed edge (v_1, v_2) is added to $A(O)$. Hence a *χ-path* joins v_1 to each vertex in $\tilde{D}(v_2)$. Therefore all pseudo-descendants $\tilde{D}(v_2)$ of v_2 are descendants of v_1.

$$D(v_1) \leftarrow D(v_1) \cup \tilde{D}(v_2)$$

Moreover, since a *χ-path* joins each ancestor of v_1 to v_1, the descendants of v_1 are descendants of each ancestor of v_1

$$D(\tilde{v}) \leftarrow D(\tilde{v}) \cup D(v_1) \quad \forall \tilde{v} \in A(v_1)$$

It follows that an upper bound on the number of ancestor-descendant relations induced by the column folding is :

$$\rho_1 = |A(v_1)|\,|\tilde{D}(v_2)|$$

It is reasonable to conjecture that the fewer relations are induced, the lower is the probability of finding *χ-cycles* at further steps of the algorithm. Hence a good choice for a candidate folding pair v_1, v_2 is the one for which ρ_1 is minimal. Unfortunately $\frac{n(n-1)}{2}$ candidate pairs have to be tried to find the minimum ρ_1 for an array with n unfolded columns. This procedure is too time consuming for large arrays. Therefore, an alternative selection strategy is used: select the candidate folding pair (v_1, v_2) such that:

$$v_1 = arg \min_{v \in \bar{V}} |A(v)|$$

$$v_2 = arg \min_{v \in \bar{V}} |\tilde{D}(v)|$$

where $\bar{V} \subseteq V$ is the vertex subset corresponding to the unfolded columns.

Similar considerations apply to the candidate selection according to folding pattern 2. When a column c_1 is folded on top of an ordered folding list $(c_{2,1}, \cdots, c_{2,n})$, a directed edge $(v_1, v_{2,1})$ is added to $A(O)$. Hence a *χ-path* joins v_1 to each vertex v_k, such that $v_k \in \tilde{D}(v_{2,1})$. Therefore an upper bound on the number of ancestor-descendant relations induced by the column fold is:

$$\rho_2 = |A(v_1)|\,|\tilde{D}(v_{2,1})|.$$

Conversely when a column c_2 is folded on the bottom of an ordered folding list $(c_{1,1}, c_{1,2}, \cdots, c_{1,n})$ an oriented edge $(v_{1,n}, v_2)$ is added to $A(O)$. Hence a *χ-path* joins every vertex $A(v_{1,n})$ to every vertex in $\tilde{D}(v_2)$. Therefore, an upper bound on the number of ancestor-descendant relations induced by the column fold is:

$$\rho_2 = |A(v_{1,n})|\,|\tilde{D}(v_2)|$$

The strategy for candidate selection according to folding pattern 2 is based on the same considerations used for folding pattern 1.

A slightly different strategy is used for candidate selection according to folding pattern 3.

Example 2.4.2 : Consider the PLA shown in Fig. 2.1.1. Let us suppose that column c_7 is folded into the folding list $o_1 = (c_{10}, c_9)$ to give (c_{10}, c_7, c_9), as shown by Fig. 2.3.3. The ancestors of c_7 become ancestors of c_9 and the ancestors of c_{10} become ancestors of c_7. ■

In the general case suppose that column $\bar{c}$ is folded into a folding list

$(c_{i,1}, c_{i,2}, \cdots, c_{i,n})$ to give $(c_{i,1}, c_{i,2}, \cdots, c_{i,k-1}, \bar{c}, c_{i,k}, \cdots, c_{i,n})$. An oriented edge joins vertex $v_{i,k-1}$ to $\bar{v}$ and $\bar{v}$ to $v_{i,k}$. Hence the ancestors $A(\bar{v})$ become ancestors of the vertices in $\tilde{D}(v_{i,k})$ and the ancestors $A(v_{i,k-1})$ become ancestors of the vertices in $\tilde{D}(\bar{v})$. Therefore, an upper bound on the number of ancestor-descendant relations is:

$$\rho_3 = |A(v_{i,k-1})|\,|\tilde{D}(\bar{v})| + |A(\bar{v})|\,|\tilde{D}(v_{i,k})|$$

Unfortunately the computation of the minimum ρ_3 may be too time consuming for large arrays. Hence the candidate for insertion is determined first as:

$$\hat{v} = \underset{v \in \bar{V}}{arg\,min}\ (\ |\tilde{D}(v)| + |A(v)|\ )$$

and then the folding list and the insertion position such that :

$$\hat{\rho}_3 = |A(v_{i,k-1})|\,|\tilde{D}(\hat{v})| + |A(\hat{v})|\,|\tilde{D}(v_{i,k})|$$

is minimal.

When the "best" folding candidates have been selected according to the three folding patterns, the selection of the folding pattern is based on a weighted comparison of the upper bounds ρ_i , $i = 1, 2, 3$. Weighting factors allow to privilege a folding pattern with regard to the others, as, for example, multiple folding versus simple folding.

Remark 2.4.2: The Master Algorithm and the candidate selection strategy applies *mutatis mutandis* to the multiple unconstrained row folding problem.

■

The Master Algorithm is used for multiple row and column folding also. Order relations induced by the folds are described by the row constraint and

column constraint graphs. A candidate fold is rejected at Step 2 of the algorithm if it induces a direct cycle in any of the two graphs. The folding candidate selection strategy is similar to the one used for column folding, provided that some definitions are changed to be compatible with the different graph representation.

For this problem, a vertex d is descendant of v if there is a direct path from v to d; the adjacency set of a vertex is not defined and the pseudo-descendant set is equivalent to the descendant set. Hence the "best" column and the "best" row folding candidates and patterns can be found by a procedure similar to the one described above. Let ρ^c (ρ^r) be the related upper bounds on the number of relations induced in G_R (G_C) by a column (row) fold. A column (row) fold is attempted if :

$$\alpha * \rho^c < \beta * \rho^r$$

$$(\alpha * \rho^c \geq \beta * \rho^r)$$

where $\alpha = \frac{C(O)-1}{C(O)}$ and $\beta = \frac{R(O)-1}{R(O)}$ are dynamic weighting factors which take into account the relative area saving achieved by a column (row) fold at that step of the algorithm and $C(0)$ $(R(O))$ is the column (row) cardinality.

It is important to remark that this strategy allows to achieve more folds in comparison with other algorithms performing column (row) folding after row (column) folding. Nevertheless it is straight-forward to constrain the selection so that all column (row) folds are tried first, if desired.

2.5. MULTIPLE CONSTRAINED FOLDING

As stated in Section 2.2 the PLA constrained folding problems are related to the interconnection of the array to the outside circuitry. Constraints on folding are classified into two major categories:

1) Architectural or primary constraints

2) Secondary constraints.

Architectural constraints are related to the structure of the array and to the positions of input/output busses relative to the array. Secondary constraints are related to the positions of input and output lines inside the busses. 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 Master Algorithm presented in Section 4 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 busses 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 assignment algorithms are presented in this Section that assign positions to rows and/or columns and check if the secondary constraints are satisfied. The assignment algorithm for problem 2A is presented first. From this, an algorithm for problem 2B can be easily derived by interchanging rows with columns. Problems 2C and 2D are solved by a double assignment algorithm, based on the assignment algorithm of problem 2A.

2.5.1 Column folding with bounded product-row assignment

In this Section the problem of constraining product-term row positions only is considered. Therefore product-term rows are referred to as rows throughout this Section.

The **lower (upper) row bound map**:

$$L_R:\{r_i;\ i = 1, 2, \cdots, nr\} \rightarrow \{1, 2, \cdots, nr\}$$

$$(U_R:\{r_i;\ i = 1, 2, \cdots, nr\} \rightarrow \{1, 2, \cdots, nr\})$$

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

A **row assignment** $P : \{r_i;\ i = 1, 2, \cdots, nr\} \rightarrow \{1, 2, \cdots, nr\}$ is a permutation of the rows and an **implementable row assignment** a permutation compatible with an ordered column-folding set O; i.e. $P(r_x) < P(r_y)$ $\forall r_x < r_y \in QR^+(O)$

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

$$L_R(r_j) \leq P(r_j) \leq U_R(r_j) \quad \forall\ j = 1, 2, \cdots, nr$$

Example 2.5.1.1 : For the logic array shown in Fig. 2.1.1, the following lower and upper bounds are given:

$$L_R = 1, 1, 1, 4, 4, 6$$

$$U_R = 1, 3, 3, 6, 6, 6$$

This means that r_1 is constrained to the first position, r_2 and r_3 are constrained between position 1 and 3 , and so on. The implementable row assignment ($r_1, r_4, r_2, r_3, r_5, r_6$) induced by the column folding shown in Fig. 2.2 does not satisfy the given bound maps. On the contrary, the folded PLA shown in Fig. 5.1 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.

■

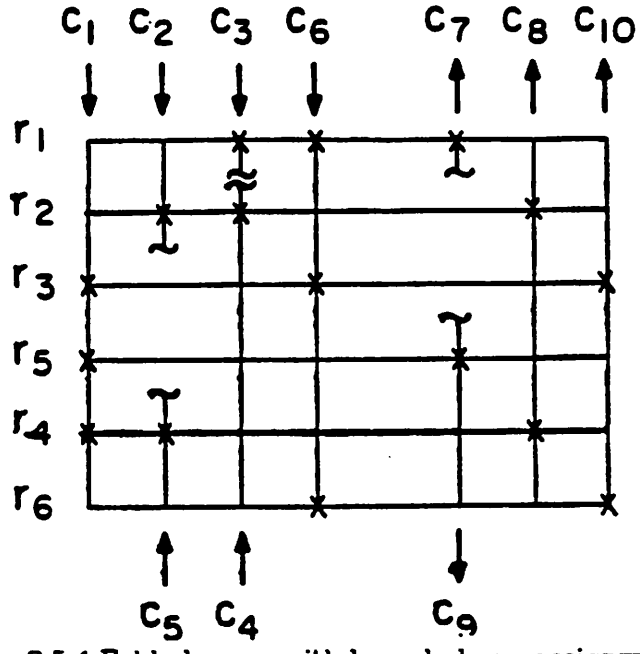


Fig. 2.5.1 Folded array with bounded row-assignment

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 a graph interpretation of the following subproblem:

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

The graph interpretation is useful to understand the underlying structure and to develop an algorithm and related heuristics. This subproblem is

described by a directed graph $G(R, N, A)$, with two node sets N and R, and a set of directed edges A.

The node sets R and N are in one to one correspondence with the row set and the set of the first nr natural numbers representing the possible row positions. The problem consists in finding a matching between R and N, i.e. coupling each row-node to a position-node, so that all the required bounds are satisfied. Position bounds are represented by a set of directed edges :

$$A = A_1 \cup A_2 \cup A_3 \cup A_4 \cup A_5$$

where : $A_1 \equiv \{(n_j, n_{j+1});\ j = 1, 2, \cdots, n-1\}$ represents the order on the sequence of the first nr natural numbers; $A_2 \equiv \{(n_i, r_j) | L(r_j) = i+1,\ j = 1, 2, \cdots, nr\}$ and $A_3 \equiv \{(r_j, n_i) | U(r_j) = i-1,\ j = 1, 2, \cdots, nr\}$ take into account the lower and upper bound maps ; $A_4 \equiv \{(r_i, r_j) | r_i < r_j \in QR(O)\}$ represents the order relations among the rows induced by the column folding.

Example 2.5.1.2 : Fig. 2.5.2a shows graph $G(R, N, A')$ $A' = A_1 \cup A_2 \cup A_3 \cup A_4$ for the PLA of Fig. 2.1.1, the row bounds of example 2.5.1.1 and the ordered folding set $O = \{(c_7, c_9), (c_3, c_4), (c_2, c_5)\}$. ■

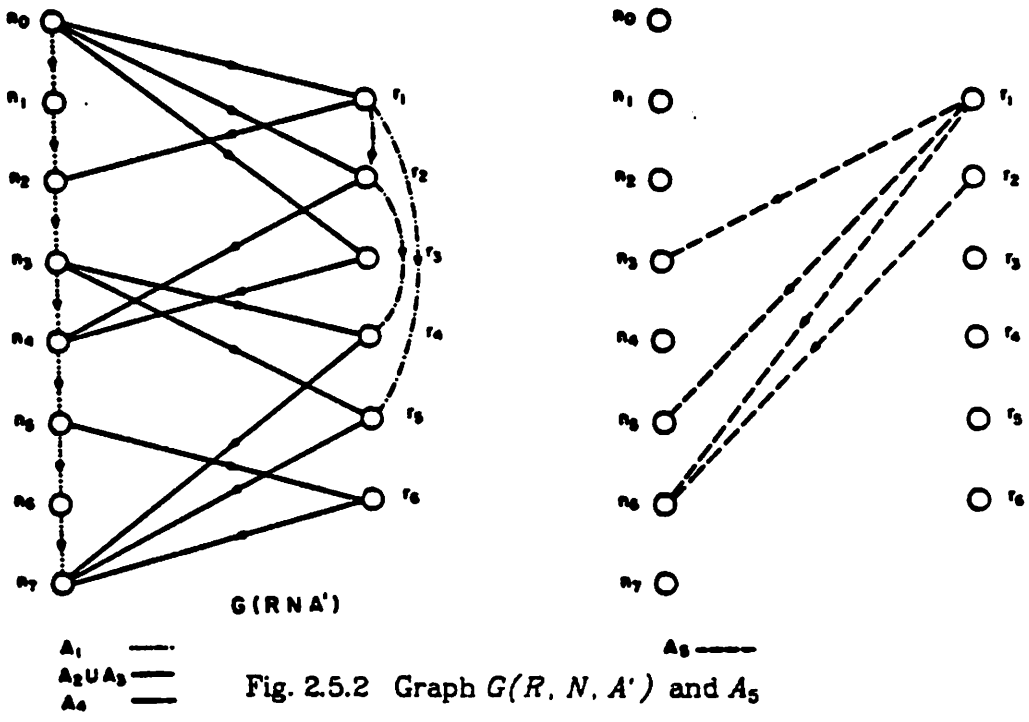


Fig. 2.5.2 Graph $G(R, N, A')$ and A_5

Note that an edge from a node in N (R) to a node in R (N) represents now a strict lower (upper) bound. If a lower (upper) bound on a row position is 1 (πr) , it can be represented by appending nodes n_0 (n_{nr+1}) to set N and by adding appropriate directed edges to A.

Moreover note that if a row, say $\tilde{\tau}$, has the position w as strict upper bound (i.e. $(\tilde{\tau}, n_w) \in A_3$) and must follow another row, say $\hat{\tau}$ (i.e. $(\hat{\tau}, \tilde{\tau}) \in A_4$), then row $\hat{\tau}$ has as strict upper bound a position lower or equal to $w-1$.

Example 2.5.1.3 : Row r_1 must be above r_2 which in turn must be above r_4. Since r_4 is required to be assigned to a position lower or

equal to 6, τ_1 must be assigned to a position lower or equal to 4. (In this case τ_1 has already the more stringent constraint to be in position 1).

■

Therefore: $A_5 = \{(\tau_k, n_{i-l}) | \exists \tau_j$ such that $(\tau_j, n_i) \in A_3$ and $\exists\ l+1$ distinct nodes in R along the directed paths in A_4 from τ_k to $\tau_j\}$. Similar considerations apply to lower bounds, but the assignment algorithm does not require that the set of directed edges is further increased.

Example 2.5.1.4 : The edges in subset A_5 are represented by dashed lines in Fig. 2.5.2b.

■

Our problem is to find an additional set of undirected edges E matching every node in R to one and only one node in N so that the resulting mixed graph $G(R, N, E, A)$ is acyclic.

Remark 2.5.1 : Column folding with bounded row assignment is equivalent to the sequencing problem with release times and deadlines where all task length are equal to one [GARE78][LAWL73b] and where a partial order on the tasks is given.

■

The following algorithm will either construct a set of undirected edges such that graph $G(R, N, E, A)$ is acyclic or will return a flag if no possible edge set exists. The in-degree of a node is the number of directed edges incident to that node and the deletion of a node from a graph corresponds to remove the node from the node set and all edges incident to/from it from the edge

set. The algorithm is described in Pidgin C.

ASSIGNMENT ALGORITHM

$E = \phi$;
delete n_o from graph G;
for ($i = 1 ; i \leq nr ; i = i + 1$) {
    if (*in-degree* (n_i) $\neq 0$) **return** (FALSE) ;
    $Q = \{ r \in R$; *in-degree* (r) $= 0 \}$;
    if ($Q = \phi$) **return** (FALSE) ;
    $r_j = r \in Q$ such that $(r_j, n_k) \in A$ and k is minimal;
    $E = E \cup \{ n_i, r_j \}$;
    delete n_i from graph G;
    delete r_j from graph G;
}
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 node in R (i.e. selects the product-row with lowest upper bound). The algorithm finds an implementable bounded row assignment, if one exists, as proven by the following theorem.

Theorem 2.5.1 : The Assignment Algorithm returns TRUE if and only if there exists a matching E such that graph $G(R, N, E, A)$ is acyclic.

Proof:

(if)

Suppose that the algorithm returns " false " at step i ; i.e. after having matched $i-1$ row nodes to position nodes. For the sake of contradiction, suppose that there exists a matching $E' = \{ \{r'_j, n_j\} , j = 1, 2, \cdots, nr\}$, such that $G(R, N, E', A)$ is acyclic.

The algorithm returns "false" in one of the following two cases:

Case 1 : $Q = \phi$ at step i.

There are $nr - i + 1$ row nodes that must be matched to position nodes $n_j, j>i$. Since $|\{n_j \in N, j>i\}| = nr - i$, no row assignment can be found satisfying the given bounds. In fact, since $\exists j > i$ such that (n_j, r'_i) $\in A$, then $[n_i, \cdots, n_j, r'_i, n_i]$ is a cycle in $G(R, N, E, A)$. Therefore we have a contradiction.

Case 2 : in-degree(n_i) $\neq 0$ at step i.

Let E^p be the partial assignment constructed by the algorithm, i.e. $E^p = \{\{n_j, r^p_j\}, j = 1, 2, \cdots, i-1\}$.

We show first that the matching E' can be transformed into another matching E'', such that $G(R, N, E'', A)$ is acyclic and the row nodes matched to $n_j, j = 1, 2, \cdots, i-1$ in E^p and E'' are identical. For this reason let:

$$a = arg \min \{j \mid r'_j \neq r^p_j\}$$

Nodes r'_a and r^p_a have no incoming directed edges from $\{n_j, j \geq a\}$.

Moreover $\exists n_h, n_k \in N, k \geq h > a$, such that $(r'_a, n_k) \in A$ and $(r^p_s, n_h) \in A$. Let $n_b \in N$, s.t. $\{n_b, r^p_s\} \in E'$. Then $a < b < h \leq k$. Let us consider the matching:

$$E'' = E' \cup \{r'_a, n_b\} \cup \{r^p_s, n_a\} - \{r'_a, n_a\} - \{r^p_s, n_b\}$$

We claim that $G(R, N, E'', A)$ is acyclic. If not, there would be at least a directed path joining one of the following node pairs:

i) n_b, r'_a

ii) n_a, r^p_s

iii) r'_a, n_b

iv) r^p_s, n_a

and $G(R, N, E', A)$ would have a cycle. In fact:

i) Since $b > a$ and there is a directed path from n_a to n_b, there would be the cycle $[n_b, r'_a, n_a, \cdots, n_b]$.

ii) Since r^p_s has no incoming directed edges from n_j, $j \geq a$ there would be a directed path from n_a to a node n_j, $j < a$ and therefore there would be the cycle $[n_a, \cdots, n_j, \cdots, n_a]$.

iii) Since r'_a has no directed edges into n_j, $j < h$, there would be a directed path from a node n_j, $j \geq h$ to n_b , and therefore there would be the cycle $[n_b, \cdots, n_j, \cdots, n_b]$.

iv) Since $b > a$ and there is a directed path from n_a to n_b, there would be the cycle $[r^p_s, n_a, \cdots, n_b, r^p_s]$.

Let now $\tilde{E}'' = \{\{n_j, r''_j\} \in E'', j = 1, 2, \cdots, i-1\}$. If $\tilde{E}'' = E^p$, then n_i has no incoming directed edges from $\{r''_j \in R \mid j > i\}$. Suppose that $\{(r''_k, n_i) \in A$ and $k > i$. Then $[r''_k, n_i, \cdots, n_k, r''_k]$ would be a cycle in $G(R, N, E'', A)$. We therefore have a contradiction. If $\tilde{E}'' \neq E^p$, then we can construct a finite sequence of matchings

E'', E''', $\cdots$, E^* using the procedure shown above , so that $G(R, N, E^*, A)$ is acyclic and $\tilde{E}^* = E^p$, where : $\tilde{E}^* = \{\{n_j, r_j^*\} \in E^*, j = 1, 2, \cdots, i-1\}$. Also in this case we have a contradiction.

(only if)

The algorithm terminates in a finite number of steps, because it attempts at most nr assignment. Let $E = \{ \{n_j, r_j\}, j = 1, 2, \cdots, nr\}$ be the assignment constructed by the algorithm. Since n_j and r_j have no incoming directed edges from $\{\{n_k \mid k>j\} \cup \{r_k \mid k>j\}$ $j = 1, 2, \cdots, nr\}$ by construction, then $G(R, N, E, A)$ is acyclic.

■

Example 2.5.1.5 : Consider the column folded logic array shown in Fig. 2.5.1, and the related graph $G(R, N, A)$ shown in Fig. 5.2. The implementable bounded-row assignments given by the algorithm is $(r_1, r_2, r_3, r_5, r_4, r_6)$.

■

The Assignment Algorithm replaces Step 3 of the Master Algorithm for column folding with bounded row assignment.

A different strategy for folding candidate selection is used. Since folding is limited by row positions, we try to fold columns incident to rows constrained to be in the top part of the array with columns incident to rows constrained to be in the bottom part of the array. Therefore two "induced bound" maps can be computed for each column:

$$\tilde{L}(c_j) = \min_{r \in R(c_j)} L_R(r) \qquad j = 1, 2, \cdots, nc.$$

$$\tilde{U}(c_j) = \max_{r \in R(c_j)} U_R(r) \qquad j = 1, 2, \cdots, nc.$$

The column with the lowest (highest) entry in $\tilde{U}$ $(\tilde{L})$ is the most constrained to be folded on the top (bottom).

Example 2.5.1.6 : For the logic array of Fig. 2.3.1 and the row bound maps of Example 2.5.1, the induced bound maps are the following:

$$\tilde{L} = 1, 1, 1, 1, 4, 1, 1, 1, 4, 1$$

$$\tilde{U} = 6, 3, 1, 3, 6, 6, 1, 6, 6, 6$$

Hence columns c_3 and c_7 are the most constrained to be folded on the top part of the array and c_5 and c_9 on the bottom.

■

Hence a "good" selection is the candidate pair (c_i, c_k) such that

$$c_i = arg_{j=1,2,\ldots,nc} \min \tilde{U}(c_j)$$

$$c_k = arg_{j=1,2,\ldots,nc} \max \tilde{L}(c_j)$$

A more considerate choice takes also care of the number of ancestor-descendant relations induced in the mixed graph, as shown in Section 4. Therefore a weighted selection criterion is used:

$$c_i = arg_{j=1,2,\ldots,nc} \min [\alpha |A(v_j)| + \beta \tilde{U}(v_j)]$$

$$c_k = arg_{j=1,2,\ldots,nc} \min [\alpha |\tilde{D}(v_j)| - \beta \tilde{L}(v_j)]$$

Example 2.5.1.7 : The first folding pair selected by the algorithm is (c_7, c_9).

■

Similar considerations apply, ***mutatis mutandis***, to the multiple folding candidate selection.

Remark 2.5.2 : The graph interpretation and an algorithm for the row folding with bounded column assignment problem can be derived ***mutatis mutandis*** from this problem.

■

2.5.2 Column folding with bounded connection-row assignment

This Section deals with a logic array implemented with connection-rows for routing input and output signals as described in Section 2. 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, constrained folding of one array only will be considered.

Both proposed architectures support at most as many connection-rows as product-rows. Since each column is contacted to a connection row, the number of columns in the considered array is required to be at most equal to the number of rows throughout this Section. Most PLAs satisfy this assumption.

A **connection-row assignment** is a one-to-one map: $T:\{c_i,\ i = 1, 2, \cdots, nc\} \rightarrow M \subseteq \{1, 2, \cdots, nr\}$ such that $j = T(c_i)$ if column c_i is contacted to the connection row in the j^{th} position.

Example 2.5.2.1 : Consider the OR plane of the PLA shown in Fig. 2.3.1. Fig. 2.5.3 shows the unfolded array with the connection-row assignment:

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

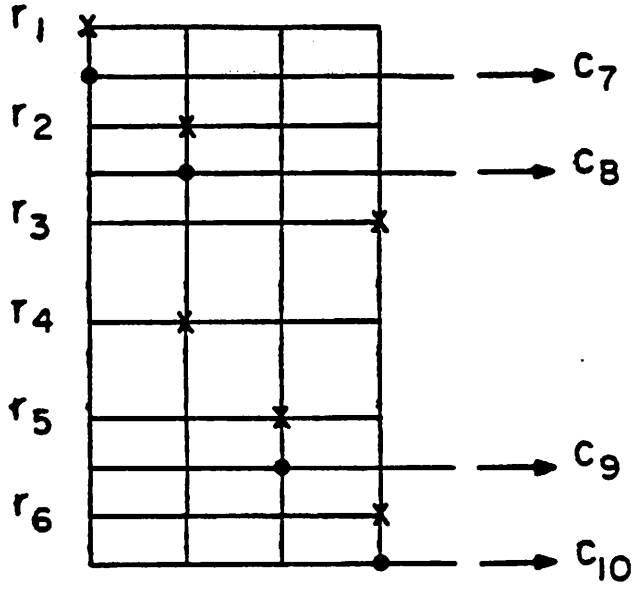


Fig. 2.5.3 Unfolded OR plane

The physical connection-row set M is 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.

A **lower (upper) connection-row bound map** is a map:

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

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

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

Example 2.5.2.2 : For the OR plane of the PLA shown in Fig. 2.3.1 , the following bounds are given:

$$L_C = 1, 1, 4, 6$$

$$U_C = 1, 3, 6, 6$$

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 :

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

$\forall$ column $c_{i,j}$ in folding list o_i with cardinality n, where by definition:

$$\max(P(R(c_{i,0}))) = 0 \quad \text{and} \quad \min(P(R(c_{i,n+1}))) = \infty$$

Example 2.5.2.3 : Consider the folded OR plane shown in Fig. 2.1.2 with the ordered folding set $O = \{(c_7, c_9), (c_8, c_{10})\}$. An implementable connection-row assignment would be:

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

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 contacted 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 :

$$L_C(c_j) \leq T(c_j) \leq U_C(c_j) \quad j = 1, 2, \cdots, nc$$

Example 2.5.2.4 : The implementable connection row-assignment of Example 2.5.2.3 does not satisfy the bounds given in Example 2.5.2.2. An implementable bounded connection row-assignment is:

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

Fig. 2.5.4 shows a folded implementation of the OR plane compatible with the bounded connection-row assignment.

■

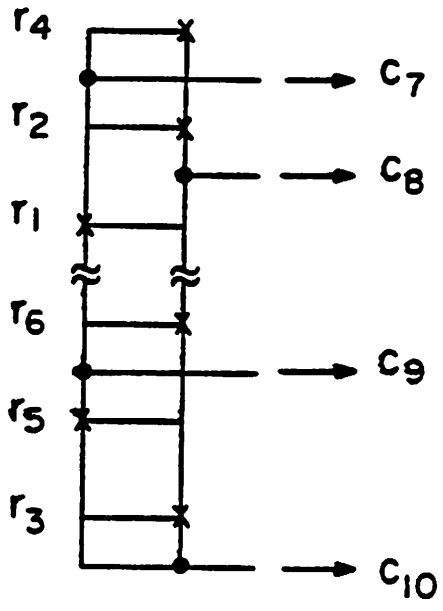


Fig. 2.5.4 Folded OR plane with bounded connection-row assignment

The column folding with bounded connection-row assignment problem can be stated 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.***

A graph interpretation of the following subproblem is considered:

> ***Given an ordered column-folding set and a 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 both sets influence each other. Hence the problem consists in finding the two row assignments compatible with the ordered column-folding set, if they exist.

This subproblem is represented by a directed graph $G(R, N, C, A)$, with three node sets R, N and C and a directed set of edges A. The node sets R, C and N are in one to one correspondence with the row set, the column set and the set of the first nr natural numbers respectively.

Bounds on the row positions are represented by a set of directed edges:

$$A = A_1 \cup A_2 \cup A_3 \cup A_4 \cup A_5 \cup A_6 \cup A_7 \cup A_8$$

where A_1 and A_4 are defined as in Section 2.5.1, $A_2 = \{(n_i, c_j) | L_C(c_j) = i+1 ;\ j = 1, 2, \cdots, nc\}$ and $A_3 = \{(c_j, n_i) | U_C(c_j) = i-1 ;\ j = 1, 2, \cdots, nc\}$ take into account the lower and upper bound maps.

Example 2.5.2.5 : Fig. 2.5.5a shows graph $G(R, N, , C, A')$, $A' = A_1 \cup A_2 \cup A_3 \cup A_4$ in the case that the OR plane of the PLA of Fig. 2.3.1 is folded and the ordered column-folding set $O = \{(c_7, c_9), (c_8, c_{10})\}$ is compatible with the bounds given in Example 2.5.2.2.

■

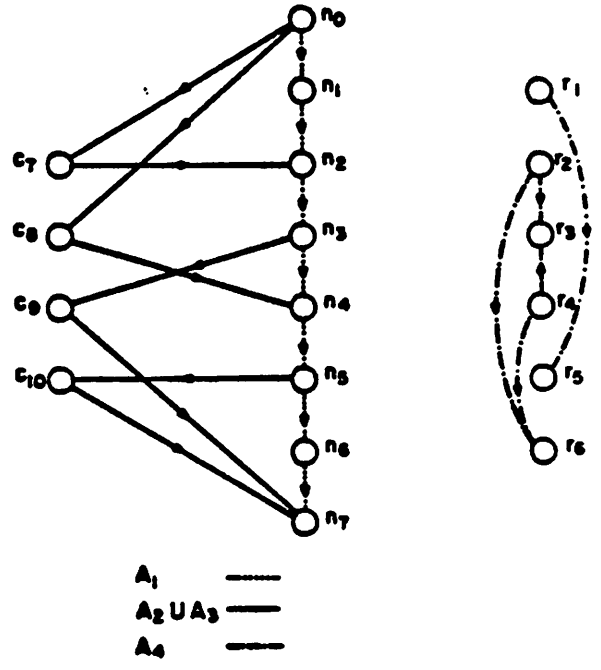


Fig. 2.5.5a Graph $G(R, N, , C, A')$

The mutual relations among products and connection-rows is represented by the edge subsets: $A_6 = \{(\tilde{r}, \hat{c}) | \tilde{r} \in R(\tilde{c})$ and $\tilde{c}$ is split on top of $\hat{c}\}$ and $A_7 = \{(\hat{c}, \tilde{r}) | \tilde{r} \in R(\tilde{c})$ and $\hat{c}$ is split on top of $\tilde{c}\}$. In words , if column $\tilde{c}$ is folded on top of $\hat{c}$, then all the rows (product and connection) connected to $\tilde{c}$ must be assigned to positions with index lower than the positions of all the rows connected to $\hat{c}$.

Example 2.5.2.6 : Fig. 2.5.5b shows the edges in subsets A_6 and A_7 for the problem of Example 2.5.2.5.

■

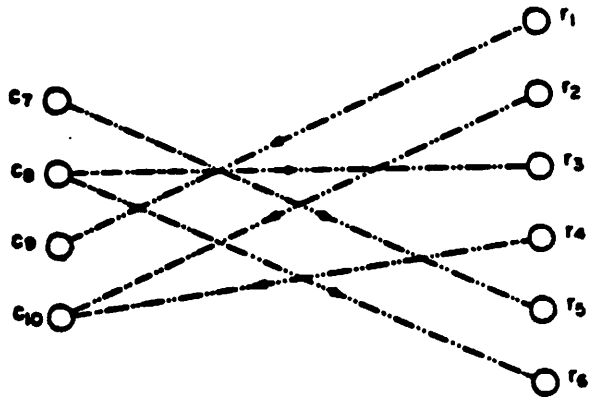


Fig. 2.5.5b Edge sets A_6 and A_7

Moreover note that if a column , say $\hat{c}$, has as strict upper bound the position w (i.e. $(\hat{c}, n_w) \in A_3$), $(\tilde{r}, \hat{c}) \in A_6$ and $(r, \tilde{r}) \in A_4$, then r has as upper bound the position $w-2$. Therefore: $A_5=\{(r_k, n_{i-l}) | \exists \tilde{r}$ ($\tilde{r}$ not necessarily distinct from r_k) and $\exists \hat{c}$ such that $(\tilde{r}, \hat{c}) \in A_6$, $(\hat{c}, n_i) \in A_3$ and $\exists l+1$ distinct nodes along the directed paths in $A_4 \cup A_6$ from r_k to $\hat{c}\}$. The edges in this set represent the upper bounds on the position of each product-row induced by folding. Note that all nodes in R must be assigned to a position lower than $nr+1$. Hence the edges (r_k, n_{nr+1}) $\forall r_k \in R$ having no explicit upper bound are appended to A_5.

Example 2.5.2.7 : Fig. 2.5.5c shows the edges in subset A_5 for the problem of Example 2.5.2.5.

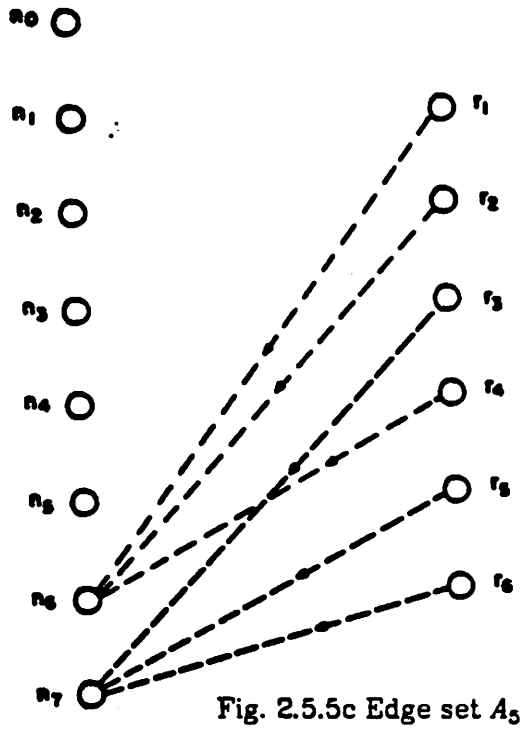


Fig. 2.5.5c Edge set A_5

Similarly , upper bounds induced on the column positions are represented by: $A_8 \equiv \{(c_k, n_{i-l}) | \exists l > 0$ nodes $\hat{r} \in R$, such that $(c_k, \hat{r}) \in A_7$ and $(\hat{r}, n_i) \in A_5\}$.

Example 2.5.2.8 : Fig. 2.5.5d shows the edges in subset A_8 for the problem of Example 2.5.2.5.

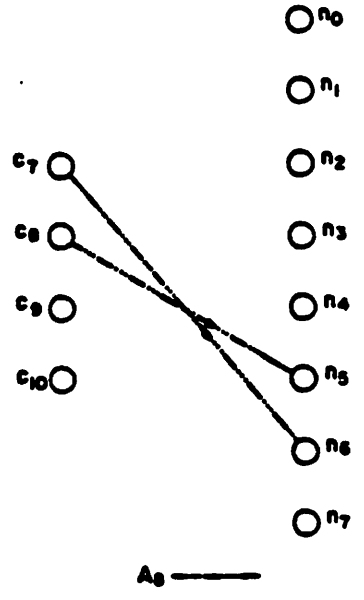


Fig. 2.5.5d Edge set A_8

In graph terms, this problem is to find a set of undirected edges E matching every node in R and in C to one and only one node in N so that the resulting mixed graph $G(R, N, C, E, A)$ is acyclic. Note that in general the number of columns and hence of physical connection-rows required is smaller than the number of rows by Δ and the double assignment algorithm takes advantage of this.

DOUBLE ASSIGNMENT ALGORITHM

$E = \phi$;

$\Delta = nr - nc$;

delete n_0 from graph G;

for ($i = 1$; $i \leq nr$; $i = i + 1$) {

if (*in-degree* (n_i) $\neq 0$) **return** (FALSE);

$Q = \{r \in R$; *in−degree* (r) $= 0\}$;

if ($Q = \phi$) **return** (FALSE) ;

$r_j = r \in Q$ such that $(r_j, n_k) \in A$ and k is minimal;

$E = E \cup (n_i, r_j)$;

$H = \{c \in C$; *in−degree* (c) $\neq 0\}$;

if ($H = \phi$) {

$\Delta = \Delta - 1$;

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

}

else {

$c_l = c \in H$ such that $(c_l, n_k) \in A$ and k is minimal ;

$E = E \cup (n_i, c_l)$;

delete c_l from graph G;

}

delete r_j from graph G;

delete n_i from graph G;

}

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 or at most equal to Δ.

Theorem 2.5.2: The assignment algorithm returns TRUE if and only if there exists a set of undirected edges E matching each node in R and in C to one and only one node in N such that $G(R, N, C, E, A)$ is acyclic.

Proof:

(if)

Suppose that the algorithm returns " false " at step i . For the sake of contradiction, suppose that there exists a matching E' such that $G(R, N, C, E', A)$ is acyclic. In particular:
$E' = \{ \{r'_j, n_j\} , j = 1, 2, \cdots, nr\} \cup \{ \{c'_j, n_j\},$
$\forall j \in M' \subseteq \{1, 2, \cdots, nr\} \}$, where M' is the physical connection row set corresponding to the matching E'.
The algorithm returns "false" in one of the following three cases:

Case 1 : $Q = \phi$ at step i.

There are $nr - i + 1$ row nodes that must be matched to position nodes $n_j, j > i$. Since $|\{n_j \in N, j > i\}| = nr - i$, no row assignment can be found satisfying the given bounds. In fact, since $\exists j > i$ such that $(n_j, r'_i) \in A$, then $[n_i, \cdots, n_j, r'_i, n_i]$ is a cycle in $G(R, N, C, E', A)$. We therefore have a contradiction.

Case 2 : $H = \phi$ and $\Delta<0$ at step i.

There are $nr-i+1$ connection-row nodes that must be matched to position nodes $n_j, j>i$. Since $|\{n_j \in N, j>i\}| = nr-i$, no connection-row assignment can be found satisfying the given bounds. In fact, since $\exists j>i$ such that $(n_j, c'_i) \in A$, then $[n_i, \cdots, n_j, c'_i, n_i]$ is a cycle in $G(R, N, C, E', A)$. We therefore have a contradiction.

Case 3 : in-degree(n_i) $\neq 0$ at step i.

Let E^p be the partial assignment constructed by the algorithm.
We show first that the matching E' can be transformed into another matching E'', such that $G(R, N, C, E'', A)$ is acyclic and row and connection-row nodes matched to $n_j, j = 1, 2, \cdots, i-1$ in E^p and E'' are identical. For this reason let:

$$a = arg \min \{j \mid r'_j \neq r^p_j\}$$

$$d = arg \min \{j \mid c'_j \neq c^p_j \text{ or } \{c'_j, n_j\} \notin E' \text{ and } \{c^p_j, n_j\} \in E^p\}$$

If ($a \leq d$) let:

$$E'' = E' \cup \{r'_a, n_b\} \cup \{r^p_a, n_a\} - \{r'_a, n_a\} - \{r^p_a, n_b\}$$

If ($d<a$), $c'_d \neq c^p_g$ and $\{c'_d, n_d\} \in E'$ let:

$$E'' = E' \cup \{c'_a, n_e\} \cup \{c^p_g, n_d\} - \{c'_a, n_d\} - \{c^p_g, n_e\}$$

where $n_e \in N$ s.t. $\{n_e, c^p_g\} \in E'$.

If ($d<a$), $\{ c'_d, n_j\} \notin E'$ and $\{ c^p_g, n_j \} \in E^p$ let:

$$E'' = E' \cup \{c^p_g, n_d\} - \{c^p_g, n_e\}$$

We can show with an argument similar to the one used in the proof of

theorem 5.1, that graph $G(R, N, C, E'', A)$ is acyclic, because otherwise graph $G(R, N, C, E', A)$ would have a cycle and violate our assumption.

Let now $\tilde{E}'' \subseteq E''$ be the subset of the undirected edges having an end-point in n_j, $j = 1, 2, \cdots, i-1$. If $\tilde{E}'' = E^p$, then n_i has no incoming directed edges from $\{r''_j \in R \mid j>i\} \cup \{c''_j \in C \mid j>i\}$. Suppose that $\{(r''_k, n_i) \in A$ and $k>i$. Then $[r''_k, n_i, \cdots, n_k, r''_k]$ would be a cycle in $G(R, N, C, E'', A)$. Suppose that $\{(c''_k, n_i) \in A$ and $k>i$. Then $[c''_k, n_i, \cdots, n_k, c''_k]$ would be a cycle in $G(R, N, C, E'', A)$. We therefore have a contradiction.

If $\tilde{E}'' \neq E^p$, then we can construct a finite sequence of matchings $E'', E''', \cdots, E^*$ using the procedure shown above, so that $G(R, N, C, E^*, A)$ is acyclic and $\tilde{E}^* = E^p$, where : $\tilde{E}^* \subseteq E^*$ is the subset of the undirected edges having an end-point in n_j, $j = 1, 2, \cdots, i-1$. Also in this case we have a contradiction.

(only if)

The algorithm terminates in a finite number of steps, because it attempts at most $2 * nr$ assignment. Let E be the assignment constructed by the algorithm. Since n_j, r_j and c_j have no incoming directed edges from $\{\{n_k \mid k>j\} \cup \{r_k \mid k>j\} \cup \{c_k \mid k>j\}$ $j = 1, 2, \cdots, nr$ $\}$ by construction, then $G(R, N, C, E, A)$ is acyclic.

■

The double assignment algorithm replaces Step 3 of the Master Algorithm for column folding with bounded connection-row assignment

The selection of folding candidates is based on the following strategy. Try to fold columns incident to connection-rows constrained to be in the top part of the array with columns connected to connection-rows constrained to be in the bottom part of the array. Therefore the candidate selection follows the outlines presented in Section 5.1, where $\tilde{L}(c_j) = L(c_j)$ and $\tilde{U}(c_j) = U(c_j)$ $j = 1, 2, \cdots, nc$. Also in this case, a considerate choice of folding candidates uses a selection criterion weighting the number of ancestor-descendant relations induced by the fold and the required row positions in the array.

2.5.3 Column folding with ordered connection-row assignment

The considerations on multiple column folded PLA implementation and the basic definitions presented in Section 5.2 are extended to this Section.

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

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

Example 2.5.3.1 : Consider the OR plane of the PLA shown in Fig. 2.3.1 and the following order map:

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

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 the order. Fig. 2.5.6 shows a folded im-

plementation with the implementable ordered connection-row assignment: $T(c_7) = 2$ $T(c_8) = 1$ $T(c_9) = 3$ $T(c_{10}) = 4$.

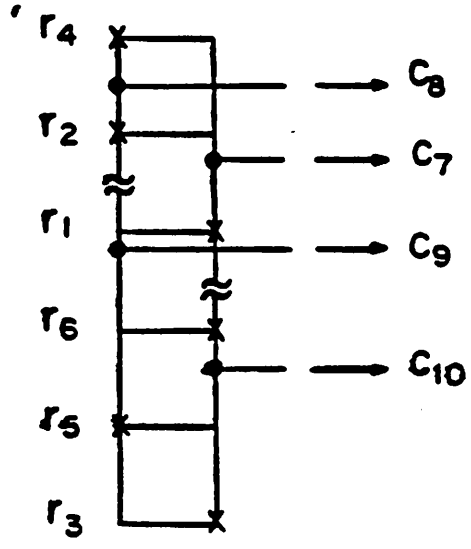


Fig. 2.5.6 Folded OR plane implementation

The column folding with ordered connection-row assignment problem can be stated 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:

$$L_C(c_i) = S(c_i) \quad \forall i = 1, 2, \cdots, nc$$

$$U_C(c_i) = S(c_i) + \Delta \quad \forall i = 1, 2, \cdots, nc$$

with the additional constraint on the order of the connection-rows.

A graph representation of a subproblem is considered:

Given an ordered column-folding set and an order map, find an implementable ordered connection-row assignment, if it exists.

The graph representation of this subproblem is given by graph $G(R, N, C, A)$ introduced in Section 5.2 where an additional subset of directed edges is added to take care of the order map:

$$A_0 \equiv \{(c_i, c_j) | i = S(c_k), j = S(c_{k+1}), k = 1, 2, \cdots, nc - 1\}$$

The Double Assignment Algorithm can be used to replace Step 3 of the Master Algorithm for the column folding with ordered connection-row assignment problem.

Example 2.5.3.2 : Fig. 2.5.7 shows graph $G(R, N, C, A)$ for the order map of Example 2.5.3.1 and the ordered folding set $O = \{(c_8, c_9)\}$ ■

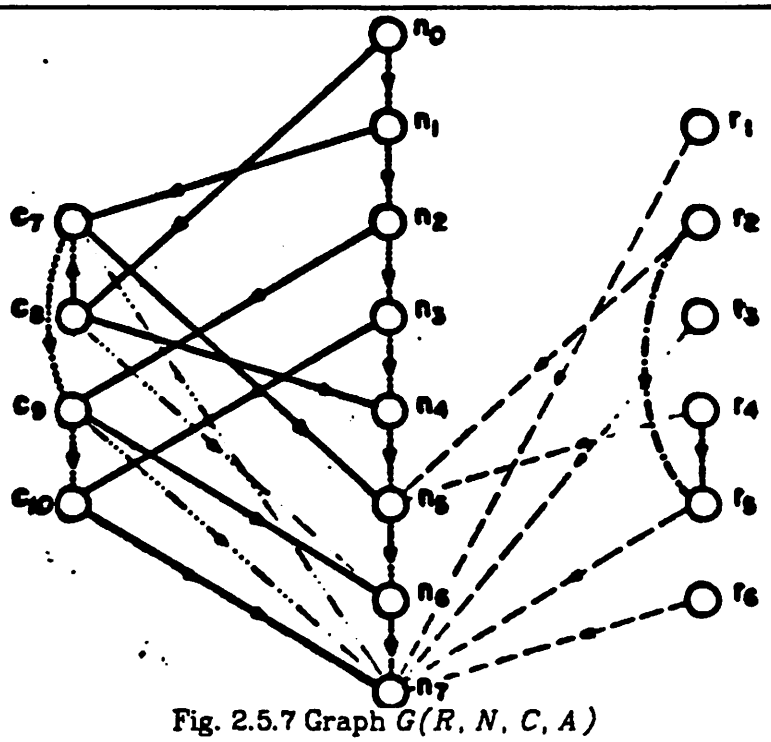


Fig. 2.5.7 Graph $G(R, N, C, A)$

Remark 2.5.3 : In the case that there are no slack positions or in the case that we are not interested in taking advantage of the slack positions, the column-folding with ordered connection-row assignment problem can be solved more easily by the following equivalent formulation: ***column folding with bounded product-row assignment, where bounds on row positions are dynamically induced by column-folding.*** In particular:

$$U_R(c_{i,j}) = S(c_{i,j+1}) + \delta - 1$$

$$L_R(c_{i,j+1}) = S(c_{i,j}) + \delta + 1$$

$\forall c_{i,j} \in o_i$, $\forall o_i \in O$ and *any fixed* δ *s.t.* $0 \leq \delta \leq \Delta$

An implementable product-row assignment satisfying the above bounds is a necessary and sufficient condition for the existence of the implementable ordered connection-row assignment $T(c_j) = S(c_j) + \delta$.

■

The selection of folding candidates is based on the following strategy. Try to fold columns incident to connection-rows constrained to be in the top part of the array with columns connected to connection-rows constrained to be in the bottom part of the array. Therefore the candidate selection follows the outlines presented in Section 5.1, where now: $\tilde{L}(c_j) = S(c_j)$ and $\tilde{U}(c_j) = S(c_j)$ $j = 1, 2, \cdots, nc$. Also in this case, a considerate choice of folding candidates uses a selection criterion weighting the number of ancestor-descendant relations induced by the fold and the required row positions in the array.

2.6. PLEASURE

PLEASURE is an interactive program for simple/multiple constrained/unconstrained row and/or column folding of Programmable Logic Arrays.

The PLA description is given as input to the program in the form of two-level sum-of-products logical implicants. The output file of logic minimizer POP can be used as input to PLEASURE.

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 ad connection rows. The symbolic table 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 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 row 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 PLA's.

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 on 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. Simply 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.

Program PLEASURE is coded in *ratfor* and consists of about 5000 lines. Intermediate code in *fortran* 77 is available. PLEASURE runs in a VAX-UNIX® environment, but is easily transportable to other machines.

We consider now a simple example to show how the folded array structure can be implemented. The representation of a PLA as a sum-of-product logical implicants is shown in Fig. 2.6.1.

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

Fig. 2.6.1 Logical PLA description

The topological structure is represented in Fig. 2.1. Assume that the rows and the columns of the array have to be folded so that:

i) the folded array has an AND-OR-AND structure;

ii) inputs to columns 1, 2 and 4 are connected from the left side of the array;

iii) inputs to columns 3, 5 and 6 are connected from the right side of the array;

iv) outputs from columns 8 and 9 are connected to the top of the array;

v) outputs from columns 7 and 10 are connected to the bottom of the array.

The PLEASURE output file is shown in Fig. 6.2. Fig. 2.6.3 shows a nMOS implementation according to the design rules suggested in [MEAD80].

The same PLA can be folded with the additional constraint that input connection-rows are positioned according to the input-column label order. Fig. 6.4 shows the PLEASURE output file. Note that the additional constraint leads to a less compacted structure. For example, it is not possible to fold the bottom two rows in Fig. 6.4, because two connection-rows are needed to contact both column segments c_2 and c_4 and be positioned below the fourth row from the top.

The layout of small folded arrays (Fig. 2.6.3) shows that a considerable area fraction is taken by the extra contacts and power and ground lines required by the folded structure. However this overhead is negligible in larger arrays.

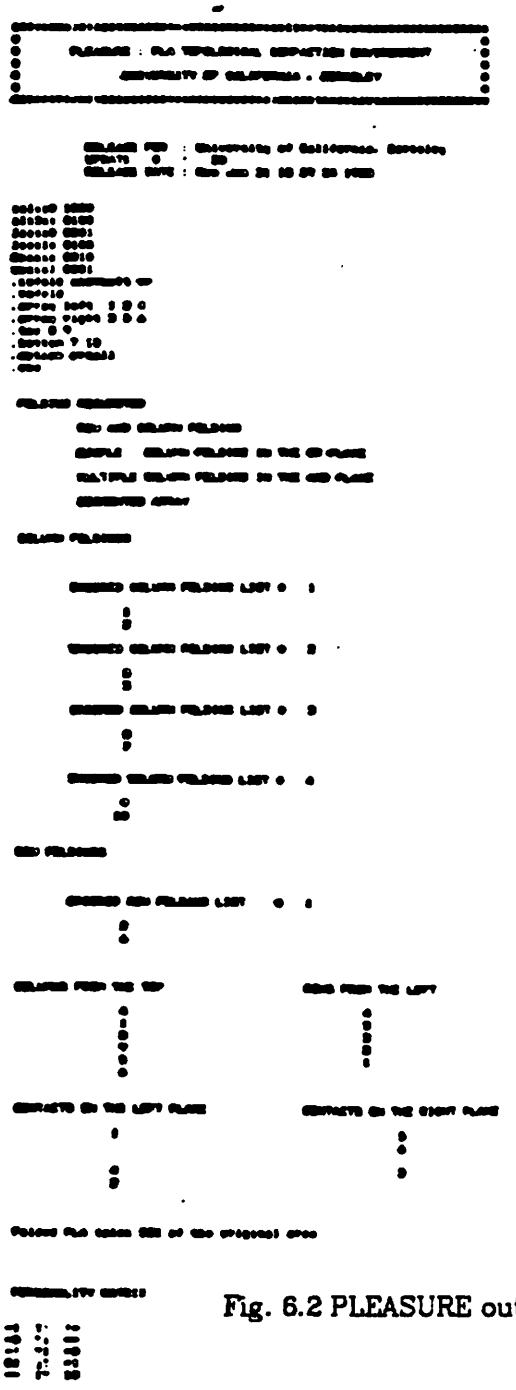


Fig. 6.2 PLEASURE output file

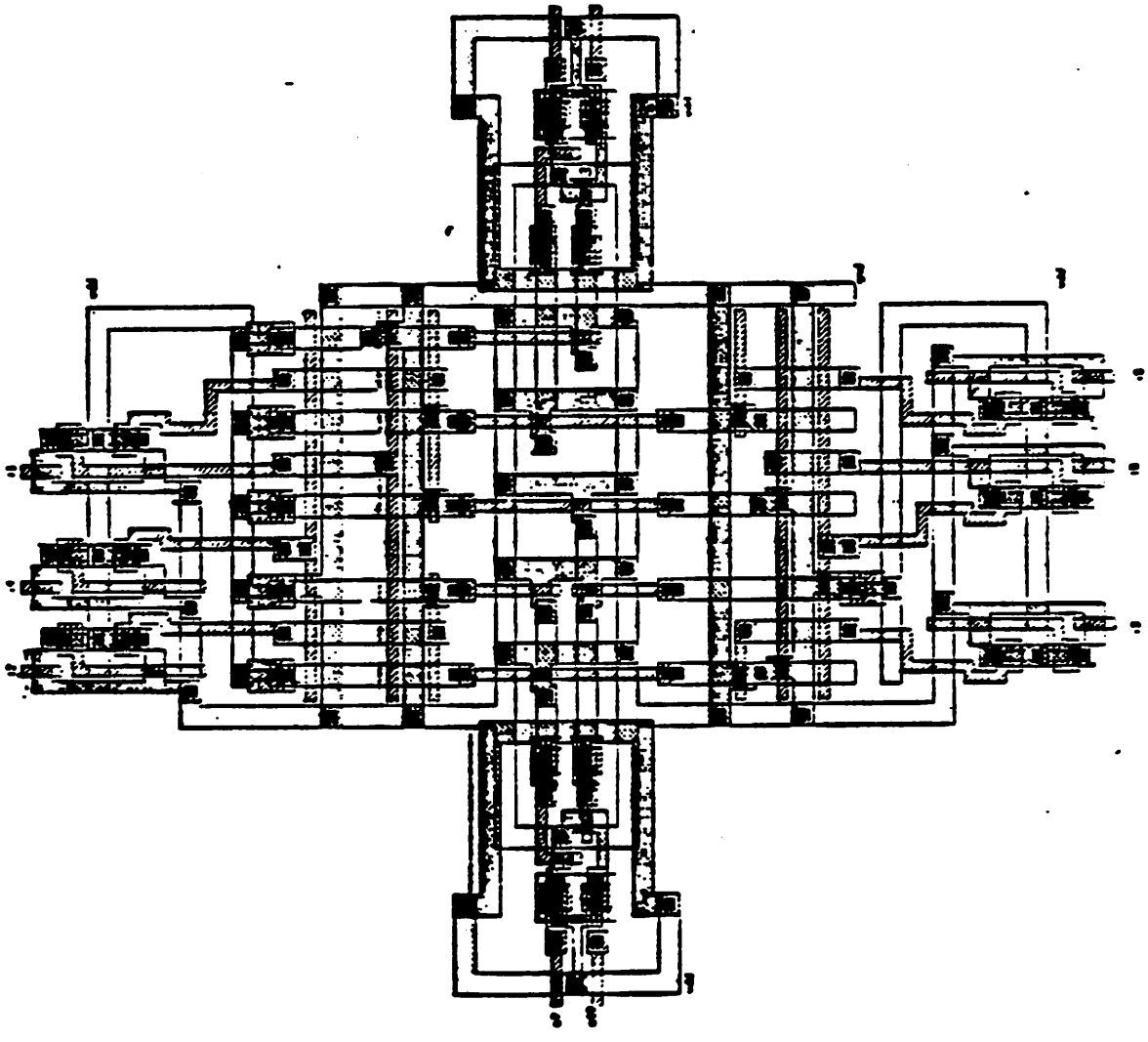


Fig. 2.6.3 Layout of the folded PLA

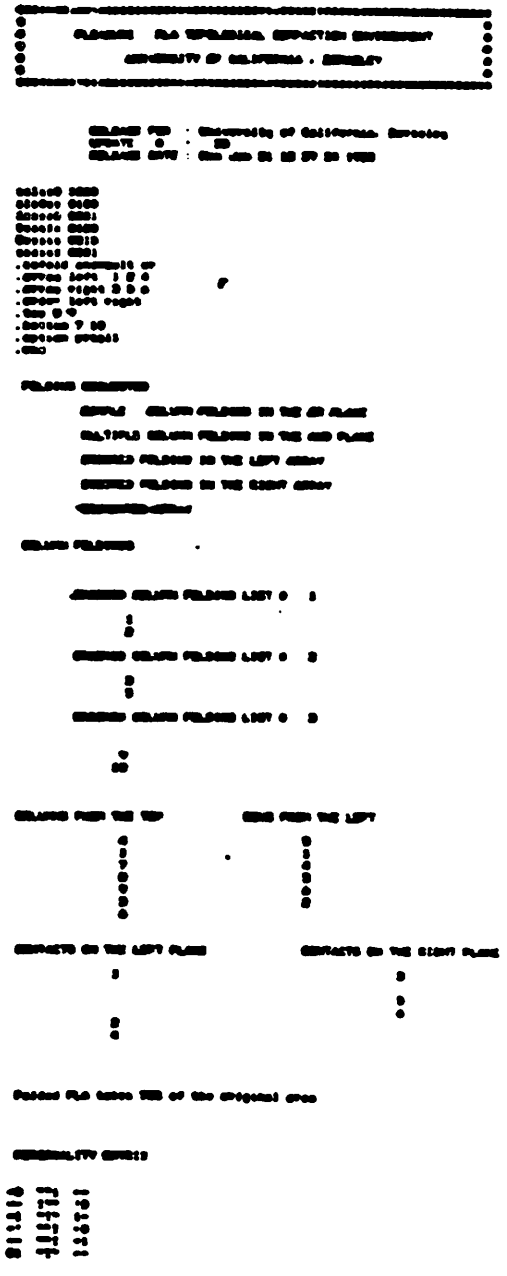


Fig. 6.4 PLEASURE output file

2.7. EXPERIMENTAL RESULTS

Some PLEASURE output files are reported in App. B . 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: $L(r_i) = max(i-\alpha,0)$; $U(r_i) = min(i+\alpha,nr)$; $\alpha = \frac{nr}{10}$; iii) column folding with constrained connection-row positions: $L_C(c_i) = max(i-\alpha,0)$; $U_C(c_i) = min(i+\alpha,nr)$; $\alpha = \frac{nr}{10}$; iv) column folding with ordered connection-row assignment: $S(c_i) = i$, $i = 1, 2, \cdots, nc$. The folding results and execution time on a VAX 11/780 computer are reported in Table 2.1.

Comparison of PLAs folded by PLEASURE with different constraints.

PLA	size nr*(ni+no)	Constraints	Folding lists	Folded_Area Unfolded Area = 100	Time (sec)
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

TABLE 2.1

CHAPTER 3

ARRAY PARTITIONING

3.1. PROGRAMMABLE LOGIC ARRAYS PARTITIONING

There are two motivations in exploiting partitioning techniques for PLAs. The first is to explore and compare a topological compaction technique alternative to folding. The second is to provide a means of designing PLAs partitioned in subarrays of bounded size, when a technological limitation on the array size is given.

Kang approached for the first time the PLA partitioning problem [KANG81]. He proposed two heuristic partitioning algorithms whose objectives were PLA area reduction only. The partitioning technique reported in this Chapter introduces a graph theoretic interpretation of the problem and addresses both PLA-area reduction and design of bounded size arrays.

Programmable Logic Array folding allows to compact an array by exploiting its sparsity only. Array partitioning techniques attempt to achieve a PLA implementation in a minimal area, by taking advantage of the knowledge of array connectivity in exploiting the array sparsity.

A logical array (or a plane of a logical array) is **connected** if the rows of the topological personality matrix (input/output TPM) cannot be partitioned into two or more subsets, having non-zero entries only in mutually disjoint column subsets. It is evident that disconnected arrays can be implemented by an appropriate connection of smaller arrays. It is also easy to show that such an implementation is always convenient in terms of silicon area

[KANG81]. However a good logical design of a switching function seldom leads to a disconnected array. On the other hand, several PLAs show weakly-connected structures. In this case, PLA partitioning techniques allow to transform the array into a disconnected one, which has a convenient implementation in terms of silicon area.

PLA partitioning does not exclude folding. In particular, partitioned arrays can be folded to achieve an ultimate array compaction. As an example, Suwa presented a technique to obtain segmented-folded array in [SUWA81]. In particular PLAs are partitioned into blocks, and column pairs folded inside each block.

Partitioning a PLA may be a design necessity, in order to satisfy some technological constraints. In particular, timing delay through an array grows with the array size. Therefore it is often necessary to set an upper bound on the physical array size and partition the original array into sub-units, which satisfy the given size bounds.

The PLA structure partitioning problem is addressed first. Area-effective implementations of partitioned arrays are presented in Section 3.5.

3.2. BASIC CONCEPTS AND DEFINITIONS

The PLA structure is represented by a set of rows and columns of the input and output topological personality matrix, which are referred to in this Chapter as $A \in \{1,0\}^{P \times N}$ and $B \in \{1,0\}^{P \times M}$ respectively. The **input** array (**output** array) column set is defined as: $I=\{i_1, i_2, \ldots, i_N\}$ ($O=\{o_1, o_2, \ldots, o_M\}$). The row set of the topological personality matrix $[A|B]$ is denoted by: $P=\{p_1, p_2, \ldots, p_P\}$. A row p_j is split into two parts: p_j^A and p_j^B corresponding to the input and output topological personality rows.

The logical disjunction (conjunction) of two 1-0 vectors x,y, $x \vee y$ $(x \wedge y)$ is defined to be the vector obtained by the component-wise disjunction (conjunction) of x and y. The logical disjunction (conjunction) of n vectors $x_1, x_2, \cdots, x_n$ will be indicated as: $\vee_{i=1}^{n} x_i$ $(\wedge_{i=1}^{n} x_i)$. Two vectors x,y are **independent** (orthogonal) if $x \wedge y = 0$, where 0 is the null vector. Two independent vectors are denoted by $x \perp y$. Two vector sets X,Y are **independent** if $x \perp y$, $\forall x \in X$ and $\forall y \in Y$

Logic array partitioning relies on determining independent sets of vectors in the topological personality matrix. 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-rows $p_j^A (p_j^B)$. A logic array is said to be **partitionable** if there exist row independent sets.

Remark 3.2.1: A partitionable array is input and output partitionable, but the inverse is not true because input independent row sets and output independent rows sets not necessarily coincide. ■

3.3. EQUIVALENT ARRAYS AND PARTITIONING

In general the TPMs of logic arrays do not have input and/or output independent sets of products 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] [SUWA82] or by rearranging the TPM of the array by a reshape [HONG74] 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 substitution of the input, output column or product row with a set of input, output columns or product rows which gives an equivalent logic array. Three rules to obtain equivalent arrays by augmentation are defined here:

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 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 array 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 disjunction 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 array 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

$$\vee_{k=1}^{s} p_{jk}^{B} = p_j^{B}$$

$$p_{jk}^{A} = p_j^{A} \quad \forall k = 1,2,\ldots,s$$

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. 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.

3.4. GRAPH INTERPRETATION OF THE PARTITIONING PROBLEM

A graph interpretation of the partitioning problem gives a pictorial representation of the array connectivity and is useful in understanding the underlying structure.

The AND plane (OR plane) of a PLA can be represented by a bipartite graph $G^A(I,P,E^A)$ $(G^B(P,O,E^B))$ whose adjacency matrix is: $\begin{bmatrix} 0 & A^T \\ A & 0 \end{bmatrix}$ ($\begin{bmatrix} 0 & B \\ B^T & 0 \end{bmatrix}$). The whole logic array is therefore represented by the union of such graphs, i.e. the tripartite graph $G(I,P,O,E)$, where $E=E^A \cup E^B$, as shown in Fig. 3.4.1.

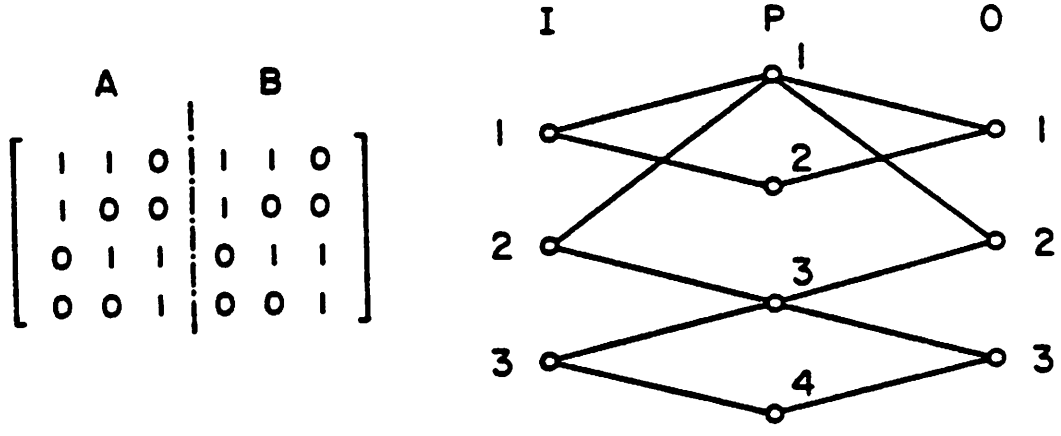


Fig. 3.4.1 Tripartite graph $G(I,P,O,E)$

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.

In order to give an estimate of the silicon area taken by the PLA, an area function F_0 on G is defined as follows:

$$F_0 = (a|I|+b|O|)|P| + c|I| + d|O| + e|P|$$

where coefficients $a-e$ are parameters depending on the physical layout of the PLA. The first term takes into account the area of the array and the last three terms the area taken by the drivers, the output inverters and the loads.

3.4.1 Input partitioning

In this case, only graph $G^A(I,P,E^A)$ is considered, because input partitioning does not affect the OR plane. Let us consider first the trivial case in which set P is the disjoint union on n input independent sets P_j , $j=1,2,...,n$. Because of independence, input columns are also partitioned into n disjoint sets I_j. As a consequence graph G^A is disconnected into subgraphs $G_j^A = (I_j, P_j, E_j^A)$ $j=1,2,...,n$. Each subgraph G_j^A represents a block of an input partitioned PLA. It is straightforward that in this case an input partitioned array takes an area smaller than the original one.

However, in general, graph G is connected and the input array is not partitionable. A transformation of the input array into an equivalent input partitionable one is then required: this corresponds to transform graph G^A into an equivalent disconnected one. This goal can be achieved by an input node splitting which is the counterpart of the input augmentation.

Example 3.4.1: Input node 2 is split into two nodes 2' and 2'' (PLA input column augmentation) and the edges incident to 2 are now incident either to 2' or to 2''. The equivalent augmented PLA is shown in Fig. 3.4.2 with its input partitioned implementation.

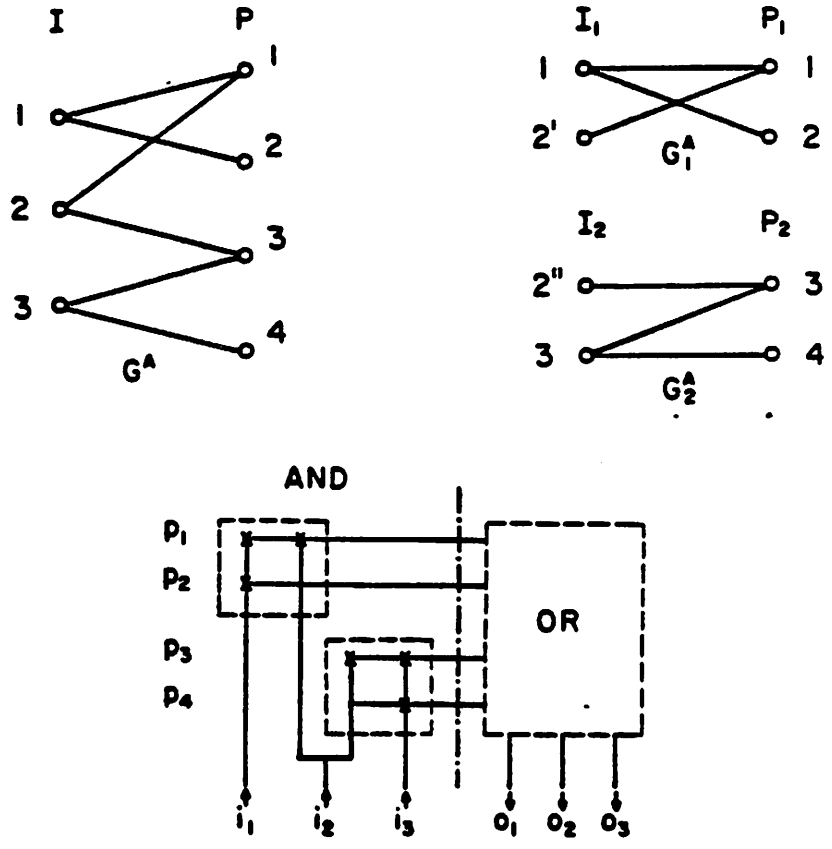


Fig. 3.4.2 Input partitioning

In general let us denote by $\Pi_n(E^A)$ a partition of the edge set E^A into n subsets $E_1^A, E_2^A, \ldots, E_n^A$. Let $G_j^A(I_j, P_j, E_j^A)$ be any subgraph induced by the partition where I_j and P_j are the sets of input and product nodes which are adjacent to edges in E_j^A. Because of input node splitting in general $|I| \leq \sum_{j=1}^{n} |I_j|$ while $|P| = \sum_{j=1}^{n} |P_j|$ (no product augmentation is allowed). Subgraphs

G_j^A, $j=1,2,...,n$ correspond to the blocks of the input partitioned array. An estimate of the input partitioned array area is given by:

$$F^A = \sum_{j=1}^{n} |P_j|(a|I_j|+b|O|) + c\sum_{j=1}^{n}|I_j| + d|O| + e|P|$$

$$+ f(\sum_{j=1}^{n}|I_j|-|I|)$$

where the last term takes into account the overhead due to the routing of the augmented input columns.

The input partitioning optimization problem OP1 can be stated as follows:

"Problem OP1"

Find a partition $\Pi_n^*(E^A)$ such that:

$$F^A(\Pi_n^*(E^A)) \leq F^A(\Pi_n(E^A)) \quad \forall \Pi_n(E^A) \text{ and } \forall n$$

$$P_j \cap P_k = \phi \quad \forall j,k=1,2,...,n;\ j \neq k$$

■

Note that the optimal solution can be not unique.

3.4.2 Output partitioning

In this case only graph $G^B(P,O,E^B)$ is considered. As stated in Section 3.3, an output-partitionable array can be obtained by a sequence of output-columns and/or product-row augmentations.

Example 3.4.2: Product node 1 is split into two nodes 1' and 1" (product row augmentation) and the edges incident to 1 are now incident either to 1' or to 1". The equivalent augmented PLA is shown in Fig.

3.4.3 with its output-partitioned implementation.

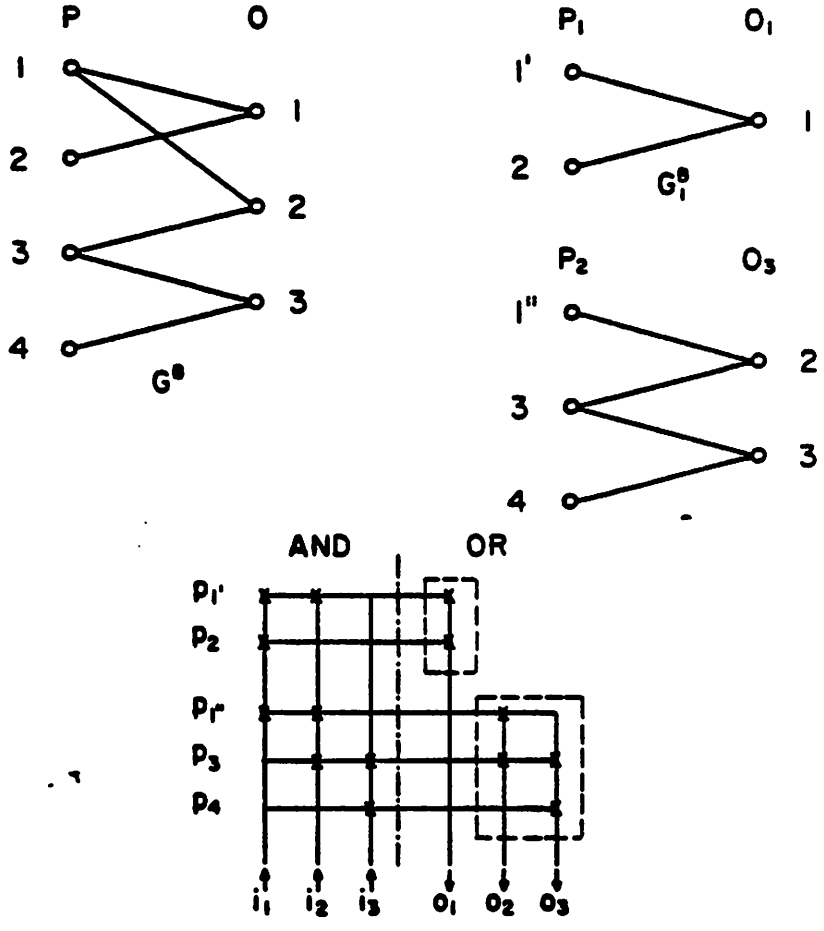


Fig. 3.4.3 Output partitioning

In general let us denote by $\Pi_m(E^B)$ a partition of the edge set E^B into m subsets $E_1^B, E_2^B, \ldots, E_m^B$. Let $G_j^B(P_j, O_j, E_j^B)$ be any subgraph induced by the partition where P_j and O_j are the sets of product and output nodes which are

adjacent to edges in E_j^B. Because of output node splitting in general $|O| \leq \sum_{j=1}^{m} |O_j|$ and $|P| \leq \sum_{j=1}^{m} |P_j|$. Subgraphs G_j^B , $j=1,2,...,m$ correspond to the blocks of the output partitioned array.

An estimate of the output partitioned array area is given by:

$$F^B = \sum_{j=1}^{m} |P_j|(a|I|+b|O_j|) + c|I| + d\sum_{j=1}^{m} |O_j| + e\sum_{j=1}^{m} |P_j|$$

$$+ g[(\sum_{j=1}^{m} |O_j|)-|O|] + h[(\sum_{j=1}^{m} |P_j|)-|P|]$$

where the last terms take into account the overhead due to the routing of the augmented output columns and product rows. The output partitioning optimization problem OP2 can be stated as follows:

"Problem OP2"

Find a partition $\Pi_m^*(E^B)$ such that:

$$F^B(\Pi_m^*(E^B)) \leq F^B(\Pi_m(E^B)) \quad \forall \Pi_m(E^B) \text{ and } \forall m$$

■

Note that the optimal solution can be not unique.

Remark 3.4.1: If only output column augmentations are allowed $|P| = \sum_{j=1}^{m} |P_j|$ and therefore F^B can be obtained from F^A by interchanging I with O. In this case the output partitioning is exactly the ***dual*** of the input partitioning. Problem OP2 is then obtained from the problem OP1 by adding the constraint equation

$$P_j \cap P_k = \phi \quad \forall j,k=1,2,\ldots,n;\ j \neq k$$

■

3.4.3 Partitioning

The whole logic array is represented by graph $G(I,P,O,E)$. A partitionable array is obtained by transforming the original PLA into an equivalent one whose graph is disconnected.

This goal can be achieved by node splittings i.e. by means of input ,product and/or output augmentations. The procedure is shown in Fig. 3.4.4 on the same simple example. The equivalent augmented PLA is also shown with its parallel partitioned implementation.

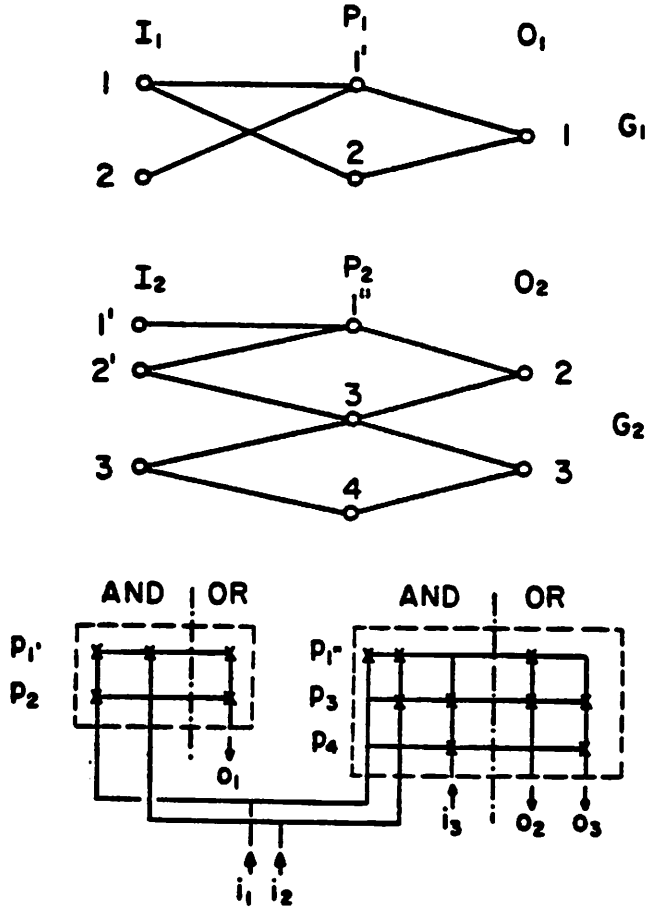


Fig. 3.4.4 Partitioning (general case)

In general let us denote by $\Pi_l(E^B)$ a partition of the edge set E^B into l subsets $E^{B_1}, E^{B_2}, \ldots, E^{B_l}$. Let $G^B(P_j, O_j, E_j^B)$ the subgraph induced by the partition where P_j and O_j are the node sets of product and output nodes which are adjacent to edges in E_j^B. Let E_j^A be the set of edges incident to nodes in P_j and I_j be the set of nodes adjacent to P_j.

In general, because of output and product node splittings $|O| \leq \sum_{j=1}^{l} |O_j|$ and $|P| \leq \sum_{j=1}^{l} |P_j|$. Moreover also $|I| \leq \sum_{j=1}^{l} |I_j|$ because of the input augmentations required by Rule 3. Subgraphs $G_j(I_j, P_j, O_j, E_j^A \cup E_j^B)$, $j = 1, 2, \cdots, l$ correspond to the j^{th} sub-units of the partitioned PLA.

An estimate of the area taken by the l logic subarrays and by the interconnect to route them is given by:

$$F = \sum_{j=1}^{l} |P_j|(a|I_j| + b|O_j|) + c\sum_{j=1}^{l} |I_j| + d\sum_{j=1}^{l} |O_j| + e\sum_{j=1}^{l} |P_j|$$

$$+ f[(\sum_{j=1}^{l} |I_j|) - |I|] + g[(\sum_{j=1}^{l} |O_j|) - |O|] + h[(\sum_{j=1}^{l} |P_j|) - |P|]$$

The partitioning optimization problem OP3 can be stated as follows:

Problem OP3

Find a partition $\Pi_l^*(E^B)$ such that:

$$F(\Pi_l^*(E_j^B)) \leq F(\Pi_l(E_j^B)) \quad \forall \Pi_l(E_j^B) \text{ and } \forall l$$

■

Note that the optimal solution can be not unique.

Remark 3.4.2: The unconstrained partitioning of the edge set E_j^B may lead to several product augmentations and consequently input augmentations as required by Rule 3. The augmentation may induce a kind of "chain reaction". It is therefore more convenient to consider a constrained partitioning of the set E_j^B which avoids product aug-

mentations, i.e. such that:

$$P_j \cap P_k = \phi \qquad \forall j,k=1,2,\ldots,l;\ j \neq k$$

An example of PLA partitioning with no product augmentations is shown in Fig. 3.4.5.

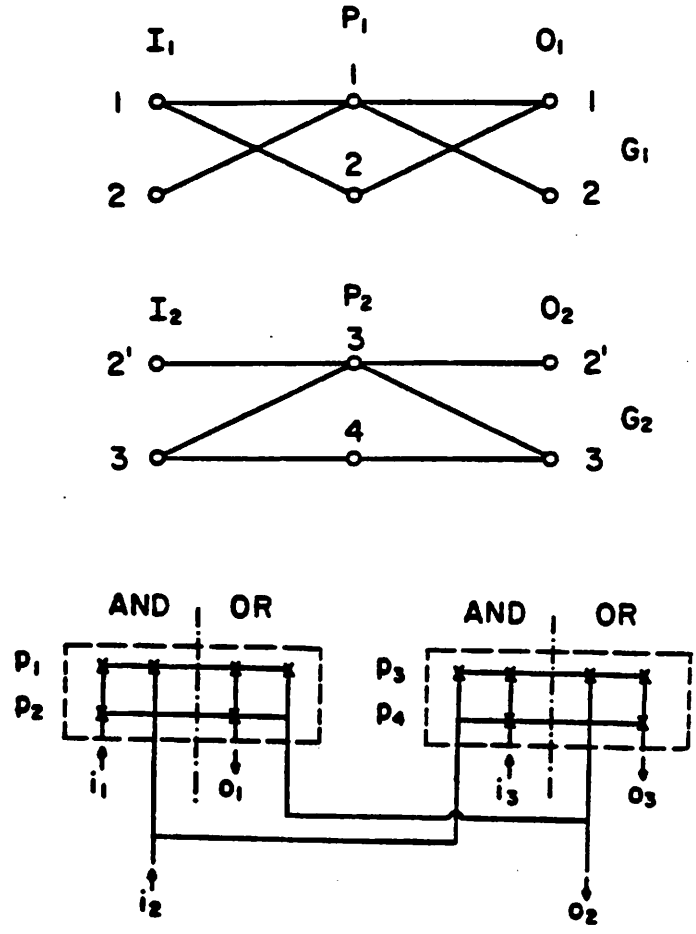


Fig. 3.4.5 Partitioning (no product augmentations)

3.5. PARTITIONED PLA IMPLEMENTATIONS

Different implementations of partitioned PLAs are possible. In particular, MOS technology implementations are considered here.

Two-fold partitioned input-arrays can be implemented as simply column block-folded arrays. This implementation is referred to as **bipartite folded** implementation by Egan [EGAN82], and shown in Fig. 3.5.1.

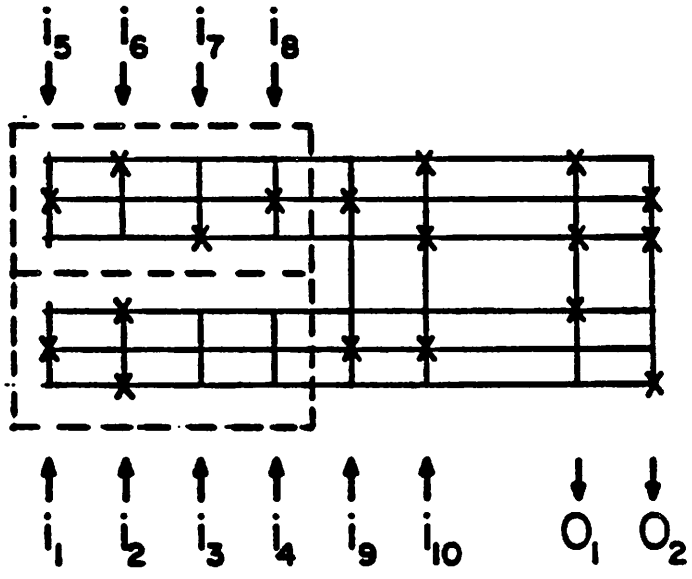


Fig. 3.5.1 Column block-folded implementation of a partitioned input-array

Note that augmented input columns are implemented as unsplit columns and therefore require connection to the input signal line from one side of the array only. The column positions in each block are arbitrary and can be assigned as to route optimally the PLA input signal lines from/to other cir-

cuit blocks.

Input arrays partitioned into more than two blocks can be implemented by a multiply column block-folded array, where inputs are routed by means of connection rows as described in Section 2.2 (Fig. 3.5.2).

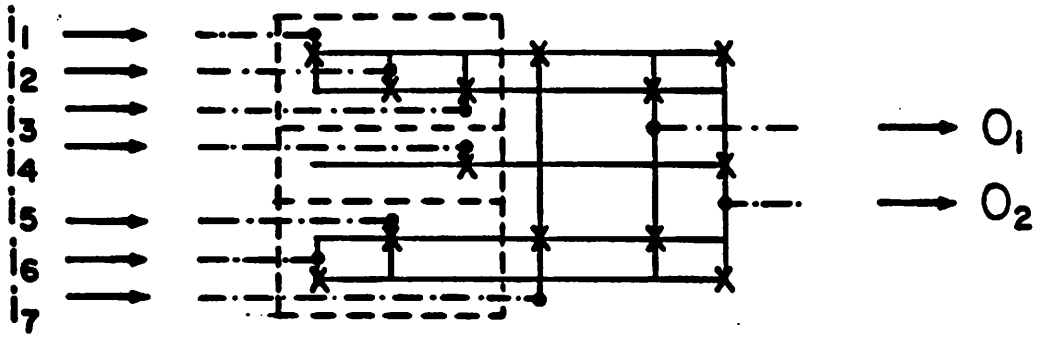


Fig. 3.5.2 Multiply column block-folded implementation of a partitioned input-array

Note that augmented input columns do not necessarily require extra connection-rows. In particular column segments in adjacent blocks corresponding to the same input can be connected internally.

Partitioned output-arrays are implemented similarly. Augmented outputs from different blocks must be OR-ed together with the proper phase (Fig. 3.5.3a). However, in NOR implementations, column segments

corresponding to the augmented columns can be connected internally by a wired-NOR (Fig. 3.5.3b).

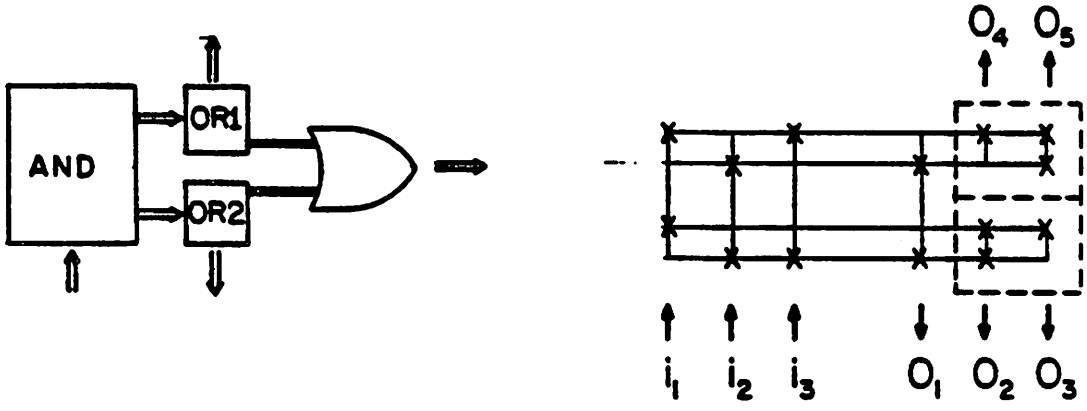


Fig. 3.5.3 Column block-folded implementation of partitioned output-array.

Partitioned arrays can be implemented as parallel connected arrays, as shown in Fig. 3.5.4. In this case the component-PLAs are placed as to simplify the routing between them.

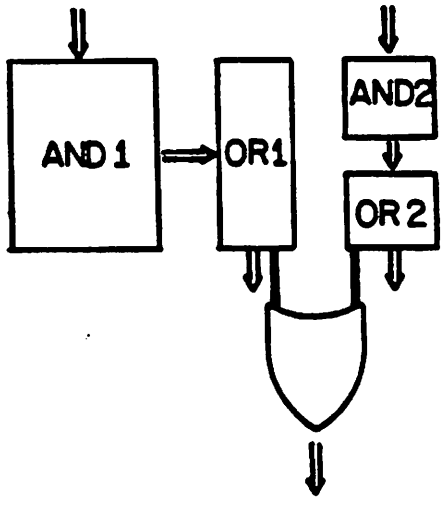


Fig. 3.5.4 Parallel connected implementation

In particular, partitioned arrays can be stacked or arranged in a line. The former implementation is similar to a column block-folded implementation (Fig. 3.5.5a), while the latter to a row block-folded one (Fig. 3.5.5b).

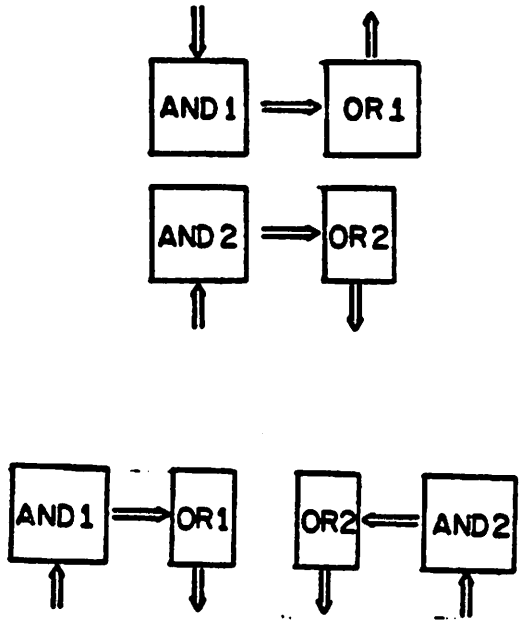


Fig. 3.5.5 Column (row) block folded implementation of a partitioned array

3.6. A HEURISTIC CLUSTERING ALGORITHM FOR PLA PARTITIONING

The optimization problems arising from PLA partitioning require to minimize a nonlinear function with integer constraints. The objective functions depend on the cardinality of the node subsets induced by an edge set partitioning.

A heuristic algorithm based on a cluster search [SPAT80] and on array transformations is proposed here. The same cluster search strategy is used for the three partitioning problems. For this reason the graph related to a partitioning problem is denoted by $G(V,E)$. The node set V is defined as $I \cup P$, $P \cup O$ and $I \cup P \cup O$ and the edge set E as E^A , E^B and $E^A \cup E^B$ for input

partitioning, output partitioning and partitioning respectively.

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. Let $\bar{E} \subset E$ be the set of edges joining nodes in V_1 to nodes in V_2. If $\bar{E}$ is empty, the node partition induces a graph partition into two disjoint subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$. If $\bar{E}$ is not empty, the algorithm modifies the graph by adding to V_1 and V_2 appropriate nodes incident to $\bar{E}$, so that 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. This operation corresponds to node splitting (augmentation) and is described in detail in the sequel according to the different partitioning problems. 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 selection of cluster nodes is driven by the values taken by the objective function .

Different authors have dealt with clustering related problems [LUCC69],[LAWL73a],[KERN70]. The partitioning algorithm 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*) and its entries are nodes of the graph. The second one is the *adjacency set* (*AS*) and its entries are sets of nodes of the graph. The third column is the *objective function* vector (*OF*) and its entries are the values of the area estimates F^A , F^B and F for the three partitioning problems.

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≠ φ ) {
        IS = φ ; AS = φ ; OF = φ ;
        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) );
        };
}.
```

Procedure adj(i) returns the nodes adjacent to node i.

Procedure nextadj($IS,AS(i)$) returns all the nodes adjacent to node $IS(i+1)$ not contained in $\cup_{j=1}^{i} IS(j)$. An efficient way to evaluate the procedure is described in [SANG77]: the nodes returned by procedure nextadj are obtained from $AS(i)$ by deleting $IS(i+1)$ and adding the set of all the nodes which are adjacent to $IS(i+1)$ that are not already in $AS(i)$ or in $\cup_{j=1}^{i} IS(j)$.

Procedure inselect(V) selects an initial node of the graph $G(V,E)$ and procedure nextselect($AS(i)$) selects the next iterating node in $AS(i)$. Both selections follow a heuristic criterion described in the sequel.

Procedure update($G(V,E)$) stores subgraph $G_1(V_1, E_1)$ and returns 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. At each step of the internal *while* loop of the algorithm, the set V is partitioned into three disjoint subsets, namely:

$$X = \cup_{j=1}^{i} IS(j) \qquad Y = AS(i) \qquad Z = V-X-Y. \tag{5.1}$$

The nodes in X are inside the cluster and are adjacent only to nodes in $X \cup Y$. Nodes in set Y are "border" nodes. By construction, the nodes in Z are not adjacent to any node in X. Let $W \subset X$ be the subset of nodes adjacent to Y at the current step of the algorithm. Let us define $X_I\ (X_P, X_O)$, $Y_I\ (Y_P, Y_O)$, $W_I\ (W_P, W_O)$, $Z_I\ (Z_P, Z_O)$ the subsets of input (product and output) nodes of X, Y, W and Z respectively (i.e. $X_I = X \cap I$).

In the case of input partitioning only input columns are augmented. Hence the set V_1 is obtained by adding to cluster nodes X the input nodes Y_I adjacent to cluster nodes. Set V_2 is obtained by adding to the cluster complement set $Y \cup Z$ the input cluster nodes W_I adjacent to them. Note that the product node set P is partitioned into two subsets X_P and $Z_P \cup Y_P$. The edge set E is partitioned accordingly: E_1 and E_2 are the subsets of E , whose elements are incident to nodes in X_P and $Z_P \cup Y_P$ respectively. Hence:

$$G_1(V_1, E_1) = G_1(X \cup Y_I, E_1) \qquad G_2(V_2, E_2) = G_2(Y \cup Z \cup W_I, E_2)$$

Example 3.6.1 : Consider the AND plane of PLA shown in Fig 3.3.1. Suppose that at one step of the internal *while* loop the cluster set

contains the following nodes: $X = \{I_1, P_1, P_2\}$. The adjacency set is: $Y = \{I_2\}$. The other two sets defined by the partitioning algorithm are: $W = \{P_1\}$ and $Z = \{I_3, P_3, P_4\}$ (Fig. 3.6.1). Then $V_1 = \{I_1, I_2, P_1, P_2\}$ and $V_2 = \{I_2, I_3, P_3, P_4\}$.

A similar definition applies , *mutatis mutandis* , to the output partitioning problem with product (output) augmentations only:

$$G_1(V_1, E_1) = G_1(X \cup Y_P, E_1) \qquad G_2(V_2, E_2) = G_2(Y \cup Z \cup W_P, E_2)$$

$$(\quad G_1(V_1, E_1) = G_1(X \cup Y_O, E_1) \qquad G_2(V_2, E_2) = G_2(Y \cup Z \cup W_O, E_2) \quad)$$

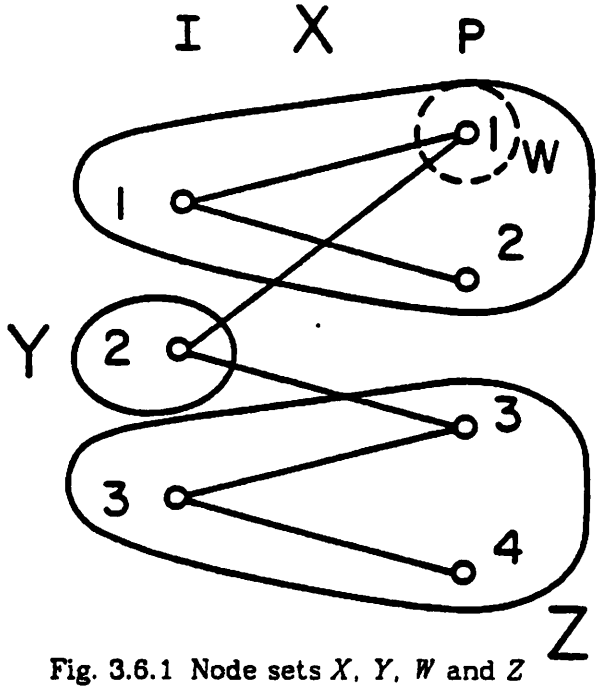


Fig. 3.6.1 Node sets X, Y, W and Z

In the case of parallel partitioning with input and output augmentations only, the set V_1 is obtained by adding to cluster nodes X the input nodes Y_I and the output nodes Y_O adjacent to cluster nodes. Set V_2 is obtained by adding to the cluster complement set nodes $Y \cup Z$ the input and the output cluster nodes $W_I \cup W_O$ adjacent to them. Note that the product node set P is partitioned into two subsets X_P and $Z_P \cup Y_P$ as in the input partitioning problem. The edge set E is partitioned accordingly: E_1 and E_2 are the subsets of E, whose elements are incident to nodes in X_P and $Z_P \cup Y_P$ respectively. Hence:

$$G_1(V_1, E_1) = G_1(X \cup Y_I \cup Y_O, E_1) \quad G_2(V_2, E_2) = G_2(Y \cup Z \cup W_I \cup W_O, E_2)$$

Remark 3.6.1: In the case of output partitioning with product and output duplications and parallel partitioning with input, output and product duplications, subgraphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ are defined differently. Since these definitions do not affect the analysis of the algorithm, they are not reported here for the sake of simplicity. ■

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

$$|AS(i)| = 0$$

$$\gamma(|X_I|, |X_P|, |X_O|, |Y_I|, |Y_P|, |Y_O|, |W_I|, |W_P|, |W_O|) > \gamma_{max}$$

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

The first condition guarantees that a cluster is found if graph $G(V,E)$ is not connected. The second condition allows the user to define a scalar function γ of

the cardinality of the subsets X_I, X_P, X_O, Y_I, Y_P, Y_O, W_I, W_P and W_O in order to specify the maximum size of each block. The third condition is a heuristic rule to determine a cluster. It can be also required that $OF(i)$ be smaller than a proper fraction of the initial area $OF(0)$ to ensure that partitioning is performed only if it gives a considerable saving in the total area. Since the objective function vector may have several local minima close to each other, 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 $AS(i)$. When any node in $AS(i)$ is added to the cluster node set X , graph $G(V,E)$ can be partitioned accordingly and the corresponding value of the objective function 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 [SANG77], a node connecting two clusters is a bad selection of 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 interesting to show that the time computational complexity of the algorithm is polynomially bounded , though the total number of nodes may increase at each iteration. Let $n = |V|$.

Theorem 3.1: The time computational complexity of the Partitioning Algorithm is bounded by $O(n^3)$.

Proof:

Every time the algorithm cycles through the external *while* loop, procedure update($G(V,E)$) returns $G_2(V_2, E_2)$. At least one node of the cluster set is not added to V_2,because otherwise $G(V,E) = G_2(V_2, E_2)$ and the cluster condition cannot be met. Hence $V_2 \subset V$ and $|V|$ is decreasing at every step of the external loop. The algorithm cycles at most n times through the external *while* loop. Moreover since $AS(i) \subset V$ and $|AS(i)| < n$, the algorithm will execute at most n inner inner *while* loops, because there is necessarily an integer m , $m < n$, such that $|AS(m)| = 0$ and a cluster condition is satisfied. Since procedures nextselect and nextadj can perform at most n comparisons and objective function evaluations, the time complexity of the algorithm is bounded by $O(n^3)$.

■

3.7. SMILE

SMILE is an interactive program for Programmable Logic Arrays partitioning. The PLA description is given as input to the program in the form of two-level sum-of-products logical implicants. The output file of logic minimizer PRESTO [BROW80] can be used as input to SMILE. Partitioning instructions are entered to 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 require to limit 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. The SMILE output file 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 array is technology independent.

Some SMILE output files are reported in App. C. As an example consider the PLA whose personality is shown in Fig. 3.7.1a. Since the OR plane is very sparse, output partitioning is attempted by the program as requested by the user. SMILE partitions the output column set into two disjoint subsets: $\{o_5, o_8, o_{10}, o_{11}, o_{14}, o_{17}, o_{18}, o_{20}\}$ and $\{o_6, o_7, o_9, o_{12}, o_{13}, o_{15}, o_{16}, o_{19}\}$ Three product terms, namely p_3, p_{11} and p_{12}, are augmented in order to transform the original array into an equivalent partitionable one. Fig. 3.7.1b shows the output partitioned array. The OR plane has been implemented as a block folded array. Note that the array size has changed: eight columns are not needed for the PLA implementation at the expenses of adding three extra rows. A global area saving of 29% has been achieved.

Program PLAID [HOFF81] was used to assemble the output-partitioned PLA as a column block-folded array. The block-folded implementation of the partitioned array in Fig. 3.7.2.

Program SMILE is coded in *ratfor* and consists of about 2000 lines of code. Intermediate code in *fortran*77 is available. SMILE runs in a VAX-UNIX® environment, but is easily transportable to other machines.

```
1        0000 1000101010101010
2        0001 0000101000001000
3        0010 01000000010001000
4        0011 0001010100010101
5        0100 0010100000101000
6        0101 0000010000010000
7        0110 0000100000101010
8        0111 0000001010100010
9        1000 0101010101000100
10       1001 0100010100000001
11       1010 0100000100100001
12       1011 0101010100100101
13       1100 0101000100010001
14       1101 0101000100000100
15       1110 0100010000010100
16       1111 1010000000101010


                  1111111
         1234 1234567890123456
```

BLOCK FOLDED OR PLANE

```
                    000002
                  68024680

* 3      0010     10000000
  4      0011     01110111
  6      0101     00100100
  9      1000     11111010
 10      1001     10110001
* 11     1010     10010001
* 12     1011     11110011
 13      1100     11010101
 14      1101     11010010
 15      1110     10100110
                  --------
  1      0000     10111111
  2      0001     00110010
  3      0010     00001010
  5      0100     01100110
  7      0110     00100111
  8      0111     00011101
 11      1010     00000100
 12      1011     00000100
 16      1111     11000111


                     11111
         1234     57913579
```

Partitioned PLA takes 71% of the original area

Fig. 3.7.1 Personality of the original and block-folded PLA

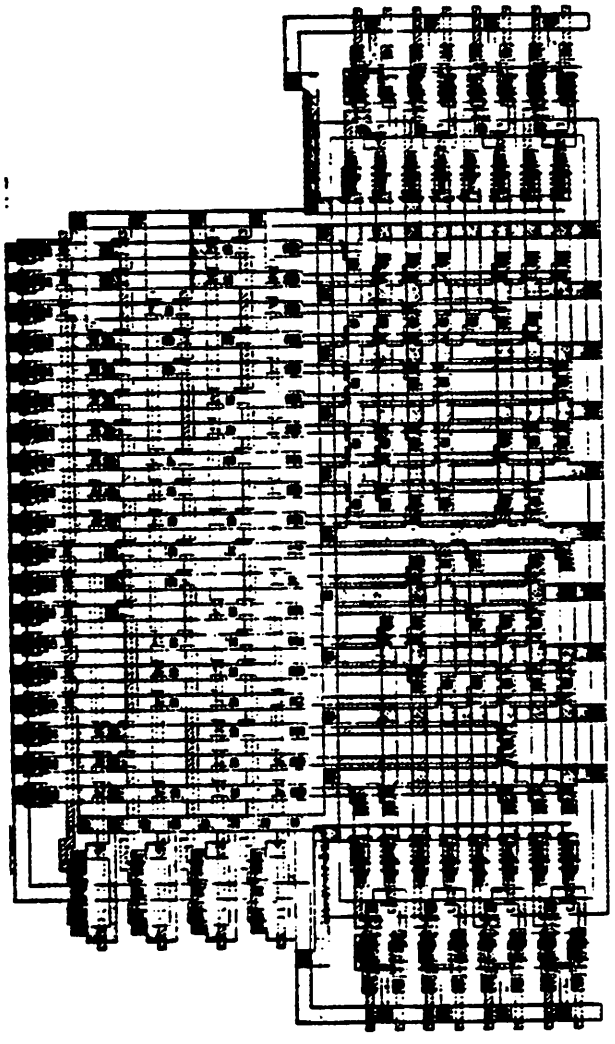


Fig. 3.7.2 Check plots of the block folded PLA .

3.8. EXPERIMENTAL RESULTS

Program SMILE has been tested on a large set of industrial PLAs. Some results are reported in Table 3.1. The time spent by the algorithm ranges from a few hundreds of milliseconds for PLA 1 to several seconds for larger arrays (PLA 7). Since execution time is small, circuit designers may want to use the program with different requirements in order to compare the different partitioned structures.

Note that it is not possible to achieve an area reduction of PLA 2 by means of input partitioning, because the AND plane has a full structure (no "don't cares") (Fig. 3.7.1).

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

TABLE 3.1

3.9. A COMPARISON BETWEEN FOLDING AND PARTITIONING TECHNIQUES

In order to compare the effectiveness of the two topological compaction techniques, two similar array implementations must be considered. Therefore simply column-folded PLAs are compared to simply block-folded arrays.

In principle, the column block-folded structure could be obtained by folding the array with the constraint of placing the cut in each folding pair at the same level. Therefore implementing partitioned PLAs as block-folded structures can be seen as a constrained folding problem. Therefore folded PLAs can be expected to be smaller than the corresponding block-folded implementation.

As a final remark, it is important to stress that heuristic algorithms are used for both folding and partitioning. Hence some partitioned PLA implementations may be found smaller than the corresponding folded ones and *vice versa*. However, experimental results show that the algorithm of program PLEASURE yields in general a better compaction than the one of program SMILE.

Normalized folded versus partitioned array areas. Initial area = 100.			
PLA	size P*(N+M)	Folding	Partitioning
PLA 1	6*(6+4)	60	61
PLA 2	16*(4+16)	60	65
PLA 3	30*(19+10)	68	67
PLA 4	75*(35+29)	53	46
PLA 5	62*(24+14)	58	60
PLA 6	84*(27+10)	54	59
PLA 7	84*(27+10)	51	57

TABLE 3.2

CHAPTER 4

DESIGN OF PLA-BASED FINITE STATE MACHINES

4.1. SEQUENTIAL LOGIC IMPLEMENTATION

Sequential circuits play a major role in the control part of digital systems. Digital computers are very complex examples of sequential systems and involve a combination of sequential functions.

A sequential function can be represented by several models [HOPC79]. The **Deterministic Finite State Machine** or **Deterministic Automaton** representation is used in the sequel and is referred to as Finite State Machine (FSM) for the sake of simplicity.

Finite State Machines consist of two major components: a combinational circuit and a memory. The memory stores a representation of the state of the machine at any given time and the combinational circuit generates the machine primary outputs as a function of the machine state and/or the machine primary inputs (Fig. 4.1.1).

A customized design of the combinational component can be achieved by interconnecting logic gates. For example, the control part of the Z8000 micro-processor was implemented by *random logic* gates. Such a design may require a small silicon area, but it is highly dependent on the particular logic function. Moreover design time is longer in comparison to other structured implementations and engineering changes may require a complete redesign.

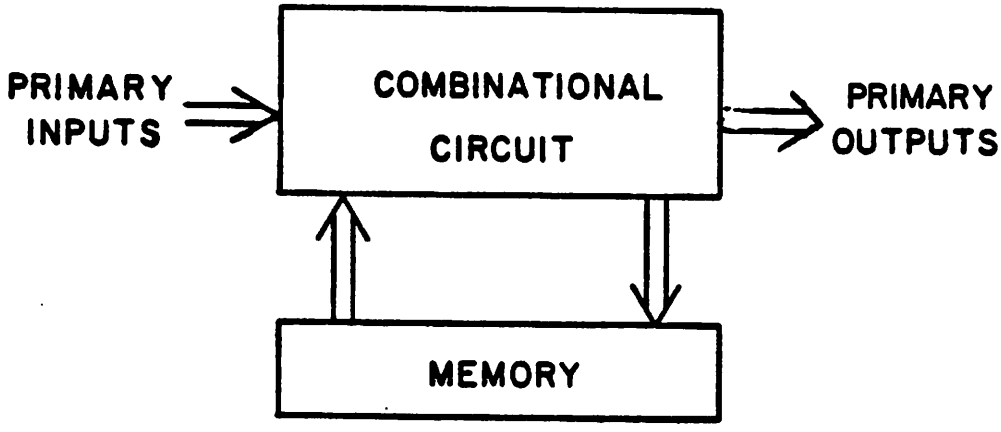


Fig. 4.1.1 Finite State Machine

On the other hand, digital controllers are often implemented by Finite State Machines whose combinational component is a ROM. The controller operation can be altered by replacing or re-programming the ROM, giving a high degree of flexibility. The control units of micro-programmed digital computers are designed according to this methodology.

The implementation of sequential functions in VLSI system design has to satisfy two major requirements:

i) regular and structured design that can be supported by computer-aided tools;

ii) size and performance of the silicon implementation.

A PLA implementation of the FSM combinational component can satisfy both requirements. Since FSM memory components, as well as PLAs, can be designed by means of regular structures, the entire FSM implementation can be regular and structured. This allows the automation of FSM-based sequential-circuit design. Moreover several techniques, like logic minimization and topological compaction, allow the of design area-effective PLA implementations. Therefore PLA-based FSM design can be optimized with regard to silicon area requirement and subsequently to switching-time performance.

The memory component of a FSM consists of a set of latches that store the machine state representation. Several types of latches (Delay, Toggle, JK) can be used [HILL81]. Some functions can be implemented more efficiently

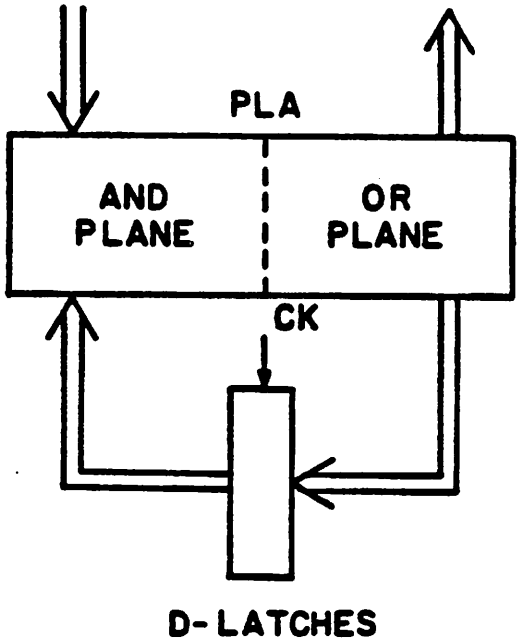


Fig. 4.1.2 Model of synchronous PLA-based FSM using D-latches.

with a particular type of latch: for example counters are usually implemented by means of Toggle-latches and generic sequential functions by Delay-latches. Memory is implemented by Delay (D) latches in the FSM model used in the sequel for the sake of simplicity.

The operation of VLSI systems is often synchronized to a system clock. The main goal is to keep a *race-free* design even when the circuit size is large. For this reason the model of a sequential function implementation used here is a synchronous Finite State Machine as shown in Fig. 4.1.2.

4.2. SEQUENTIAL FUNCTION REPRESENTATION

Different functional representation of a Finite State Machine are possible. Formally a FSM can be defined as a 5-tuple $(X, Y, Z, \delta, \lambda)$ where:

$X = \{x_1, x_2, \cdots, x_{n_x}\}$ is the set of primary inputs;

$Y = \{y_1, y_2, \cdots, y_{n_y}\}$ is the set of internal states;

$Z = \{x_1, x_2, \cdots, z_{n_z}\}$ is the set of outputs;

δ: $X \times Y \rightarrow Y$ is the next state function and

λ: $X \times Y \rightarrow Z$ (λ: $Y \rightarrow Z$) is the output function for a Mealy (Moore) machine.

A Finite State Machine representation is said to be **incompletely-specified** if the next-state and/or output function are not specified for some input and/or present-state. A FSM representation is incompletely-specified when some inputs (outputs) never occur (are never sampled) in some machine state.

Other representations are often used in designing FSMs. The most common are: **state tables**, **flow-charts** and **Hardware Description Language** (HDL) programs. Note that all these representations are equivalent to the abstract mathematical formulation.

A tabular representation of the FSM functionality is given by **state tables** [HILL81]. The state table is an array in which each column (row) is associated to an input (state). The entries of the array are the corresponding values of the next-state function δ and the output function λ. A "don't care" symbol (*) represents the next-states and/or outputs when these are not specified.

Example 4.2.1: The following state tables describes a FSM having $n_x = 2$ inputs, $n_y = 7$ states and $n_z = 2$ outputs ■

	input = 0		input = 1	
present state	next state	output	next state	output
START	state_6	00	state_4	00
state_2	state_5	00	state_3	00
state_3	state_5	00	state_7	00
state_4	state_6	00	state_6	10
state_5	START	10	state_2	10
state_6	START	01	state_2	01
state_7	state_5	00	state_6	10

Remark 4.2.1: A state table has an equivalent graph representation in the **transition graph**. The transition graph is a directed graph: the node set is in one-to-one correspondence with the state set Y and directed edges represent the state transition function δ. Each edge has a label corresponding to the primary input and output associated with that transition [HILL81],[MEAD80]. ■

State tables and state diagrams have been used extensively in the literature in the past years. Unfortunately state-table representations of large, incompletely-specified Finite State Machines can be cumbersome. For this reason designers use to describe the machine functionality by means of **Flow-charts** or **Hardware Description Languages.**

Flow-chart descriptions, such as the Algorithmic State Machine chart [CLAR75], allow an algorithmic description of the FSM and enable the designer to visualize the functionality of the entire machine. Hardware Description Language, such as ISP [BARB77], AHPL [HILL78] or FTL2 [DEUT83], allow to describe a sequential function as a software program. Both have advantages and disadvantages and a choice between them is left to the designer's preference.

Example 4.2.2: The FSM of Example 4.2.1 is described by the following flow-cart, according to the notations given in [CLAR75]. Every square box (*state box*) represents a state. Every directed path joining two square boxes represents a transition between the two states depending on the machine inputs tested by the *condition boxes* along the path. Outputs are represented either in the state blocks or by *conditional output boxes* (rounded edge boxes) along the path (Fig. 4.2.1). ■

Note that the ASM representation is equivalent to the formal description. Every specified value of the next state function δ (output function λ) is related to one and only one directed path joining a state box to a state (output) box. The values of the functions are related to the present-state

represented by the state box that is origin of the directed path and to the input associated to the conditional boxes along the path.

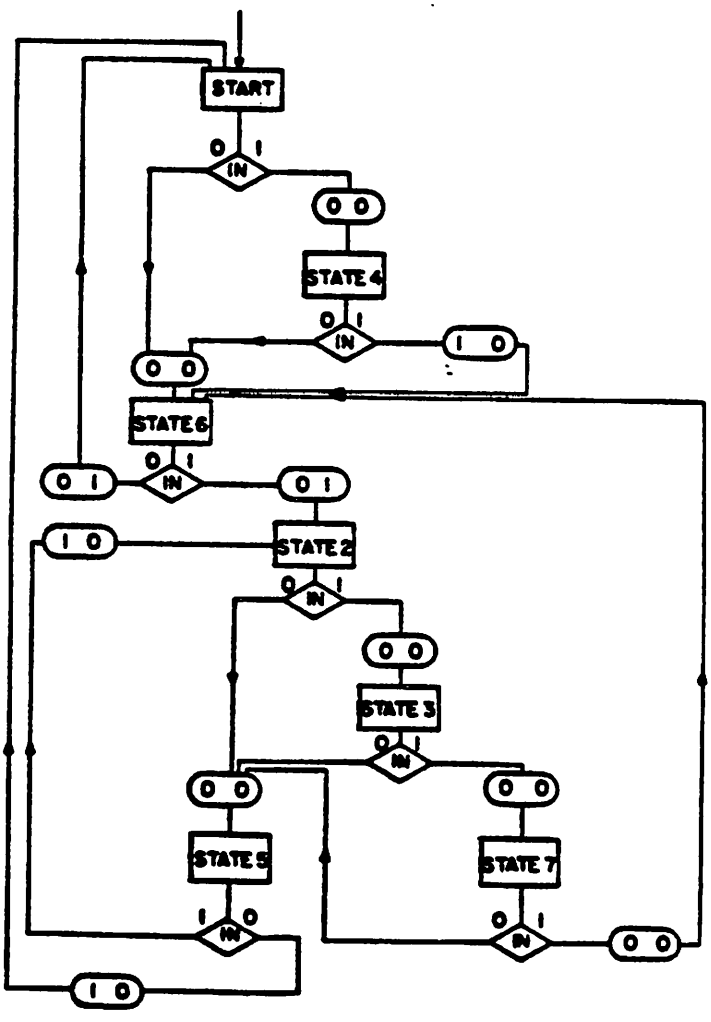


Fig. 4.2.1 ASM description

Example 4.2.3: The following FTL2 [DEUT83] program is an equivalent description of the FSM of Examples 4.2.1 and 4.2.2:

■

```
(which? state

 (START (if (= input 0)
        then
        (<- state_6)
        else
        (<- state_4))

     (<- output 00b))

 (state_2 (if (= input 0)
        then
        (<- state_5)
        else
        (<- state_3))

      (<- output 00b))

 (state_3 (if (= input 0)
        then
        (<- state_5)
        else
        (<- state_7))

      (<- output 00b))

 (state_4 (if (= input 0)
        then
        (<- output 00b)
        else
        (<- output 10b))

      (<- state_6))

 (state_5 (if (= input 0)
   then
   (<- START)
  else
   (<- state_2))

   (<- output 10b))

 (state_6 (if (= input 0)
   then
   (<- START)
  else
   (<- state_2))

   (<- output 01b))

 (state_7 (if (=input 0)
    then
    (<- state_5)
     (<-output 00b)
      else
     (<- state_6))

 (<- output 10b))))
```

Note that a formal description of a FSM can be mapped into a HDL program in a straight-forward way. In particular the values of the next-state function δ and output function λ are represented by conditional statements, whose clauses are the corresponding values of the inputs and present states.

4.3. OPTIMAL DESIGN OF FINITE STATE MACHINES: STATE ASSIGNMENT

State tables, flow-charts and HDL programs specify Finite State Machines at the functional level. Optimal FSM design involves several transformations, as described in Section 1.5. In particular: **state minimization, state** and **memory element assignment, logic minimization** and **topological compaction** of the combinational component. Optimal state assignment is the subject of this Chapter. Basic concepts and definitions related to the representation of a switching function at the logic level are reported in App.A.

The **state assignment**, or **state encoding** problem consists of choosing a Boolean representation of the internal states of the machine. State encoding affects substantially the complexity of the FSM combinational component [HART66]. In particular, the PLA size depends heavily on the state assignment. Therefore the optimum state assignment problem can be stated as follows:

Find the assignment corresponding to a PLA implementation of minimum area.

This task is formidable and some simplifying assumptions are needed. As a first step, topological compaction techniques to reduce the PLA area, such as folding and partitioning, are not considered. Under this assumption, the PLA area is proportional to the product of the number of rows (product-terms) times the number of columns. Both row and column cardinality depend on state encoding. The (minimum) number of rows is the cardinality of the (minimum) cover of the FSM combinational component according to a given assignment. The code-length (i.e. the number of bits used to represent the states) is related to the number of PLA columns and in particular to the

number of PLA input (output) columns corresponding to the present (next) states. Therefore the PLA area has a complex functional dependence on state assignment.

For this reason two simpler optimal state assignment problems are defined:

i) Find the assignment of minimum code length among the assignments that minimize the number of rows of the PLA.

ii) Find the assignment that minimizes the number of rows of the PLA among the assignments of given code length.

The optimum solution to the state assignment problem can be seen as a trade-off between the solutions to problem i) and ii). Note that the above problems are still computationally very hard and to date no method (other than exhaustive search) is known that solves them. Therefore heuristic strategies are used to approximate their solution.

A method that attempts a solution to problem i) is presented in the sequel, as an intermediate step toward the solution of the complete problem. Problem i) is referred to as the optimal state assignment problem throughout this Chapter. Note that most of the previous state assignment techniques attempted to solve problem ii) with minimum code length (i.e. $\log_2 n_y$). The relevance of problem ii) was related to minimizing the number of storage elements in discrete component implementations of Finite State Machines. Today, optimizing the total usage of silicon area (related only partially to the number of storage elements) is the major goal in integrated circuit implementations of PLA-based Finite State Machines.

Many papers dealing with the state assignment problem can be found in the literature (see Section 1.7). Most techniques can be reduced to algebraic

methods based on partition theory and on a *reduced dependence* criterion [HART66]. However no theoretical result was ever presented that related reduced dependencies to optimal implementation of the FSM and in particular to the minimality of a cover of the FSM combinational component.

Armstrong [ARM62a] developed a method that related state encoding to the number of gates required to implement the combinational component of the FSM (he simplified the problem by neglecting the output function). In particular his method can be specialized to PLA-based FSMs. The technique is based on an inspection of the state table and on the following considerations:

i) If an input, say i, maps two states, say s_A and s_B into the same next-state, i.e. $\delta(i, s_A) = \delta(i, s_B)$ and the code of s_A differs from the code of s_B in one coordinate only, then both transition $s_A \rightarrow \delta(i, s_A)$ and $s_B \rightarrow \delta(i, s_B)$ can be expressed by two implicants having distance one (that can be merged by a logic minimizer into one implicant).

ii) If two inputs, say i_A and i_B map a state s into $s_A = \delta(i_A, s)$ and $s_B = \delta(i_B, s)$ and the code of s_A differs from the code of s_B in one coordinate only, then both transition $s \rightarrow s_A$ and $s \rightarrow s_B$ can be expressed by two implicants having distance two. (Heuristic minimizers transform distance-two implicants into one implicant and one minterm. This operation is defined as *reshape* [HONG74] and reduces the number of literals. Moreover it is possible that the minterm is covered by some other implicant and can be dropped from the cover.)

If a state assignment satisfies distance relations according to the above rules, then logic minimization of the encoded cover of the FSM combinational component leads to a reduction of the number of product-terms and literals.

Therefore state encoding is constrained to satisfy appropriate *adjacency* relations. Since *adjacency* relations can be represented by a graph, then state assignment is equivalent to embedding a graph into a Boolean cube.

Armstrong's approach can in principle handle rather large machines, but it has three serious drawbacks. The first is related to the fact that the criteria suggested by Armstrong do not take into account the techniques of fast heuristic logic minimizers such as MINI [HONG74], or ESPRESSO-II [BRAY84] in use today (Armstrong's paper appeared before the work on heuristic minimizers started). The second is that the state assignment problem is transformed into a particular graph embedding problem, which represents only partially the state encoding problem [DEMI83f]. In particular, state assignment is related to a more complex embedding problem, that cannot be expressed only in terms of adjacency relations, as shown in Section 4.5. The third is that the graph embedding technique used by Armstrong to solve the problem he introduced was ineffective. In particular, a fixed (minimal) code-length was chosen *a priori*. In this case graph embedding is equivalent to a subgraph isomorphism problem, where a one-to-one relation (coding) is sought between the set of the states (vertices of the adjacency graph) and a subset of the vertices (codes) of the Boolean cube. Note that even questioning the existence of a subgraph isomorphism is a hard problem: in particular it was shown to belong to the class of NP-complete problems [GARE78]. Since such an isomorphism may not exists, Armstrong relaxed some adjacency requirements and proposed heuristic techniques to embed a subgraph of the adjacency graph into the Boolean cube.

An enhanced technique, overcoming these difficulties, was presented in [DEMI83f]. The approach was based, as Armstrong's, on the use of distance

relations among the codes of the internal states. Armstrong rules were replaced by a prediction of the effects of heuristic minimization [HONG74] of the combinational logic related to a symbolic description of the FSM. State encoding was then achieved by a particular graph embedding technique: states were assigned to vertices or faces of the Boolean hypercube. This technique is equivalent to embed the adjacency graph into a **squashed hypercube** , i.e. a hypercube having appropriate faces squeezed into vertices. Such an embedding can always be achieved in polynomial time [GRAH72].

This technique has proven to produce acceptable results even for large machines, but has two drawbacks. The former is related to the prediction of the effects of heuristic logic minimization, that is a difficult task, because minimizers use rather complex techniques. The operations considered in the prediction phase are a simple subset of the operations performed by a sophisticated logic minimizer. Therefore an encoding based on the prediction step might not be the one that minimizes the combinational component of the FSM. The latter drawback is related to code length. Since some states may be assigned to rather large faces, the state set is contained in a cube of large dimensions, i.e. the code length tends to approach cardinality of the state set.

The state encoding technique reported in the sequel is based on an innovative strategy: instead of trying to estimate the effects of heuristic minimizers, logic minimization is applied **before** state assignment. Logic minimization is performed on a symbolic (code independent) representation of the combinational component of the FSM. A constrained encoding problem relates the minimal symbolic representations to the class of assignments that implement the combinational component of the FSM with at most as

many product-terms as the cardinality of the minimal symbolic cover. In this class, an optimal encoding is one of minimal length.

4.4 LOGIC MINIMIZATION OF THE FSM COMBINATIONAL COMPONENT

Symbolic minimization of the FSM combinational component is performed on an intermediate representation of the FSM: the **symbolic cover** . The concept of symbolic cover is a generalization of the logic cover representation of combinational-logic functions.

A symbolic cover is a set of primitive elements called **symbolic implicants** . A symbolic implicant is a set of two-input and two-output character strings and is denoted by the 4-tuple (i, s, s', o). The two input strings, represent a binary-valued representation of a primary input (i) and a symbolic representation of a present state (s). The two output strings represent the corresponding symbolic representation of the next-state ($s' = \delta(i, s)$) and a binary-valued representation of the primary outputs ($o = \lambda(i, s)$). Therefore a symbolic cover is equivalent to the formal mathematical representation. Note that this representation can be generalized by letting i and o describe symbolic inputs and outputs.

Example 4.4.1: Consider the Finite State machine of Example 4.2.1: The following is a symbolic implicant:

0 START state_6 00

showing that a "0" primary-input value maps state "START" into "state_6" and asserts output 00. The symbolic cover is the collection of the symbolic implicants representing the state transitions:

0	START	state_6	00
0	state_2	state_5	00
0	state_3	state_5	00
0	state_4	state_6	00
0	state_5	START	00
0	state_6	START	00
0	state_7	state_5	00
1	START	state_4	00
1	state_2	state_3	00
1	state_3	state_7	00
1	state_4	state_6	00
1	state_5	state_2	00
1	state_6	state_2	00
1	state_7	state_6	00

Remark 4.4.1: An ASM chart can be easily transformed into a symbolic cover. In particular every path joining two state-boxes corresponds to a symbolic implicant. The symbolic implicant has the s and s' entries as the labels of the boxes. The input string i is determined by the condition-boxes along the path. The j-th entry in i is a "1" or "0" if a condition-box with a "true" or "false" exit respectively is placed along the path and corresponds to the j-th input qualifier. Else the entry is a "*". The j-th entry in o is a "1" or "0" if the corresponding output assertion is defined in any conditional-box along the path, or in the terminating state-box. Else the entry is a "*".

Remark 4.4.2: Hardware Description Language programs can be transformed as well into a symbolic cover representation. For example, a symbolic implicant is defined for each of the assertions of the conditional constructs of FTL2. The clauses of the conditional statements correspond to the i and s parts while the statements them-

selves correspond to the s' and o parts.

Note that a symbolic cover is a logic cover of a multiple-valued logic function [SU72] [HONG74], where each state takes a different logic level and is represented by a character string. A symbolic implicant having n (m) primary input (output) bits can be seen as a (n+1)-input, (m+1)-output multiple-valued logic implicant.

Several notations are used to represent multiple-valued logic covers. For example, the different logic levels can be represented by integer values : 0, 1, 2, $\cdots$, p-1. This is an extension of the binary notation to a p-valued representation.

The **positional cube** notation is used in the sequel [SU72]. A p-valued logical variable is represented by a string of p binary symbols. Value r is represented by a "1" in the $r-th$ position, all others being "0". Note that the positional cube notation allows to represent a set of values with one string. The disjunction (multiple-valued logical OR) of several values is represented by a string having "1"s in the corresponding positions. Therefore the "don't care" value is represented by a string of "1"s and the empty value by a string of "0"s.

The transformation of a symbolic cover into a multiple-valued cover with positional cube notation is straight-forward, since the latter is itself a symbolic cover and the transformation involves only symbol translations. Therefore definitions and properties of multiple-valued logic covers carry over to symbolic covers as well.

Example 4.4.2: The symbolic cover of Example 4.2.4 can be translated

into a multiple-valued positional-cube representation by associating a value to each state. START is represented by 1000000, state_2 by 0100000, etc.

0	1000000	0000010	00
0	0100000	0000100	00
0	0010000	0000100	00
0	0001000	0000010	00
0	0000100	1000000	10
0	0000010	1000000	01
0	0000001	0000100	00
1	0000010	0100000	01
1	0000100	0100000	10
1	0001000	0000010	10
1	0000001	0000010	10
1	1000000	0001000	00
1	0100000	0010000	00
1	0010000	0000001	00

■

Finding a minimum multiple-valued cover is a computationally expensive problem. Heuristic multiple-valued logic minimizers, such as MINI [HONG74] can be used to compute a minimal (local minimum) cover. (Program MINI [HONG74] is used in general for binary-valued logic minimization; however it supports multiple-valued minimization as well.) Alternatively the positional-cube representation can be seen as a binary-valued encoding of a multiple-valued function. This encoding is referred to as **1-hot coding**, because each value of the multiple-valued function corresponds to one and only one binary value "1" (HIGH) in the coded representation. (See App. A for details.) By using this representation, binary-valued minimizers, such as ESPRESSO-II [BRAY84], can be used to obtain minimal symbolic covers. Experimental results have shown that ESPRESSO-II yields minimal (symbolic) covers that are quite close to the minimum (symbolic) cover, for problems for which the minimum cover can be determined.

Example 4.4.3: Consider the symbolic cover of Example 4.4.2. The related minimal symbolic (multiple-valued) cover is the following:

0	0110001	0000100	00
0	1001000	0000010	00
1	0001001	0000010	10
0	0000010	1000000	01
1	0000100	0100000	10
0	0000100	1000000	10
1	1000000	0001000	00
1	0000010	0100000	01
1	0100000	0010000	00
1	0010000	0000001	00

Consider now the first symbolic implicant from the top:

0	0110001	0000100	00

This implicant shows that input "0" maps states "2", "3" and "7" into next-state "5" and assert output "00". A similar condition is expressed by the second and third implicants. ■

The example above shows that the effect of symbolic (multiple-valued) logic minimization is to group together the states that are mapped by some input into the same next-state and assert the same output. In other words, while the original symbolic cover is a set of implicants:

$$i \quad s \quad \delta(i, s) \quad \lambda(i, s)$$

where s represents a single state, the minimal multiple-valued cover may contain symbolic implicants in which s represents a set of states. Each state subset having more than one element and represented by a s string is termed **state group** . A **group set** is a collection of state groups. Given a state encoding and a state group, a **group face** (or simply **face**) is the smallest dimension subspace containing the codes of the states assigned to that group (or equivalently the disjunction of the codes assigned to the states in

that group).

The goal of the state assignment technique presented here is to group together the state codes in binary-valued logical implicants in the same way states are grouped in the minimal symbolic (multiple-valued) cover. In particular, a state encoding is sought, such that each symbolic implicant can be coded by one binary-valued implicant. For this assignment, there exist a binary-valued cover of the FSM combinational component having as many implicants as the minimal symbolic cover.

An encoding such that each group face contains all and only the codes of the states included in the corresponding group satisfies the above requirement. In fact, each coded implicant represents all and only the state-transitions related to the corresponding symbolic implicant. For this reason, a **constrained encoding problem** is considered:

> *Given a group set, find an encoding such that each group face does not intersect the code assigned to any state not contained in the corresponding group.*

In view of the previous considerations, any solution to the constrained coding problem is a state assignment such that the coded Boolean cover has the same cardinality of the minimal symbolic cover.

Example 4.4.4: Consider the minimized symbolic cover of Example 4.4.3. If the states are coded as follows:

START	100
state_2	110
state_3	011
state_4	000
state_5	001
state_6	101
state_7	010

then the following Boolean cover specifies the FSM combinational component:

0*0*	00100
0*00	10100
10*0	10110
0101	10001
1001	11010
0001	10010
1100	00000
1101	11001
1110	01100
1011	01000

The Boolean cover cardinality is the same as the minimal symbolic cover cardinality.

■

Remark 4.4.3: It is important to note that most state assignment technique are based on some grouping of the states. Hartmanis and Stearns [STEA62] presented an approach based on ***partition pairs*** of the state set. A partition pair (π, π') is an ordered pair of partitions such that any pair of states in the same block of π are mapped by any input to states in the same block of π'. When each block of π' is a single state, i.e. $\pi' = 0$, the blocks of π are state groups as defined above. Similarly Armstrong's approach [ARMS62a] lead to state grouping. However both Hartmanis' and Armstrong's approach were inefficient for large incompletely specified machines, because state grouping did not take into account the primary-input don't care set.

For this reason, in a previous approach [DEMI83f], symbolic implicants were considered pair-wise and the following rule was used to determine a group: Let $A = \{i_A, s_A, s'_A, o_A\}$ and $B = \{i_B, s_B, s'_B, o_B\}$ be two symbolic implicants such that: $i_A \supseteq i_B$ and $o_A \supseteq o_B$. Let $S(A)$ be the set of states which are mapped by any input representation $i \subseteq i_A$ either into a next-state different from s'_A or into an output representation not covered by o_A or both. Then s_A and s_B share the same group, while any state $s_Q \in S(A)$ cannot be in that group. The rationale is the following: if the code of s'_A s chosen to cover the code of s'_B, the two coded implicants can be merged. This approach produced efficient state grouping, because it simulated better the effects of logic minimization.

Symbolic minimization outperforms the other techniques in capturing the group structure of a Finite State Machine. Suppose a minimum symbolic cover is known and the related group set is determined. Then a solution to the corresponding coding problem allows to express each next-state function by a minimum number of product-terms. Even if a minimum symbolic cover is seldom found, experimental results have shown that the grouping related to the minimal symbolic covers obtained by programs ESPRESSO-II and MINI is very effective.

However there is still room to improve this technique. In the symbolic cover, the components of the next-state function δ_i, $i = 1, 2, \cdots, n_y$ have disjoint on-sets. However, in the coded Boolean cover, some states codes have a non-zero entry in the same position and therefore the components of the next-state functions are not necessarily disjoint. Therefore the minimum

(minimal) Boolean cover of the FSM combinational component may require fewer implicants than the sum of the each minimum (minimal) cover related to each next-state. In other words, state encoding transforms a minimal symbolic cover into a non-necessarily-minimal Boolean cover.

Example 4.4.5: Consider the coded cover of Example 4.4.2. Since the code of state_2 is bitwise OR of the code of START and state_7, then the transition state_5 → state_2 can be represented by the cover:

```
*001 10010
10*1 01000
```

that represents at the same time the transitions state_5 → START and state_3 → state_7.

The following is a minimal Boolean cover:

```
0*0*  10100
10*0  10110
*101  10001
*001  10010
1*01  01000
1110  01100
10*1  01000
```

■

In the present formulation, symbolic minimization groups present-states only. Therefore the assignment optimizes the choice of present states, and the implications of the corresponding next-state encoding is neglected. State assignment techniques that take into account next-states encoding are still under investigation.

Remark 4.4.4: 1-hot encoding of the Finite State Machine combinational component is widely used in industrial design. The reason is that in general no technique capable of encoding FSM with large state sets is available.

The penalty of using 1-hot coding is related to the number of PLA

columns required, that grows linearly with the state set cardinality. Note that the minimal number of PLA columns required grows with the logarithm of the state set cardinality.

Moreover 1-hot coding does not even minimize the number of PLA rows required, because the components of the next-state function representation have disjoint on-sets.

■

Remark 4.4.5: In general symbolic minimization is affected by the machine primary inputs and/or outputs. If a minimal PLA implementation is sought, primary inputs (outputs) can be considered as machine input (output) states and coded as well as the internal states.

However interfacing a FSM to other circuit building blocks often limits the possibility of finding an optimal coding for primary inputs and/or outputs.

■

4.5 CONSTRAINED STATE ENCODING

Symbolic minimization is used to define a constrained encoding problem, whose solutions are the state assignments that allow to implement the FSM combinational component with at most as many product terms as the minimal symbolic cover cardinality. According to the definitions given in Section 4.4, the constrained encoding problem consists of finding a state assignment such that each group face does not intersect the code assigned to any state not contained in the corresponding group. An optimal state assignment (as defined in Section 4.3) is a minimal code-length solution to

the constrained encoding problem.

The geometric interpretation of the optimal encoding problem is *finding the minimal dimension Boolean space in which group faces are subspaces and state encodings lay only in those subspaces corresponding to groups containing the states themselves.* State assignments are restricted here to one-to-one mappings between the state set and a subset of the vertices of the Boolean hypercube. This restriction is motivated by the objective of finding a solution to the constrained encoding problem in the Boolean space of minimal dimension.

Some definitions are introduced now to allow a formal statement of the problem. The coding problem is studied using matrix notation. Let n_s be the cardinality of the state set cardinality, n_i the cardinality of the group set and n_b the code length.

To be consistent with the positional-cube notation, state groups are represented by a Boolean matrix and in particular by the subset of the columns of the minimal multiple-valued cover corresponding to the present-states.

The **constraint matrix** A is a matrix: $A \in \{0, 1\}^{n_i \times n_s}$

$$A = \begin{bmatrix} a_{1\cdot} \\ a_{2\cdot} \\ \cdots \\ a_{n_i\cdot} \end{bmatrix} = \left[a_{\cdot 1} \mid a_{\cdot 2} \mid \cdots \mid a_{\cdot n_s} \right]$$

representing n_i state groups. State j belongs to group i if $a_{ij} = 1$.

A row of the constraint matrix is said to be a **meet** if it represents the intersection of two or more state groups. A row of the constraint matrix is said to be **prime** if it is not a meet.

Example 4.5.1: The following constraint matrix is derived from the

minimal symbolic cover of Example 4.4.3 and represent the state groups:

{ state_2 , state_3 , state_7 } , { START , state_4 } and { state_4 , state_7 }.

$$A = \begin{bmatrix} 0110001 \\ 1001000 \\ 0001001 \end{bmatrix}$$

All the rows of A are prime. If row a = 0001000 is appended to A, then a is a meet because it represents { state_4 }, which is the intersection between the second and the third group.

■

The **state code matrix** S is a matrix $S \in \{0, 1\}^{n_a \times n_b}$

$$S = \begin{bmatrix} s_{1*} \\ s_{2*} \\ \cdots \\ s_{n_b*} \end{bmatrix}$$

whose rows are state codes. The state code matrix is the unknown of the problem.

Definition 4.5.1: Let $a \in \{0, 1\}$ and $b \in \{0, 1, *\}$. The **selection** of b according to a is:

$$a*b = \begin{matrix} b & \text{if } a = 1 \\ \phi & \text{if } a = 0 \end{matrix}$$

Selection can be extended to two dimensional vectors and is similar to matrix multiplication.

Definition 4.5.2: Let $A \in \{0, 1\}^{p \times q}$ and $B \in \{0, 1, *\}^{q \times r}$. Then

$$A*B = C = \{c_{ij}\}^{p \times r}$$

where:

$$c_{ij} = \bigvee_{k=1}^{q} a_{ik} \cdot b_{kj}$$

Selection is useful to determine group faces corresponding to a group set and a given coding.

The face matrix $F \in \{0, 1, *, \phi\}^{n_1 \times n_0}$

$$F = \begin{bmatrix} f_{1*} \\ f_{2*} \\ \cdots \\ f_{n_1*} \end{bmatrix}$$

is the matrix whose rows are the group faces. Note that the empty group corresponds to the empty face ϕ. The face matrix is the selection of S according to A:

$$F = A \cdot S$$

Example 4.5.2: Consider the constraint matrix of Example 4.5.1 and the state assignment represented by:

$$S = \begin{bmatrix} 100 \\ 110 \\ 011 \\ 000 \\ 001 \\ 101 \\ 010 \end{bmatrix}$$

Then the face matrix is:

$$F = A \cdot S = \begin{bmatrix} *1* \\ *00 \\ 0*0 \end{bmatrix}$$

■

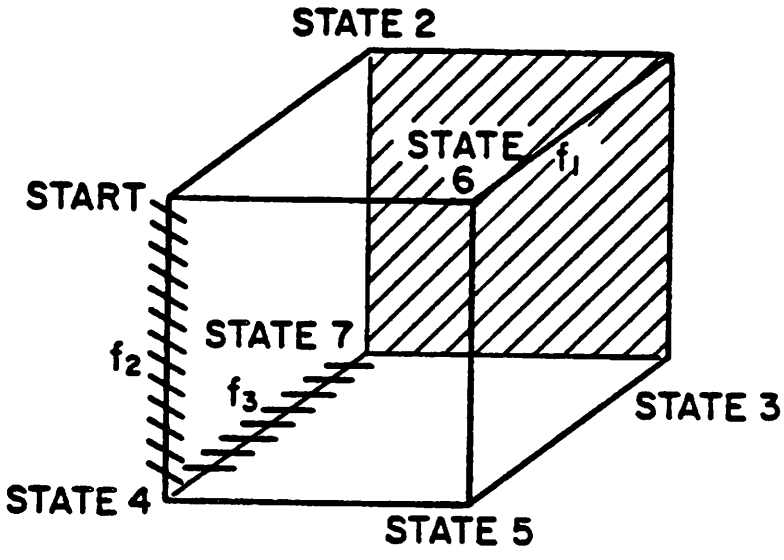


Fig. 4.5.1 Representation of the faces on the Boolean cube.

A solution to the constrained coding problem satisfies the **constraint relation:**

$$\bar{a}_{s_i} \cdot s_i \cdot \Lambda \; F \equiv \begin{bmatrix} \bar{a}_{1i} \cdot s_i \cdot \Lambda \; f_1 \cdot \\ \bar{a}_{2i} \cdot s_i \cdot \Lambda \; f_2 \cdot \\ \cdots \\ \bar{a}_{n_i i} \cdot s_i \cdot \Lambda \; f_{n_i} \cdot \end{bmatrix} = \begin{bmatrix} \phi \\ \phi \\ \cdots \\ \phi \end{bmatrix} = \Phi \qquad \forall i = 1, 2, \cdots, ns.$$

where $\bar{a}_{ij}$ is the complement of a_{ij}. In particular $\bar{a}_{s_i} \cdot s_i$ is a matrix whose rows are the encoding of state i, if state i does not belong to the corresponding group, else containing the empty value. Therefore the constraint relation is satisfied if and only if S is a solution of the constrained state encoding problem.

Example 4.5.3: The state matrix of Example 4.5.2 satisfies the constraint relation. However if the 6-th row is changed to 111, the constraint relation is no more satisfied, because the code of state_β intersects the first face, or equivalently:

$$\bar{a}_{\beta}s_{\beta} = \begin{bmatrix}1\\1\\1\end{bmatrix}\cdot[111] = \begin{bmatrix}111\\111\\111\end{bmatrix} \quad \Lambda \quad F = \begin{bmatrix}111\\\phi\\\phi\end{bmatrix} \neq \Phi$$

■

Remark 4.5.1: An implementable state assignment is such that codes are disjoint from one another. This requirement can be embedded in the constraint relation when n_s groups, consisting of one different state each, are added to the problem. In this case, n_s faces correspond to the n_s state codes, and any coding satisfying the constraint relation is such that codes are disjoint from one another.

■

The **optimal constrained encoding problem** can be stated as follows:

Find a state code matrix S with minimal number of columns that satisfies the constraint relation.

It is important to point out that there always exists a solution to this problem.

Theorem 4.5.1: Given any constraint matrix A, $\exists$ a state code matrix $S \in \{0, 1\}^{n_s \times n_s}$ satisfying the constraint relation. In particular the identity matrix satisfies the constraint relation.

Proof:

Let $F = A \bullet S$. Without loss of generality let us assume that no row of F represents an empty group. Then $f_{ij} = 0$ when $a_{ij} = 0 \ \forall$ $i = 1, 2, \cdots, n_t$ and $\forall j = 1, 2, \cdots, n_s$. Since $\bar{a}_{ij} \bullet s_{jj} = 1 \quad \forall a_{ij} = 0$ $\forall i = 1, 2, \cdots, n_t$ and $\forall j = 1, 2, \cdots, n_s$, the constraint relation is satisfied. ■

Theorem 4.5.1 shows that 1-hot coding satisfies any constraint relation. However it is relevant to determine shorter codes that satisfy the constraint relation as well. The constraint relation is invariant under a set of transformations on matrices A and S. These transformations allow to construct S as will be shown in Section 4.6. For these reasons, addition (deletion) of rows and/or columns to (from) matrices A and S are investigated. This corresponds to modify parameters n_t , n_b and n_s of the problem. The addition (deletion) of a row in matrix A is considered first.

Proposition 4.5.1: If S satisfies the constraint relation for a given A, then S satisfies the constraint relation for A' when A' is obtained from A by dropping a row.

Let A' be a constraint matrix obtained from A by adding a row. Adding a row corresponds to add a set of constraints to the state codes and in general a state code matrix S does not satisfy the constraint relation for an augmented constraint matrix A.

Example 4.5.4:

Let: $A = [1100]$ and $S = \begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix}$. If $A' = \begin{bmatrix} 1100 \\ 1011 \end{bmatrix}$, then S does not satisfy

the constraint relation. In fact:

$$A' \cdot S = \begin{bmatrix} 0^* \\ ** \end{bmatrix}$$

and

$$\bar{a}_{2'} \cdot s_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot [01] = \begin{bmatrix} \phi\phi \\ 01 \end{bmatrix} \quad \Lambda \quad A' \cdot S = \begin{bmatrix} \phi \\ 01 \end{bmatrix} \neq \Phi$$

■

However note that given any constraint matrix A and a state code matrix S satisfying the constraint relation, S satisfies the constraint relation for $\bar{A} = \begin{bmatrix} A \\ A_m \end{bmatrix}$, where the rows of A_m represent all the state group intersections, i.e the rows of A_m are all the meets of $\bar{A}$. This is proven by:

Lemma 4.5.1: If S satisfies the constraint relation for a given A, then S' satisfies the constraint relation for $A' = \begin{bmatrix} A \\ a_{m\cdot} \end{bmatrix}$, where $a_{m\cdot}$ is a meet of A'.

Proof:

Consider matrix $F' = A' \cdot S = \begin{bmatrix} A \\ a_{m\cdot} \end{bmatrix} \cdot S = \begin{bmatrix} A \cdot S \\ a_{m\cdot} \cdot S \end{bmatrix}$.

Suppose by contradiction that $\exists$ a state, say k , such that:

$$\begin{bmatrix} \bar{a}_{ik} \\ \bar{a}_{mk} \end{bmatrix} \cdot s_k \quad \Lambda \quad A' \cdot S \neq \Phi$$

Since $\bar{a}_{ik} \cdot s_k \quad \Lambda \quad A \cdot S = \Phi$, then $\bar{a}_{mk} \cdot s_k \quad \Lambda \quad f'_m \neq \phi$. Let now $\{a_{j\cdot}\}$ be the set of rows of A corresponding to the groups whose intersection is represented by $a_{m\cdot}$. Then $\exists a_{i\cdot} \in \{a_{j\cdot}\}$ such that $\bar{a}_{ik} \cdot s_k \; \Lambda \; f'_i \neq \phi$ and

then

$\bar{a}_{i} \leq s_{k}$. $\wedge$ $A \cdot S \neq \Phi$. Hence we have a contradiction. ■

The problem of modifying the code length is considered now.

Proposition 4.5.2: If S satisfies the constraint relation for a given A, then S' satisfies the constraint relation when S' is obtained from S by dropping: i) any column equal to some other column; ii) any column with all "0" ("1") entries.

Lemma 4.5.2: If S satisfies the constraint relation for a given A, then S' satisfies the constraint relation when S' is obtained from S by column permutation or complementation.

Proof:

i) column permutation:

Let $P \in \{0, 1\}^{n_s \times n_s}$ be a permutation matrix. Then

$$\bar{a}_{i} \leq s_{i}. \quad \wedge \quad AS = \Phi$$

$$\Rightarrow \bar{a}_{i} \leq s_{i} P \quad \wedge \quad A \cdot SP = \Phi \quad \forall i = 1, 2, \cdots, n_s$$

$$\Rightarrow \bar{a}_{i} \leq s'_{i}. \quad \wedge \quad AS' = \Phi \quad \forall i = 1, 2, \cdots, n_s$$

ii) column complementation:

Suppose by contradiction that by complementing a column, say j, the constraint relation is not satisfied. Then $\exists i$ such that $\bar{a}_{i} \leq s'_{i}$. $\wedge$ $AS' \neq \Phi$ and in particular $\exists l$ such that $\bar{a}_{il} \leq s'_{i}$. $\wedge$ $f'_{i} \neq \phi$. Therefore: $\bar{a}_{il} = 1$ and

$$s'_{ij} \quad \Lambda \quad f'_{ij} \neq \phi$$

$$s'_{ik} \quad \Lambda \quad f'_{ik} \neq \phi \ \forall k = 1, 2, \cdots, n_b \ ; \ k \neq j$$

Since:

$$f'_{ij} = \begin{cases} * \text{ if } f_{ij} = * \\ 1 \text{ if } f_{ij} = 0 \\ 0 \text{ if } f_{ij} = 1 \end{cases}$$

and $f'_{ik} = f_{ik} \quad \forall k = 1, 2, \cdots, n_s$; $j \neq k$ then $s_{i\cdot} \Lambda f_{i\cdot} \neq \phi$ and we have a contradiction. ■

Corollary 4.5.1: If S satisfies the constraint relation for a given A, then S' satisfies the constraint relation when S' is obtained from S by dropping any column that is the complement of some other column.

Lemma 4.5.3: If S satisfies the constraint relation for a given A, then $\forall T \in \{0, 1\}^{n_s \times n_a}$ and $\forall n_a$, $S' = [S \mid T]$ satisfies the constraint relation.

Proof:

Let $s'_{i\cdot} = [s_{i\cdot} \mid t_{i\cdot}] \quad \forall i = 1, 2, \cdots, n_s$ and suppose by contradiction that $\exists k$ such that:

$$\bar{a}'_{k\cdot} \circ s'_{k\cdot} \quad \Lambda \ A \circ S' \neq \Phi$$

Then:

$$[\bar{a}_{k\cdot} \circ s_{k\cdot} \mid \bar{a}_{k\cdot} \circ t_{k\cdot}] \quad \Lambda \quad [A \circ S \mid A \circ T] \neq \Phi$$

$$\Rightarrow \ \bar{a}_{k\cdot} \circ s_{k\cdot} \ \Lambda \ A \circ S \neq \Phi$$

and we have a contradiction. ■

The problem of adding (deleting) an element to (from) the state set is considered now. Adding (deleting) a state corresponds to adding (deleting) a row to the state code matrix S and a column to the constraint matrix A.

Proposition 4.5.3: Let S be a state code matrix satisfying the constraint relation for a given A. Let A' be the constraint matrix obtained from A by dropping a column and S' be a reduced state code matrix obtained from S by dropping the corresponding row. Then S' satisfies the constraint relation for A'.

Adding an element to the state set is relevant in constructing a state code matrix: given a set of state codes represented by S that satisfies the constraint relation for a given A, let us add a new state to the state set and a new column vector α to A, i.e the new constraint matrix is $A' = [A \mid \alpha]$. The non-zero entries in column α are related to the groups to which the new state belongs. The problem consists of determining an augmented state code matrix S', that satisfies the augmented constraint relation. The most desirable situation is to obtain $S' = \begin{bmatrix} S \\ \sigma \end{bmatrix}$, where σ is the new state code. Unfortunately it is not always possible to determine such an S'. However it is in general possible to determine a binary matrix T, such that $S' = \begin{bmatrix} S \mid T \\ \sigma \end{bmatrix}$ satisfies the augmented constraint relations.

Theorem 4.5.2: Let S satisfy the constraint relation for a given A. Then $\exists$ a matrix $S' = \begin{bmatrix} S \mid T \\ \sigma \end{bmatrix}$; $T \in \{0\}^{n_s \cdot 1}$ that satisfies the constraint relation for $A' = [A \mid \alpha]$ if one of the following conditions is met:

i) The new state does not belong to any state group, i.e. α is a column of "0"s;

ii) No state belongs to any group that includes the new state, i.e. $\alpha_k = 1 \Rightarrow \alpha_{k\bullet}$ is a row of "0"s;

iii) The groups containing the new state have a non-empty intersection, i.e. $\exists j$ *s.t.* $\alpha_{jk} = 1$ $\forall k$ *s.t.* $\alpha_k = 1$ and $\alpha_{k\bullet}$ is not a row of "0"s.

Proof:

The new face matrix is:

$$F' = A'\bullet S' = [A \mid \alpha]\bullet\begin{bmatrix} S \mid T \\ \sigma \end{bmatrix} = [A\bullet S \mid A\bullet T] \quad \mathrm{V} \quad \alpha\bullet\sigma$$

Let us consider the three following cases:

i) α is a column of "0"s

Let C be the set of coordinates of length n_b. Let $\sigma = [c \mid 1]$; $c \in C$. There are 2^{n_b} choices for σ, and σ is disjoint from any other code.

Since $\alpha\bullet\sigma$ is a matrix of empty values and from Lemma 4.5.3: $\bar{\alpha}_i \prec s_i$, Λ $F = \Phi$ $\forall i = 1, 2, \cdots, n_s$, we need only verify that:

$$\bar{\alpha}\bullet\sigma \quad \Lambda \quad [A\bullet S \mid A\bullet T] = \Phi$$

Let us consider the right-most column of $\bar{\alpha}\bullet\sigma$ and matrix $A\bullet T$. By construction, the right-most column of $\bar{\alpha}\bullet\sigma$ has only "1" entries and $A\bullet T$ has no "1" entry. Therefore the constraint relation is satisfied.

ii) $\alpha_k = 1 \Rightarrow \alpha_{k\bullet}$ is a row of "0"s

Let C be the set of coordinates of length n_b. Let $\sigma = [c \mid 1]$; $c \in C$.

Hence $F' \equiv [F \mid \varphi] = [A\bullet S \mid A\bullet T]$ V $\alpha\bullet\sigma$ is such that $\varphi_k = 1$ and $\varphi_j = 0$ $\forall j = 1, 2, \cdots, n_s; j \neq k$. Therefore:

$$\bar{\alpha}_i \circ s_i \quad \Lambda \quad AS \;=\; \Phi \;\; \forall \;\; i=1, 2, \cdots, n_s$$

$$\Rightarrow \quad \bar{\alpha}'_i \circ s'_i \quad \Lambda \; A' \cdot S \;=\; \Phi \qquad \forall i = 1,2, \cdots, n_s$$

Let us consider now the right-most column, ψ, of $\bar{\alpha} \circ \sigma$.

Since $\psi_k = \phi$ and $\psi_j = 1 \;\; \forall j = 1, 2, \cdots, n_s;\; j \neq k$ then $\bar{\alpha} \circ \sigma \quad \Lambda \; F' = \Phi$.

iii) $\exists j \;\; s.t. \;\; \alpha_{jk} = 1 \;\; \forall k \;\; s.t. \;\; \alpha_k = 1$ and α_k is not a row of "0"s.

Let C be the set of coordinates covered by all the faces f_k for each face $k \;\; s.t. \;\; \alpha_k = 1$ and α_k is not a row of "0"s. Note that C is non-empty. Let $\sigma = [c \,|\, 1]$; $c \in C$.

Hence $F' \equiv [F|\varphi] \;=\; [A \cdot S \,|\, A \cdot T] \quad \forall \quad \alpha \circ \sigma$ is such that:

$$\varphi_k = \begin{cases} * & \forall k \;\; s.t. \;\; \alpha_k = 1 \text{ and } \alpha_k \textit{ is not a row of } 0\text{s} \\ 1 & \forall k \;\; s.t. \;\; \alpha_k = 1 \text{ and } \alpha_k \textit{ is a row of } 0\text{s} \\ 0 & \forall k \;\; s.t. \;\; \alpha_k = 0 \end{cases}$$

Therefore:

$$\bar{\alpha}_i \circ s_i \quad \Lambda \quad AS \;=\; \Phi \;\; \forall \;\; i=1, 2, \cdots, n_s$$

$$\Rightarrow \quad \bar{\alpha}'_i \circ s'_i \quad \Lambda \; A' \cdot S \;=\; \Phi \qquad \forall i = 1,2, \cdots, n_s$$

Let us consider now the right-most column, ψ, of $\bar{\alpha} \circ \sigma$.

Since $\psi_k = \phi \qquad \forall k \;\; s.t. \; \alpha_k = 1 \quad (\psi_k = 1 \;\; \forall k \;\; s.t. \; \alpha_k = 0)$ then $\bar{\alpha} \circ \sigma \quad \Lambda \; F' = \Phi$. This completes the proof.

■

Remark 4.5.2: In some cases it is not necessary to increase the code-length when adding a state to the state set. Consider for example:

$$A = \begin{bmatrix} 110 \\ 101 \\ 111 \end{bmatrix} \qquad S = \begin{bmatrix} 00 \\ 01 \\ 10 \end{bmatrix}$$

If:

$$A' = \begin{bmatrix} 1100 \\ 1010 \\ 1111 \end{bmatrix}$$

Then $\exists S' = \begin{bmatrix} S \\ \sigma \end{bmatrix}$; $\sigma = 11$, that satisfies the augmented constraint relation.

■

Let us consider now the general case in which no assumption is made on the entries in column α.

Theorem 4.5.3: Let n_p be the number of prime rows in A' having a non-zero entry in column α. If S satisfies the constraint relation for a given A, then $\exists S' = \begin{bmatrix} S \mid R \mid T \\ \sigma \end{bmatrix}$; $R \in \{0, 1\}^{n_s \times n_p}$; $T \in \{0\}^{n_s \times 1}$ that satisfies the constraint relation for $A' = [A \mid \alpha]$.

Proof:

Without loss of generality, let us permute the rows of A', so that

$$PA' = \left[\begin{array}{c|c} A^{(1)} & \alpha^{(1)} \\ A^{(2)} & \alpha^{(2)} \\ A^{(3)} & \alpha^{(3)} \end{array}\right]$$

where P is a permutation matrix, the entries of $\alpha^{(1)}$ and $\alpha^{(2)}$ ($\alpha^{(3)}$) are all "1" ("0") and the rows of $A^{(1)}$ are prime.

Let: $R = A^{(1)T}$.

Let C be the set of codes of length n_s. Let $\sigma = [c \mid x]$; $c \in C$ and $x \in \{1\}^{1 \times np+1}$.

We show that the constraint relation is satisfied for the three submatrices $A^{(1)}$, $A^{(2)}$ and $A^{(3)}$. Consider $A^{(1)}$ first. Let:

$$F^{(1)} = [A^{(1)} \mid \alpha^{(1)}] \cdot S' = [A^{(1)} \cdot S \mid A^{(1)} \cdot R \mid A^{(1)} \cdot T] \ \mathrm{V} \ \alpha \cdot \sigma =$$

$$[\tilde{F} | \varphi_{\cdot 1} | \cdots | \varphi_{\cdot n_p+1}]$$

Note that the diagonal elements of $A^{(1)} \cdot R = A^{(1)} \cdot A^{(1)T}$ are all "1" and therefore $\varphi_{ii} = 1$; $\forall i = 1, 2, \cdots, n_p$.

Suppose by contradiction that $\exists$ a state , k, and a row, j, of $F^{(1)}$, such that:

$$\bar{a}'_{jk} \cdot s_k \cdot \Lambda \; f'_{j \cdot} \neq \phi$$

Then $a_{jk} = 0$ and $t_k \cdot \Lambda \; \varphi_{j \cdot} \neq \phi \;\; \Rightarrow \;\; t_{kj} = 1$. Since $a_{jk} = 0$ we have a contradiction.

Consider now the constraint submatrix $A^{(2)}$. Since the rows of $A^{(2)}$ are meets of $A^{(1)}$, it follows from Lemma 4.5.1 that:

$$\bar{a}^{(2)}_{i} \cdot s_i \cdot \Lambda \; A^{(2)} \cdot S' = \Phi \qquad \forall i = 1, 2, \cdots, n_s$$

Consider last the constraint submatrix $A^{(3)}$ and the corresponding face submatrix $F^{(3)} = [A^{(3)} \cdot S \,|\, A^{(3)} \cdot R \,|\, A^{(3)} \cdot T]$. From Lemma 4.5.3:

$$a^{(3)}_{i} \cdot s_i \cdot \Lambda \; A^{(3)} \cdot S' = \Phi \qquad \forall i = 1, 2, \cdots, n_s$$

Moreover $A^{(3)} \cdot T$ is a column of "0" entries. Since the trailing bit of σ is "1", then $\bar{a} \cdot \sigma \;\; \Lambda \;\; F^{(3)} \cdot = \Phi$. This completes the proof.

■

4.6 AN ALGORITHM FOR OPTIMAL STATE ASSIGNMENT

Optimal constrained coding is a complex combinatorial optimization problem. To date, it is not known whether an optimal solution can be computed by an non-enumerative procedure. A heuristic algorithm is presented here, that constructs a state assignment satisfying the constraint relation. Experimental results show that the length of the encoding generated by the algorithm is reasonably short. However it is not known whether the com-

puted solution has shortest possible code length.

The coding algorithm constructs the state code matrix S by an iterative procedure. At each step a larger set of states is considered. The theoretical results proven in Section 4.5 show that it is possible to construct a state matrix S satisfying the constraint relation for a given constraint matrix A. However such a procedure would be ineffective, if the code length grows at each step. Therefore the heuristics of the algorithm take advantage of the possibility of ordering the states and selecting the state codes, to keep the code-length short.

The coding algorithm constructs the state code matrix starting from an initial *seed* matrix $S = [0]$. An initial state is selected and assigned code "0". The constraint relation is always satisfied for a state set consisting of one element only. Thereafter states are selected one at a time and the state code matrix grows in a way that the constraint relation restricted to the coded states is satisfied .

The input to the algorithm is the constraint matrix A and the set of states U. The output is the state code matrix S, having n_s columns. The algorithm is described in Pidgin C.

CODING ALGORITHM

u = inselect(U);
$U = U - \{u\}$;
$A' = a_u$;
$S = [0]$;
$n_b = 1$;
while ($U \neq \phi$) {
    u = state_select(U);
    do {
        C = candidates(S, A');
        σ = code_select(C);
        if (σ is ϕ) {
            $n_b = n_b + 1$;
            S = adjoin(S);
        }
    }
    while (σ is ϕ);
    $U = U - \{u\}$;
    $S = \begin{bmatrix} S \\ \sigma \end{bmatrix}$;
    $A' = [A' | a_u]$;
}

■

Procedure inselect(U) and state_select(U) sort the states according to a heuristic strategy. The constraint matrix A' (face matrix F') represents a permutation of the columns of A (F') corresponding to the coded states in

the given order.

Procedure candidates(S,A') returns the set of codes of length n_b that can be assigned to state u, while satisfying the constraint relation. Let $F'(c)$ denote the face matrix obtained by assigning code c to state u. By definition:

$$F'(c) = F' \vee a_{u}c$$

The set of candidate codes $C = \{c\}$ for state u must satisfy the following requirements:

i) Code c must not intersect any face corresponding to a group not including state u, i.e. $a'_{u}c \wedge F'(c) = \Phi$.

ii) The faces (rows of $F'(c)$) must not intersect the code of any state not included in the corresponding group, i.e. $a'_{i}c_{i} \wedge F'(c) = \Phi$ for each coded state i;

The candidate set C is computed in three steps.

First C is initialized to the complement of the set of the assigned state codes, represented by S, because no candidate can be equal to an already assigned code: $C = \{0, 1\}^{n_b} \tilde{\#} S$. Complementation is done using the disjoint sharp operation, because computationally efficient and leading to a compact representation of C. Note that other techniques, such as *unate complementation* [BRAY84], might be efficiently used.

Then all the faces corresponding to state groups not containing u are deleted from C, to satisfy requirement i). Since these faces are independent of c, then: $C = C \tilde{\#} f'_{j}, \forall j \ s.t. \ a'_{ju} = 0$, where $f'_{j} = f'_{j}(c)$ is a row of the face matrix F'. The disjoint sharp operation is used again at this step. Now the candidate set C is split into minterms (by procedure split(C)) because state assignment is restricted to one-dimensional codes.

Eventually a new face matrix $F'(c) = F' \ \mathrm{V} \ a'_{ul} \prec c$ is computed for each element of C. An element c is dropped from the candidate set C, if a face of $F'(c)$ intersects the code of a state not belonging to the corresponding group. This step is performed last, because computationally expensive.

The remaining elements of C satisfy the constraint relation. Note that candidates(S,A') may return an empty set ϕ, when no code of length n_b satisfy the constraint relation.

candidates(S, A' **)**

$C = \{0, 1\}^{n_b} \ \tilde{\#} S$;
for (j=1 ; j ≤ n_i ; j=j+1) {
    if (a_{ju} *is 0*) $C = C \ \tilde{\#} \ f'_{ju}$;
}
C = split(C);
foreach (c **in** C) {
    $F'(c) = F' \ \mathrm{V} \ a'_{ul} \prec c$;
    if ($a_i \prec s_i \ \Lambda \ F'(c) \neq \Phi$) $C = C - \{ c \}$;
}
return (C);

■

The code_select routine returns an element of C according to a heuristic criterion. If C is empty, then code_select(ϕ) returns ϕ and the dimension of the code space, n_b, has to be increased.

The rationale of the choice of a code σ is the following. Codes should be selected in a way that the final code length n_b is as short as possible. Therefore it is desirable to have a non-empty candidate set C for each selected

state u. The candidate set C does not contain the vertices covered by some faces; in particular $\{f'_j, \forall\ j$ such that $a'_{ju} = 0\}$. It is therefore desirable that all the faces cover the fewest possible number of vertices of the n_b-dimensional Boolean space. Let n be the number of coded states at the current step of the algorithm and $m(c)$ be the total number of vertices covered by at least one face. Then $m(c)$-n vertices are not assigned to state codes, but still occupy a part of the n_b dimensional space that cannot be assigned to some other state. Therefore σ is chosen as: $\sigma = \arg\min m(c)$

Procedure adjoin(S) is invoked when the candidate set is empty, and the code space dimension needs to be increased. Let R be defined as in Section 4.5: $R = A^{(1)T}$, where $A^{(1)}$ is the subset of prime rows of A having a non-zero entry in column $a_{\sigma u}$. Procedure adjoin(S) returns:

$[S \mid T]$; $T = \{0\}^{n_s \times 1}$ at the first time procedure adjoin is invoked for a given selected state u or when the right-most columns of S are the columns of R;

$[\bar{S} \mid r]$ in the other cases. Column vector r is the column of R with minimal "1"-count not already adjoined to S ; $\bar{S}$ is obtained by deleting the right-most column of S at the second time procedure adjoin(S) is invoked for a given selected state; else $\bar{S} = S$.

adjoin(S**)**

```
if (called for the first time for a given u) return (S|T);
else {
        if (called for the second time for a given u)
                delete right-most column of S;
        R' = set of the columns of R not already adjoined to S;
        if ( R' is not φ ) {
                r = column of R' with minimal 1-count;
                return ([S|r]);
        }
        else return ([S|T]);
};
```

■

Lemma 4.6.1: The candidate set C in not empty after a finite number of iterations through procedure adjoin(S).

Proof:

Let n_p be the number of prime rows in A' having a nonzero entry in column a_{uj}. After $n_p + 2$ iterations, through procedure adjoin the state code matrix is $[S \mid \tilde{R} \mid T]$, where $\tilde{R}$ is obtained from R by permuting its columns. By Theorem 4.5.3 the candidate set is not empty.

■

The rationale of procedure adjoin(S) is the following. The code space dimension, n_ϕ, is increased by one. By Lemma 4.6.1 a code σ is found after a finite number of iterations through adjoin (S). However it is desirable that a code σ is found while adding the fewest columns to S, i.e. by the minimum increase of the code space dimension. For this reason procedure adjoin(S) adds to S a column at a time.

Procedure adjoin(S) returns $\overline{S} = [S \mid T]$ the first time it is invoked. In this case, the size of the faces not related to state u is not increased. Moreover a state code σ is always found after one iteration through procedure adjoin(S), when one of the conditions of Theorem 4.5.2 is met. The strategy changes if the candidate set is still empty after one iteration through procedure adjoin(S). The columns of R are adjoined to S one at a time. This corresponds to reshaping the prime faces related to state u, i.e. the faces corresponding to the prime rows in A having a non-zero entry in column a_{uj}. Reshaping consists of adding one dimension to the state code space: the new coordinate of the state codes in a prime face is set to "1", while is set to "0" for the remaining state codes. Reshaping is performed considering one prime face at a time, and by considering first the faces involving fewest states. Since in general states are related to many faces, reshaping a prime face leads to a size increase of some other face. Therefore the heuristic strategy tries to increase the least the face dimensions. If a state code σ is not found after adjoining to S all the columns of R, then vector T is adjoined to S.

Example 4.6.1: Let:

$$A' = \begin{bmatrix} 10 \\ 01 \end{bmatrix} \qquad S = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Note that S satisfies the constraint relation. Let $a_{uj} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Since

n_b = 1 and all one dimensional codes have been assigned, the candidate set is empty. Then procedure adjoin returns $[S \mid T] = \begin{bmatrix} 00 \\ 10 \end{bmatrix}$. The candidate set is still empty, because $\begin{bmatrix} S \\ c \end{bmatrix}$; $c \in \{01,11\}$ does not satisfy the constrain relation for $[A \mid a_u]$. Therefore procedure adjoin is invoked a second time and returns $[S \mid \tau] = \begin{bmatrix} 01 \\ 10 \end{bmatrix}$. The candidate set is now $C = \{00, 11\}$, because $\begin{bmatrix} S \\ c \end{bmatrix}$; $c \in \{00,11\}$ satisfies the constraint relation for $[A \mid a_u]$. Since m(00) = m(11) = 3, either candidate can be chosen for state u.

■

State ordering is crucial to obtain a short coding. Procedures inselect(U) and state_select(U) return the initial and current state to be coded respectively. The constraint matrix A is assumed to be connected: i.e. the rows of A cannot be partitioned into two or more subsets having nonzero entries only in mutually disjoint column subsets.

Remark 4.6.1: If matrix A is not connected, it can be rearranged into a block diagonal matrix, whose blocks are connected matrices. Then state ordering can be achieved by considering each block at a time.

■

The objective of state ordering is the following. States sharing the same group should have codes whose distance is as short as possible, in order to keep the size of the corresponding face small and eventually the code length short. This leads to a very intricated problem, because states belong to different state groups. Therefore it is important to consider all the intersections among state groups. Moreover the states belonging to a state group

intersection should be coded first. In this way matrix A' satisfies the assumptions of Theorem 4.5.2 and a code σ satisfying the constraint relation can be found for each state u by increasing at most by one the state code space dimension.

For this reason matrix $\bar{A} = \begin{bmatrix} A \\ A_m \end{bmatrix}$ is considered as far as state selection is concerned. The rows of A_m correspond to all the state group intersections; i.e. the rows of A_m are all meets of $\bar{A}$. The definition of state groups is extended to state groups intersections for the remaining part of this Section. A state group is said to be completely coded, if all the elements of the group are coded.

At each iteration of the algorithm, the state constraint relation is satisfied for $\bar{A}'$, where $\bar{A}'$ is obtained from $\bar{A}$ by considering only the columns related to coded states.

The rows of $\bar{A}'$ represent:

i) state groups (completely coded)

ii) proper subsets of state groups

Let n_b be the code space dimension at the current iteration of the algorithm. The faces corresponding to the rows of type i) are group faces. The first n_b coordinates of these faces cannot change, because all the elements of the groups have been coded. The faces corresponding to the rows of type ii) may change when the other states in the group are coded.

The state selection strategy is based on the following criterion: order the states so that the largest number of state groups is completely coded at each step of the algorithm. As a result the states in group intersections are coded first.

For this reason, let $\bar{A}''$ be the matrix obtained from $\bar{A}$ by deleting the columns corresponding to the coded states. Then all the rows in $\bar{A}''$ corresponding to the rows of type i) have "0" entries only, because they are related to groups of coded states. The nonzero elements in each row of $\bar{A}''$ correspond to the states that should be selected to complete the coding of the corresponding state group.

Therefore procedure state_select(U) finds first the row in $\bar{A}''$ having minimal (excluding zero) "1"-count row, i.e. the minimal state subset that has to be coded to complete a group. The selection of the state among the elements of the subset is based on the following criterion: code the state with maximum "1"-count in the corresponding column of $\bar{A}''$. This choice minimizes the nonzero elements of $\bar{A}''$ at the next iteration of the algorithm, i.e. the number of elements in state groups that are not completely coded.

A similar strategy is used by procedure inselect(U), that returns the initial state to be coded. Procedure inselect(U) returns the state corresponding to the column of $\bar{A} = \bar{A}''$ with maximum "1"-count. This corresponds again to the state that minimizes the number of elements in the state groups that are not completely coded at the next iteration of the algorithm.

Example 4.6.2: Consider the constraint matrix of Example 4.5.1 related to the FSM described in Example 4.2.1. Note that two columns have only "0" entries; i.e. the corresponding states (state_5 and state_6) do not belong to any state group.

Consider the reduced matrix:

$$A = \begin{bmatrix} 01101 \\ 10010 \\ 00011 \end{bmatrix}$$

where the columns correspond to START, state_2, state_3, state_4 and state_7 respectively. Then:

$$\bar{A} = \begin{bmatrix} 01101 \\ 10010 \\ 00011 \\ 00010 \\ 00001 \end{bmatrix}$$

ITERATION 1

Procedure inselect returns state_4. Hence state_4 is coded by "0". Then:

$$A' = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \qquad S = [0] \qquad \text{and} \qquad F' = \begin{bmatrix} \phi \\ 0 \\ 0 \\ 0 \\ \phi \end{bmatrix}$$

satisfy the constraint relation.

ITERATION 2

Now:

$$\bar{A}'' = \begin{bmatrix} 0111 \\ 1000 \\ 0001 \\ 0000 \\ 0001 \end{bmatrix}$$

The row selected is the second and the state selected is START. The code candidate set is $C = \{ 1 \}$. Then:

$$A' = \begin{bmatrix} 00 \\ 11 \\ 10 \\ 10 \\ 00 \end{bmatrix} \qquad S = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \text{and} \qquad F' = \begin{bmatrix} \phi \\ 0 \\ 0 \\ \phi \end{bmatrix}$$

satisfy the constraint relation.

ITERATION 3

Now:

$$\bar{A}'' = \begin{bmatrix} 111 \\ 000 \\ 001 \\ 000 \\ 001 \end{bmatrix}$$

The row selected is the third and the state selected is state_7. The candidate set is empty. Therefore procedure adjoin returns:

$$S = \begin{bmatrix} 00 \\ 10 \end{bmatrix}$$

The candidate set is now $C = \{ 01,11 \}$. Then:

$$A' = \begin{bmatrix} 001 \\ 110 \\ 101 \\ 100 \\ 001 \end{bmatrix} \qquad S = \begin{bmatrix} 00 \\ 10 \\ 01 \end{bmatrix} \qquad \text{and} \qquad F'' = \begin{bmatrix} 01 \\ *0 \\ 0* \\ 00 \\ 01 \end{bmatrix}$$

satisfy the constraint relation.

ITERATION 4

Now:

$$\bar{A}'' = \begin{bmatrix} 11 \\ 00 \\ 00 \\ 00 \\ 00 \end{bmatrix}$$

The selected row is the first. The next states to be coded are state_2 and state_3. The candidate set is: $C = \{ 11 \}$. Then:

$$A' = \begin{bmatrix} 0011 \\ 1100 \\ 1010 \\ 1000 \\ 0010 \end{bmatrix} \qquad S = \begin{bmatrix} 00 \\ 10 \\ 01 \\ 11 \end{bmatrix} \qquad \text{and} \qquad F'' = \begin{bmatrix} *1 \\ *0 \\ 0* \\ 00 \\ 01 \end{bmatrix}$$

satisfy the constraint relation.

ITERATION 5

State_3 is selected. Since the candidate set is empty, procedure adjoin returns:

$$S = \begin{bmatrix} 000 \\ 100 \\ 010 \\ 110 \end{bmatrix}$$

Now $C = \{ 011, 001, 101, 111 \}$. Then:

$$A' = \begin{bmatrix} 00111 \\ 11000 \\ 10100 \\ 10000 \\ 00100 \end{bmatrix} \qquad S = \begin{bmatrix} 000 \\ 100 \\ 010 \\ 110 \\ 011 \end{bmatrix} \qquad \text{and} \qquad F' = \begin{bmatrix} *1* \\ *00 \\ 0*0 \\ 000 \\ 010 \end{bmatrix}$$

satisfy the constraint relation.

ITERATIONS 6 and 7

State_6 and state_7 can be assigned to any code not covered by any face. Hence $C = \{ 001, 101 \}$. State_5 is coded by 001 and state_6 by 101.

Theorem 4.6.1: The coding algorithm terminates in a finite number of steps and constructs a state code matrix satisfying the constraint relation.

Proof

The coding algorithm iterates n_s times through the external **while** loop. For every selected state u the algorithm loops through the internal **do-while** loop until a valid code σ is found: i.e. $\begin{bmatrix} S \\ \sigma \end{bmatrix}$ satisfies the constraint relation for matrix A'. By Lemma 4.6.1 the number of iterations through the internal **do-while** loop is finite. Therefore the algorithm terminates in a finite number of steps.

Since every state-code σ is selected so that $\begin{bmatrix} S \\ \sigma \end{bmatrix}$ satisfies the constraint relation for matrix A', eventually matrix S satisfies the constraint relation for matrix A.

4.7. KISS

KISS is a computer program for state assignment of Finite State Machines. The FSM description is given as input to the program in the form of symbolic cover. Primary inputs (outputs) can be described by symbolic strings and coded as well as the internal states. Kiss generates an output file containing a minimal Boolean cover of the FSM combinational component. Information about state coding is provided on request by the user. The KISS output file can be processed by a topological compaction program, such as PLEASURE or SMILE, and eventually by a silicon assembler which generates the mask layout of a PLA with clocked feedback registers (Fig. 4.5.2) according to a given technology.

Kiss performs the following tasks. First a symbolic cover is read and a 1-hot coded representation of the FSM combinational component is written to a temporary file. The *don't care set* related to the 1-hot representation (see App. A) is generated and appended to the temporary file. Second a two-level binary-valued logic minimizer is invoked to minimize the cover; i.e. to perform the equivalent operation of a symbolic minimization. Then the minimized representation is used to define a constrained coding problem and the coding algorithm constructs a state code matrix. Eventually the coded states and state groups are replaced into the minimal symbolic cover.

Some Kiss output files are reported in App. D. For example, consider the FSM described by a symbolic cover in Example 4.3.1. The coding generated by KISS and the related minimal Boolean cover are reported in Fig

4.7.1 and 4.7.2 respectively.

STATE CODES

```
state = _START_   label =  1   code = 110
state = state_2   label =  2   code = 100
state = state_3   label =  3   code = 101
state = state_4   label =  4   code = 010
state = state_5   label =  5   code = 111
state = state_6   label =  6   code = 011
state = state_7   label =  7   code = 000
```

Fig. 4.7.1 State assignments

```
0*0*   10100
10*0   10110
*101   10001
*001   10010
1*01   01000
1110   01100
10*1   01000
```

Fig. 4.7.1 Minimal Boolean cover

Program KISS is coded in *ratfor* and consists of about 2000 lines of code. Intermediate code in *fortran*77 is available. KISS runs in a VAX-UNIX® environment, but is easily transportable to other machines.

4.8. EXPERIMENTAL RESULTS

KISS has been tested on a set of industrial FSMs. Some results are reported in Table 6.1 along with the execution times in seconds.

Note that the logic minimizer invoked by KISS performs a key role to obtain a good coding. If the minimized symbolic cover is far from the minimum one, the coded binary-valued cover contains redundant product-terms. Moreover a partially minimized symbolic cover corresponds to a partial information about state groups and eventually to a coding close to a binary enumeration of the states.

KISS has been tested in connection with minimizers POP, MINI and ESPRESSO-II. Experimental results have shown that ESPRESSO-II outperforms the other logic minimizers and enables KISS to obtain codings leading to the minimal-area PLA implementing the FSM combinational component.

Table 6.2 compares the assignments generated by KISS to those obtained using a previous approach [DEMI83f], 1-hot coding and a random assignment of minimal length. Table 6.3 compares the area estimates $c_2 \times n_\phi$ of the PLA segment depending on the state representation.

LEGEND

n_i = number of inputs

n_s = number of states

n_o = number of outputs

c_1 = symbolic cover cardinality

c_2 = minimal Boolean cover cardinality

n_b = number of bits

Parameters of some Finite State Machines coded by KISS

Logic minimizer: ESPRESSO-II

FSM	n_i	n_s	n_o	c_1	c_2	n_b	time
FSM 1	4	5	1	20	9	3	4
FSM 2	8	7	5	56	21	5	31
FSM 3	8	4	5	32	12	4	10
FSM 4	4	27	3	108	29	9	748
FSM 5	4	8	3	32	16	4	11
FSM 6	2	7	2	14	7	3	4
FSM 7	2	15	3	30	17	5	26

TABLE 4.1

Comparison of state assignments using different techniques								
	KISS		*SAP**		1-hot		minimal n_b	
FSM	c_2	n_b	c_2	n_b	c_2	n_b	c_2	n_b
FSM 1	9	3	9	3	13	5	13	3
FSM 2	21	5	33	6	24	7	44	3
FSM 3	12	4	18	2	16	4	24	2
FSM 4	29	9	80	22	52	27	87	5
FSM 5	16	4	25	5	18	8	26	3
FSM 6	7	3	9	3	10	7	8	3
FSM 7	17	5	26	14	23	15	23	4

TABLE 4.2

	KISS	*SAP**	1-hot	minimal n_b
FSM 1	27	27	65	39
FSM 2	105	198	168	132
FSM 3	48	36	64	48
FSM 4	261	1760	1404	435
FSM 5	64	125	144	78
FSM 6	21	27	70	24
FSM 7	85	312	345	92

TABLE 4.3

* SAP (State Assignment Program) implemented a previous technique [DEMI83f].

CHAPTER 5

CONCLUSIONS AND FUTURE DIRECTIONS

This dissertation has addressed the automated synthesis and optimization of PLA-based systems. It is shown how combinational and sequential functions can be effectively implemented by means of PLAs and PLA-based FSMs respectively. In particular three major contributions have been presented:

i) An innovative PLA **folding** method that allows to compact the array area while ensuring effective routing of the folded array.

ii) A new approach to PLA topological **partitioning** based on a graph interpretation of the problem.

iii) A **state assignment** technique that optimizes the size of the PLA implementing the FSM combinational component.

The above problems have been studied using theoretical formulations. The related techniques have been implemented on three computer programs: PLEASURE, SMILE and KISS. The programs are a part of the U.C. Berkeley VLSI design system and have been used effectively in chip design by several industries. Experimental results have shown their effectiveness in designing macro-cells of complex electronic systems.

There are still some open problems related to the optimal automated synthesis of PLA-based systems. Future work in the area is listed:

i) Design of a global software package for the automated optimal design of PLA-based systems, implementing all the transformations among representations at different levels of abstraction. Such an automated design system is very complex, because a complete design of PLA-based systems involves multiple-choice decisions. For example, PLA topological design can involve folding or partitioning, or a combination of both. Therefore it will be interesting to explore the use of knowledge-based systems in this perspective.

ii) Automated translation of HDL descriptions into symbolic cover and the functional minimization of a FSM. The theoretical aspect of the state minimization problem was addressed by [GRASS75] and others, but no computer program has shown to be effective to achieve functional minimization of large sequential functions.

iii) The present state assignment algorithm determines the class of present-state encodings that minimizes the number of PLA rows and selects one of minimal length. Two directions can be explored to attempt a solution to the optimum state assignment problem. First investigate the relations between next-state encoding and minimality of the PLA cover. Second study an enhanced state assignment technique that achieves minimal PLA area implementation exploiting the trade-offs between code-length (related to PLA column cardinality) and the cardinality of the minimal PLA cover (PLA rows).

iv) Memory element assignment techniques for FSMs. Curtis addressed this problem in relation with reduced dependency criteria for state assignment [CURT69]. However a unified theory of state and memory assignment to achieve minimal PLA area implementation of the FSM

combinational component has not yet been presented.

v) Better techniques for the assembly of the symbolic layout of a PLA-based FSM from a symbolic matrix with parametrized design rules and flexible structures.

vi) Exploring the combination of two or more transformations for optimal design. For example, study the interaction between logic minimization (or state assignment) and topological compaction techniques.

vii) Eventually explore alternative structures - other than PLAs - to implement combinational functions. Weinberger arrays and gate matrices have been effectively used, as well as PLAs, in VLSI design. The choice of a structure is still left to the designer's experience. However formal criteria to decide the optimal implementation should be investigated.

REFERENCES

[AHO74] A.V.Aho J.E.Hopcroft and J.D.Ullman , "*The Design and Analysis of Computer Algorithms*", Addison Wesley, 1974.

[ALLE81] J.Allen and P.Penfield, "VLSI Design Activities at MIT", *IEEE Trans. on Circ.* and *Syst.*, vol CAS-28, No. 7 pp. 645-653, jul. 1981.

[ARMS62a] D.B.Armstrong, " A Programmed Algorithm for Assigning Internal Codes to Sequential Machines ", *IRE Trans. Elect. Comp.* , vol. EC-11, pp. 466-472 , aug. 1962.

[ARMS62b] D.B.Armstrong , " On the efficient Assignment of Internal Codes to Sequential Machines ", *IRE Trans. Elect. Comp.* , vol. EC-11, pp. 611-622, oct. 1962.

[AYRE79] R.Ayres, "Silicon Compilation- A Hierarchical use of PLAs", *Proc. Des. Aut. Conf.* pp. 314-326, 1979.

[BARB77] M.Barbacci, D.Siewiorek, R.Gordon, R.Howbrigg and S.Zuckermann, "An Architectural Research: ISP Description, Simulation and Data Collection", *Proc. Nat. Comp. Conf.* vol 46, 1977.

[BARB81] M.Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Specification", *IEEE Trans. on Comp.*, vol. C-30,

pp. 24-40, jan. 1981.

[BLUM79] R.Blumberg and S.Brenner, "A 1500-gate Random-logic Large Scale Integrated (LSI) Masterslice, *IEEE Solid State Jour.*, vol. SC-14, pp. 818-822, oct. 1979.

[BOSE83] A. Bose, B.Chawla and H.Gummel, "A VLSI Design System", *Proc. Int. Symp. on Circ.* and *Syst.*, pp. 734-737, 1983.

[BRAY82a] 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", *Proc. Int. Symp. on Circ. and Syst.*, pp. 43-49, Rome, may 1982.

[BRAY82b] R.Brayton and K.McMullen, "The Decomposition and Factorization of Boolean Expressions", *Proc. Int. Symp. on Circ.* and *Syst.*, pp. 49-54, Rome 1982.

[BRAY84] R.Brayton,G.D.Hachtel,C.McMullen and A.L.Sangiovanni- Vincentelli, "ESPRESSO-II A New PLA Logic Minimization Program ", in preparation.

[BROW81] D.W.Brown, "A State-Machine Synthesizer - SMS", *Proc. Des. Aut. Conf.*, pp. 301-304, Nashville, jun. 1981.

[CHUQ82] S.Chuquillanqui and T. Perez Segovia, " PAOLA: A Tool for Topological Optimization of Large PLAs", *Proc. Des. Aut. Conf.*, pp. 300-306, Las Vegas, jun 1982.

[CLAR75] C.R.Clare, "*Designing Logic Systems using State Machines*", McGraw Hill, 1975.

[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. on Elec. Dev.*, vol ED-26, No 4, pp. 333-345, apr. 1979.

[CURT69] H.A. Curtis, "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part 1", *IEEE Trans. on Comp.*, vol. C-18, pp. 1121-1127, dec. 1969.

[CURT70] H.A. Curtis, "Systematic Procedures for Realizing Synchronous Sequential Machines Using Flip-Flop Memory: Part 2", *IEEE Trans. on Comp.*, vol. C-19, pp. 66-73, jan. 1970.

[DEMI81] G. De Micheli, "Pleasure: A Program for topological compaction of PLAs" , *Internal Report*, Harris Corporation , 1981.

[DEMI83a] G. De Micheli and A.Sangiovanni-Vincentelli, "Multiple Folding of Programmable Logic Arrays", *Proc. Int. Symp. on Circ. and Syst.*, Newport Beach (CA), pp.1026-1029, may 1983.

[DEMI83b] G. De Micheli and A.Sangiovanni-Vincentelli, "PLEASURE: A Computer Program for Simple/Multiple Constrained/Unconstrained Folding of Programmable Logic Arrays", *Proc. Des. Aut. Conf.*, Miami Beach (FL) pp. 530-537, jun 1983. and invited paper to be published on *Computer-Aided Design*.

[DEMI83c] G.De Micheli and A.L.Sangiovanni Vincentelli , "Multiple Constrained Folding of Programmable Logic Arrays: Theory and Applications", *IEEE Trans. on CAD of Int. Circ.* and *Syst.*, vol. CAD-2, No. 3 pp. 167-180 jul. 1983.

[DEMI83d] G.De Micheli and M.Santomauro, "SMILE: A Computer Program for Partitioning of Programmed Logic Array", *Computer Aided Design* No. 2 pp. 89-97, mar. 1983 and *Memorandum UCB/ERL No. 82/74.*

[DEMI83e] G. De Micheli and M.Santomauro, "Topological Partitioning of Programmable Logic Arrays", *Proc. Int. Conf. on Comp. Aid. Des.*, Santa Clara, pp. 182-184, sep. 1983.

[DEMI83f] G. De Micheli, A.Sangiovanni-Vincentelli and T.Villa, "Computer-Aided Synthesis of PLA-based Finite State Machines", *Proc. Int. Conf. on Comp. Aid. Des.*, Santa Clara pp. 154-157, sep 1983.

[DEMI84] G.De Micheli, M.Hoffman, A.R.Newton and A.L.Sangiovanni Vincentelli, "A Design System for PLA-based Digital Circuits", *Advances in Computer Engineering Design*, Jai Press (in preparation).

[DEUT83] J.T.Deutsch and A.R.Newton, "Data-flow based Behavioral-level Simulation and Synthesis", *Proc. Int. Conf. on Comp. Aid. Des.*, pp 83-64, Santa Clara, CA sep. 1983.

[DIET68] D.L.Dietmeyer and J.R. Duley, "Translation of a DDL Digital System Specification to Boolean Equations", *IEEE Trans. on Comp.*, vol

C-18, pp. 305-313, apr. 1969.

[DIRE81] S.Director, A.Parker, D.Siewiorek and D.Thomas, "A Design Methodology and Computer Aids for Digital VLSI Systems", *IEEE Trans. on Circ.* and *Syst.*, vol. cas-28, No 7, pp. 634-644, aug. 1981.

[DOLO64] T.A.Dolotta and E.G McCluskey, " The coding of internal states of sequential machines", *IEEE Trans. Elect. Comp.*, vol EC-13, pp. 549-562, oct. 1964.

[DUTT81] R.Dutton, "Stanford Overview in VLSI Research", *IEEE Trans. on Circ.* and *Syst.*, vol CAS-28, No 7, pp 654-665, jul. 1981.

[EGAN82] J.R.Egan and C.L.Liu , "Optimal Bipartite Folding of PLA", *Proc. Des. Aut. Conf.*, pp. 141-146, Las Vegas, jun. 1982.

[ELLI82] S. Ellis K. Keller A. Newton D. Pederson A. Sangiovanni-Vincentelli C. Sequin, "A Symbolic Layout Design System", *Proc. Int. Symp. on Circ.* and *Syst.*, Rome, Italy may 1982.

[FANG83] S.Fang, *High Speed Bipolar PLA Design Techniques*, Ph.D. Dissertation, U.C.Berkeley 1983.

[FELL76] A.Feller, "Automatic Layout of Low-cost Quick-turnaround Random-logic Custom LSI devices", *Proc. Des. Aut. Conf.*, pp. 79-85, jun. 1976.

[FLEI75] H.Fleisher and L.I.Maissel, "An Introduction to Array Logic", *IBM Jour. of Res.* and *Devel*, vol. 19, pp. 98-109, mar. 1975.

[FLET80] W.Fletcher, "*An Enginnering Approach to Digital Design*", Prentice Hall, 1980.

[FLOY82] R.Floyd and J.Ullman, "The Compilation of Regular Expressions into Integrated Circuits", *ACM Jour.*, vol 29, No. 3, pp. 603-622, jul. 1982.

[GARE78] M.R.Garey and D.S.Johnson, "*Computers and Intractability*", W.H.Freeman and Company San Francisco, 1978.

[GLAS80] L.Glasser, "An Interactive PLA Generator as an Archetype for a new VLSI design Methodology", *Proc. Int. Symp. on Circ.* and *Syst.* pp. 608-611, oct 1980.

[GRAH71] R.L.Graham and H.O.Pollak, "On The Addressing Problem for Loop Switching", *Bell Syst. Techn. Jour.*, vol. 50 No. 8 , pp. 2495-2519, oct. 1971.

[GRAH72] R.L.Graham and H.O.Pollak, "On Embedding Graphs in Squashed Cubes", *Graph Theory and Applications*, Lecture notes in mathematics, no. 303, Springer Verlag 1972.

[GRAS65] A. Grasselli and F. Luccio, "A Method for Minimizing the Number of States in Incompletely Specified Sequential Networks", *IRE Trans. on Elect. Comp.* , vol. EC-14, pp. 350-359 jun. 1965.

[GRAS82] W. Grass, "A Depth-first Branch-and-Bound Algorithm for Optimal PLA Folding", *Proc. Des. Aut. Conf.* pp. 133-140, Las Vegas 1982.

[GREE76] D.L.Greer, " An Associative Logic Matrix ", *IEEE Solid State Jour.* vol. SC-11, no 5, pp. 679-691, oct 1976.

[HACH80] G.D.Hachtel, A.L.Sangiovanni Vincentelli and A.R.Newton, "An Algorithm for Optimal PLA Folding" , *Proc. Int. Conf. on Circ. and Comp.* , pp. 1023-1028, New York, N.Y. oct. 1980.

[HACH82a] G.D.Hachtel,A.R.Newton and A.L.Sangiovanni Vincentelli, "An Algorithm for Optimal PLA Folding",
IEEE Trans. on CAD of Int. Circ. and Syst. , pp. 63-77 vol. 1, No. 2, apr. 1982.

[HACH82b] G.D.Hachtel,A.R.Newton and A.L.Sangiovanni Vincentelli, "Techniques for Programmable Logic Arrays Folding", *Proc. Des. Aut. Conf.*, pp. 147-152, Las Vegas, jun. 1982.

[HART61] J. Hartmanis, "On the State Assignment Problem for Sequential Machines 1", *IRE Trans. Elect. Comp.* , vol. EC-10 pp. 157-165, jun. 1961.

[HART66] J.Hartmanis and R.E.Stearns, "*Algebraic Structure Theory of Sequential Machines*", Prentice Hall, 1966.

[HENN83] J. Hennessy, "Partitioning Programmable Logic Arrays. Summary", *Proc. Int. Conf. on Comp. Aid. Des.*, pp. 180,181 , Santa Clara, CA sep. 1983.

[HILL78] F.Hill and G.Peterson, "*Digital Systems: Hardware Organization and Design*", Wiley, 1981.

[HILL81] F.Hill and G.Peterson, "*Introduction to Switching Theory and Logical Design*", Wiley, 1981.

[HOFF81] M. Hoffman, "A Method for Topological Compaction of Programmed Logic Arrays", *Master Report, University of California Berkeley*, 1981.

[HONG74] S.J.Hong,R.G.Cain and D.L.Ostapko, "MINI:a Heuristic Approach for Logic Minimization", *IBM Jour. of Res.* and *Devel* vol. 18, pp. 443-458, sep. 1974.

[HOPC79] J. Hopcroft and J. Ullman, "*Introduction to Automata Theory, Languages and Computation*", Addison-Wesley 1979.

[HU83] T.C.Hu and Y.S.Kuor, "Graph Folding and Programmable Logic Array", *Computer Science Technical Report*, No CS-71, UCSD 1983.

[JONE75] J.W.Jones, "Array Logic Macros", *IBM Jour. of Res.* and *Devel*, vol. 19, pp. 120-126, mar. 1975.

[KANG81] Sungho Kang, "Automated Synthesis of PLA Based Systems", *Ph.D. Dissertation*, 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. on CAD of Int. Circ. and*

Syst., vol. CAD-2 No. 1 pp. 18-29 , jan. 1983,

[KARP64] R.Karp, " Some Techniques for State Assignment for Synchronous Sequential Machines ", *IEEE Trans. Elect. Comp.* , vol. EC-13 pp. 507-518 , oct. 1964.

[KELL82] K. Keller A. Newton, "A Symbolic Design System for Integrated Circuits", *Proc. Des. Aut. Conf.*, jun. 1982.

[KELL83] K.Keller, "An Electronic Circuit CAD Framework", *Ph.D. Dissertation*, U.C.Berkeley 1983.

[KERN70] B.W.Kernigham and S.Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell Syst. Techn. Jour.*, vol. 49 No. 2 pp. 291-307 feb. 1970.

[KOHA64] Z. Kohavi, "Secondary State Assignment for Sequential Machines", *IEEE Trans. on Elect. Comp.* pp. 193-203 jun. 1964.

[LAND81] H.Landman, "Automatic Layout of Optimized PLA Structures", *M.S. Report*, dept. EECS, U.C.Berkeley 1981.

[LAWL73a], E.L.Lawler "Cutset and Partitions of Hypergraphs", *Networks*, No. 3, pp. 275-285, jul. 1973.

[LAWL73b] E.L.Lawler, "Optimal Sequencing of a Single Machine Subject to Precedence Constraints", *Management Science* , vol. 19 No. 5, pp. 544-546, jan. 1973.

[LAWL76] E.Lawler, "*Combinatorial Optimization : Networks and Matroids*", Holt Rinehart and Winston 1976.

[LOGU75] J.C.Logue N.F.Brickman F.Howley J.W.Jones and W.W.Wu, "Hardware Implementation of a Small System in Programmable Logic Arrays", *IBM Jour. of Res.* and *Devel*, vol. 19, pp. 110-119, mar. 1975.

[LUBY82] M.Luby U.Vazirani V. Vazirani and A. Sangiovanni-Vincentelli, "Some Theoretical Results on the Optimal PLA Folding Problem", *Proc. Int. Conf. on Circ.* and *Comp.*, pp. 165-170, New York,N.Y., oct. 1982.

[LUCC69] F.Luccio and M.Sami, "On the Decomposition of Networks in Minimally Interconnected Subnetworks", *IEEE Trans. on Circuit Theory*, vol. CT-16 pp. 184-188, may 1969.

[MAH83] G.Mah, "PANDA - A PLA Generator for Multiply Folded Arrays", *M.S. Report*, dept. EECS, U.C.Berkeley 1983 (in preparation).

[MKCL56] E.J.McKluskey, "Minimization of Boolean Functions", *Bell Syst. Techn. Jour.*, vol. 35, pp. 1417-1444, apr. 1956.

[MEAD80] C.Mead and L.Conway, "*Introduction to VLSI Systems*", Addison Wesley 1980.

[NEWT81] A.R.Newton, D.O.Pederson, A.L.Sangiovanni Vincentelli and C.H.Sequin, "Design Aids for VLSI: the Berkeley Perspective", *IEEE Trans. on Circ.* and *Syst.*, vol. CAS 28 pp. 618-633 jul. 1981.

[OGBU70] E.C.Ogbuobiri,W.F.Tinney and J.W.Walker, "Sparsity-directed Decomposition for Gaussian Elimination on Matrices", *IEEE Trans. on Power App. and Sys.*, vol. PAS-89, No. 1, pp. 141-150, jan. 1970.

[OUST84] J. Ousterhaut, G.Hamachi, R.Mayo, W.Scott and G.Taylor, "Magic: A VLSI Layout System", submitted for presentation at *Des. Aut. Conf.* 1984.

[PAIL81] J.F.Paillotin, " Optimization of the PLA Area", *Proc. Des. Aut. Conf.*, pp 406-410, Nashville, jun. 1981.

[PATI79] S.Patil and T.Welch, "A Programmable Logic Approach for VLSI", *IEEE Trans. on Comp.*, vol. C-28, No. 9, pp. 594-601, sep. 1969.

[PROE76] R.Proebsting, "*Electronics*", p. 82, oct. 28, 1976.

[ROTH58] J.P.Roth, "Algebraic Topological Methods for the Synthesis of Switching Functions", *Trans. Amer. Math. Soc.*, vol. 88, pp 301-326, jul. 1958.

[ROTH80] J.P.Roth, "*Computer Logic, Testing and Verification*", Computer Science Press 1980.

[SANG77] A.Sangiovanni Vincentelli, Li-Kuan Chen and L.O.Chua, "An Efficient Cluster Algorithm for Tearing Large-Scale Networks", *IEEE Trans. on Circ. and Syst.*, vol CAS-24, no. 12, pp. 709-717, dec. 1977.

[SASA78] T.Sasao and H.Terada, "An Application of Multiple-valued Logic to a Synthesis of Programmable Logic Arrays", *Proc. ISMVL-78*, pp. 65-72, may 1978.

[SAUC72] G.Saucier, "State Assignment of Asynchronous Sequential Machines Using Graph Techniques", *IEEE Trans. on Comp.* vol. C-21 pp. 282-288, mar. 1972.

[SCHM80] M.S.Schmookler, "Design of Large ALUs Using Multiple PLA Macros", *IBM Jour. of Res.* and *Devel*, vol 24, pp. 2-14, jan 1980.

[SEQU83] C.Sequin, "Managing VLSI Complexity: An Outlook", *Proceedings of the IEEE*, vol 71, No 1, pp. 149-166, jan 1983.

[SIGN79] *Signetics Bipolar and MOS Memory Data Manual*, pp 156-188, 1979.

[SIMA83] P.Simany, A.R.Newton and A.Sangiovanni Vincentelli, "The POP PLA Optimizer", *U.C.Berkeley CAD Group Users Guide*, 1983.

[SPAT80] H.Spath, "*Cluster Analysis Algorithms*", Ellis Horwood 1980.

[STEA61] R.E.Stearns and J. Hartmanis, "On the State Assignment Problem for Sequential Machines 2", *IRE Trans. Elect. Comp.*, vol. EC-10 pp. 593-603, dec. 1961.

[STOR72] J.R.Story H.J.Harrison and E.A.Reinhard, "Optimum State Assignment for Synchronous Sequential Circuits", *IEEE Trans. on Comp.*, vol. C-21 pp. 1365-1373, dec. 1972.

[SU72] S.Y.H.Su and P.T.Cheung, "Computer Minimization of Multi-Valued Switching Functions", *IEEE Trans. on Comp.*, vol 21, pp. 995-1003, 1972.

[SUWA81] I.Suwa and W.J.Kubitz, "A Computer Aided Design System for Segment-Folded PLA Macro cells", *Proc. Des. Aut. Conf.*, pp. 398-405, Nashville, jun. 1981.

[TISO67] P.Tison, "Generalization of Consensus Theory and Application to the Minimization of Boolean Functions", *IEEE Trans. on Elect. Comp.*, vol. EC-16, pp. 446-456, aug. 1967.

[TORN68] H.C.Torng, "An Algorithm for Finding Secondary Assignments of Synchronous Sequential Circuits", *IEEE Trans. on Comp.*, vol. C-17 pp. 416-469, may 1968.

[TRAC66] J.H.Tracey, "Internal State Assignment for Asynchronous Sequential Machines", *IEEE Trans. on Elect. Comp.*, vol. EC-15, pp. 551-560, aug. 1966.

[TRIM81] S. Trimberger, J. Rowson, C.R.Lang and J.P.Gray, "A Structured Design Methodology and Associated Software Tools", *IEEE Trans. on Circ. and Syst.*, vol. CAS-28, pp. 618-633, jul. 1981.

[WEBE79] H.Weber, "High Level Design of Programmed Logic Arrays", *Proc. Int. Symp. on CHDL*, oct. 1979.

[WEIN67a] A.Weinberger, "Large Scale Integration of MOS Complex Logic: a Layout Method", *IEEE Sol. St. Jour.* vol. SC 2, No. 4, pp. 182-190, dec 1967.

[WEIN67b] P.Weiner and E.J.Smith, "Optimization of Reduced Dependencies for Synchronous Sequential Machines", *IEEE Trans on Elect. Comp.*, vol. EC-16, pp. 835-847, dec. 1967.

[WEIN79] A.Weinberger, "High-speed Programmable Logic Array Adders", *IBM Jour. of Res.* and *Devel*, vol. 23, pp. 163-178, mar. 1979.

[WOOD79] R.A.Wood, "A High Density Programmable Logic Array Chip", *IEEE Trans. Comput.*, vol. C-28, pp. 602-608, sep. 1979.

APPENDICES

APPENDIX A

BASIC DEFINITIONS OF SWITCHING THEORY

The representation of combinational functions at the logic level is based on the use of logic variables. Most implementations of logic functions, as those addressed by this dissertation, use **binary-valued** circuits, i.e. circuits having two stable states, **LOW** and **HIGH** that are represented by the logic values {0,1}. A logic (boolean) variable is a variable that can take a value in the set {0,1}. A logic variable is represented by "*" when the variable can take either value in the set {0,1}. A variable represented by "*" is called a *don't care* condition. In the other cases it is called *care* .

The n-dimensional boolean space $\{0,1\}^n$ can be represented by an n-dimensional cube [ROTH80] and is referred to as n-dimensional **cube**. The subspaces of the n-dimensional cube are termed **subcubes** or **faces**. An n-dimensional cube has 2^n **vertices** corresponding to each element of the n-dimensional boolean space.

A switching function f in n input variables and m output variables is a map :

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

A switching function is called **single output (multiple output)** if $m=1$ ($m \geq 1$).

Many representation of a combinational function are possible. The most straight-forward one is the **tabular form** or **truth table** . In this form the

value of the outputs are specified for each input combination.

Example A.1: The following is the truth table of a one-bit adder:

00 00
01 01
10 01
11 10 ■

A ROM implementation of a logic function represented by a truth table is straightforward, since inputs can be put in one-to-one relation with the memory addresses and outputs with the corresponding memory-cell contents.

A more compact representation of a combinational logic function is given by a logical cover, which has a straight-forward PLA implementation. To define a logical cover, some basic concepts are reported here [HILL81].

For each component of f, f_i, $i = 1, 2, \cdots, m$ the **on-set** $X_i^{ON} \subseteq \{0,1\}^n$ (**off-set** $X_i^{OFF} \subseteq \{0,1\}^n$, **don't care set** $X_i^{DC} \subseteq \{0,1\}^n$) is the set of the input values such that $f_i(X_i^{ON}) = 1$ ($f_i(X_i^{OFF}) = 0$, $f_i(X_i^{DC}) = *$).

A logical **implicant** is a pair of row vectors in $\{0,1,*\}^n$ and $\{0,1,*\}^m$ respectively. The former is called input part and the latter output part of the implicant. An **implicant of a switching function** f is such that the subspace specified by the input part belongs to the on-set (off-set, don't care set) of f_i, $i = 1, 2, \cdots, m$ if the i-th entry in the output part is "1" ("0","*").
A don't care condition in the $j-th$ position of an input part means that the $j-th$ input variable may be either 1 or 0. A don't care condition in the $i-th$ position of an output part means that f_i is not specified for the corresponding input and can be either 1 or 0.

A **minterm** is an implicant with no don't cares values ("*") and only one "1" in the output part.

Implicants and minterms have a geometrical representation. The representation of multiple-output implicants (or sets of multiple-output implicants) is obtained by considering each output component at a time. Since each output component can have three values (0,1,*), three n-dimensional cubes are used. An implicant (minterm) is represented in the n-dimensional cube corresponding to the value of the output part component. In particular an implicant is represented by the face corresponding to the subspace specified by the care entries of the input part. The **size** of the implicant is the dimension of the corresponding subspace. A minterm is represented by the vertex specified by the care entries of the input part.

Relations and operations on implicants represented in the same n-dimensional cube are defined on the basis of this representation. Let A and B be two implicants (or sets of implicants).

A is **equivalent** to the set of minterms corresponding to the vertices of the face representing A. $A = \phi$ if the equivalent minterm set is empty.

A **covers (contains)** B ($A \supset B$) if the set of minterms equivalent to A contains the set of minterms equivalent to B.

A **intersects** B ($A \cap B$) if the set of minterms equivalent to A has a non-empty intersection with the set of minterms equivalent to B.

A is **disjoint** from B if their intersection is empty.

The **union** of A and B ($A \cup B$) is the set of implicants equivalent to the union of the minterm sets equivalent to A and B.

The (disjoint) **sharp** of A and B , A # B , ($A\tilde{\#}B$) is the set of (mutually disjoint) implicants equivalent to the minterm set equivalent to A after having deleted the minterms equivalent to $A \cap B$.

The **conjunction** (AND) of two implicants is the largest size implicant contained in both of them.

The **disjunction** (OR) of two implicants is the smallest size implicant containing both of them.

The **distance** between two implicants is the number of positions in which they differ and both entries are cares.

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 i-th 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 $i = 1, 2, \cdots, m$. Note that a truth table is a cover.

A **prime implicant of a switching function** f is an implicant not contained in any implicant of f. A cover is said to be **irredundant** if no proper subset is a cover of f. A **minimal** cover is an irredundant cover of prime implicants. A **minimum** cover is a minimal cover of minimum cardinality.

A cover of a switching function, and in particular a truth table, can be seen as a pair of matrices. These matrices are referred to as **input** and **output personality** matrices. The PLA **personality** matrix is the partitioned matrix whose components are the input and output personality matrices.

Example A.2 : A single-output switching functions can be specified as follows:

X^{ON} = { 001, 100, 110, 111 }

X^{OFF} = { 000, 010 }

X^{DC} = { 011, 100 }

The truth table of the function isa set of minterms:

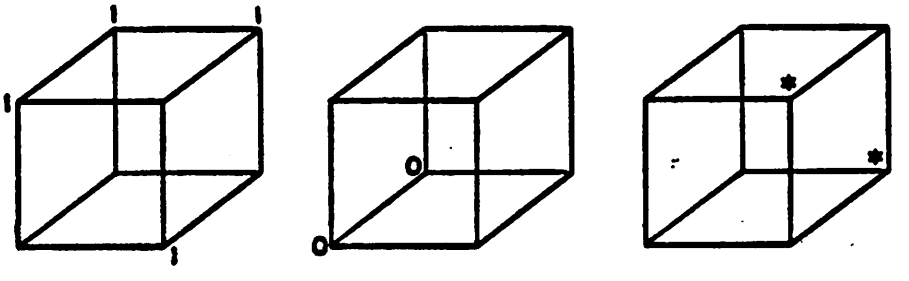
000	0
001	1
010	0
011	*
100	*
101	1
110	1
111	1

Alternatively the function can be specified by a cover i.e. by a set of implicants:

1*0	1
111	1
001	1

The cover is irredundant, but not minimal because the first implicant is not prime. The following cover is prime and minimal:

1**	1
**1	1



Boolean cube representation

Multiple-valued logic functions have been studied in connection with circuits having several stable equilibrium points [SU72]. Multiple-valued logic functions are used in this dissertation in connection with symbolic (code independent) representations of a logic function.

Several notations are used to represent multiple-valued logic values. For example, the different logic levels can be represented by integer values : 0, 1, 2, $\cdots$, p-1. This is an extension of the binary notation to a p-valued representation.

The **positional cube** notation is used in this dissertation [SU72]. A p-valued logical variable is represented by a string of p binary symbols. Value r is represented by a "1" in the r-th position, all others being "0". Note that the positional cube notation allows to represent a set of values with one string. The disjunction (multiple-valued logical OR) of several values is represented by a string having "1"s in the corresponding positions. Therefore the "don't care" value is represented by a string of "1"s and the empty value by a string of "0"s.

Example A.3: Let x be a 7-valued variable. Variable x assuming value 2 can be represented by 0010000. Variable x assuming the disjunction of values 2 and 4 is represented by 0010100. ■

Multiple-valued logic variables can be represented by means of a set of binary-valued variables. This representation is very convenient, because it allows to use techniques and methods for binary-valued functions on multiple-valued functions. In particular, binary-valued logic minimizers can be used to perform multiple-valued logic minimization [BRAY84].

In a binary-value representation, a p-valued variable can be represented by the coordinates of a vertex of the p-dimensional Boolean cube, or equivalently by a minterm. (Minterm and implicant input parts are referred to as minterms and implicants for the sake of simplicity in the sequel.) Value r is represented by the minterm having a "1" in the $r-th$ position, all others being "0"s. Therefore the binary-coded representation of a value is the same as the positional-cube notation. This representation is referred to as **1-hot coding** , because each value of the multiple-valued variable corresponds to one and only one binary value "1" (HIGH) in the binary representation.

The disjunction of two or more values is represented in the binary notation by the disjunction of the corresponding minterms, i.e. by the implicant having "*"s in the corresponding positions, all others being "0"s.

Example A.4: Let x be a 7-valued variable. The disjunction of values 2 and 4 is represented by: 00*0*00, corresponding to 0010000 V 0000100. ■

Therefore the *don't care* condition is represented by a string of *don't care* values ("*"). Note that the representation of the disjunction of some values differs from the corresponding positional-cube notation. However these positional-cube notations can be transformed into binary-valued representations by replacing "1"s with "*"s.

In the binary notation every value is a vertex of the p-dimensional cube. Since there are 2^p vertices and p values, only a subset of the vertices represent a value, and in particular the subset of vertices adjacent to the origin, i.e. having one and only one coordinate equal to "1". All the remaining

vertices do not represent existing values and are included into the don't care set of any multiple-valued function represented by 1-hot coding.

Example A.5: Minterm 1100000 does not represent any existing value. Implicant *100000 is equivalent to minterms 0100000 and 1100000 and therefore represents value 1. ■

A multiple-valued logic function can be represented by binary-valued variables by specifying the appropriate don't care set.

Example A.6: Consider the multiple-valued function specified in the positional cube notation by:

X^{ON} = { 0011, 1000 }

X^{OFF} = { 0100 }

The corresponding binary-valued representation is:

X^{ON} = { 00**, 1000 }

X^{OFF} = { 0100 }

X^{DC} = { 0000 , 11** , 1*1* , 1**1 , *11* , *1*1 , **11 }

The don't care set specifies the non-existing values. ■

A binary-valued representation of a multiple-valued function can be interpreted as follows. Each binary-valued implicant can be represented in the positional-cube notation by the equivalent minterms. Note that only a subset of these represent logic values, and in particular those having one and only one coordinate equal to "1". The implicant represents the multiple-valued

disjunction of the values corresponding to the elements of the subset.

Example A.7 : The binary-valued representation:

X^{ON} = { *00* , 1100 }

X^{OFF} = { 0100 }

is equivalent to:

X^{ON} = { 0000 , 1000 , 0001 , 1001 , 1100 }

X^{OFF} = { 0100 }

Therefore the multiple-valued on-set is value "0" and "3" and the off-set is value "1".

■

APPENDIX B

PLEASURE PROGRAM AND EXAMPLES

APPENDIX C

SMILE PROGRAM AND EXAMPLES

APPENDIX D

KISS PROGRAM AND EXAMPLES