TOPOLOGICAL OPTIMIZATION OF MULTIPLE LEVEL ARRAY LOGIC

by

Srinivas Devadas

Memorandum No. UCB/ERL M86/95

12 December 1986

TOPOLOGICAL OPTIMIZATION OF MULTIPLE LEVEL ARRAY LOGIC

by

Srinivas Devadas

Memorandum No. UCB/ERL M86/95

12 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TOPOLOGICAL OPTIMIZATION OF MULTIPLE LEVEL ARRAY LOGIC

by

Srinivas Devadas

Memorandum No. UCB/ERL M86/95

12 December 1986

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ACKNOWLEDGEMENTS

I am indebted to my research advisor Professor Richard Newton for his encouragement and support throughout the course of this work. I am also grateful to Professors Alberto Sangiovanni-Vincentelli and Carlo H. Sequin for the interest they have shown in my research project.

This work is supported in part by the Digital Equipment Corporation, the Semiconductor Research Corporation, and the Defense Advanced Research Projects Agency under contract N00039-86-R-0365. Their support is gratefully acknowledged.

All the members of the "CAD group" here at Berkeley have helped me in many ways over the past two years — I wish to especially thank Tony Ma, Karti Mayaram, Jeff Burns, Fabio Romeo, Albert Wang, George Jacob and Dr. Ruey-sing Wei.

Last but not the least, I thank Almighty God for unflagging moral support during the ups and downs in my life at UCB.

ABSTRACT

A generalized topological optimization tool for array based layout styles is presented. This tool can be used for automated layout synthesis of logic networks in a variety of technologies and design styles, including static CMOS, static NMOS and dynamic CMOS Domino structures. Results obtained compare favorably with technology and design-style-specific synthesis systems.

The topological optimization tool is a generalized array optimizer, which can be used for the multiple constrained folding of Programmable Logic Array, Gate Matrix, Weinberger Array, Multi-Level Matrix, and Storage/Logic Array structures. The optimizer uses simulated-annealing-based algorithms and performs as good as or better than existing specialized PLA folding programs and Gate Matrix folders. The different layout style alternatives allow area-efficient synthesis of logic circuits in various technologies. Layout for sequential logic in the form of Storage/Logic Arrays has been automated for the first time.

A multi-processor implementation of the simulated-annealing-based algorithms for generalized array optimization has been developed on the Sequent Balance 8000 multi-processor. Dynamic windowing and dynamic partitioning techiques have resulted in an efficient parallel implementation of simulated annealing.

TABLE OF CONTENTS

CHAPTER 1

Introduction

Much work has gone into automating the integrated circuit design process over the past few years (e.g. [park79] [newt81] [bray85]). Logic synthesis tools and tools for automatic layout of logic networks are very desirable from a viewpoint of a fast design turnaround time. However, these automatically generated designs are invariably less efficient area-wise or speed-wise than manual designs, or do not meet critical performance requirements. The large number of possible design styles, static or dynamic, single MOS or CMOS, complicate the automation process. In this report, a framework for topological optimization during automatic logic network layout in the form of array-based structures is presented. The system produces designs under various constraints and a variety of design styles resulting designs as good are as good as, or better than, corresponding manual designs.

1.1. Automatic Layout of Logic Circuits

Given a gate level description, the goal of automatic layout is to synthesize a corresponding layout implementing the logic in the input description while taking into account constraints on the area and speed of the finished module. Optimization steps are crucial in a automatic layout system to ensure that the resulting layouts are area and speed efficient.

The automatic layout step involves many decisions. Depending on the technology, a layout style has to be chosen. Two broad choices exist a standard cell place and route method[souk81], or adopting a regular array-based layout

style[wein67][lope81] [pati75][hofm85a][wood79]. Array-based layout styles are relatively easy to automate but topological compaction steps are essential. The type of electrical design-style chosen is also technology dependent. For example, a Weinberger array[wein67] is suitable for single MOS technologies, whereas a Gate Matrix[lope81] is suitable for CMOS. Multi-level matrices are best for dynamic CMOS or Domino[hofm85a].

Two phases exist in the automatic layout of regular arrays: symbolic topological compaction and actual layout generation. The logic network is represented as a symbolic array (a PLA can be represented as a matrix of 0's and 1's and don't-connects) and then area-optimized by the topological reordering of the gates and signals. The topological compaction step is typically the bottleneck as regards area optimization. It is important that the tools for area compaction be very efficient if the resulting design is to compare favorably with manually generated designs.

Array based layout structures are compacted using a technique known as folding[wood79,hach80]. Layout generation is performed using tile packing methods[mayo83]. The compacted symbolic layout is converted into tiles using a context-based tiler and the tiles are then stitched together. The tiler makes local decisions only, since the global compaction step has been taken care of.

1.2. Previous Work in Automatic Layout of Array Structures

Structured forms of layout for combinational logic include Weinberger Arrays[wein67], Programmable Logic Arrays[flei75] (PLA) and Gate Matrices[lope81]. Sequential logic can be realized as a Storage/Logic Array[pati79] (SLA). Methods of automatic layout in the form of arrays can be termed as tiled methods because connection between cells is by abutment, like tiling a floor.

Topological compaction of array based structures is possible by means of a technique known as folding. Folding comes in two flavors, simple folding[hofm80,hach80] and multiple folding. In simple folding at most two signals can occupy the same row. In multiple folding, there is no limit to the number of signals which can occupy the same row. Constrained folding allows constraints on where the signals are to appear outside the array. Multiple constrained folding of PLAs was first proposed by De Micheli[demi83] and implemented in program PLEASURE. PLEASURE row and column folds PLAs under various constraints. Folding is also possible in Gate Matrices and Weinberger arrays. This is one dimensional folding as opposed to two dimensional PLA folding. Algorithms for optimal packing on an one dimensional interval have been investigated[asan82,ohts79,wing82].

Most of proposed algorithms are based on graph-theoretic interpretations and are easily trapped in local minima. These algorithms also break in highly constrained situations since they have been designed primarily for the unconstrained cases. Most algorithms for Gate Matrix layout have no provision for constraints on the signals and differing sizes of transistors in the matrix. SLAs have resisted automated implementation until now because of their very complicated structure and widely varying cell sizes.

1.3. Organization of this report

The six following chapters are organized as follows. Various regular array-based layout styles are described in Chapter 2 and previous work in topological compaction of the different structures is reviewed. The combinatorial optimization techniques known as Simulated Annealing is introduced in Chapter 3, and theoretical work done in the field of probabilistic hill climbing (PHC) algorithms as well

as previous applications of simulated annealing the physical design of integrated circuits are reviewed.

A generalized array optimization program GENIE, based on simulated annealing, for two dimensional multiple constrained folding of PLAs, Multi-Level Matrices, and SLAs is described in Chapter 4. Comparisons with specialized array compaction programs are drawn. Modified algorithms for one dimensional multiple constrained folding also based on simulated annealing are presented in Chapter 5. These algorithms can be used for Gate Matrix and Weinberger array compaction. The multi-processor implementation of these simulated annealing based algorithms using dynamic windowing and partitioning schemes to preserve the convergence properties of simulated annealing to the global minimum are described in Chapter 6. Conclusions are drawn and directions for future work indicated in Chapter 7.

1.4. Results achieved

An automatic layout system consisting of a topological optimization tool, GENIE, a context based tiler ELECTRA, and a layout generator MKARRAY[krin86] has been developed. This system produces efficient layouts and encompasses a wide range of array based layout styles.

GENIE is a generalized array optimizer which can be used for the multiple folding of PLAs, as well as for compacting Gate Matrix layouts, SLA, and Weinberger arrays. The cells in the array can be of non-uniform sizes and any sort of constraint can be placed on the input and output terminals. GENIE uses the combinatorial optimization technique called Simulated Annealing. Results obtained are uniformly better than existing specialized array optimizers and folding programs, particularly when the input locations are constrained. GENIE is the first program

for automated SLA compaction.

GENIE has been compared with a number of existing tools for topological folding and compaction of array logic. TWIST, a part of the MAMBO pipeline[hofm85a] is a program for the folding of multi-level connectivity matrices. Results up to 50% better than TWIST have been obtained. PLEASURE[demi83] is a PLA folding program also developed at Berkeley. GENIE typically produces better results than PLEASURE more so in constrained cases which are more likely in real chip designs. Up to 30% better results have been obtained in constrained examples. Detailed comparisons with specialized array compacters can be found in Chapters 4 & 5.

A multi-processor implementation of the simulated-annealing-based algorithms in the array optimizer GENIE has been developed on the Sequent Balance 8000 multi-processor. Dynamic windowing and dynamic partitioning schemes have been used to preserve the global convergence properties of simulated-annealing-based algorithms to the global minimum and efficiences of up to 75% have been achieved over 8 processors. This implementation is described in Chapter 6.

CHAPTER 2

Array Based Layout Structures

Logic circuits in various technologies and design styles can be implemented as regular arrays. Array-based layout structures have been extensively used for large scale integration of MOS logic [wein67,flei75,pati79,lope81]. These structures are used to retain a reasonably fast design turnaround time and to permit the design steps to be automated. However, standardized layouts obtained by synthesizing the logic circuits as arrays are area efficient compared to manual random logic designs only if topological compaction algorithms are applied. The algorithms do not change the logic function of the circuit but instead try to find an optimal ordering of gates, signals or transistors, as the case may be, so as to minimize the eventual area. Compaction is achieved by a process called *folding* where more than one signal or more than one gate can occupy a single row or column of the array.

2.1. Programmable Logic Array

Programmable logic arrays (PLA) are two dimensional arrays implementing a two-level combinational logic function[flei75]. The PLA consists of two planes the AND plane and the OR plane. The AND plane maps the primary inputs into minterms (product terms) required and the OR plane maps the minterms into the outputs. In practice, both these planes are implemented as NOR structures in dynamic CMOS or NMOS technologies. An array can be programmed for any arbitrary two level combinational logic function by the presence or absence of transistors in various AND or OR locations. The functionality of a PLA can be represented as a 0-1 matrix. PLA folding algorithms work on this symbolic representation using the

information as to the presence or absence of transistors at each array node. Various architectures exist for implementing PLAs[demi83].

The area of a PLA is optimized by means of row and column folding[hach82]. The technique reported in [hach82] is referred to as simple folding. A generalization of simple folding is multiple folding. All previous techniques for PLA folding rely on a graph theoretic interpretation of the problem. Multiple/simple constrained/unconstrained folding is possible in PLEASURE[demi83]. In PLEASURE a column intersection graph (CIG) is defined whose nodes are in one-to-one correspondence with the columns of the logic array. Nodes have edges between them if the corresponding columns have a transistor in the same row. Folded columns are represented by a directed edge between the two corresponding nodes. The problem is now to find disjoint clusters of nodes in the graph, fold columns without creating alternating cycles (alternating cycles render the PLA unimplementable). Constraints on the positions of rows and columns, as well as ordering constraints on the rows and columns can be handled by this technique. Unfortunately, like most heuristic techniques, this method is easily trapped by local minima. Good heuristics however ameliorate the problem and PLEASURE produces area efficient folded PLAs with reasonable cpu time expenditure.

Simulated annealing has been applied to the PLA folding problem[moor85,wong86]. In [moor85] heuristics similar to the ones used in PLEASURE are applied initially, but when folding pairs become more difficult to find hill climbing moves are generated in an effort to escape local minima. [Wong86] gives an algorithm based on simulated annealing to solve a column folding problem — their approach cannot be used for general two-dimensional row and column PLA folding.

2.2. Weinberger Array

Weinberger arrays are one-dimensional structures for standardized layout of multi-stage combinational logic networks[wein67]. In a Weinberger array a gate occupies a column, and signals occupy rows. The gates are elongated structures and the output of the gate can be tapped at any location. Thus, it is not necessary to have signals cross each other (similar to the PLA). Figure 2.1 shows a Weinberger array from [wein67]. The array usually has diffusion/metal columns and poly/metal rows. It is possible to order the gates in a Weinberger array so as to have more than one signal occupy a single row. Like in the PLA rows can be folded. Algorithms for optimal packing on a one dimensional interval have been investigated[asan82,ohts79]. Asano[asan82] describes an exact algorithm for ordering the gates of a one dimensional array to minimize row cardinality. The algorithm searches for an optimal net ordering as opposed to a gate ordering. The gate sequence corresponding to the net ordering obtained is easily constructed. Optimality-preserving pruning methods, namely branch and bound are used. Ohtsuki et. al[ohts79] give a graph-theoretic interpretation to the one dimensional packing problem. Unlike a PLA, the transistors in a Weinberger array may be of varying sizes. The algorithms mentioned above assume all the transistors are of equal sizes and minimize for row cardinality instead of row height. The optimization does not include minimizing total net length. Net length may be important when delay through the circuit must be considered.

Weinberger arrays are area efficient for single MOS (e.g. NMOS, PMOS) technologies. In a complementary MOS static technology the signals have to drive two transistors instead of one. Using a Weinberger array this can be accomplished by duplicating all the signals for the p-channel and n-channel devices but this reduces area efficiency. A layout style more suitable for a static CMOS technology

is Gate Matrix.

2.3. Gate Matrix

In a Gate Matrix[lope81] the p-channel transistors and n-channel transistors occupy different halves of the matrix. The transistors are transposed in such a fashion so all those having a common input are placed on a common polysilicon line. This common line serves a dual purpose, i.e. it is the gate of many transistors which lie on the line and it serves as the common contact among the transistors

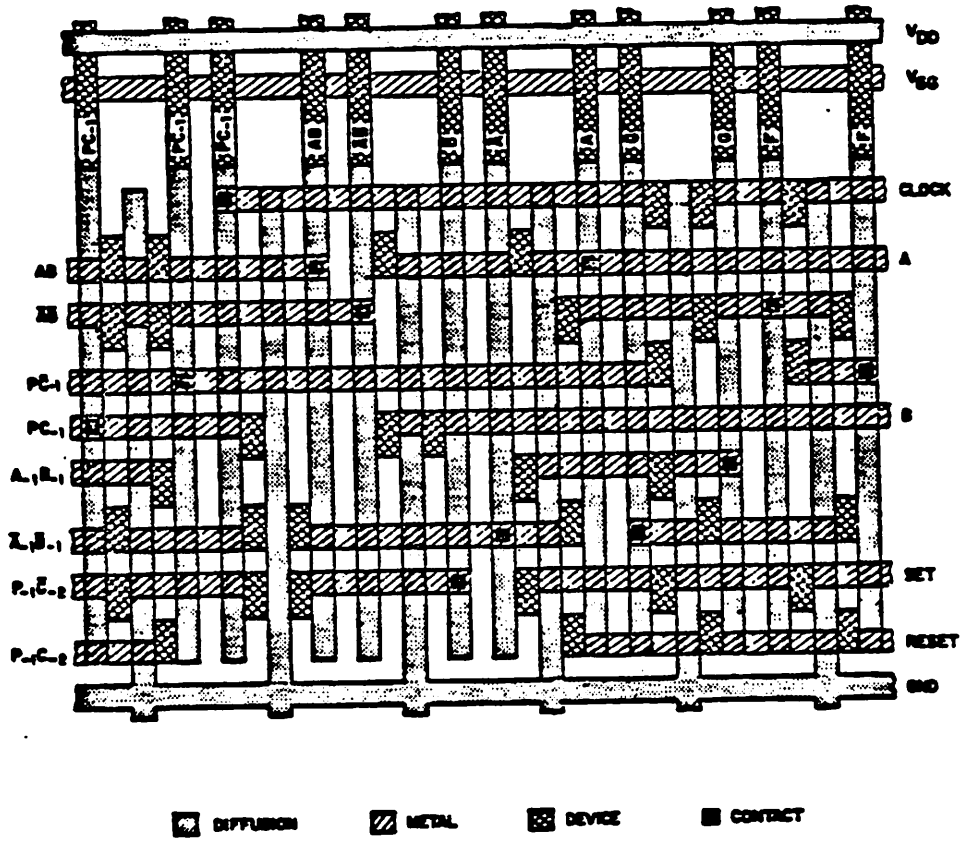


Fig. 2.1 Weinberger Array

which have a common input. The polysilicon lines which are all parallel become the columns of the gate matrix. The rows are formed by grouping together those transistor diffusions which associate with each other in a series or parallel fashion. The Gate Matrix layout style is the complement of the Weinberger array layout style in a sense — logic gates occupy rows and signals occupy columns. Thus each signal drives both the n and p transistors of the gates. Figure 2.2 shows a Gate Matrix.

The one-dimensional placement algorithms for a Weinberger array apply equally well to a gate matrix. Here the problem is to find an ordering of signals so the gates can be maximally folded. However, gate matrices have evolved to support complex gates like AND-OR-INVERT and pass transistors and the relatively straightforward techniques described above have to be modified. Wing et. al [wing82,wing85] give a graph-theoretic algorithm for compacting gate matrix layouts for complex gates and pass transistors. The technique involves formulating two assignment functions f and h such that the layout $L(f,h)$ requires the minimum number of rows as a gate matrix. The function f maps the distinct gates of the transistors to the columns of the gate matrix and the function h maps the nets of the circuit to the rows such that all the vertical diffusion runs which connect nets on different rows are realizable. A two stage approach is used which first obtains layout without regard to the vertical constraints and then permutes the rows to satisfy the constraints.

The algorithms proposed thus far for gate matrix layout have no provision for constraints on the signal columns. For example, it may be desirable to have an input signal near the right end of the matrix as opposed to the left end or middle. Also, like the Weinberger array case, transistors are all assumed to be of the same size; row cardinality alone is minimized disregarding total row height.

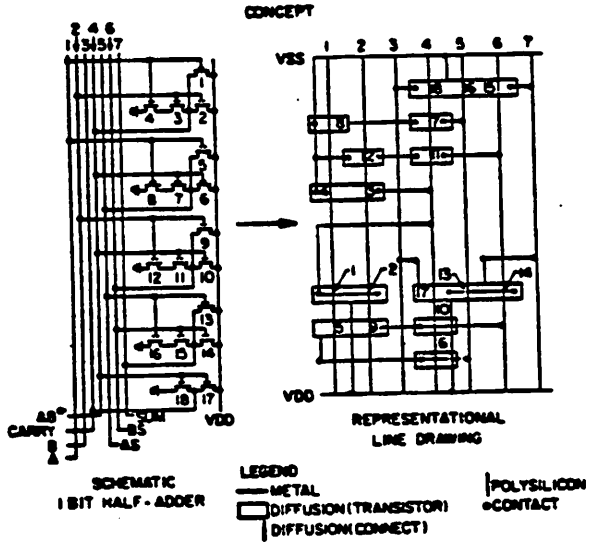


Fig. 2.2 Gate Matrix

2.4. Multi-Level Matrix

A new layout style suitable for NMOS and Domino logic circuits was proposed by Hofmann[hofm85a]. A Multi-Level Matrix (MLM) is a two dimensional structure like a PLA but supports multi-stage logic circuits unlike a PLA. It is a hybrid structure having characteristics of both gate matrix and Weinberger array structures and allows mixing of static and dynamic functions. In a MLM more than one gate can occupy a column (like a gate matrix), and more than one signal can occupy a row (like a Weinberger array). Figure 2.3a shows a symbolic representation of a folded MLM. The characters s and S denote a series transistor in the first and second gates on a column respectively, similarly characters p and P denote parallel transistors in the first and second gates on a column, and the characters o and O denote output connections. Figure 2.3b shows the actual layout of the MLM in a technology with two levels of metal interconnect.

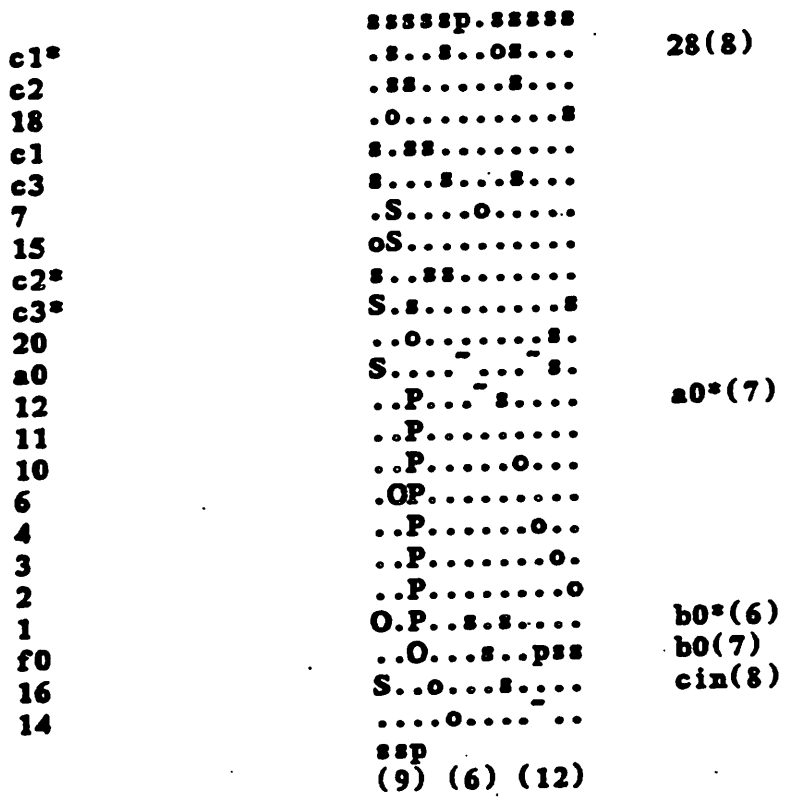


Fig. 2.3a Multi-Level Matrix

Like a PLA a MLM can be both row and column folded. The intermediate inputs can be multiply folded without any area penalty. Hofmann et.al[hofm85b] proposed an algorithm similar to [demi83] for MLM folding. TWIST is a program for multiple constrained folding of MLMs. Rows are multiply folded in TWIST but the layout style constrains the columns to be simply folded. The load devices for an NMOS circuit would be on top or bottom of the matrix as would be the buffers and clocked load of a Domino circuit. The ordering of signals is significant in TWIST as delay optimization is performed along with folding. Also, since signals are ordered relative to the output buffer when columns are folded, the ordering of

signals to the flipped gate need to be inverted. The necessity to invert or "flip" constraints when a column is folded differentiates the folding of these arrays from other structured arrays like PLAs or gate matrices.

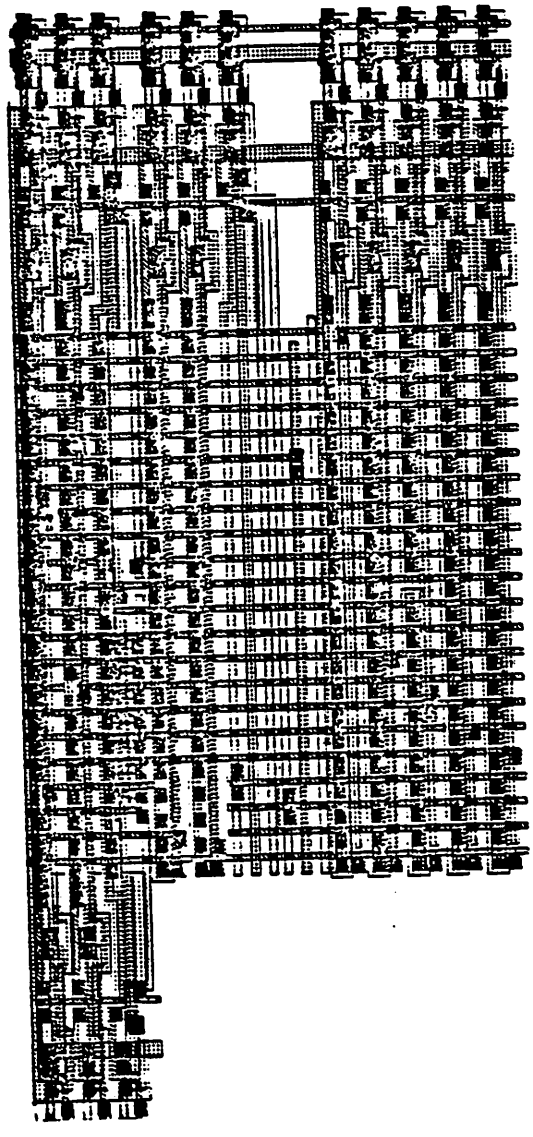


Fig. 2.3b Layout of Multi-Level Matrix in Fig. 2.3a

2.5. Storage/Logic Array

Storage/Logic Arrays were first described in 1975[pati75]. They are a form of structured logic derived from PLA where the AND and OR planes of the PLA are folded into a single plane. The SLA supports multi-stage combinational logic with embedded memory elements. Memory elements are placed on the grid itself and can be randomly distributed within the SLA logic. Columns may also contain boolean combinations of state values. Using columns to generate these boolean expressions permits multiple levels of logic. All columns and rows can be broken at arbitrary locations.

SLAs are ideally technology independent. That is, one SLA program should be portable, without change, between different process technologies. Unfortunately, this is not the case and it becomes necessary to select a particular process and implementation based on the individual circuit needs[smit82]. For example I^2L is extremely limited in allowing large number of actions being controlled by a row or column because of poor fanout. NMOS can handle heavy loads, but at the price of low speed. CMOS overcomes these objections but needs a more complex process and larger inverters. CMOS SLAs appear the most popular.

An SLA program is a two-dimensional array of symbols that specifies the placement of cells for a given circuit. The elements of the SLA program are taken from an SLA cell set which is predefined and dependent on the given technology which has been chosen. The SLA cell set will include memory elements, inverters, combinational elements for the folded AND-OR plane, row and column breaks, and row and column connection cells. The memory elements might be as sophisticated as a set/reset read/write-enabled master/slave flip-flop or as simple as a set/reset latch. Figure 2.4 shows a SLA program and the physical realization of an oscillator[smit82]. A more complicated SLA realization of and adder/subtractor is

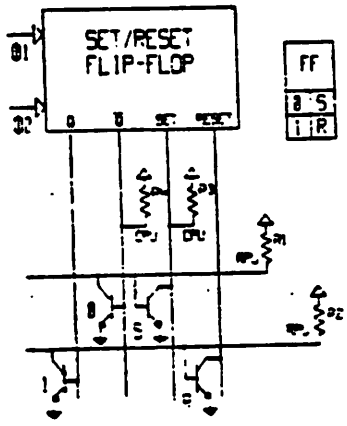


Fig. 2.4 SLA program and physical realization of an oscillator

shown in Figure 2.5.

Topologically compacting an SLA is much more difficult than folding a PLA. The cells in the SLA are of widely differing sizes; one may be ten times the area of the other. Defining large cells to be multiple cells causes adjacency constraints on the constituent cells. Ordering constraints on cells exist due to the presence of inverting buffers (a buffer cannot be placed after a transistor it is feeding into). PLA optimization techniques fail to produce good results for SLAs. No efficient program for automated SLA compaction existed before GENIE.

2.6. Generalized Array Optimization

A generalized array optimizer should perform multiple folding of PLA, Weinberger Array, Gate Matrix, MLM, and SLA structures under various constraints. The array optimizer should minimize for total row and column height and not for row or column cardinality.

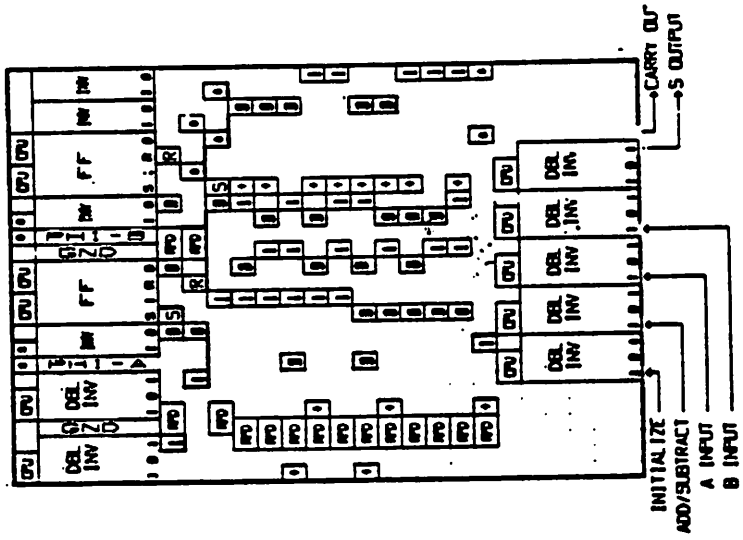


Fig. 2.5 The USCM adder/subtractor

2.6.1. Defining the Problem

The initial unfolded array can be represented as a matrix, M; non-zero elements denote the existence of a cell in the corresponding array. Each location in the matrix is a record consisting of four elements: the length of the cell in the actual array location, the width of the cell, the initial column number of the cell (in the unfolded matrix) and the initial row number of the cell. (entry.length, entry.width, entry.hor_net, entry.ver_net)

Two matrices can be derived from this matrix, *Row* and *Col*, the row and column constraint matrices. Physically, they represent the constraint that nets/cells may not overlap. The constraint matrices have entries corresponding to the *extent* of the horizontal or vertical nets i.e. if a horizontal net extends from (1,1) to (3,1) the entries Row_{11}, Row_{21} and Row_{31} would be incremented by 1. No

element of the constraint matrix can be greater than unity.

$$Row_{ij} \leqslant 1 \text{ for } all\ i\ .\ j \qquad (1a)$$

$$Col_{ij} \leqslant 1 \text{ for } all\ i\ .\ j \qquad (1b)$$

The area of a matrix is measured as the sum of row heights times the sum of column widths. The row height is the maximum of the heights of the cells in the row and the column width is the maximum of the widths in the column. If no cells exist on a row the height is obviously zero.

$$Height = \sum_{i=1}^{N_r} max\,(M_{ij}\ length)$$

$$Width = \sum_{j=1}^{N_c} max\,(M_{ij}\ .width)$$

$$A = Height\ *\ Width$$

The problem is thus to find an arrangement of cells which satisfies the constraint matrix condition (1), and minimizes area A.

2.6.2. GENIE: A Generalized Array Optimizer

GENIE is a generalized array optimization package using simulated-annealing-based algorithms which performs the above tasks. GENIE needs a symbolic representation of the array in the form of a connectivity matrix and information about the sizes of the cells, to produce a compact simply/multiply folded layout. Adjacency constraints among rows(columns), ordering constraints on rows(columns), edge constraints on signals and bounded row(column) constraints are provided for by GENIE.

GENIE is the first program to produce efficient automated SLA implementations. For the PLA and MLM case GENIE produces better results than the best specialized folders available for comparison.

CHAPTER 3

Simulated Annealing

The subject of combinatorial optimization aims at developing techniques to find the minimum or maximum values of a function of very many independent variables. A function, usually called the cost function or objective function, represents a quantitative measure of the quality of the solution. The number of variables may be of the order of tens of thousands and the number of possible solutions may be equally large.

Most of the combinatorial optimization problems dealing with the physical design of integrated circuits are NP-complete (nondeterministic, polynomial-time complete) which means that no method for an exact solution with a computing effort bounded by a power of N has been found for any of these problems. Optimal solutions to these problems generally require exponentially bounded or worse computing time. Heuristic methods with computational requirements proportional to small powers of N have been developed for many of these problems. These techniques are rather problem specific: there is no guarantee that a heuristic procedure for finding near-optimal solutions for one NP-complete problem will be effective for another.

3.1. Introduction to Simulated Annealing

Kirkpatrick, Gelatt and Vecchi[kirk83] showed there was a deep and useful connection between the behavior of systems with many degrees of freedom in thermal equilibrium at a finite temperature and multivariate or combinatorial optimization. The analogy drawn is as follows: Atoms in a solid always try to

move toward a configuration of lowest possible energy (maximum stability). If a solid is heated to a high temperature and cooled very slowly, which is the physical annealing process, the annealed solid has better mechanical properties than the original, because the atoms are now in a lower energy (more stable) state. Heating to a high temperature increases the thermal motion of atoms in the solid allowing them to redistribute themselves into a more stable configuration. At a high temperature, the atoms are likely to move to higher energy positions, but as the temperature is lowered, the movement of atoms is restricted to locations of lower energy. The physical annealing process, for best results, should have a cooling process which is very gradual and a high starting temperature. In the combinatorial optimization problem, the energy of the configuration corresponds to the cost function, thermal motion to randomly generated moves, and the temperature corresponds to the parameter controlling the acceptance of these moves. While at a high temperature, *hill climbing moves* which increase the cost function are accepted, but the probability of accepting these moves reduces as the temperature is lowered. Thus the physical annealing process is being simulated to solve the combinatorial optimization problem.

The general structure of the basic simulated annealing algorithm is shown below.

```
T = T0 ;
X = Starting_Configuration;
while("cost is changing") {
   for("a certain number of times") {
      Generate_New_State(j)
      if(accept(c(j),c(X),T)) {
         X = j;
      }
   }
   T = update(T);
}
```

Whether or not a new state is accepted is determined by the function accept():

```
accept(c(j),c(i),T ) {
   change_in_cost = c(j) - c(i);
   if (change_in_cost < 0 ) return(1);
   else {
      Y = exp(-change_in_cost/T );
      R = random(0,1);
      if ( R < Y ) return(1);
      else return(0);
   }
}
```

There are two loops in the simulated annealing algorithm which correspond to changing the temperature and generating a number of random states at a given temperature point. The temperature profile depends on the number of states generated in the inner loop and the function **update**. The acceptance function **accept** shown is the well known exponential acceptance function, variations exist.

3.2. Theoretical Results

Markov chains have been proposed as a mathematical model for simulated annealing[lund84,gema84,rome84] Using these mathematical models, simulated annealing has been shown to be a special form of a general class of algorithms called probabilistic hill climbing algorithms[rome84] having the same asymptotic properties. The asymptotic properties underline the fact that under certain assumptions on the number of moves generated by the algorithm at a certain temperature, simulated annealing produces the global optimal solution with probability 1.

Romeo *et al*[rome85] provide results to estimate how many steps should be attempted for each value of T. These results give necessary conditions to preserve the global convergence properties of the simulated annealing algorithm. One particular result which gives the expected value of the number of iterations to leave a state is reproduced from [rome84] below.

Proposition: Let i be a state such that $P_{ii}(T) < 1$, where

$$P_{ij}(T) = G_{ij}(T) * f(C_{ij}, T)$$

$G_{ij}(T)$ is the probability of generating state j being in state i and C_{ij} is c(j) - c(i). The expected value of the number of iterations required to leave i, N is given by

$$N_i = 1/(1 - P_{ii}(T))$$

A conservative estimate of the $P_{ii}(T)$ is obtained by assuming that C_{ij} is constant and equal to C_{kl} where k is the worst configuration and l the best configuration found thus far.

Unfortunately, these results are asymptotic and indicate that an infinite number of moves have to be generated in order to obtain a globally optimal solution with probability 1, which is in fact worse than exhaustive search. However, these results provide information as to what kind of cooling schedules need to be used in order to obtain good solutions minimizing cpu time requirements[huan86].

3.3. Previous applications

Simulated annealing has been applied to various problems relating to the physical design of integrated circuits like global wiring[vecc83], cell placement[sech84,sech85] and channel routing[wong85]. It has also been proposed as a tool for logic partitioning[kirk83]. Techniques used in these applications are reviewed.

3.3.1. Logic Partitioning

One of the simplest applications of simulated annealing is logic partitioning. The problem is to partition a set of modules into two or more groups such that the interconnections between the groups are minimized. Heuristic

algorithms[kern70,fidu82] have been used to solve this problem.

The annealing begins on a random set of partitions. Moves can be generated in two ways: (1) two modules in different partitions are interchanged (2) a module is displaced from one partition to another. The cost function is the total number of nets across the partitions. In general, it is desirable that the partitions should be of the same size. Moves can be allowed only if this constraint is satisfied within a tolerance limit.

3.3.2. Global Wiring

Given a placement of modules, the problem is to construct a "global" or coarse scale routing for each connection from which the ultimate detailed wiring can be completed[souk81]. Package technologies and structured image chips have prearranged areas of fixed capacity for the wires. For the global routing to be successful, it must not call for wire density which exceeds this capacity.

The global routing problem is modeled by lumping all actual pin positions into a regular grid of N_x x by N_y points which are treated as sources and sinks of all connections. The wires are to be routed along the links which connect adjacent grid points. Pre-placed components are modeled by prefilling some of the links.

The problem is to choose paths for routable connections in such a way that the likelihood of "overflows" or wires which don't fit into the eventual detailed package is minimized. This means that the most uniform possible distribution of wires plus existing blockages is sought. The objective function used is that which rewards the most balanced arrangement, and is obtained by calculating the square of the numbers of wires on each link of the package, and summing the results over the links.

$$F = \sum_{v=1}^{n} m_v^2$$

The path between two points of a connection can be an L-shaped path (a path with one bend) or a Z-shaped path (a path with one or two bends). The moves can be L-shaped moves or Z-shaped moves. For the L-shaped moves, F, the objective function has a relatively simple form. The computational effort required to obtain a good result for Z-shaped paths is larger than that for L-shaped paths because of the larger number of states available for each connection. For the same reason, a better solution can be reached for the wiring problem by allowing Z-shaped paths. So instead of using time-consuming annealing from high temperatures with the full set of Z-shaped paths two stages of annealing are performed. First, a annealing from high temperatures is performed using L-shaped paths. Then, the resulting configuration is used as a starting point for another annealing process using Z-shaped paths, but only from low temperatures. The time required for generating good solutions is thus modest. Global wiring optimization for 3000 nets on a 11 by 11 grid required just 2 seconds on a IBM VM/370 system with a 3033 processor[vecc83].

3.3.3. Cell Placement

TimberWolf[sech84,sech85] is a placement and global routing package for integrated circuits. TimberWolf is a set of programs for standard cell placement, gate array placement, macro-cell placement and standard cell global routing, which uses simulated annealing based algorithms. The standard cell placement program in TimberWolf places standard cells into rows and/or columns in addition to allowing user-specified macro blocks and pads. The results obtained by TimberWolf are on the average, better than other existing placement packages.

The TimberWolf program begins with a random initial placement of cells. A new state is generated by either exchanging two fundamental units or displacing a

unit to another location. Moves are randomly generated. New states can also be generated by orientation changes of standard cells. Interchanges and displacements are controlled by a range limiter. The range of interchange/displacement of a cell is dynamically changed during the annealing process.

The objective function in TimberWolf consists of many parts. The net length calculated as the Manhattan distance between the furthest pins of the net is one constituent. Since cells can be of different sizes, interchanges may result in cell overlap. Rather than disallow these moves, the penalty function approach is used. When two cells overlap a penalty is assessed is which is proportional to the square of the quantity of linear overlap plus an offset parameter. The objective function has an additional term which controls block lengths. The sum of the actual lengths of the cells in a particular block is compared to the actual block length. A penalty is assessed which is equal to the absolute value of the difference times a parameter value.

3.3.4. Channel Routing

Wong *et al*[wong85] proposed a scheme for channel routing using simulated annealing. The basic technique is to find a valid set of partitions (V_1, V_2, V_w) V_i containing all the nets on track i, where w is the channel width, such that w is minimized i.e. the number of tracks is minimized.

Moves are generated in three ways: (1) two subnets belonging to different groups V_i and V_j can be interchanged. (2) a subnet can be moved from group V_i to group V_j. (3) a subnet can be removed from group V_i and form a new group by itself. Vertical and horizontal constraint violations can occur during the annealing process. (1) leaves the channel width unchanged. (2) may decrease channel width and (3) increases the channel width.

The objective function is as follows:

$$C(\pi) = w^2 + \lambda_p * p^2 + \lambda_u * U$$

where w is the number of groups. p is the longest path in $G(\pi)$ and λ_p and λ_u are constants. The channel width w obviously has to be minimized. The quadratic dependence places a higher penalty on solutions with large w's and lower penalty for solutions with small w's. The reason for the second term is that p is a lower bound on the number of wiring tracks needed for all the solutions derived from π by further merging of subnets. U is defined as

$$U = \sum_{i=1}^{w} u_i^2 \text{ where}$$

$$u_i = 1 - 1/l \sum_{v}^{n} \epsilon V_i \, |v|$$

l is the channel length and $|v|$ is the length of the horizontal segment of subnet v. Here, u_i is the fraction of the track i that is unoccupied. Thus, U is a measure of the sparsity of all the tracks in the corresponding routing solution. Intuitively, all good routing solutions are densely packed, and hence have small U values.

The annealing schedule can only start from a valid partition. At each temperature, enough moves are tried until there are either N downhill moves or the number of moves exceeds $2N$ where $N = \lambda_N * m^2$. The annealing process is terminated if the number of downhill moves is less than 5% of all the accepted moves or the temperature is too low. If at any time a solution with d tracks is reached, which is clearly optimal the annealing is terminated.

This particular application of simulated annealing is interesting but has little or no practical value because very good heuristic techniques for general multilayer channel routing have been developed[brau86] which produce optimal solutions with small cpu time expenditure.

3.4. Conclusions

Simulated annealing has been applied to a variety of NP-complete problems with encouraging results. The only disadvantage with probabilistic hill climbing algorithms is cpu time expenditure. This can be partly alleviated by intelligent choices of data structures (to keep cost evaluation incremental) and annealing schedules to maximize efficiency. The main advantage of PHC algorithms as opposed to heuristic algorithms is their relative immunity to local minima traps.

In the next two chapters GENIE, a generalized array optimization package, which uses simulated annealing based algorithms for the multiple constrained folding of PLAs, Gate Matrix, Weinberger Arrays, MLMs and SLA, is described.

CHAPTER 4

GENIE: A Generalized Array Optimizer

In this chapter, a generalized, simulated-annealing-based array optimization scheme is presented that has been applied to PLA, Gate Matrix, Weinberger Array, and SLA problems[deva86a]. In all cases, the program has obtained as good or better results than the best tools available to us for comparison. In practical circuits, where a number of constraints on terminal positions were involved, our new approach reduced the area of the final layout by up to 50% compared with our previous best techniques. For results of comparable area, GENIE uses comparable CPU time to the previous approaches.

The approach begins with a planar connectivity graph which represented the circuit to be connected. The nodes in the graph represent circuit components, or *tiles*, in the final layout and the arcs represent the connections among those components. The nodes may represent single transistors, as in the case of a PLA or domino CMOS implementation, or they may represent collections of transistors and interconnect, as in the case of flip-flops in a SLA. The scheme is independent of the nature of the components — it works with a connectivity matrix description of the circuit and topologically compacts it producing a fully routed result. The matrix can be of a highly regular PLA-like structure with uniform cell sizes or it can be a representation of a multi-level logic function in gate matrix form whose cells are of varying sizes. Constraints can be placed on the positions of various input and output terminals if required. The constraints may specify a particular edge for the terminal, a particular fixed location, a particular ordering of terminals, or a combination of these constraints. Terminals may also be required to be

available on more than one edge, as in the case of bus-through connections, for example. Constraints on the aspect ratio of the folded array are also taken into account during the optimization step. The eventual result will have simply or multiply folded rows and columns in a minimum area configuration; no routing is necessary.

The basic simulated annealing algorithm was described in the previous chapter. In the following section the topological compaction algorithm is described. Illustrative examples are given in Section 4.2 and implementation details are discussed. Section 4.3 is devoted to Storage/Logic Array compaction.

4.1. A Generalized Array Compaction Algorithm

4.1.1. The problem of generalized array optimization

The problem of generalized array optimization is illustrated in Figure 4.1. An unfolded general array is shown in Figure 4.1a. Note that the cells in the array are of varying dimensions. Initially, all the signals in the array occupy distinct rows and columns. It is possible to rearrange the horizontal signals without disrupting the cell connectivity in the array and *fold* disjoint vertical signals on to the same column. Similarly, the vertical signals can be rearranged, so as to fold horizontal signals on to rows. One folded version of the array is shown in Figure 4.1b. The array has been both column and row folded.

Since the cells are of varying sizes, merely minimizing row and column cardinality does not guarantee minimum area. This is illustrated in Figure 4.1c, where another folded version of the array in Figure 4.1a is shown. Though this result has the same row and column cardinality as the previous result in Figure 4.1b its

area is smaller.

Thus, a generalized array optimization algorithm must take into account the varying dimensions of the cells.

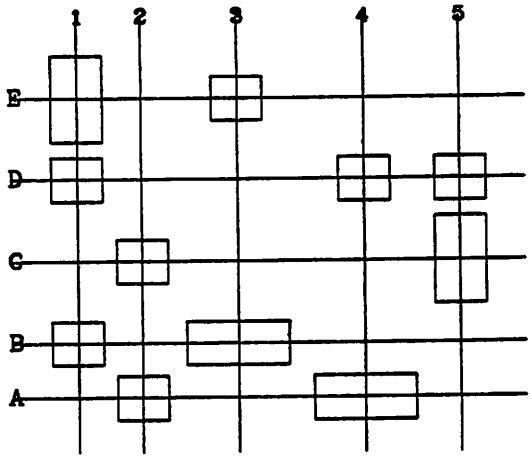
4.1.2. Array optimization using simulated annealing

The primary difference between the approach described here and the heuristic *folding* algorithms developed in the past (e.g. [hach82,chuq82,hach80]) is the existence of *hill climbing* moves during the optimization process, i.e. moves which increase the cost of the configuration (worsen the solution) are accepted based on a random criterion in an effort to escape local minima.

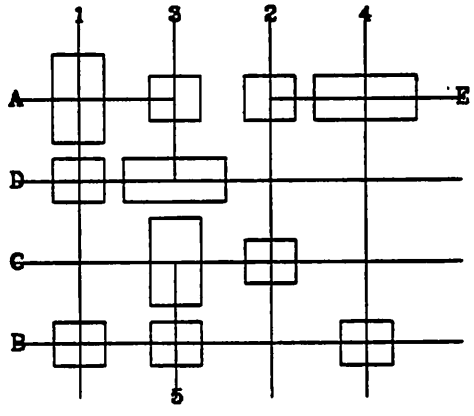
In GENIE the basic structure is a cell. Cells are interconnected by nets. We intend that the output of GENIE be the input to a tiling program, like PANDA[mah84] or TINKER[hofm85a], for actual layout. The cell denotes the existence of a transistor, or a collection of transistors, at that particular location. Cells may have different sizes depending on the width and length of devices and existence of multiple devices or internal contacts.

Nets are of two kinds, vertical and horizontal. Cells are connected to two nets — one horizontal and one vertical. Special *pad* cells are connected to input and output nets, including nets that must connect to peripheral load devices, and they serve as ports. Location of pads are constrained to the periphery of the array throughout the annealing process. As explained later, constraints on the locations of input and output terminals are implemented as additional pads.

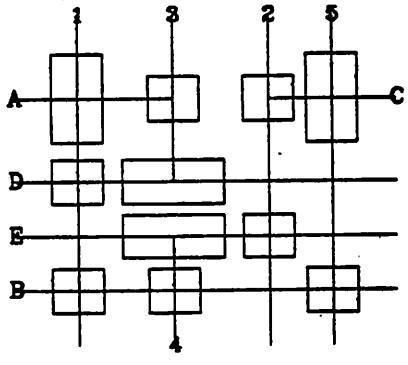
An example of an input to GENIE is shown in Figure 4.2[hofm85a] The connectivity matrix shown in Figure 4.2b has been derived from the Domino CMOS gate diagram shown in Figure 4.2a. Similar personality matrices can be derived from gate diagrams for static CMOS/NMOS design styles. In this format, the



(a) Unfolded General Array



(b) After folding disregarding cell dimensions



(c) After folding taking cell dimensions into account

Fig. 4.1 Generalized Array Optimization

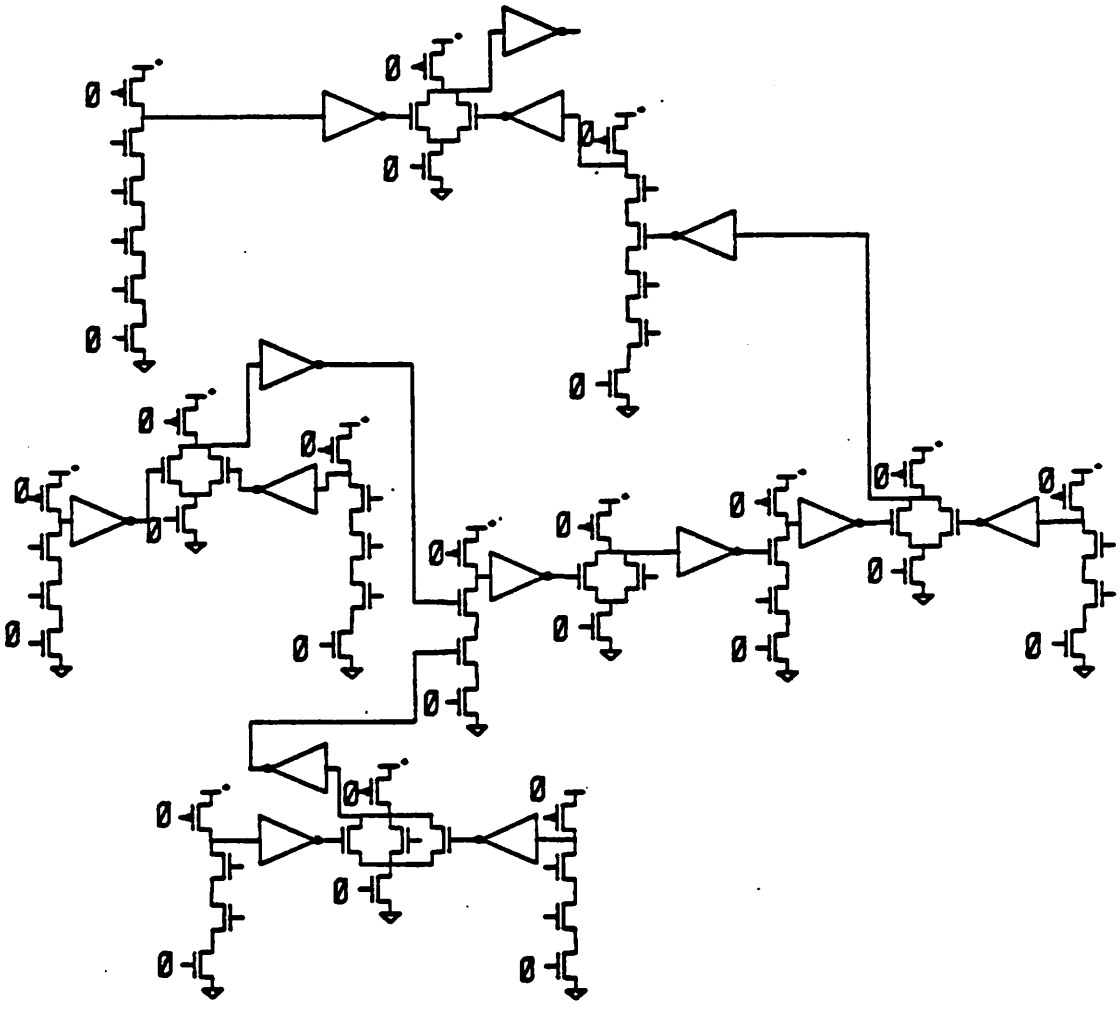


Fig. 4.2(a) Domino CMOS gate diagram.

```
new 36 27

A     s....s....s.............s..
11    .....s...........o.........
6     ............os............
8     ............oss...........
15    os........................
B     s....s....................s
C     s............s...........
D     s............s........p....
26    .op.......................
14    ..os......................
13    ..p.o.....................
out   ..o.......................
12    ....so....................
C*    .....s....s.............s.s
19    ......o...................
A*    ......s...................
B*    ......s...s.........s.....
17    .......os.................
18    .......p..................
23    .......p.o................
16    ........o..s..............
22    .........so...............
5     ...........os.............
4     ...........s.........o.....
7     ............os............
27    ..............o...........
9     ..............o.p.........
10    ................so........
25    ................po........
24    ..................so......
D*    ..................s.......
3     ...................so.....
20    ....................po....
2     ....................p..o.
21    ......................so..
1     ........................so
```

Fig. 4.2(b) Personality Matrix input to Genie

character s declares that the gate in this column is series, or AND, in nature in its top level. Similarly the character p declares that the cluster in this column is parallel, or OR, in nature at its top level. The character o indicates a gate-column

output connection to a signal row; the . character indicates that signals are bussed through this tile without connection to the current gate. The addition of the character ~ , interpreted to mean toggle, allows two-level logic in a single gate to be expressed symbolically. Toggling implies the switching between series and parallel nature in a gate. Two-level structures may need two columns to be represented and realized. A typical set of tiles that these symbols represent for a double-metal technology is illustrated in Figure 4.3.

The problem of array optimization differs from that of placement because in the former case routability is a primary consideration. Our result must be fully routed with the nets perfectly straight and non overlapping. This complicates the problem since simulated annealing is apt to generate random unacceptable configurations like overlapped cells, staggered nets or overlapping nets. Cell displacements can cause staggering of nets unless cells are constrained to move only in a particular direction through out the annealing process. We cannot disallow these configurations, if the power of simulated annealing is to be exploited fully, since the convergence properties of simulated annealing algorithms to the global optimum improves with the enlargement of the configuration space, but these configurations are not allowed in the final compacted array.

Three approaches to the compaction problem were tried using different means of generating new configurations:

(1) Net displacements.

(2) Net and constrained cell displacements.

(3) Unconstrained cell displacements.

In (3) nets at any point may be bent and may overlap each other. Constraining cell displacements to either the X or Y directions alone in (2) may result in either the vertical nets or the horizontal nets being staggered and/or overlapping. In the

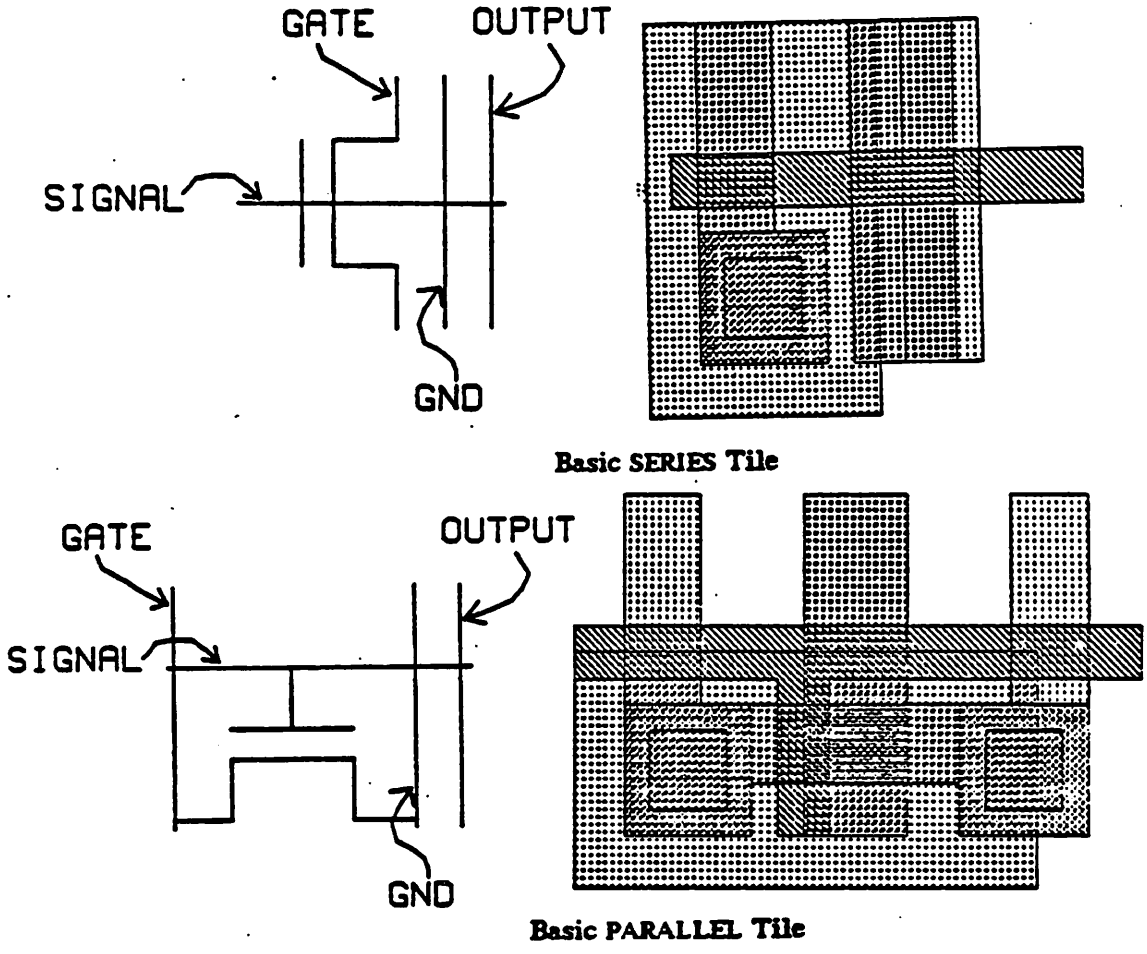


Fig. 4.3. Tiles for a double-metal, CMOS Domino implementation

case of (1) the nets are always perfectly straight but they may be overlapping.

All three approaches were implemented and it was found that (1), where the horizontal and vertical nets alone are displaced or interchanged with each other, produced the best results overall. Approach (2) yielded reasonably good results but was wasteful of CPU time and Approach (3) was discarded almost immediately as being unacceptable, again for CPU time reasons.

In (1), there is a three stage compaction process with a first stage of simulated annealing and two stages of routing nets. The routing stages are necessary for removing cell/net overlaps if they exist. Briefly these steps are:

```
Anneal_Matrix();
Route_Horizontal_Nets();
Route_Vertical_Nets();
```

These three steps are described in detail in the remainder of the section. Approach (2) is described briefly at the end of the section.

4.1.3. Annealing the matrix

The program begins with the initial configuration of nets, cells, number of rows and columns, as specified by the unfolded matrix. The objective is to reduce the total width of the columns and total length of the rows to as great an extent as possible, thus minimizing area. In the case of varying cell dimensions the column width is deemed to be the width of the widest cell on the column and the row length is the length of the longest cell in the row, since these cells set the column or row pitch respectively. All the primary inputs and outputs which are constrained to be at any edge of the matrix are terminated by *pads*. Pads are fixed cells restricted to lie at the periphery of the array. Adjacency constraints may also be included. For example, in a PLA the input and its complement should lie side-by-side in the folded matrix for an efficient implementation. This constraint to the program takes the form of an extra vertical net connecting two dummy

cells in the input and it's complement rows during annealing.

4.1.4. Generating New States

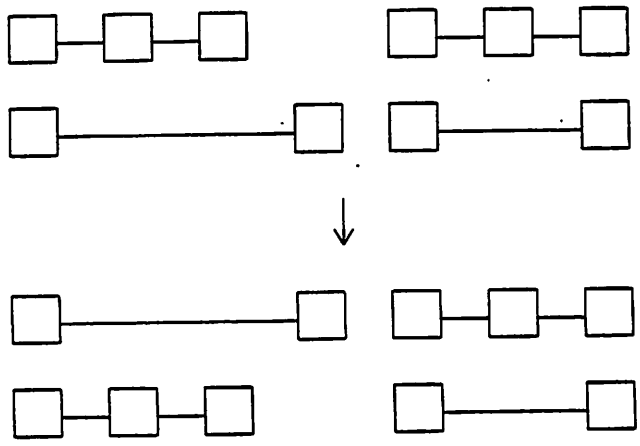
A new state is generated in this stage of annealing process by either exchanging two fundamental units or moving a unit to another location. The fundamental units in this case are vertical or horizontal nets. The overlap of horizontal and vertical nets is penalized, as is cell overlap. The penalty-function approach was first described in[kirk83]. Thus there are four different ways of generating new states: moving a vertical net and all of its cells on it to another column, interchanging two vertical nets on different columns, moving a horizontal net with all its cells into another row, and interchanging two horizontal nets (i.e. interchanging all the cells on two nets). Pads are treated the same as cells but are only connected to a horizontal net and hence the x location of a pad can never change. Pads can be left-pads or right-pads depending on whether they are constrained to the left or the right of the array. A particular horizontal net may have both left and right pads if necessary as in the case of bussed through input signals in connectivity matrices. Net interchanges and displacements are illustrated in Figure 4.4a and 4.4b.

The selection of new states is based on the following considerations:

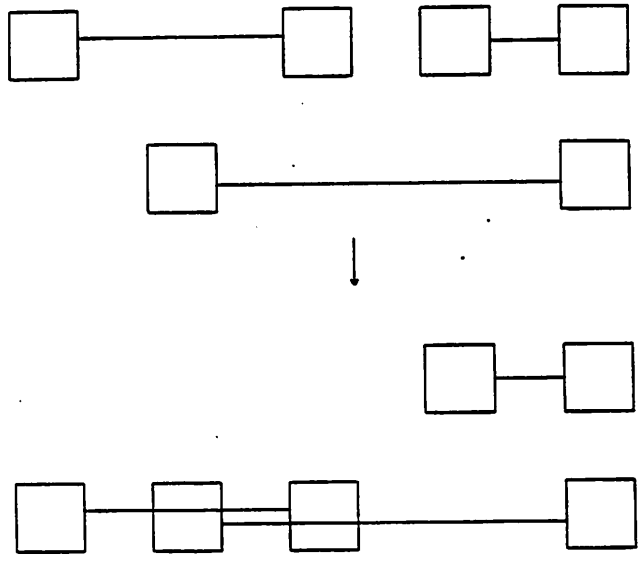
(1) A random number between one and the total number of nets is generated.

(2) A second random number is generated between one and the number of nets times RATIO1 (typically 5).

(3) If the second number happens to be less than the number of nets and the nets corresponding to the two numbers happen to be both horizontal or both vertical then the two nets represented by the two numbers are interchanged.



(a) Net Interchange



(b) Net Displacement

(4) Otherwise, the net corresponding to the first number is displaced: vertically if it is a horizontal net to a new row, horizontally if it is a vertical net to a new column.

The final solution is affected by the ratio of net displacements to net interchanges i.e. the number RATIO1. A value of 5 has been experimentally found to be effective in all test cases. The area of an array is reduced only by net displacements and not net interchanges. Net interchanges are useful in removing violations which represent unroutable configurations for example, overlapped nets and overlapped cells. Since the goal is a minimum area configuration, net displacements are in higher proportion.

The displacement of a net to a new row or column is controlled by a range limiter[sech85], which limits the range of displacement of a net either vertically or horizontally. The new location is selected randomly within the acceptable range. The range limiter is used because in the latter stages of the annealing the displacement of a net has very little chance of being accepted unless it is local. So to generate states which have high probability of being accepted, the range of possible displacement of a horizontal(vertical) net is gradually reduced from the total number of columns(rows) at the beginning to a single column(row) when the temperature approaches zero as a logarithm function of the temperature. The logarithm function was experimentally found to be very effective. Net interchanges are also allowed only over a small range toward the end of the annealing process using the range limiter.

The range limiter is also used for implementing *bounded column* or *bounded row constraints*. The range of displacement or interchange of a gate is restricted to

the specified bounds by not generating any states which violate the constraints.

4.1.5. The Cost Function:

The cost function is made up of several components. The first portion is *total net length* which is defined as the distance between its extreme cells. Nets may be weighted differently in which case the cost function is the length times the net weight. Critical nets, which must be kept short, can be weighted heavily (typically, 3-5 times higher) so as to minimize their length in the final layout. As mentioned above, adjacency constraints embody themselves as vertical nets and these nets are weighted highly.

The second portion of the cost is the *sum of overlap penalties* of all the cells. If a cell or pad is at the same location as any other, a penalty is assessed. This penalty is the sum of all the cell overlaps times a overlap parameter which is large (typically, 30-50) so as to obtain a final configuration without any overlaps.

The third portion is the *area penalty* for increasing the number of rows or columns in the array. Since the eventual objective is to reduce area, not just wire length, every time a net moves to an empty row(column) a penalty is assessed. Similarly, a negative penalty, or gain, is assessed if a net moves out of a row(column) leaving it empty. There is a corresponding area penalty parameter as in the overlap case (typically 5-10).

The fourth portion is the *net overlap penalty*. This portion exists because net overlap is directly related to routability and is assessed for both horizontal and vertical nets. Two horizontal/vertical nets lying on top of each other represents an unroutable configuration and hence the configuation is penalized.

The fifth portion is required when cells have non-uniform sizes. At every stage the width of each column and length of each row is stored. As mentioned

before. the width of a column is the width of the widest cell on it; likewise for rows. If a net displacement or interchange increases or decreases the width(length) of a column(row) then a penalty or gain is assessed similar to the area penalty. In the case of non-uniform cell width, this serves to minimize the total width and length of the array, rather than simply the number of rows and columns, which is what is needed is to minimize area.

4.1.6. The Stopping and Inner Loop Criteria:

A certain number of states (moves) per fundamental unit are generated in the inner loop, which corresponds to a temperature point in the physical annealing process. Horizontal and vertical nets represent fundamental units. The best results were obtained when the cost function attained equilibrium at every temperature point. However, at high temperatures, equilibrium is attained faster, i.e. in a fewer number of generated states, hence the number of states per fundamental unit is gradually increased from a low value (typically 2) to a high value (typically 10) near the end of the annealing process. The value is computed an inverse function of the logarithm of the temperature.

$$S(T) = S_i + (S_i - S_f)*\log(T)/\log(T_0)$$

where $S(T)$, S_i , S_f are the present, initial and final values of the number of generated states in the inner loop, and T is the present temperature.

The annealing process ends when the cost function has not changed for a cer-

tain number of temperature points at a sufficiently low temperature.

4.1.7. Temperature Profile

The function update determines to a certain extent the quality of the final result. Initially, when the cost of the configuration is increasing the temperature is decreased quickly. At every step the temperature becomes a fraction ALPHA (typically 0.8) of what it was at the previous step. This fraction ALPHA is gradually increased to about 0.90 as the rate of increase of the cost function slows down and begins to decrease. The cost function should attain equilibrium at every temperature point for best results[rome85]. Since equilibrium is reached relatively quickly at high temperatures, initially the temperature is decreased quickly.

4.1.8. Routing the Horizontal nets

After annealing, the vertical nets and horizontal nets may overlap each other. A possible overlapped configuration is illustrated in Figure 4.5a. In this second stage all the overlap of the horizontal nets is removed with the minimum change in the placement of cells produced by the anneal.

The straightening procedure is as follows. First the horizontal nets are sorted in the order of decreasing length. All locations are free at the beginning. A net is picked and if it lies on free locations alone it is fixed in its position along with its cells. If any of the locations it lies on isn't free then a new row has to be found for the net. An attempt is made to place the net in any of the other rows without altering the x coordinates of its cells and such that there is no resulting overlap due to this move. If more than one row exists where the net can be placed, the row whose height is closest to the present net's height is chosen. If this is not

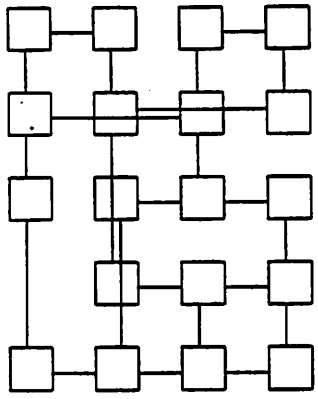
possible, a new row is added beside the net's initial row and all the cells of the net are placed on this new row again without changing their x coordinates. The net and the locations are then fixed on this new row. This process is repeated till all the horizontal nets have been placed.

The result of removing horizontal net overlap in the configuration of Figure 4.5a is shown in Figure 4.5b. Vertical net overlap, however, still exists.

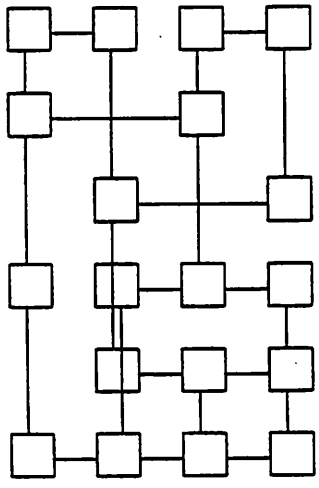
4.1.9. Routing the Vertical Nets - The Final Stage

At the end of this stage the array is compacted, fully routed and all cell and net overlaps have been eliminated. Typically, the simulated annealing produces a good placement of cells and horizontal nets minimizing vertical net violations. However those violations that do exist must be removed. This stage of routing is more complicated than the previous stage of horizontal net routing as the vertical nets have to be separated from each other without disturbing the horizontal nets. Care must be taken while routing these nets that unnecessary columns are not added.

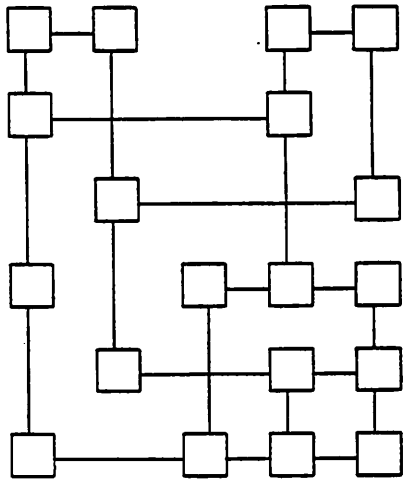
First the nets are ordered according to decreasing number of cells on the net. They are routed in this order. As before all locations are freed initially. If the chosen net and has no overlaps and lies on free locations it is merely locked in place. If a violation exists then various columns adjacent and neighboring it are examined to see if the net can be placed on that column without overlapping other vertical nets and *without causing horizontal net overlap*. Thus there is a more stringent condition here for an acceptable column than in the previous case of acceptable rows. If many columns are available for placing the vertical net the column closest to the present one is chosen. If more than one column satisfies this condition, the column whose width is closest to the present net's width is chosen.



(a) Array after Annealing with Overlaps



(b) Horizontal Net Overlap removed



(c) Vertical Net Overlap removed

Fig. 4.5

If no existing column satisfies the condition then a new column is added next to the original location and the net with its cells is placed on the column. The final result with all overlap removed of the configuration of Figure 4.5a is shown in Figure 4.5c.

The aspect ratio of the folded matrix depends to a certain extent on whether horizontal net straightening precedes vertical net straightening or *vice versa*. Relatively speaking, in the former case more rows are folded than in the latter. The same can be said for column folding in the latter case in comparison to the former. The order can be decided upon depending on initial row and column sparsity, initial aspect ratio and final desired aspect ratio. The multiplicative row and column penalty parameters in the cost function also control to a certain extent the relative number of row and column folds.

4.1.10. A Brief Description of Approach 2

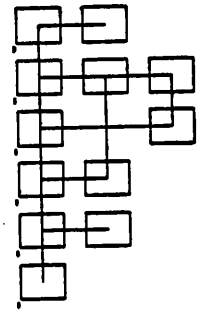
In this approach, two annealing stages are performed. The generation of states proceeds differently in the two stages. In the first stage (a) horizontal nets are displaced vertically or interchanged with other horizontal nets and (b) cells are displaced horizontally on a row. After the first stage of annealing, the vertical nets may be staggered and overlapping, but the horizontal nets will be straight though horizontal overlap may exist. The cost function has an added constituent in this approach, vertical net stagger. If a net is not straight because all its attached cells are not in the same column, a penalty is assessed. This penalty is proportional to the horizontal distance between the leftmost and rightmost cells. After the annealing, all horizontal net overlap is removed using techniques described earlier in the section.

The second stage of annealing attempts to minimize vertical net stagger and overlap without causing any horizontal net violations. Generation of states is constrained to horizontal cell displacements *within the range of the cell's horizontal net*. Displacement within the range of the net ensures that horizontal net overlap does not occur. Nets are not interchanged. The cost function in this stage is the vertical net stagger and overlap. After this second annealing stage, vertical net violations due to vertical nets not being straight or overlapping are removed and the compaction process is ended.

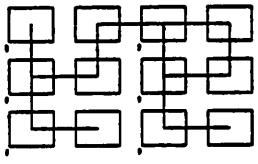
4.1.11. Splitting long nets

The minimum possible area after folding is bounded by the length of the longest nets, horizontal or vertical. One single input feeding to many product terms in a PLA can constrain the final size of the array even if the rest of the array is very sparse. Similar problems can occur in other array structures.

A solution to this problem for the general case is possible in GENIE by pre-processing the array to split the nets possessing a number of cells greater than a threshold value into two or more subnets. Route-through-cells are inserted to connect up the different subnets. If a horizontal net is split, two route-through-cells are appended, one to each subnet and connected up by a vertical route net. Vertical nets can be similarly treated. If the array is a PLA, it now becomes a pseudo MLM. It has been found that this technique results in large improvements over the conventional techniques for sparse PLAs with some long nets. An example



(a) Long Vertical Net constraining Folding



(b) Result after splitting net in two
Fig. 4.6

of how splitting a long net can reduce area is shown in Figure 4.6.

4.1.12. Parameter extraction

The cost function has many components and each component has an associated weight given by its parameter. For example, the overlap penalty parameter is typically in the range of 10-50 units and the gain for reducing the number of rows/columns is in the range of 5-10 units. Finding the absolute best set of parameters for a array or a given set of arrays is very difficult. The problem is further compounded by the fact that the efficacy of a particular set of parameters is dependent on the annealing profile i.e. the number of states generated per temperature point and the temperature profile. Also, the size and sparsity of arrays is

a variable factor affecting the parameter efficacy.

However, it is universally true that provided certain inequalities in parameter values are satisfied, if a sufficient number of states are generated per temperature point and if the temperature is decreased sufficiently slowly, the quality of the final solution shows a remarkable insensitivity to the actual parameter values. When CPU time is a factor and the annealing process is short absolute parameter values become important. The relative values of the row and column gain/penalty parameters, however, affect the aspect ratio of the final result and this fact is used to advantage to obtain a desirable aspect ratio as is explained in greater detail in Section 4.2.2.

With these in mind, extraction of parameters for four different array types was done for short annealing times. The four different types were small-dense, small-sparse, large-sparse, and large-dense arrays. The solutions obtained for a wide range of parameter values are summarized in Table 4.1. The parameter values marked with an asterisk were hardwired into the program.

The table gives normalized sets of values for three parameters. These parameters are associated with the gain for reducing rows/columns, the penalty for net overlap and the penalty for cell overlap. Normalization was done so the percentage changes in parameter values could be directly found. The absolute values are 1, 10 and 50 respectively.

4.1.13. Temperature versus Cost Graphs

It is interesting to study temperature-cost graphs for an annealing process to determine the point at which one would like to operate. The marginal utility of simulated annealing goes down as the temperature profile becomes more and more

parameter	in1	in2	in3	in4	in5	in6	in7
row	0	0	0	1	1	1	1*
net	0	1	1	0	0	1	1*
cell	1	0	1	0	1	0	1*
gm0.mat(34/20)	33/20	34/19	34/20	32/19	28/19	28/19	28/16
apla.mat(42/22)	25/20	24/19	29/19	28/19	22/20	23/20	19/16
xcpla1.mat(73/72)	43/61	42/64	50/61	45/61	33/63	31/64	27/59
ex15(68/52)	43/52	65/52	64/52	60/51	42/52	41/52	37/52

parameter	in8	in9	in10	in11	in12	in13	in14
row	1	1	1	1	1	1	1*
net	1	1	0.8	1	1	1	1*
cell	0.2	0.5	1	0.2	0.5	0.8	1*
gm0.mat(34/20)	28/19	28/19	29/18	27/19	27/18	28/17	28/16
apla.mat(42/22)	22/20	21/19	18/17	22/19	21/15	18/19	19/16
xcpla1.mat(73/72)	30/62	28/64	28/60	31/60	30/59	26/60	27/59
ex15(68/52)	42/52	42/52	40/52	41/52	41/52	39/52	37/52

Table 4.1 - Parameter variations

gradual as the solution is getting closer and closer to the global minimum. Ideally, one operates at a point where the cpu times involved are reasonable, and the solutions being obtained very close to the global minimum. Various temperature versus cost graphs for different temperature profiles all starting at the same temperature on one particular example are shown in Figure 4.7. In each of these cases, during the annealing the number of states generated per temperature point was a constant and is indicated above the corresponding curve. The parameter ALPHA was also held constant at 0.90 throughout the annealing process. The final costs are smaller for the temperature profiles with a larger number of states generated, but the difference becomes less and less marked as the number of states per temperature point increases.

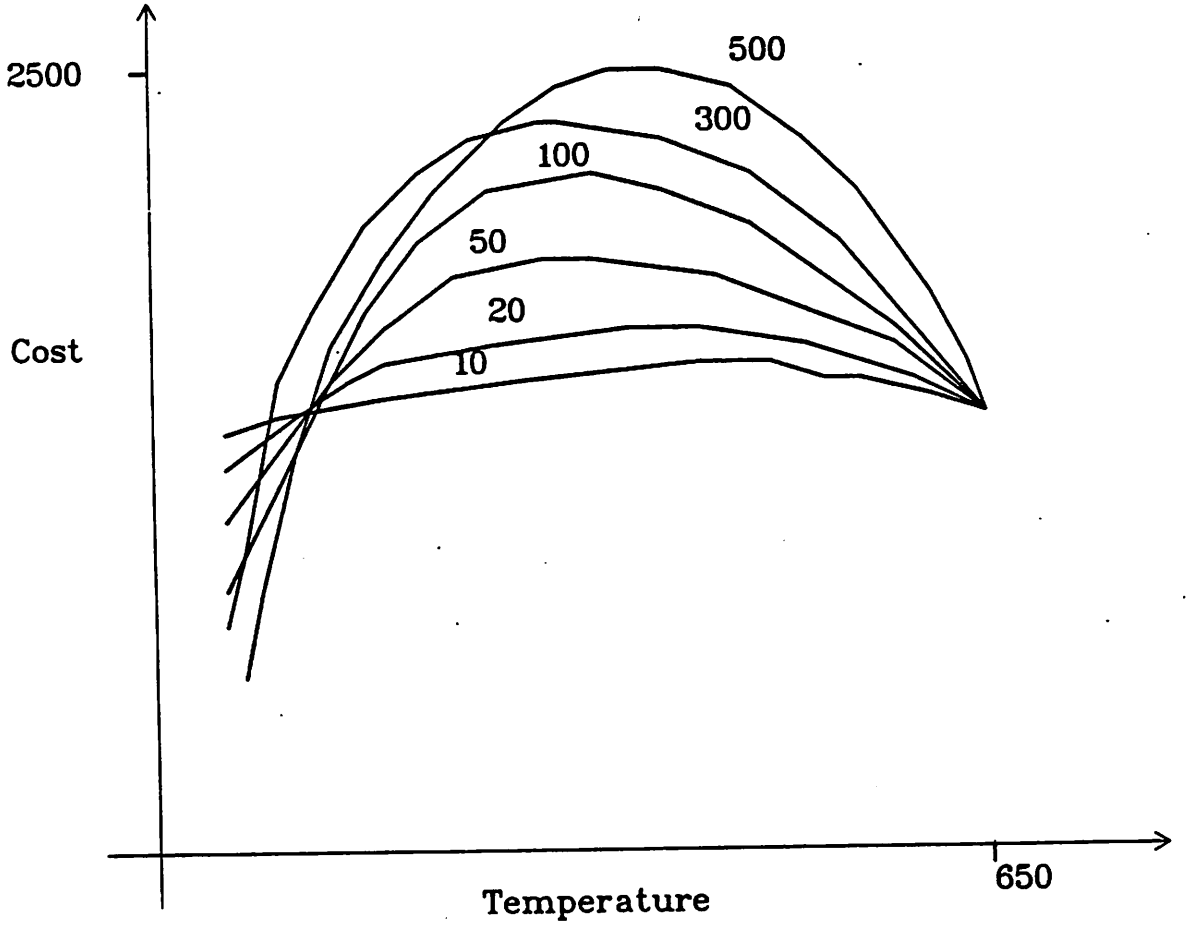


Fig. 4.7 Temperature versus Cost Graphs

4.2. Examples, Comparisons and Implementation

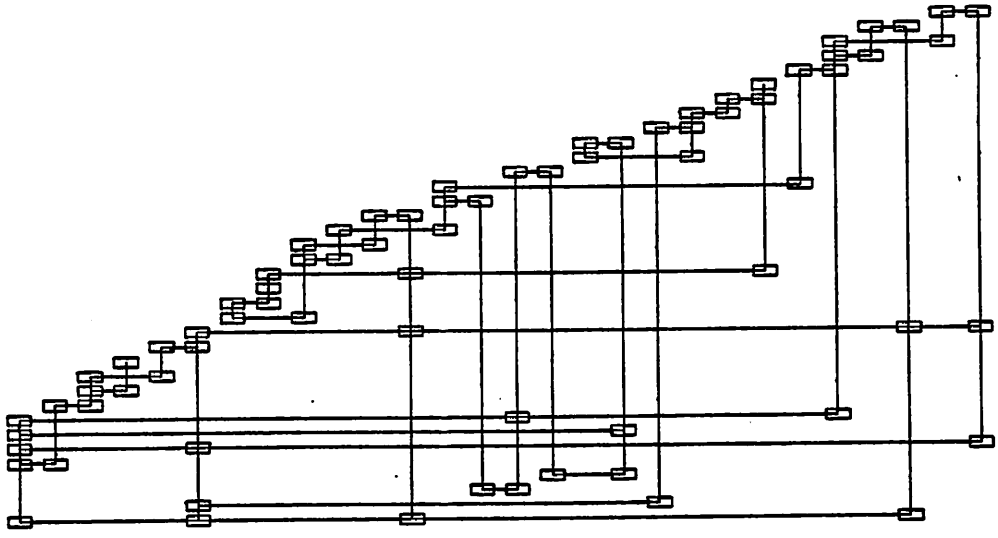
In this section we first give illustrative examples of compaction of various array structures using GENIE. Due to the complexity of the general folding problem, the only way to measure the performance of a new approach is to compare it with the best methods known previously. GENIE is compared with existing, specialized layout/folding programs.

The Gate-Matrix layout of the CMOS Domino circuit of Figure 4.2 is shown in Figure 4.8a. The folded Gate Matrix without any constraints is shown in Figure 4.8b. Constraining all the inputs to the left of the matrix and all the outputs to the right of the matrix the layout shown in Figure 4.8c was obtained. Measuring area as the number of rows multiplied by the number of columns and ignoring the area occupied by peripheral circuits, unconstrained folding reduced the area to 33% of the original as compared to 43% for the constrained case. Additional area is required for the routing of the inputs/outputs in the first case which can cause the total area, including routing, to be larger in the unconstrained result.

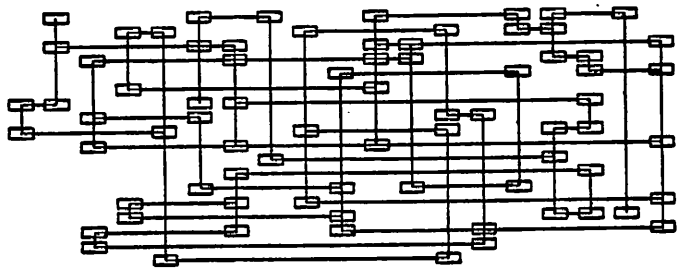
While the above example was for CMOS Domino logic, gate structures in static CMOS or static NMOS can be represented similarly in a multi-level matrix form. Gates can be two or more levels deep. Multi-level gates do complicate matters since more than one column may be required for each gate.

The unfolded and folded layouts of a large circuit are shown in Figure 4.9. There are a total of 647 single-transistor cells in the circuit. The area has been reduced to almost half (53%) of the original in this case. The final layout was obtained in approximately 14 minutes on a DECVAX 8600 running Berkeley UNIX[1] 4.3. A slightly larger layout was obtained in 8 minutes on the same

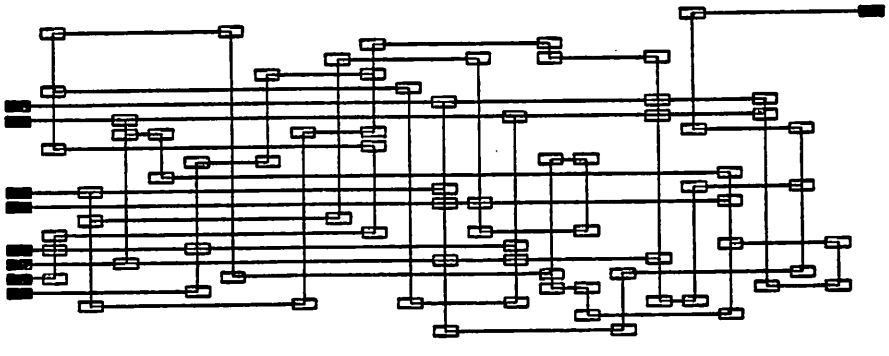
[1] UNIX is a Trademark of AT&T Bell Laboratories



(a) Layout of Fig. 4.1 in Domino Gate-Matrix, unfolded.



(b) After folding but without constraints.



(c) After folding with constraints.

Fig. 4.8. Example of constrained and unconstrained folding with GENIE

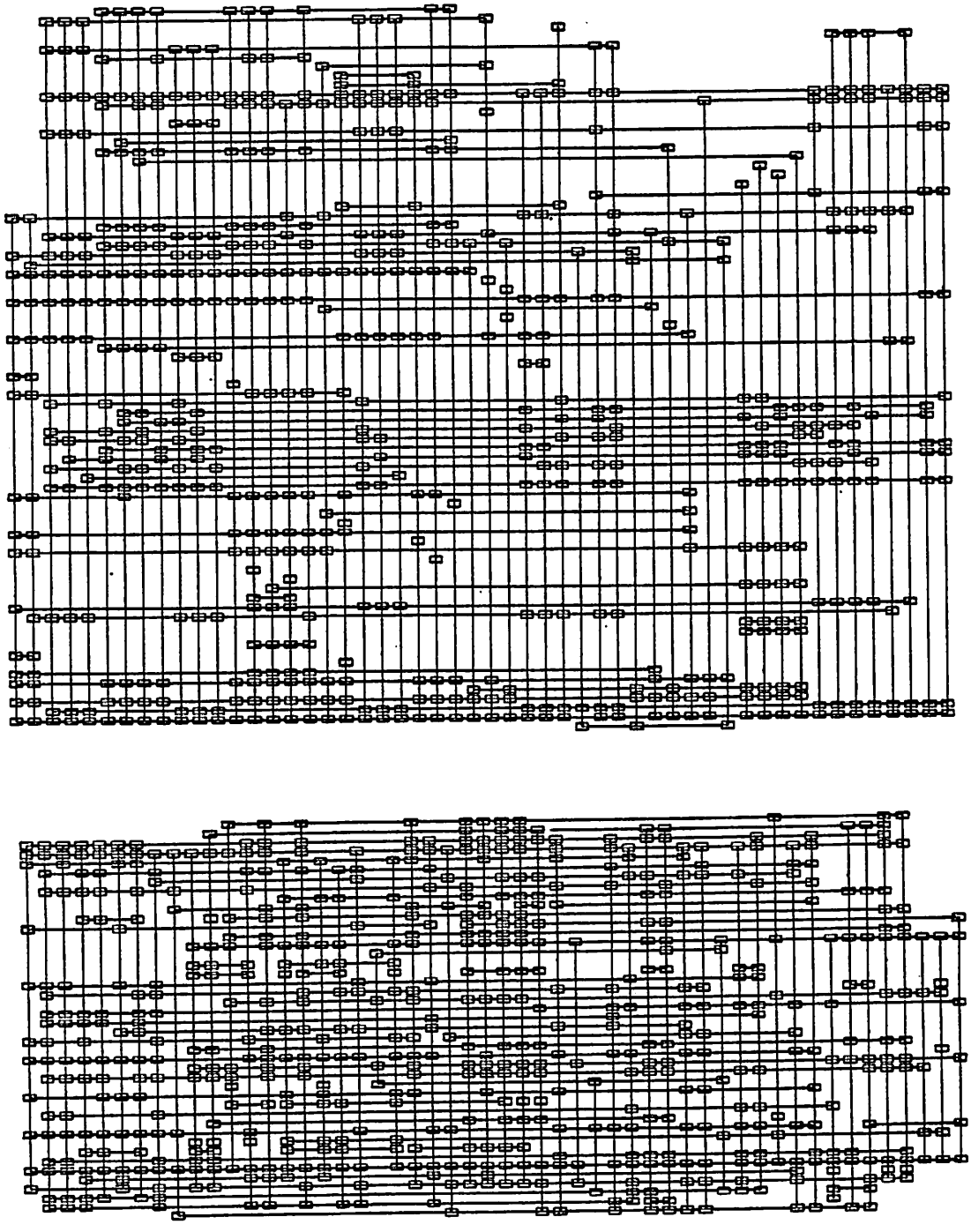


Fig. 4.9. Unfolded (top) and folded layouts of a large circuit

machine.

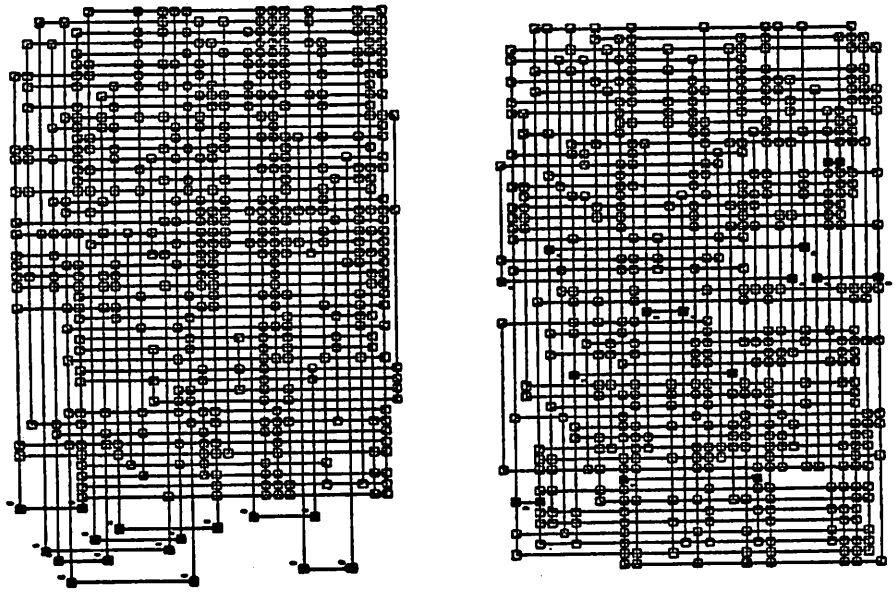
The combinational part of finite state machines is often implemented as a PLA. The flip flops which store the present state are typically latch transistors which are clocked and are usually added to the PLA later. So the inputs and outputs corresponding to the state variables have to be constrained to the same side of the PLA and also ordered to minimize routing. These constraints can often prove costly in terms of final area.

An alternative is to treat the latch transistors as part of the array itself with the state variables being intermediate inputs and outputs in a multi-level logic structure. The latch transistors are thus allowed to *float* into the array and no constraints are placed on the state variables. This is similar to an SLA approach. The area of an industrial FSM was reduced by 5% by GENIE using this method as opposed to the former method. In addition, there was a significant reduction in the length of the critical path through the array. The folded FSM using the above two methods is shown in Figure 4.10. The shaded cells represent the latch transistors of the flip-flops. In Figure 4.10a the array was folded without the latch transistors and they were later added. The result of letting the latches float into the array is shown in Figure 4.10b. The actual layout of the FSM is shown in Figure 4.10c.

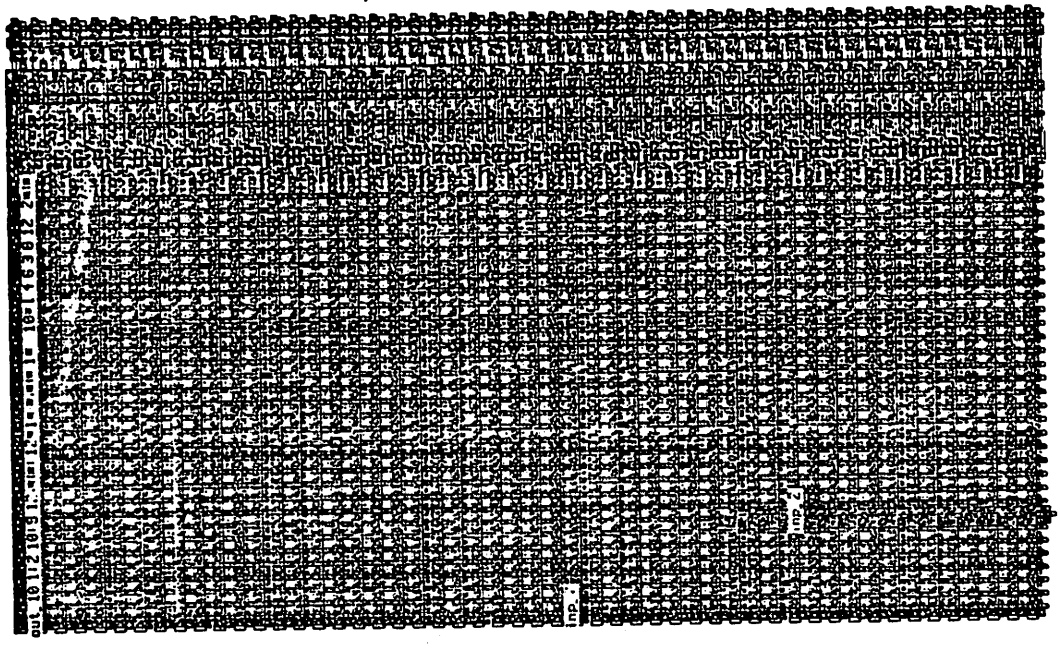
4.2.1. Implementation Details

GENIE is coded in about 5000 lines of *C* and runs in a VAX[2]-UNIX environment.

[2] VAX is a Trademark of the Digital Equipment Corporation



(a) Latches added after folding (b) Latches added before folding



(c) Actual layout
Fig. 4.10.

The output of the program is a folded connectivity matrix suitable for input to a context-based tiler for actual layout. Representations for both multiple and simply folded rows and columns are used and the representation matrix is independent of technology.

There are various folding options. The program can be used for any combination of simple, multiple or no row folding and simple, multiple or no column folding for both PLAs or multi-level logic blocks. Constraints on the inputs/outputs can be expressed in the input file using a simple left-right-both notation.

Typically, dependent on the layout tiler or the PLA generator being used there are constraints on the folding. The tiler we have used for generating Domino CMOS layouts, TINKER, is context-based and produces a layout in a three layer interconnect technology. TINKER requires simple column folding but allows multiple row folding of the intermediate input/output lines. All the primary inputs/outputs are necessarily at the edge of the array. For PLAs, the PANDA[mah84] PLA generator can be used. The style of PLA produced by PANDA is CMOS static with p-channel pull-ups as resistive loads.

The final aspect ratio of the folded array may be important to the designer. It has been found, especially in multi-level structures, that the folded area can vary to a large extent depending on the relative magnitudes of row and column folding. There is a trade-off between the desirable aspect ratio and possible folded area. Note that in Table 4.2 some folded multi-level circuits have been only row folded. In some cases that was because column folding was not possible but in the other cases it was found that only row folding gave a smaller final area as compared to equal weights for row and column folding. In MAT7, for example, when equal weights were given to the row and column gain/penalty parameters, the

folded result was 18 rows by 29 columns as compared to 16 rows by 31 columns for only row folding.

Most PLA folding programs do not interleave the AND and OR planes of the PLA whereas in multi-level logic there is almost no distinction between a series or a parallel tile: they can exist on the same row or column. GENIE can optionally interleave the AND and OR planes of a PLA or keep them separated. Interleaving may result in a smaller folded area but the layout generator used must allow it.

One of the features offered by GENIE is good control over aspect ratio using the multiplicative row-column decrease/increase parameters described in earlier. Their relative magnitude can be controlled by the user which alters the final aspect ratio.

For SLAs or static CMOS/NMOS multi-level blocks, where the cells may have varying widths, the length-width parameter described earlier is assigned a non zero value. The effect of this parameter is to minimize total length and total width instead of minimizing the number of rows or columns.

4.2.2. Comparisons with existing tools

GENIE has been compared with a number of existing tools for topological folding and compaction of array logic. TWIST[hofm85a,hofm85b] was developed at the University of California, Berkeley and is a program for the folding of multi-level connectivity matrices. TWIST can perform simple column folding and multiple row folding of intermediate inputs and signals can be constrained. The algorithms used by TWIST are similar to the ones presented in[demi83a].

A variety of constrained/unconstrained examples were run on TWIST and GENIE. The results are tabulated in Table 4.2 showing CPU time on a DECVAX

MATRIX	size nr*nc	Constraints	TWIST folded size	GENIE folded size	GENIE time (sec)	GENIE Area/ TWIST Area
MAT1	27*15	none	26*11	19*14	57	0.93
MAT2	22*42	none	19*22	18*18	76	0.77
	22*42	all inputs left	21*32	21*21	78	0.72
MAT3	30*60	none	29*59	30*54	364	0.94
MAT4	109*95	none	74*77	47*82	1029	0.67
MAT5	78*52	none	58*52	38*52	362	0.65
MAT6	85*64	none	66*61	58*41	1062	0.59
MAT7	47*31	none	28*31	16*31	312	0.57
MAT8	79*55	none	63*44	41*39	624	0.57
MAT9	47*55	none	30*53	19*44	452	0.52
MAT10	73*72	none	4*67	24*62	741	0.50

Table 4.2. Comparison of TWIST and GENIE.

8600, number of initial rows and columns, final number of rows and columns, initial and final areas with percentages. These examples are all considered to be with uniform cell sizes as TWIST has no special provision for non uniform cell dimensions. However, TWIST has additional internal constraints on the ordering of signals as it performs delay optimization. GENIE performs significantly better than TWIST and more so in the larger examples. Constraints are also handled better than TWIST. The CPU time taken for these examples is cost-effective. Area results as good as those obtained by TWIST can be achieved in much less time; the results here are the best we obtained disregarding CPU time as a factor. In all these examples simple column folding and multiple row folding were allowed.

GENIE is primarily intended for uniform/non uniform cell multi-level matrix structures. However, it is interesting to compare GENIE with a PLA folding program like PLEASURE[demi83a,demi83b]. A number of examples were run on PLEASURE and the results are tabulated in Table 4.3. All the examples were run for multiple row and column folding. GENIE produces better results than PLEASURE; more so in

PLA	size nr*(2*ni+no)	Constraints	PLEASURE folded size	GENIE folded size	GENIE time (sec)	GENIE Area/ PLEASURE
PLA1	20*(26+25)	none	14*33	17*22	50	0.80
PLA2	30*(16+31)	none	29*21	29*19	364	0.90
	30*(16+31)	some inputs top outputs bottom	29*35	29*31	374	0.88
PLA3	47*(24+12)	none	47*28	47*28	615	1.0
	47*(24+12)	all inputs top	47*32	47*30	626	0.94
PLA4	52*(54+28)	none	52*56	52*44	362	0.78
	52*(54+28)	all inputs top	52*69	52*58	376	0.84
PLA5	34*(22+2)	none	34*18	34*17	134	0.94
	34*(22+2)	some inputs top	34*23	34*20	143	0.87
PLA6	211*(64+20)	none	211*66	211*47	1745	0.71
PLA7	100*(38+16)	none	100*22	100*20	221	0.91
PLA9	119*(78+8)	none	119*42	119*29	498	0.69
PLA10	35*(68+14)	none	21*23	35*11	280	0.79
PLA11	128*(44+16)	none	128*28	128*20	129	0.71
PLA12	20*(22+5)	none	13*21	20*11	11	0.81

Table 4.3. Comparison of PLEASURE and GENIE for PLA folding.

the constrained cases which are more likely in real chip designs. Another advantage in using GENIE rather than a folding program like PLEASURE for PLA-based FSM's is that we can allow the latch transistors which store the initial state to float into the array and achieve further compaction as described earlier.

GENIE has also been compared to the PLA folder described in [moor85] which uses a rule based hill climbing technique. The results of the comparison are tabulated in Table 4.4. GENIE produces equal or better results on all the examples under the same conditions.

Simulated annealing asymptotically approaches the global optimum of the configuration space with the increase in the number of states generated per temperature point[rome85]. Various solutions can thus be obtained for a given example depending on the inner loop criterion and temperature profile. GENIE can be

PLA	size nr*(ni+no)	MO-GS folded size	MO-GS time (sec)	GENIE folded size	GENIE time (sec)	GENIE Area/ MO-GS Area
DK27	10*(9+9)	110	2.3	100	6.0	0.91
BENCH2	57*(8+18)	918	8.76	855	62	0.93
	57*(8+18)	918	8.76	840	161	0.91
SEX	21*(14+21)	266	3.33	231	11.6	0.86
	21*(14+21)	266	3.33	210	41.4	0.78
B12	42*(15+9)	525	7.06	525	61	1.0
B3	211*(32+20)	6930	65.12	6330	1745	0.91
P1	57*(8+18)	918	9.05	855	63	0.93
SHIFT	100*(19+16)	1881	10.11	1400	221	0.74
B4	54*(33+23)	1600	10.35	1404	173	0.87
LUC	26*(8+27)	696	4.72	696	47	1.0
P3	40*(8+14)	532	5.92	520	34	0.97
SPAM1	18*(9+14)	221	2.99	198	9	0.89
ALCOM	40*(15+38)	672	8.48	500	21	0.74
CLPL	20*(11+5)	168	4.28	140	11	0.83
SIGNET	119*(39+8)	2895	50.39	2023	498	0.69
MISJ	35*(34+14)	450	7.62	11.1	280	0.62
TS10	128*(22+16)	2944	6.60	1792	129	0.60

Table 4.4. Comparison of GENIE and the MO-GS Folder.

used for short or long runs depending on the CPU time the user wants to spend. Plots of CPU time versus quality of solution for eight examples are shown in Figure 4.11 — the first plot gives the solution quality/CPU time results for four PLAs. and the second plot for four MLMs. The # (hash) points on the eight plots correspond to the solution/CPU time for PLEASURE and TWIST. The plots indicate that GENIE produces solutions as good as PLEASURE and TWIST taking more or less the same order of CPU time. Further and further improvement requires more and more time as illustrated graphically by the plots. These are typical results and they span a large range of PLA and MLM sizes. The details on the sizes of the PLAs and MLMs can be found in Tables 4.2. 4.3 and 4.4. Similar plots were obtained versus the MO-GS folder.

The entire control logic of the Berkeley SOAR microprocessor was synthesised in two level and multi-level form using GENIE. The results are encouraging, both

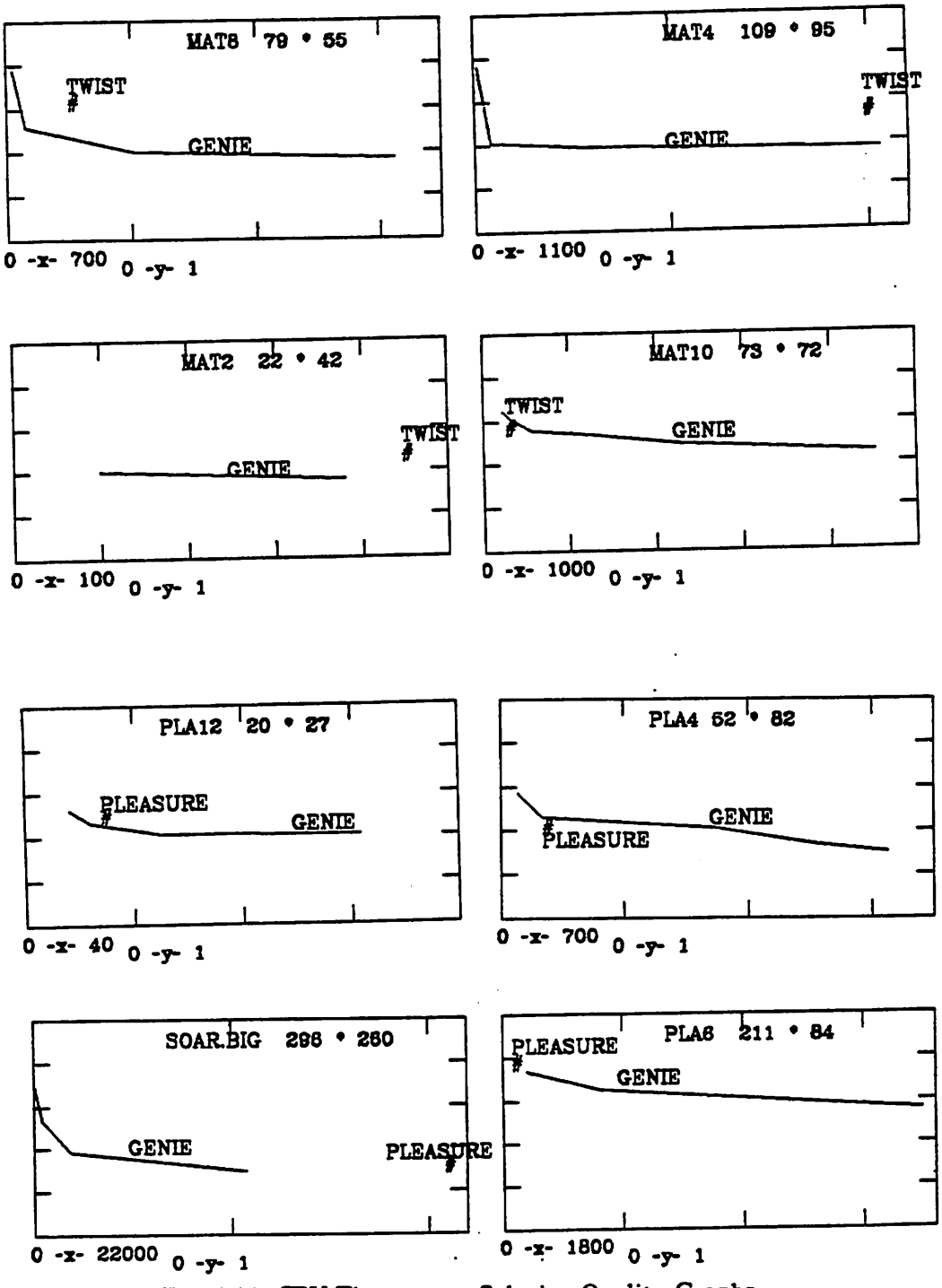


Fig. 4.11 CPU Time versus Solution Quality Graphs

for the PLA and multi-level cases and are summarized in Table 4.4. Note that PLEASURE actually takes twice the time to fold the large PLA and achieves a poorer result. The explanation for this is that the operations required to find the cost of the configuration in GENIE are purely incremental, and are independent, to a first order, of the size of the array. Hence for very large examples, simulated annealing is often faster than heuristic algorithms. The CPU time versus solution quality graph for this large PLA is shown in Figure 4.11 and corresponds to the curve marked SOAR.BIG.

Most proposed schemes for gate matrix layout using graph theoretic techniques (e.g. [wing85]) cannot handle the practical constraints of differing size transistors and fixed input/output terminals. In GENIE bounded column, ordering and adjacency constraints can be imposed on the signals. For Weinberger arrays, total net length can be minimized during compaction unlike previous approaches.

FORM	size nr*nc	Constraints	folded size	Area Reduction factor	GENIE time (sec)
PLA/GENIE	260*296	none	106*215	3.43	10740
PLA/PLEASURE	260*296	none	114*217	3.15	21100
multi-level/GENIE	644*561	none	158*248	9.22	12760

Table 4.4. Performance for both multi-level and two-level implementation of CMOS SOAR control logic.

4.3. Storage/logic Array Compaction

Storage/Logic Arrays were first described in 1975[pati75] and later extended[pati79]. They are regular structures derived from the Programmable Logic Array (PLA)[flei75]. Unlike PLAs+2 they have the ability to embed storage functions within the combinational logic array.

SLAs can be constructed from tile sets designed for particular technologies. Though SLAs have been built in various technologies the CMOS SLAs appear to be the most popular. CMOS SLA elements include a single inverter, a double inverter, a simple NAND gate latch and a pass transistor[smit82]. Schottky diodes can be used as the combinational elements. Each of these cells is represented as a single character to the designer. In much the same way as the personality matrix of the PLA can be manipulated symbolically, the characters representing the SLA elements can be arranged. Because the layout is very similar to a PLA, it is possible to employ similar folding and splitting algorithms[demi83ai,demi83b] to optimize SLA area. But if some of the tiles are quite complex and large (e.g. a D flip-flop) area utilization suffers dramatically if PLA optimization techniques are applied since then the minimum cell pitch is constrained by the largest cell in the tile set. An alternative approach is to allow cells which are the multiple of some fundamental dimension. However if this is done special "blank" and "bender" cells must be created to route signals. So far, SLA designs have been area efficient only while implementing circuits with a high ratio of combinational logic to active elements making them more and more like PLAs with external synchronizing logic. Their *embeddedness* has thus been lost[hofm85a,hofm85b].

The general algorithm described in Section 3 can be used to minimize the area of a SLA under various constraints. Since the cells are of varying sizes the height/length of the array is minimized and not just the row/column cardinality.

Also, user-specified critical signals can be made short while minimizing area. This may be vitally necessary in synchronous system design. Since the algorithm is technology independent, it can be used irrespective of the complexity of cells in the SLA.

4.3.1. Modifications for SLA compaction

A relatively simple approach was adopted for gate matrices in defining column width (row height) as the width (height) of the widest (tallest) cell in the column (row) and minimizing for the sum of column widths and row heights. However this approach assumes that cells on a row (column) cannot encroach on adjacent rows (columns) whereas in reality this encroachment is allowed and can reduce area.

This may lead to a situation depicted in Figure 4.12a after the annealing. By interchanging columns A and B in the array of Figure 4.12a and allowing cells to encroach on neighboring columns/rows without overlapping other cells, and still keeping all nets straight, we can reduce the area considerably as shown in Figure 4.12b.

Therefore, the cost function should calculate the area of a particular configuration in a more realistic way. Also, it is important for computational efficiency that this calculation be incremental and restricted to the nets being displaced/interchanged.

At every stage the necessary width (height) required for every column (row) is calculated taking into account the cells on adjacent columns (rows). The centers of the cells are assumed to be aligned. If a displacement increases the necessary width of a column or height of a row it is penalized. On the other hand, if a displacement/interchange decreases the width of a row a gain is accrued.

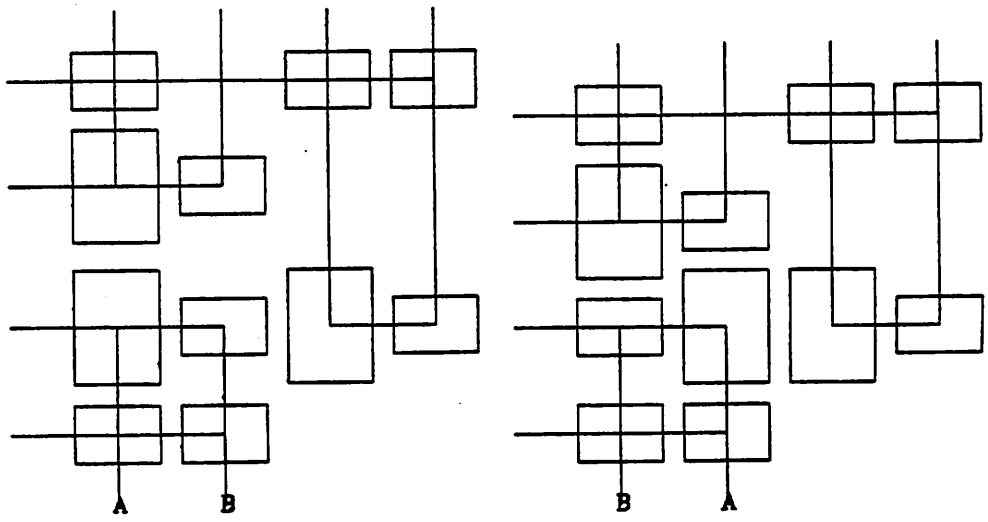


Fig. 4.12 (a) Folding disallowing encroachment (b) allowing encroachment

Displacements may cause a row or column to become empty of cells. This is the special case of the height or width becoming zero and is treated as such.

The total length of the nets is a part of the cost function like before. The associated cost is the length times the net weight. Critical nets, which must be kept short, can be weighted heavily so as to minimize their length in the final layout. Nets may be kept short even at the expense of increased area if their assigned weight is large enough. Thus the critical path through the array can be minimized.

Compound cells introduce another complication. All the constituent cells of a compound cell must eventually lie adjacent to each other in the folded array. But if the constituent cells are connected to different nets (e.g. the flip-flop cells in Figure 4.12a, some may be displaced so they are no longer adjacent to each other. Two approaches may be taken to solve this problem: the nets can be displaced/interchanged only simultaneously or we can allow their adjacency constraint to be violated by single displacements but penalize any violation. The

second approach invariably yields better results since it enlarges the exploration space. A large enough penalty ensures that the eventual configuration is acceptable.

Post processing after the annealing to remove overlap is similar except that if more than one feasible row exists on which the present overlapped row can be placed, the row with maximum height is chosen so as to minimize area increase. If no such rows exist a new row is created next to the present row and the net is placed on it and locked as before. Similarly, if more than one feasible column exists the widest is chosen. If none exists a new column is added.

4.3.2. Constraints during the annealing

Input and output terminals can be constrained to any edge of the array using pads as described earlier. Multiple/compound cells which may be made up of smaller constituent cells also introduce adjacency constraints for the annealing phase. The user may require a structured layout style within the array[pati85] e.g. all the flip-flops lying close to each other with combinational logic surrounding them. These constraints take the form of net groups or *partitions* and nets are displaced or interchanged within partitions only, which ensures that all nets in a partition neighbor each other in the optimized layout. Ordering constraints are required in the case of directional cells like inverters and buffers. Net interchanges

and displacements are such that they do not violate the required ordering.

4.3.3. Aspect Ratio sizing

The aspect ratio of the folded array may not be desirable in a SLA layout. Two approaches to aspect ratio sizing can be taken. The first involves partitioning the initial SLA into smaller parts with minimal interconnections between them. The sub-arrays can be optimized separately with constraints on the interconnecting signals. Heuristic methods[fidu82,kern70] or a simulated annealing based partitioning scheme can be used to find the optimal set of partitions.

The second approach involves modifying the cost function in the annealing step described to control the relative magnitude of row and column folding so as to achieve a desirable aspect ratio. The penalty for reducing/increasing row height (column width) can be made higher than column width (row height) if a short, squat (tall, thin) folded array is required. If this is done more rows (columns) are folded at the expense of column (row) folds. In the extreme case only row folding or only column folding can be allowed.

4.3.4. Examples

The symbolic folded layout of a representative SLA program illustrating the use of symbols which has been reproduced from [smit82] is shown in Figure 4.13a. The asynchronous cells include: 1) a set/reset latch for the memory element; 2) an inverter; 3) 1,0,S,R and + combinational logic cells; 4) row and column pullup cells; and 5) miscellaneous cells including *,#, and blank cells. The # and * cells are ohmic contacts between rows and columns and are used to hardwire signals from one position to another. Row and column interconnects are

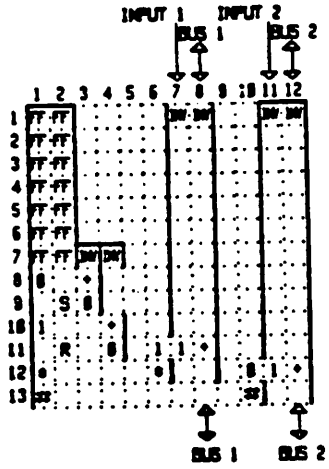
assumed to be always present between cells unless the programmer deletes them, creating a row or column break, which is indicated by the solid lines in Figure 4.13a. The single flip-flop FF occupies 2 columns and 7 rows and the rest of the cells occupy one row by one column. Neglecting the two blank columns, which are left for row pullup cells, the area of this folded layout is 130 units.

The unfolded connectivity matrix representation of the SLA with the size of the flip-flop indicated is shown in Figure 4.13b. This symbolic representation is the input to the optimization program and has been derived from the folded array placing every net on a unique row or column.

The result of running our optimization program on the unfolded symbolic representation of Figure 4.13b is shown in Figure 4.13c after expanding the flip-flop to seven rows. Simple row and column folding was allowed. The folded layout was obtained in 3.1 seconds on a DECVAX 8600 running Berkeley UNIX[3] 4.3.

The area of this layout is 98 units, 25% smaller than the hand folded array of Figure 4.13a. For large, sparse examples, or examples with widely varying cell dimensions area gains over hand optimization will be larger. A larger example from an industrial source is shown in a optimized form in Figure 4.14. As can be seen there is wide variation in the sizes of the constituent cells. It is interesting to note that GENIE clustered big cells together on rows to minimize area. GENIE actually "rediscovered" the structure present in the netlist. Multiple folding of intermediate inputs was allowed. The folded layout is less than a third the size of the original. Multiple folding of intermediate inputs was allowed. It was obtained in

[3] UNIX is a Trademark of AT&T Bell Laboratories



(a) SLA realization of a state machine from [smit82]

unfolded 19 10

```
     abcdefghij
r1   ........II
r2   .....II...
r3   ...I......
r4   ..I.......
r5   FF........
r6   0.+.......
r7   .S0.......
r8   1..+......
r9   .R.0......
r10  *...*.....
r11  $......$..
r12  ....11+...
r13  .......01+

size F (7,1)
```

(b) In unfolded form

folded 14 7

```
     abcdhij
r11  #...#II  r1(6)
r4   ..I.01+ r13(5)
r6   0.+....
r7   .S0....
r8   1..+...
r9   .R.0...
r5a  FF..11+
r5b  FF.....
r5c  FF.....
r5d  FF.....
r5e  FF.....
r5f  FF..... r12(5)
r5g  FF.I...  r3(4)
r10  *...*II  r2(6)
       efg
      (7) (7) (7)
```

(c) After automated optimization

Fig. 4.13

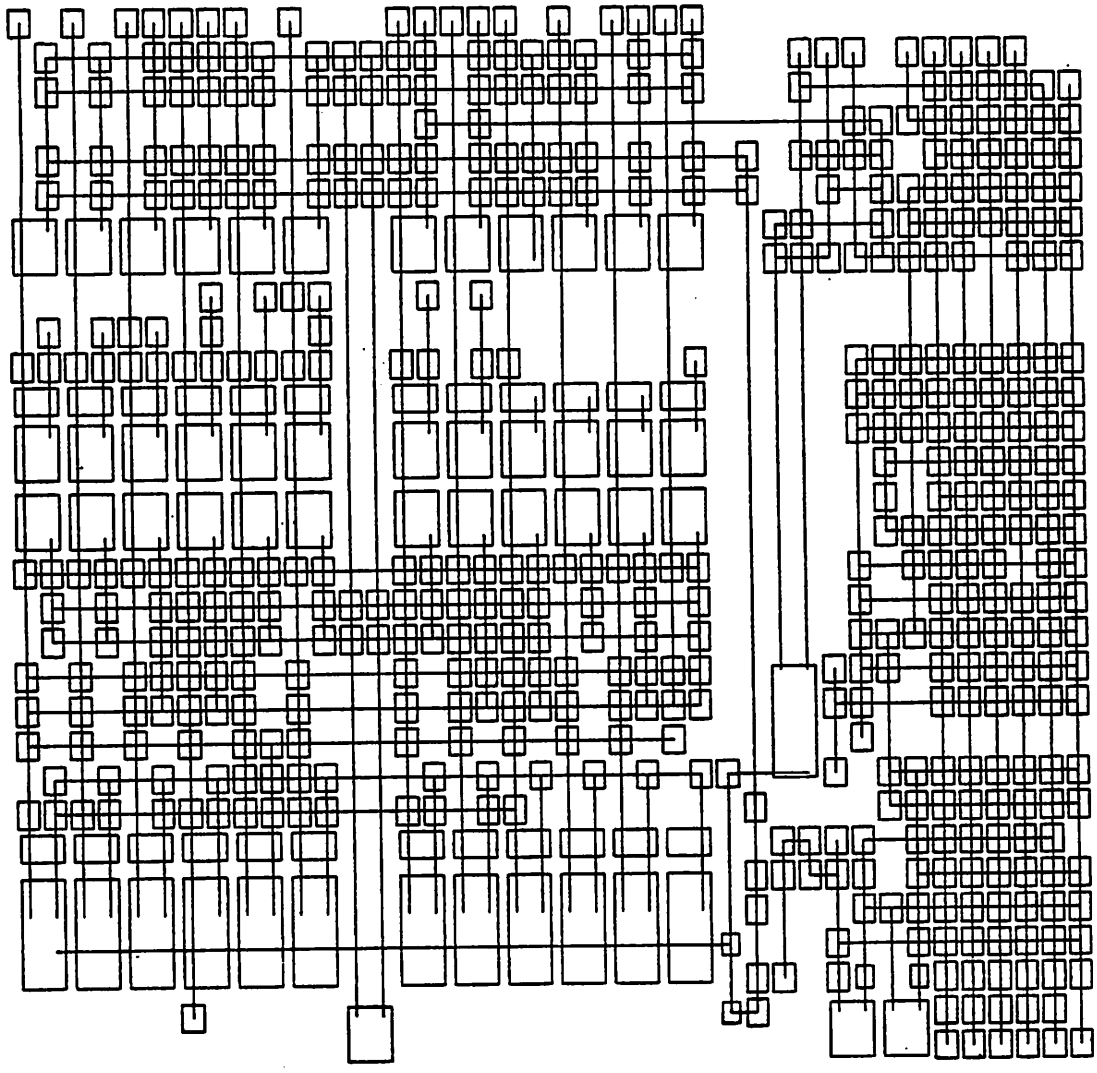


Fig. 4.14 Folded SLA

approximately 14 minutes on a DECVAX 8600.

4.4. Conclusions

In this chapter a new technique for generalized array optimization using simulated annealing was presented. GENIE has proved to be a useful tool in automatic multi-level combinational/sequential logic synthesis in various technologies because of its ability to handle practical constraints like non uniform device sizes and fixed input/output pads and because of the high quality of the final results.

It has been shown that a simulated annealing technique can be used to emulate folding of array structures ranging from highly-regular PLAs, Weinberger arrays, SLA's (Storage/Logic Arrays) to multi-level gate matrices. In all cases, GENIE outperformed our best program to date. In particular, for constrained, two-level logic GENIE over 30% reductions were achieved compared with PLEASURE. In the case of SLAs, GENIE is the first tool to provide a high-quality embedding of logic.

It has also been shown that the CPU time requirements of simulated annealing based algorithms can be reduced to a manageable level, while still producing solutions of higher quality than heuristic algorithms.

A modified algorithm for one-dimensional Gate Matrix and Weinberger Array folding, which is more cpu time efficient than the general two dimensional folding algorithm presented in this chapter, while producing equally good results, is described in the next chapter.

CHAPTER 5

Weinberger Array and Gate Matrix Layout

Multistage logic circuits in various technologies and design styles can be represented symbolically as Gate Matrices[lope81] or Weinberger Arrays[wein67]. The gate matrix has proved to be an effective layout style for implementing multi-level combinational functions in CMOS technology. In contrast, Weinberger arrays produce area efficient layouts for single MOS (e.g. NMOS,PMOS) technologies.

In the gate matrix style the p-transistors and n-transistors are placed in different halves of the matrix. The transistors are connected by lines which form the rows of the matrix. Various algorithms[ohts79,wing82,mlyu85] have been proposed to find an ordering of the signals/columns of the matrix to obtain minimum row cardinality and an area efficient layout. However, none of these methods provide for *differing size transistors*, which means that, though the row cardinality may be optimally small, the height/area of the matrix may not be the minimum possible.

Most proposed methods[wing82,wing85] lack the ability to handle practical constraints like fixed input and output terminals. In [ohts79] all the inputs are restricted to the left of the matrix and all the outputs to the right. Also, good control over the aspect ratio of the gate matrix[ohts79,wing82,wing85] is not provided by previous techniques.

Weinberger arrays are one-dimensional. Optimization algorithms to obtain the best packing on a one dimensional interval have been investigated[yosh75][ohts79] [asan82][mlyu85]. However, no previous approach provides for differing size transistors like in the gate matrix case. Provisions for

minimizing total net length or optimizing circuits with arbitrarily *complex gates* are non existent.

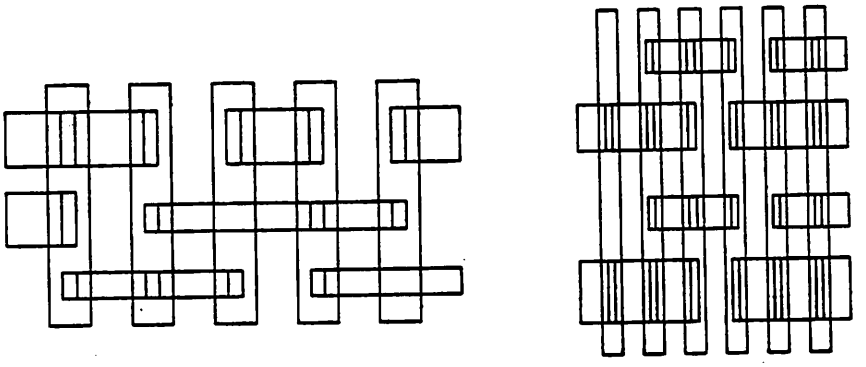
In this chapter we present simulated-annealing-based algorithms for gate matrix and Weinberger array layout which effectively solve all the above problems[deva86b]. These algorithms minimize the area of a Weinberger array taking into account arbitrary user-specified constraints on *any* terminal. *Bounded column, ordering,* or *adjacency constraints* can be placed on the input and output terminals of a gate matrix. Since transistors can be varying sizes the height of the array/matrix which is proportional to the area is minimized and *not* just the row cardinality. These techniques can also be used for partitioning the array/matrix into smaller sub-matrices to achieve a desirable aspect ratio while minimizing interconnections. Each of the sub-matrices can then be optimized. User-specified critical signals can be made short or the total net length can be minimized while minimizing area. The algorithms are technology independent since they operate on a symbolic layout and can be used irrespective of the complexity of logic gates in the circuit. Pass transistor structures are supported. A context based tiler then translates the symbolic layout into CIF (Caltech Intermediate Format). Results obtained are uniformly good over a wide range of examples.

The layout algorithm for Weinberger arrays is described in the following section. Modifications required for gate matrix layout are described in Section 5.2 and Section 5.3 contains illustrative examples.

5.1. Weinberger Array Layout Optimization

A typical Weinberger array layout is shown in Figure 5.1a. The two basic entities of interest are the gate and signal. Note that the gate may occupy one or more column and more than one signal may occupy a row in the compacted Weinberger array. The objective is to find an ordering of gates which minimizes the overall area. Given a ordering of gates the *range* of signals can be determined. Signals can exist on the same row if their ranges do not overlap. The ordering must be such that signals can be maximally folded on rows. Constraining a signal to either or both edges of the array results in expanding its range.

The input to the program is a gate net-list. A gate may be of the simple NAND or NOR type or may be a more complex AND-OR-INVERT structure. The widths and lengths of the constituent transistors in the gates may vary widely. The program generates a symbolic layout area-optimized under the various constraints which is suitable for input to a tiling program, like TINKER[hofm85a,hofm85b], a context based tiler which produces the actual layout in CIF.



(a) Weinberger Array (b) Gate Matrix

Fig. 5.1

An example of an input to the program is shown in Figure 5.2a. It is a list of logic gates interconnected by nets. Figure 5.2b is the schematic of the gates described by Figure 5.2a in transistor form. One of the logic gates is a two-level structure and may require two columns to be realizable. A pass transistor also exists in the input description. The ordering of input signals of complex gates determines the number of columns the gate needs to be realized (Section 5.1.5).

Pass transistors are handled by representing the pass transistor as a simple gate with two input signals attached to the source and gate, and an output signal attached to the drain as illustrated in Figure 5.2a. The gate net-list is technology

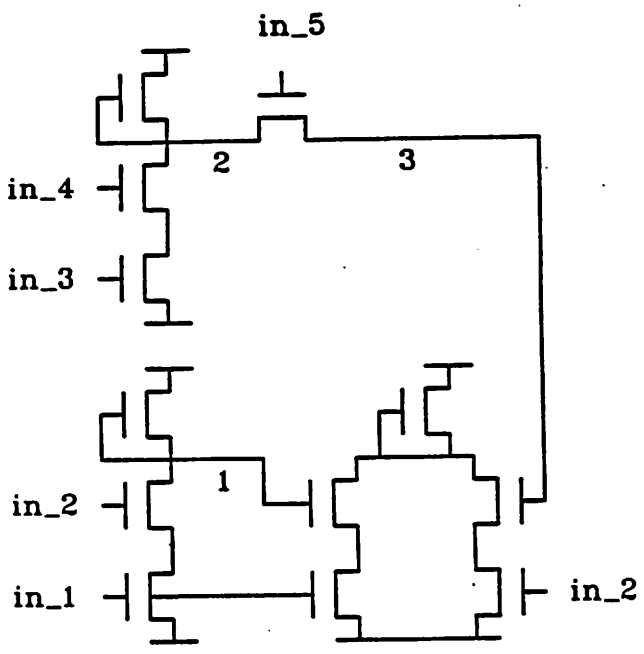
```
(o 1
 (s in_1 in_2 ));

(o 2
 (s in_3 in_4 ));

(o 3
 (k 2 in_5));

(o out
 (p
  (s 1 in_1 )
  (s 3 in_2 )
 ));
```

(a) Input gate net-list



(b) Transistor schematic

Fig. 5.2

independent, however information about the technology and design style is necessary for deriving the transistor schematic and array structure of the net-list.

The algorithm consists of two annealing stages; each with an associated cost function. New states are randomly generated and their acceptance is governed by the acceptance criterion described in the previous section. Briefly these steps are:

```
Order_Gates();
Order_Rows();
```

In the first stage the gates are ordered in such a fashion as to minimize the maximum column height while satisfying the specified constraints. In the second stage the best possible arrangement of rows is found so as to minimize the total number of columns required to realize the gates. This second stage is not necessary when all the gates are *simple* or only one level deep as assumed in [asan82]. The two stages can be merged together into a single annealing stage or performed sequentially. These two steps are described in detail in the remainder of the section.

5.1.1. Ordering the gates

The program begins with an initial arbitrary ordering of gates. The objective of this stage is to reduce the maximum of the heights of the columns to as great an extent as possible, a MIN-MAX problem. The column height is the sum of the heights of the transistors existing in the column plus the bus-through signal row widths. All the primary inputs and outputs and signals which are constrained to be at any edge of the matrix are terminated by *pads*. Pads are restricted to the periphery of the matrix; they are treated as immovable gates.

5.1.2. Generating New States:

A new state is generated in this stage of annealing process by exchanging two fundamental units, in this case, gates. Pads can be left-pads or right-pads depending on whether they are constrained to the left or the right of the array. A particular signal may have both left and right pads if necessary as in the case of bussed through input signals in connectivity matrices. Pads cannot be interchanged with each other or gates in this stage.

The selection of new states is based on the following considerations:

(1) Two random numbers between one and the total number of gates are generated.

(2) If the range limiter's condition is satisfied then an attempt is made to interchange the gates represented by the two numbers.

A range limiter[sech84] limits the range of interchange of a gate. The range limiter is used because in the latter stages of the annealing the interchange of two gates has very little chance of being accepted unless it is very local. So to generate states which have high probability of being accepted the range of possible interchange of a gate with another gate is gradually reduced from the total number of gates at the beginning to only allow neighboring gate interchanges as the temperature approaches zero.

5.1.3. The Cost Function:

The cost function is the key to any algorithm using simulated annealing. It is crucial that it be truly representative of the optimization problem.

The most important part is the maximum column height for the given ordering of gates. Constraints modify the calculation of this maximum height. The

total length of the signals is also a part of the cost function, though less important. The length of a signal is defined as the number of gates the signal spans. The associated cost is the length times the net weight. Critical nets, which must be kept short, can be weighted heavily so as to minimize their length in the final layout. Thus total net length or selected net's lengths can be minimized while minimizing area.

The procedure to find the cost of a configuration is shown below.

```
procedure find_cost_of_configuration()
{

 cost = 0;
 column_heights = 0;
 forall( signals ) {
     l = infinity;
     r = 0;
     forall( gates attached to signal ) {
       l = MIN(gate_column,l);
       r = MAX(gate_column,r);
     }
     if( signal is constrained to be on the left )
       l = leftmost_column_num;
     if( signal is constrained to be on the right )
       r = rightmost_column_num;
     cost = cost + ( r - l ) * signal_length_weight;
     for( col = l; col <= r ; col = col + 1 )
       column_heights[col] = column_heights[col] + signal_width;
 }
 forall( columns )
    max_column_height = MAX(column_heights);
 cost = cost + max_column_height * param1;
 return(cost);
}
```

The procedure is essentially calculating the maximum *density* of the signals at any particular column with weights on signal widths. The width of a signal is the width of the widest transistor on it. Constrained signals have to extend to the edges of the matrix. The relative magnitudes of *param1* and *signal_length_weight* are important in the case of a tradeoff between making the critical nets short at the expense of increased area. Typically the weight of non critical signals is zero

and highly critical signals can have weights such that they are made short even with a possible area penalty.

5.1.4. Stopping/Inner Loop Criterion and Temperature Profile

The stopping/inner loop criteria and temperature profiles used are similar to those described in Section 4.1.6 and 4.1.7.

5.1.5. Ordering the rows

After the first stage, an optimal permutation of gates would have been found which minimizes the height of the matrix. In the second stage the nets are first assigned tracks on rows using the Left Edge Algorithm (LEA)[hash71]. In the LEA nets are ordered according to increasing left edge x coordinates and routed in this order. This is similar to the channel routing problem only no vertical constraints exist between nets which guarantees that the LEA can route the nets in a number of rows equal to the maximum density at all columns. Then the rows and the signals on them are permuted such that the gates are realizable in a minimum number of columns. As mentioned before single level gates like NAND and NOR require only one column to be realized. But two and higher level structures may require additional columns if the ordering of the signals to the gate is not of a particular nature.

Two 2-level gate structures and the ordering of signals which allows realization in a single column in our layout style with three layers of interconnect are illustrated in Figure 5.3. This layout style has two layers of metal interconnect and a polysilicon layer and uses the basic transistor tiles shown in Figure 4.3. The valid permutations have been expressed in the form of compact *rules*. Each rule has an associated number called the *column magnitude* which is equal to the

maximum number of columns required to realize the gate provided the rule is satisfied. As an example the rule

{ permutable(a permutable(b c)) . 1 }

means the following orders are valid for a single column realization of the gate:

(a b c) (a c b) (b c a) (c b a)

The ordering (b a c) would require more than one column — using the tiles in Figure 4.3 all the required connections cannot be made by abutment in a single column.

A single rule may not suffice to span all the whole range of valid orders in the case of deep structures which can be realized over a range of columns depending on the ordering of the signals. In that case there will be two or more rules with different column magnitudes. Rules are *stronger* than other rules for the

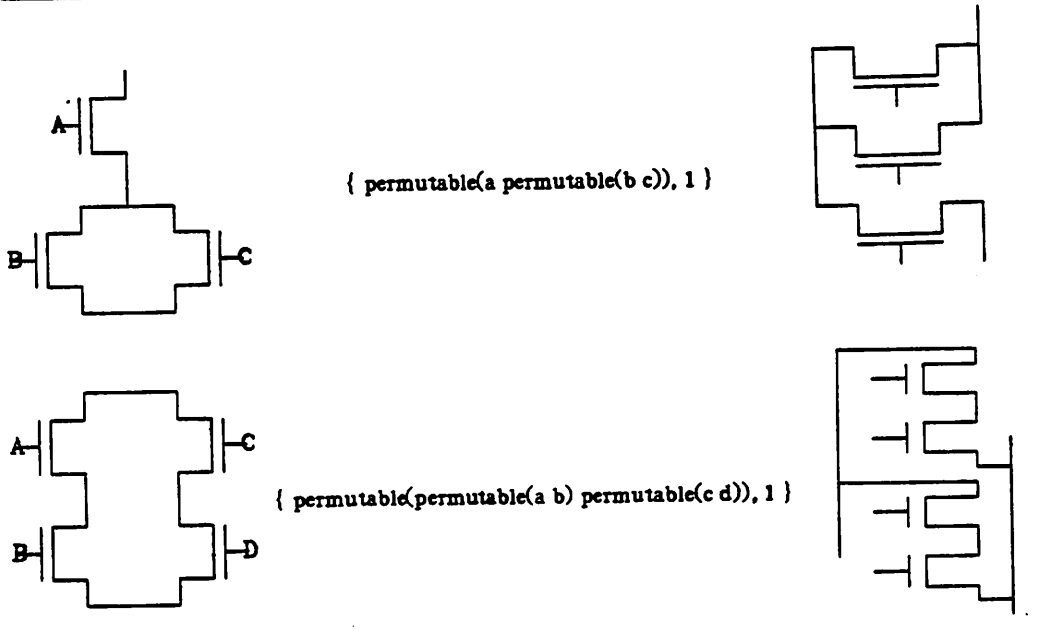


Fig. 5.3 2-level gate structures

same gate structure if their column magnitude is smaller.

The valid permutations of signals for a large complex gate to be realized in 1,2 and 3 columns are shown in Figure 5.4. Since the depth (or level) of the gate is 4 it can always be realized in 4 columns.

This annealing stage has a different cost function which computes the number of columns required at any stage given the ordering of rows. It does this by checking the applicability of the rules to every complex gate. Simple gates are *not* considered as they can always be realized in a single column.

For every complex gate the order of input signals is found. This order is compared against the set of rules for the gate and the *strongest satisfied rule* decides the number of columns required for the gate. The cost of the configuration is proportional to the number of columns or the total width of the columns if they are of varying widths.

The annealing process in this stage is similar to the first. New states are generated by exchanging two fundamental units - in this case rows, as opposed to

{ perm(A perm(B C) perm(D E) perm(F G)), 1 }

{ perm(perm(A B C) perm(D E) perm(F G)), 2 }

{ perm(perm(A B C D E) perm(F G)), 3 }

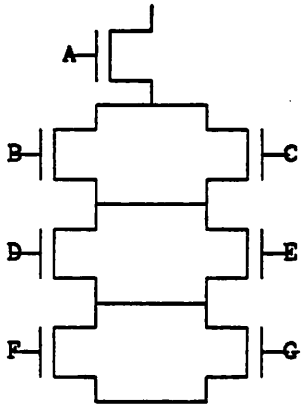


Fig. 5.4 Rules for a complex gate

gates. While exchanging rows all the signals on the corresponding rows are interchanged to maintain the row folding. The temperature profile and stopping criteria are identical to the first stage.

5.1.6. Aspect Ratio sizing

The aspect ratio of the folded array may not be desirable. The first stage of annealing can be modified slightly to find partitions of gates with minimal interconnections between them. These partitions, represented as sub-matrices can then be folded separately with constraints on the interconnecting signals.

The partitioning of the array proceeds as follows. Typically, it would be desirable to have all the sub-matrices of the same size. A range of sizes for each sub-matrix is found corresponding to the tolerance of sizes specified. Initially the gates are assigned to partitions arbitrarily without regard to interconnections and sizes. The configuration is then annealed.

The cost function in this case consists of two components. The first part corresponds to the number of interconnecting signals required between the partitions. The second part is the penalty for too small or too large a sub-matrix size. If the size of a sub matrix, measured as unfolded area, is not within the range of tolerable sizes a *penalty* is assessed which is proportional to the increment or decrement. This penalty is added to the cost function. This is vitally necessary as otherwise the best possible configuration with regard to just minimizing interconnecting nets is all the gates in one partition which defeats our purpose. This approach however allows intermediate configurations to have sub-matrix sizes outside the tolerable range while striving for an optimal solution.

5.1.7. Merging the stages

The two stages described earlier in the section can be merged together into a single annealing stage where the optimum ordering of rows and columns to minimize area is found simultaneously while satisfying the constraints imposed on the signals. This corresponds to treating the Weinberger array layout problem as a 2-dimensional placement problem rather than as two 1-dimensional ordering problems like most previous approaches[ohts79,asan82].

This merging is achieved by interchanging gates *and* rows during the annealing and using a cost function which is a composite of the cost functions described. For any particular ordering of gates total height of the rows is calculated (as in the first stage) and for any particular ordering of rows the number of required columns to realize the gates is calculated (as in the second stage). This determines the area/cost of the configuration.

The merging of the two stages produces results better than the unmerged stages especially for circuits with a high percentage of complex gates but with increases in cpu time. The increase is due to the fact that typically more states have to be generated at each temperature point. The two approaches are compared in Table 5.2.

5.1.8. Dealing with large fanin gates

A basic limitation to the minimum row cardinality possible is the maximum fanin of the gates. This could result in an area-inefficient layout if only a few gates had very large fanins. This is similar to the long net problem in PLA folding and is handled similarly. A preprocessing step is included in the algorithm which detects unusually large fanin gates. These gates are split into two gates each with half the signals attached to the original gate. During the annealing the two sub-

gates are constrained to be adjacent to each other at all times. This is necessary for simple eventual realization. If the sub-gates were not constrained to be adjacent, one would have to route a signal between the two halves to realize the gate in the final layout.

5.2. Gate Matrix Layout Optimization

A typical Gate Matrix layout is shown in Figure 5.1b. This layout style is the complement of the Weinberger array in a sense — gates occupy rows instead of columns and signals occupy columns instead of rows. A gate may need more than one row to be realizable and folding implies more than one gate occupying the same row. The problem is to find the ordering of signals which results in the row height being minimized. Simultaneous compaction of both the p and the n parts of the gate matrix is performed (unlike [wing85]) as complex gates can make the two halves asymmetric. Asymmetry implies that an optimal ordering of signals for minimizing the n part area may not be optimal for the p part.

5.2.1. Ordering the signals

The algorithm is similar to the merged Weinberger array layout optimization algorithm except that signals are ordered not gates. The ordering of signals determines the range of the p and n parts of the gates as well as the number of rows required to realize the complex gates. Rules for realization in different number of rows can be formulated for any gate and used to determine the maximum row cardinality/height given any particular ordering of signals, which in turn determines the area.

The maximum weighted density of rows at any column is minimized using an annealing step with an appropriate cost function. The procedure *find_cost_of_configuration()* has to be modified for the gate matrix case.

```
procedure find_cost_of_configuration()
{

 cost = 0;
 column_heights = 0;
 forall( gates ) {
     l = infinity;
     r = 0;
     forall( signals attached to gate ) {
       l = MIN(signal_column.l);
       r = MAX(signal_column.r);
     }
     row_mag1 = find_row_magnitude( p part of gate );
     row_mag2 = find_row_magnitude( n part of gate );
     total_width = p_part_gate_width * row_mag1 +
                   n_part_gate_width * row_mag2;
     for( col = l; col <= r ; col = col + 1 ) {
       column_heights[col] = column_heights[col] + total_width;
 }
 forall( columns )
    max_column_height = MAX(column_heights);
 cost = penalty_for_disorder() + max_column_height;
 return(cost);
}
```

The function *find_row_magnitude()* calculates the number of rows required to realize the p or n parts of the gate for a particular ordering of columns, applying rules to each gate. The cost function also provides for *ordering constraints* on signals. To impose these constraints on the signals, the penalty function approach is adopted[kirk83]. Intermediate configurations may have signal orders violating the constraints but they are penalized. The penalty assigned for violations is big enough to ensure that the final solution satisfies the constraints.

Constraining positions of terminals to a particular side of an gate matrix is vitally necessary in LSI system design. Positions of signal columns can be bounded in this approach unlike previous approaches[wing85]. *Bounded column constraints*

are very simply implemented by limiting the range of interchange of the constrained column to within the specified bounds after finding an initial order which satisfies the bounding constraints. The generation of states proceeds as follows

(1) A random number between one and the total number of columns is generated.

(2) A second random number is generated between the two bounds specified on the column represented by the first number. If the first column happens to be within the specified bounds of the column represented by the second number, an attempt is made to interchange the two columns.

If unspecified, the bounds of a column are the two edges of the matrix. Inputs and outputs can be constrained to the left/right sides of the matrix with the intermediate signal columns in the middle.

It may be beneficial to have input signals and their complements next to each other in the optimized matrix for the easy placement of input buffers. This takes the form of *adjacency constraints* on the associated signals. Adjacency constraints are accommodated by treating the pair of constrained signals as a single compound signal and interchanging the compound signal with other (possibly compound) signals. The two constrained signals may also be interchanged with each other.

If all the gates are simple then the optimization is complete after the annealing; else a post processing step follows.

5.2.2. Expanding the rows

As mentioned earlier, rule formulation for realization of complex gates in different number of rows in the gate matrix is similar to the Weinberger array case.

Given the layout strategy/rules, after the annealing, the *row magnitude* (as opposed to column magnitude in the Weinberger array case) of the strongest satisfied rule for each complex gate is examined. If the row magnitude is 1, then the gate is realizable in one row and nothing is done. If the row magnitude is greater than 1, row(s) may have to be added to realize the gate. Transistors are placed on existing adjacent row positions if they are vacant and placement does not violate the constraints due to folding, else a new row is added next to the present row and the gate is realized by placing the appropriate transistors on it. Typically, if the number of complex gates is a small fraction of the total, no rows are added.

5.2.3. Dealing with dense signals

The counterpart to the large fanin gate problem in Weinberger array optimization is the dense signal problem in gate matrix compaction. However, this is slightly simpler as signals can be split or reproduced easier than gates. Signals feeding into a large number of gates are split into two or more signals, and treated as distinct nets during the annealing process. They can be constrained to lie next to each other in the final layout for easy routing, since they will have to be electrically connected if necessary. This splitting of signals can vastly increase foldability.

5.2.4. The P and the N parts of the Gate Matrix

Previous approaches assume a one to one correspondence between the pR and n parts of a gate matrix and that an optimal ordering of signals for the n part implies an optimal ordering for the p part as well. This is not true if the circuit has complex gates. Since the structure of the p and n parts of a complex static CMOS gate are duals of each other, a different set of rules may apply for the realization of the the two parts of the gate in a certain number of columns for a given layout style. Either a different set of rules can be applied for the p and n parts or they can be optimized separately. If the latter is the case then routing of signals between the two parts becomes necessary. The number of layers available for routing can be two or three, so a multi-layer router like CHAMELEON[brau86] can be used.

5.3. Examples and Results

Figure 5.5a shows the abstracted symbolic layout of a a static NMOS multi-level combinational logic circuit with 128 transistors. Series and parallel transistors (the tiles of Figure 4.3) have been symbolically represented as boxes. The lengths and widths of some transistors are twice the minimum size. While optimizing the initial layout all the transistors were assumed to be of equal dimensions. i.e. the row cardinality was minimized disregarding the individual row widths. Inputs were constrained to the left of the matrix. The layout was obtained in 20 seconds on a DECVAX 8600 running Berkeley UNIX[4] 4.3. It's size is 29 by 23 units.

[4] UNIX is a Trademark of AT&T Bell Laboratories

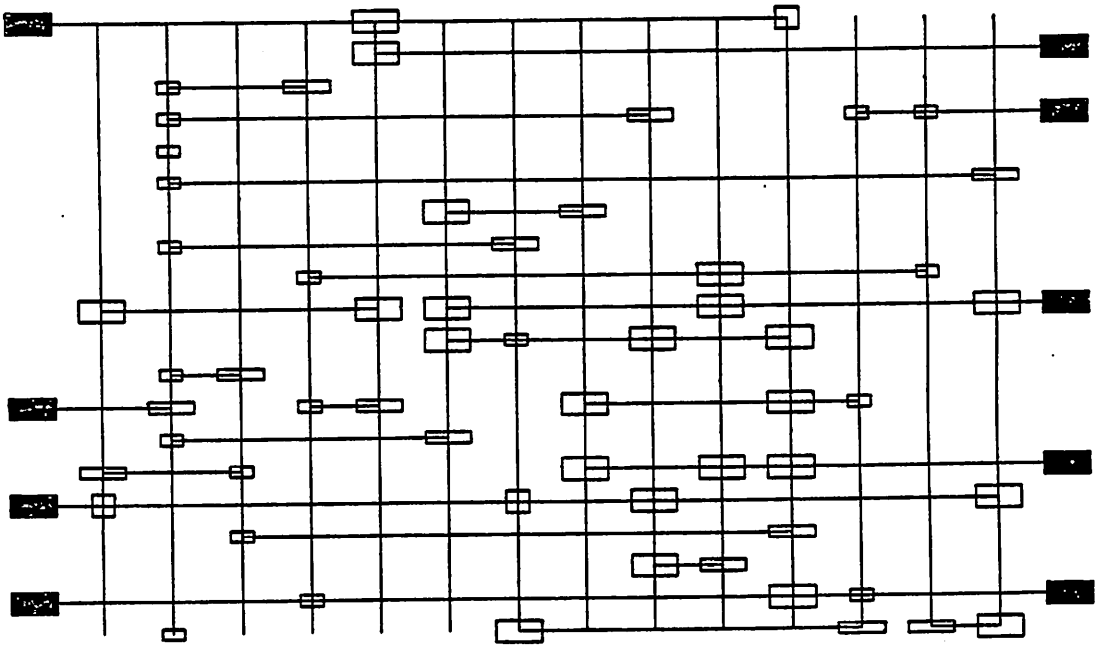


Fig. 5.5a.

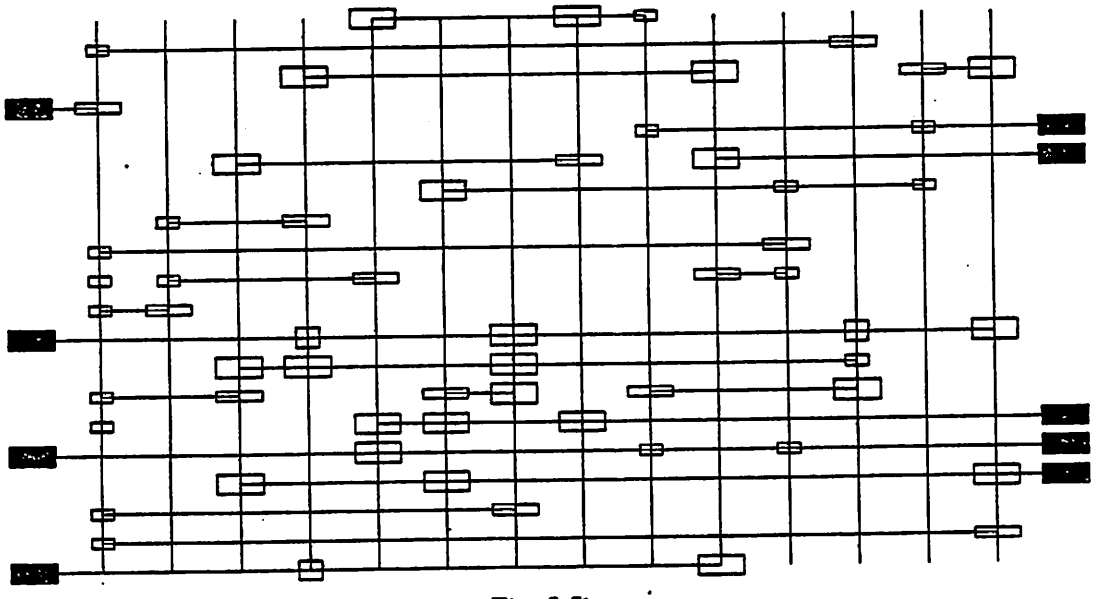


Fig. 5.5b.

The circuit was abstracted and optimized again but this time the height of the array, i.e the sum of individual row widths, was minimized. The resulting layout is shown in Figure 5.5b. This layout has the same row cardinality but its size is 27 by 25 units, 10% smaller than the previous one. This illustrates the necessity in providing for non uniform cell dimensions in a gate matrix or Weinberger array layout algorithm. If the transistors had been of widely varying the reduction would have been greater.

The results obtained by optimizing for row cardinality with results obtained by optimizing for total row height for several circuits as gate matrices and Weinberger arrays with a variety of constraints are compared in Table 5.1. (b.c. = bounded column, ord. = ordering, l.r. = inputs left and outputs right). The transistor lengths and widths varied by about a factor of 3. The results indicate that area savings of 10-20% are possible.

EXAMPLE	# gates	constraints	# signals	I Row cardinality area %	II Row height area %	Area II / Area I	Cpu Time (seconds)
MAT1	25	none	48	37	34	0.92	17
		b.c.		38	34	0.89	16
		ord.		43	39	0.91	34
MAT2	70	none	86	40	35	0.87	22
		b.c.		43	39	0.91	21
		ord.		43	40	0.93	42
MAT3	77	none	93	31	28	0.90	16
		b.c.		38	34	0.89	16
		ord.		36	32	0.89	30
MAT4	63	none	85	52	46	0.88	16
		b.c.		53	47	0.88	16
		ord.		59	50	0.85	29
MAT5	56	none	73	44	40	0.91	23
		b.c.		49	44	0.90	21
		ord.		47	44	0.93	40
MAT6	561	none	644	25	20	0.80	915
		b.c.		25	20	0.80	895
		ord.		28	24	0.85	1650
ARR1	25	none	48	38	33	0.86	15
		l.r.		60	54	0.90	17
ARR2	70	none	86	36	32	0.89	25
		l.r.		43	38	0.88	27
ARR3	77	none	93	36	30	0.83	16
		l.r.		43	36	0.83	18
ARR4	63	none	85	50	45	0.90	19
		l.r.		53	47	0.88	21
ARR5	56	none	73	37	32	0.86	21
		l.r.		59	52	0.88	23
ARR6	561	none	644	20	17	0.85	845
		l.r.		26	23	0.87	905

Table 5.1. Comparison of row cardinality with row height optimization

EXAMPLE	# gates	% complex gates	Initial area	ALG-I final area	ALG-II final area	ALG-I time (sec)	ALG-II time(sec)
ARR1	25	12	1200	650	650	17	26
ARR2	56	10	4088	2128	2128	22	29
ARR3	56	20	6020	2310	2244	16	26
ARR4	70	35	1232	756	675	16	21
ARR5	28	30	4345	1705	1643	23	31
ARR6	561	10	373520	86420	86122	905	1100

Table 5.2. Comparison of ALG-I (unmerged) and ALG-II (merged)

Cpu time has always been a major concern with simulated annealing approaches. However, though the number of states generated is large the operations required per state are minimal. In particular, the procedure *find_cost_of_configuration()* has been implemented such that all calculations are incremental. A very large circuit consisting of 561 gates and 2300 transistors, which is a multi-level random logic implementation of the Berkeley SOAR microprocessor controller was partitioned and laid out using this program. The total run time for partitioning and folding of the partitions was just 203 seconds on a 8600. The circuit was also laid out without partitioning. The area in this case was smaller but the aspect ratio may not have been desirable. The run-time in this case was 905 seconds. As indicated in Table 5.1 very large circuits have been laid out using this program with reasonable cpu time expenditure.

Table 5.2 compares the merged 2-dimensional approach (ALG-II) to Weinberger array layout with the sequential ordering approach (ALG-I). Cpu times, initial and final areas are given. Number and complexity of gates are also indicated for each circuit. As the results indicate the merged approach does better when the percentage of complex gates is high.

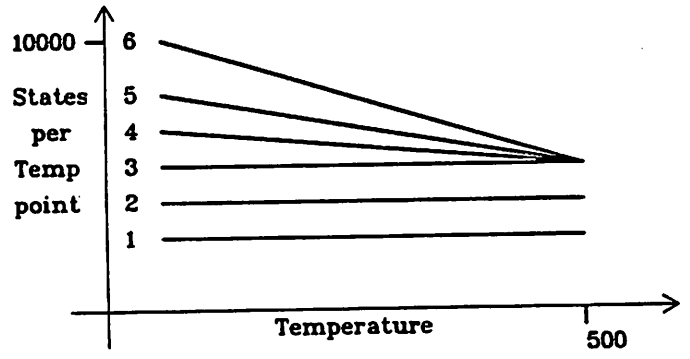
5.3.1. Solution versus Time & Temperature versus Cost Graphs

One dimensional folding being a simpler problem to solve than the corresponding two dimensional problem, cpu time requirements are smaller. One can afford to operate at a temperature profile which just about guarantees optimality. Figure 5.6a shows the plots of different temperature profiles used for the example under consideration during the annealing process. Figure 5.6b illustrates the quality of the solution for each of the temperature profiles and Figure 5.6c illustrates the behavior of the cost function during each of the annealing processes

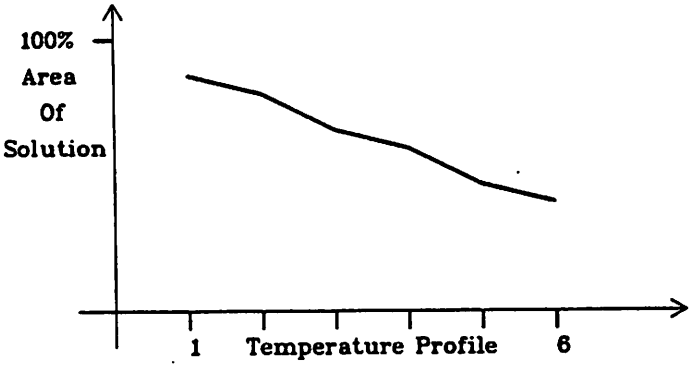
with the corresponding temperature profile.

5.4. Conclusions

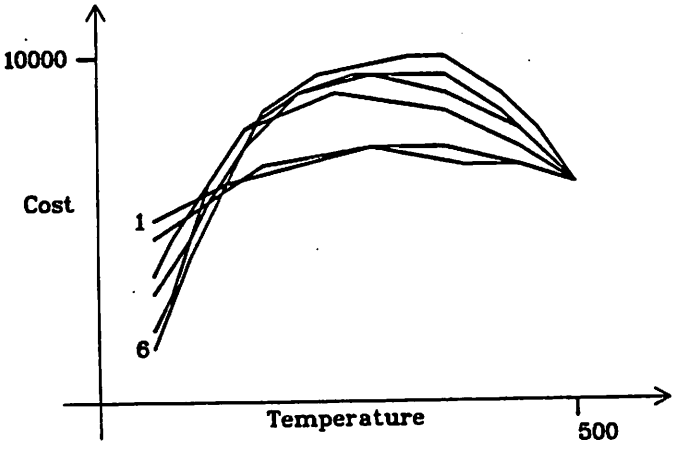
New techniques for layout optimization of gate matrices and Weinberger arrays using simulated annealing have been presented in this chapter. Constraints can be built into the simulated annealing approach and thus optimization can be done under a wide variety of constraints. Attractive features of this approach include the ability to partition large matrices into smaller ones to achieve desirable aspect ratios and selective treatment of signals if necessary.



(a) Temperature Profiles



(b) Quality of Solutions Obtained



(c) Cost Profiles

Fig. 5.6

CHAPTER 6

Multiprocessor Implementation of Simulated Annealing Based Algorithms

A parallel implementation of the simulated-annealing-based algorithms of Chapters 4 & 5 on the Sequent Balance 8000 multi-processor[deva86b] is described in this chapter. Modifications to the algorithms are necessary to fully exploit parallelism. These modifications have resulted in an efficient parallel implementation. The techniques used in this multi-processor implementation can be used to parallelize simulated annealing over a wide variety of problems.

6.1. Simulated Annealing on Multiprocessors

A multi-processor implementation of the algorithms described in Chapters 4 & 5 using static windowing of gates or signals between the various processors is relatively simple. Static windowing implies that each processor is allocated a set of gates/signals in the array — a window is implemented on each processor so it *sees* only a portion of the array. Interchanges within a window do not affect the cost outside that window, hence window interchanges can proceed in parallel since they are independent. The acceptance and rejection of states can be performed based on the initial and final window costs. Unfortunately, this scheme invariably results in a low-quality solution. This is because constraining the moves to within a window violates the simulated annealing paradigm of generating moves across the entire configuration space and the globally optimal solution cannot be reached.

Dynamic windowing implies that the portion of the array allocated to each processor changes with time. Each processor begins with an allocated region in the

array corresponding to the window implemented on that processor, but the windows of the processor move after a certain number of states have been generated within the window.

Dynamic partitioning implies that the allocation of gates/signals to processors changes with time. In our implementation dynamic partitioning of signals/gates is achieved by inter-window exchanges, signals belonging to different windows are interchanged and the cost of the global move is calculated in parallel by all the "affected" processors.

Both dynamic windowing and dynamic partitioning techniques have been used in this implementation to increase efficiency, as distinct from previous techniques for multi-processor simulated annealing[cass86,krav86].

6.2. Dynamic windowing

In the uni-processor version of the algorithm the probability of exchange of gates throughout the array is uniform especially at high temperatures. This is important to preserve the global convergence properties of simulated annealing based algorithms. A static windowing scheme (where every processor is assigned a certain region in the array which never changes, and with interchanges only within the window) results in sub-optimal solutions since the probability of exchange outside the window is zero.

The goal of dynamic windowing is to try to increase and make uniform the probability of global interchange across the array while maintaining processor efficiency. Windows implemented on processors do not overlap but *move* across the array, so a larger number of different gate interchanges are possible. Dynamic windowing is illustrated in Figure 6.1. Thus after the first window move a gate originally in window 1 can move to window 3, after two window moves to window 4

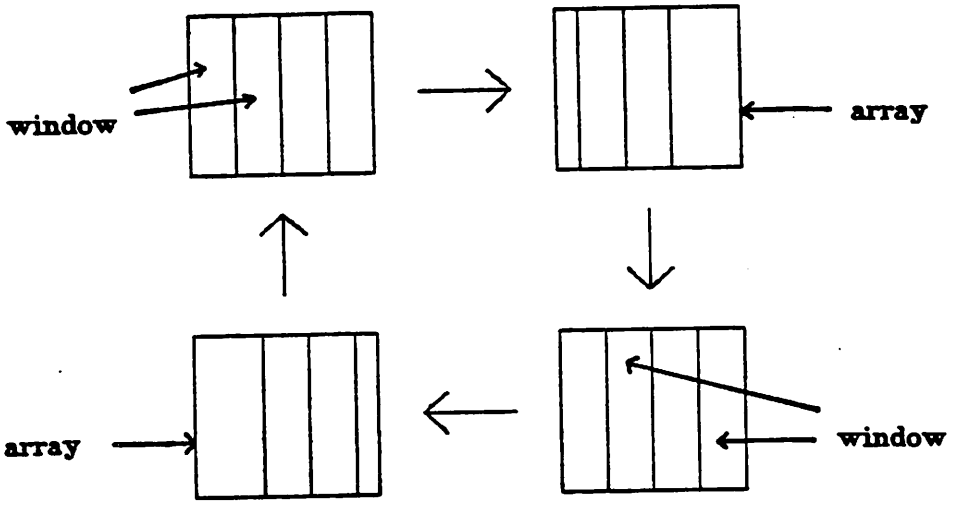
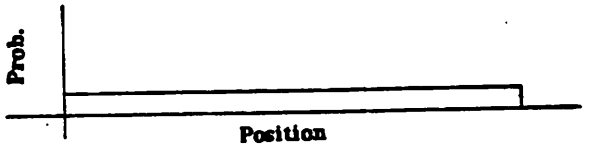


Fig. 6.1 Dynamic windowing

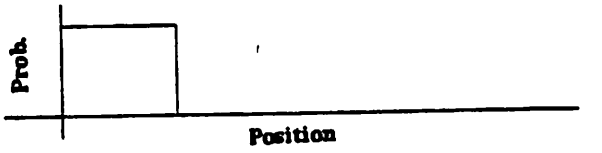
with a probability that can be computed. The probability distribution of possible column positions of a gate originally in the first column is shown qualitatively for the uni-processor (single window) case and the multi-processor (dynamic windowing) case in Figure 6.2. Dynamic windowing causes the probability distribution in the multi-processor case to smooth out with increasing number of moves and window movements and to approach the uni-processor distribution.

6.3. Implementation

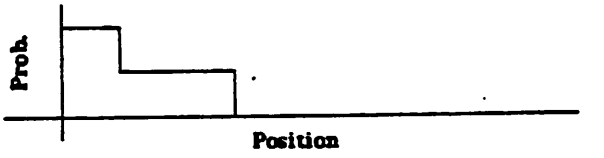
Two different cycles are possible during the annealing process. In the first kind of cycle, each processor generates gate interchanges within its current window in the array and calculates the window cost. Since the interchange is within the window, this calculation of net length and signal density is independent of other processors. Then the first available processor decides the acceptance/rejection



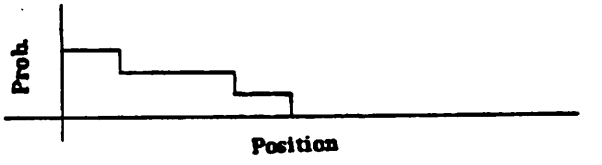
(a) Uni-processor case



Before first window move



After first window move



After second window move

(b) Multi-processor case with dynamic windowing

Fig. 6.2 Qualitative probability distributions

of these states sequentially. A window, *W*, is picked and the overall cost for the whole configuration is calculated using the old window costs for the other windows. If the move is accepted, *W*'s cost is updated, so the following interchanges (and windows) picked will now use this new cost for calculation of the total cost. This is done for all processor interchanges. The acceptance/rejection process needs to be sequential because one part of the cost function is the maximum of the signal densities across the whole array, not just a window. However, the computations involved are very few, especially in comparison with the computations

required to calculate the cost.

In the second kind of cycle, the gate interchanges are *not* restricted to within a processor window. A single processor generates a move and the evaluation of the cost function for this single global move proceeds in parallel for each processor window. Elaborating, each processor evaluates the portion of the cost accrued from the signals attached to the gates presently within its current window. If gates from windows 1 and 3 are interchanged, the densities of signals can change not only in windows 1 and 3 but also in the window between them, namely window 2. The total cost is found and the acceptance/rejection of this state is decided by the first available processor. This single global move uniformly increases the probability of the gate being in any position in the array, making the probability distribution more similar to the uni-processor version.

The first kind of cycle, which generates several moves in parallel, is more efficient in our implementation. However, the second cycle, involving a global move whose cost is calculated by several processors in parallel, is vital for generating a high quality solution since it enlarges the configuration space. Dynamic windowing as described earlier increases the efficiency by decreasing the need for the second kind of cycle. The windows of the processor are moved to the right or left after a certain number of moves have been generated. The windows can be of varying sizes during the annealing process as illustrated in Figure 6.1.

The overall multi-processing algorithm is illustrated in Figure 6.3. The routine *limbo*() is a simple loop around which processors spin while idling. The type of move to be generated, i.e. a inter-window or a set of intra-window moves is decided by the function decide_move_type() depending on the required ratio between cycles. A synchronization point is provided for initialization of windows after a window move.

```
 while( stopping_criterion() is FALSE ) {
  while( inner_loop_criterion() is FALSE ) {
/*
* Generate new states in each processor and find the cost
* of the states w.r.t the initial configuration.
*/
    goto genmain;
    generate:
        gen_state_find_cost( myid, TYPE );
        limbo();
        goto window;

    genmain:
      gen_state_find_cost( 1, TYPE );
      wait( till all kids done );
/*
* Master processor decides whether to accept these states or not.
*/
    accept_or_reject_states( TEMP );
    TYPE = decide_move_type();
    goto main;
/*
* Windows are moved to the left and initialization for next cycle.
*/
  window:
      initialize_windows( myid );
      limbo();
      goto generate;

  main:
    initialize_windows( 1 );
    wait( till all kids done );
  }
/*
* Master processor updates temperature
*/
   TEMP = update_temp( TEMP );

 }
```

Fig. 6.3 Multiprocessor Simulated Annealing

Thus in this scheme, the partition of gates amongst various processors changes even for a given window configuration due to inter-window exchanges

(second cycle) implying dynamic partitioning of gates between processors, and the window of each processor also changes with time, implying dynamic windowing across the array.

The optimal relative frequency of the two different kinds of cycles, the frequency of window movements and the amount of window movement was empirically determined over a set of benchmarks. For efficiency reasons the relative frequency of the two different kinds of cycles varies with temperature. At high temperatures, the need for global moves is higher. The lower the temperature, the smaller the probability that a interchange of gates spanning a large distance will be accepted - hence the need for such moves is lesser. In fact, in the uni-processor version of the algorithm, a range-limiter was implemented at lower temperatures to generate moves with a high probability of acceptance which effectively amounted to clustering the gates into partitions. At high temperatures the ratio is 1:1 and as the temperature decreases the number of cycles of the second kind is decreased to zero. The frequency of window movements is constant throughout - at every temperature point the number of movements is equal to the number of processors. The windows move to the right by an amount equal to half the window size.

This windowing and partitioning scheme preserves the convergence properties of simulated annealing based algorithms to the global optimum by maintaining the probability of different moves to be similar to that of the uni-processor (uni-

window) version.

6.4. Results

Results using dynamic windowing and dynamic partitioning techniques on some large examples are given in Table 6.1 for 1-8 processors on the Sequent Balance 8000 multiprocessor. The solution obtained was the same in all cases; however the number of states generated is different as indicated. Though dynamic windowing and partitioning are efficient, more states need to be generated for larger number of processors to keep the probability of reaching the global optimum the same. The efficiences have been calculated for equal quality solutions and not merely for same inner loop criteria which would be misleading.

It is interesting to note that for example-1 even though 1120 moves were generated for 8 processors as compared to 640 in the single processor case, the efficiency is as high as 74.3%. This is because the cost function computation time depends on to the distance between the gates which are being interchanged. Interchanging gates which are further apart involves greater computations while evaluating the cost of the new configuration. Given a parallel move cycle with interchanges within each window, the average distance between the gate interchanges for a N processor configuration is less than that for a M processor

No. of procs	States per temp pt	Time (sec)	Final Cost	Eff. %
1	640	742.8	25100	100
2	640	374.1	25110	99.2
3	640	254.6	25110	97.2
4	800	203.5	25090	91.2
5	880	172.7	25100	86.0
6	960	154.8	25110	79.9
7	1040	138.1	25080	76.8
8	1120	124.9	25090	74.3

Example 1

No. of procs	States per temp pt	Time (sec)	Final Cost	Eff. %
1	560	626.1	21510	100
2	560	316.8	21530	98.8
3	700	213.3	21500	97.8
4	760	169.9	21520	92.1
5	800	144.0	21510	86.9
6	920	129.4	21540	80.6
7	1000	115.4	21500	77.5
8	1050	104.2	21490	75.1

Example 2

Table 6.1. Efficiences on the Sequent Multiprocessor

configuration if $N > M$.

6.5. Conclusions

A multi-processor implementation of a simulated-annealing-based algorithm for a topological optimization problem has been presented in this chapter. New techniques, namely, dynamic windowing and dynamic partitioning schemes have been used to preserve the convergence properties of simulated annealing to the glo-

bal optimum of the configuration space while achieving high processor utilization. These techniques can be used for parallel implementations of simulated-annealing-based algorithms over a wide range of problems.

CHAPTER 7

Conclusions

This report has presented a topological optimization tool for array-based layout styles which can be used in an automated pipeline for synthesizing combinational/sequential logic circuits in silicon. The array optimization tool allows various design and layout styles to be explored, and then the best alternative for a particular circuit can be chosen. This is in contrast to design and layout style specific systems which offer virtually no alternatives in the automated synthesis process.

The tool described can be used in the lower end of a silicon compiler, after the logic has been extracted from high level behavioral or register-transfer level descriptions and minimized using logic minimization techniques[bray84a,bray84b].

The major topics covered in this report were twofold. Topological compaction algorithms are a key to producing an area-efficient layout. Topological optimization is performed on array based structures using a technique known as folding. Folding implies that more than one signal/gate occupies a row or column in a array. All previous folding algorithms break down for irregular arrays, i.e. arrays with non uniform cell sizes. A generalized array optimization scheme, which folds structures ranging from highly regular PLAs, to irregular SLAs was developed and implemented. Constraining signals to locations in the array is vitally necessary from a LSI design point of view. The algorithms allow for a variety of constraints and optimize for minimum area within these constraints. The results obtained are excellent. The last phase consists of tiling the compacted array, and generating and actual mask level layout. Context-based tiling[mayo83][hofm85a] is a power-

ful tool which separates symbolic compaction and layout generation. After the array has passed through the tiler, the actual layout is produced by stitching together the individual tiles in the array.

Simulated annealing has been applied successfully to a wide variety of combinatorial optimization problems. A parallel version of simulated annealing for a topological compaction problem has been developed. High processor utilization was achieved using dynamic windowing and dynamic partitioning techniques. These techniques are general and can be used to parallelize simulated-annealing-based algorithms for a wide variety of layout problems.

Approaches to efficient multistage combinational logic synthesis have been have been reported in [bray84b,hofm85a]. Multistage sequential logic synthesis is still an open problem. At a high level, decisions regarding the break-up of finite state machines need to be made. At a lower level, each finite state machine can be synthesised as many stages of combinational logic networks feeding into clocked registers (which store the present state). Placing the sets of registers between the combinational logic stages, so as to optimize for eventual area and speed of the synthesised logic is an non-trivial problem. Once a multistage sequential logic network has been designed at the gate/flip-flop level, the system described here provides an automatic path for layout of the circuit as a Storage/Logic Array.

REFERENCES

[asan82]
T. Asano. "An optimum gate placement algorithm for MOS one-dimensional arrays." *Journal of Digital Systems*, vol VI, no 1, 1982.pp 1-28.

[brau86]
D. Braun, J. Burns, S. Devadas, H.K. Ma, K. Mayaram, F. Romeo and A. Sangiovanni- Vincentelli."CHAMELEON: A New Multi-Layer Channel Router". *23rd Design Automation Conference*, Las Vegas, 1986.

[bray82]
R. Brayton and C. Mcmullen. "The Decomposition and Factorization of Boolean Expressions". *Proc. International Symposium on Circuits and Systems*, May 1982, pp 49-54.

[bray84a]
R. Brayton and C. McMullen. "Synthesis and Optimization of Multistage Logic". *Proc. 1984 International conference on Computer Design* , Oct 1984, pp 23-30.

[bray84b]
R. Brayton et al. "Logic Minimization Algorithms for VLSI synthesis". *Kluwer Academic Publishers*,. 1984.

[bray85]
R. Brayton et al. "A Micro-processor using the Yorktown Silicon Complier". *International Conference on Computer Design*, October 1985.

[cass86]
A. Casotto, F. Romeo, and A. Sangiovanni-Vincentelli, "A Parallel Implementation of Simulated Annealing for Cell Placement". *ICCAD-86 Digest*, Nov. 1986.

[chuq82]
S. Chuquillanqui and T. Perez Segovia, "PAOLA: A Tool for the Topological Optimization of Large PLAs". *Proc, 19th Design Automation Conf.*, pp. 300-306,1982.

[demi83a]
G. De Micheli, "Computer-Aided Synthesis of PLA-Based Systems". *PhD Dissertation, University of California*, 1983. (Chapter 4).

[demi83b]
G. De Micheli and A. Sangiovanni-Vincentelli,"Multiple Constrained Folding of Programmable Logic Arrays : Theory and Applications". *IEEE*

Transactions on Computer-Aided Design,Vol.CAD-2,No. 3,July 1983

[deva86a]
S. Devadas and A. R. Newton,"GENIE: A Generalized Array Optimizer for VLSI Synthesis",*23rd Design Automation Conference*, Las Vegas, 1986.

[deva86b]
S. Devadas and A. R. Newton, "Topological Optimization of Multiple Level Array Logic: On Uni and Multi-Processors", *International Conference on Computer Aided Design, ICCAD-86 Digest*, Nov. 1986.

[fidu82]
C. M. Fiduccia, R. M. Mattheyses, "A Linear Time heuristic for improving Network Partitions," *Proceedings of the 19th D. A. Conf.* pp 175-181, June 1982.

[flei75]
H. Fleisher and L. I. Maissel, "An Introduction to array logic", *IBM J. Res. and Dev.*, Vol 19., pp. 98-108, March 1975.

[gema84]
S. Geman and D. Geman, "Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images", *IEEE Transactions Pattern Analysis and Machine Intelligence*, Vol 6 pp 721-741.

[geus85]
A. de Geus and W. Cohen, "SOCRATES: A Rule Based System for Optimizing Combinational Logic", *IEEE Design and Test* August 1985.

[hash71]
A. Hashimoto and J. Stevens, "Wire routing by optimizing channel assignment," *Proc. 8th Design Automation Conf.*, pp 214-224, 1971.

[hach80]
G. D. Hachtel, A. R. Newton, and A. L. Sangiovanni-Vincentelli, "An algorithm for optimal PLA folding",*Proc Int. Circuits and Comp Conf. (New York N.Y.)*,Oct 1980.

[hach82]
G. D. Hachtel, A. R. Newton, and A. L. Sangiovanni-Vincentelli, "An algorithm for optimal PLA folding",*IEEE Trans. Computer Aided Design of ICAS*, vol.1, no 2,Apr 1982

[hofm85a]
M. Hofmann, "Automated Synthesis of Multi-Level Combinational Logic in CMOS Technology", *PhD Dissertation, University of California*, 1985. (Chapter 1-9).

[hofm85b]
M. Hofmann and A. R. Newton, "A Domino CMOS Logic Synthesis System", *Proc 1985 Int. Symp. on Circ. and Syst., Kyoto, Japan*,June 1985

[hong74]
S. J. Hong, R. G. Cain and D. L. Ostapko, "MINI: A Heuristic approach for logic optimization", *IBM Journal of Res. and Dev.*, Vol 18, pp 443-458, September 1974.

[kern70]
B. W. Kernighan and S. Lin, "An efficient heuristic procedure for Partitioning graphs" ,*The Bell Syst. Tech. journal* 49:2, pp 291-307.

[kirk83a]
S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing" ,Science, Vol.220,N. 4598, pp 671-680, 13 May 1983

[krin86]
C. Kring, "MKARRAY: A Tool to Build and Maintain Arrays" ,*U. C. Berkeley, Internal Report*, May 1986.

[krav86]
S. Kravitz and R. Rutenbar, "Multi-processor based placement by simulated annealing", *23rd Design Automation Conference, Las Vegas* July 1986.

[lund84]
M. Lundy and A. Mees, "Convergence of the Annealing Algorithm", *Simulated Annealing Workshop*, Yorktown Heights, April 1984.

[mah84]
G. H. Mah and A. R. Newton, "PANDA: A PLA generator for multiply folded PLAS", *Proc Int. Conf. on Computer-Aided Design 1984, pp 122-124.*

[mari85]
C. Marino, "Smalltalk on a RISC - CMOS Implementation", Masters Report, Department of EECS, University of California, Berkeley, April 1984.

[mayo83]
R. N. Mayo and J. K. Ousterhout, "Pictures in Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool", *Proc. 20th Design Automation Conference*, pp. 270-276. June 1983.

[mccl65]
E. J. McCluskey Jr, "Introduction to the Theory of Switching Functions", *McGraw-Hill*, 1965.

[moor85]
T. P. Moore and A. J. De Geus, "Simulated Annealing Controlled by a Rule Based Expert System",pp 200-202,*Digest of Technical Papers, ICCAD-85.*

[moor86]
P. Moore, "Oct Tutorial", *U. C. Berkeley Internal Report*, Feb. 1986.

[mlyu85]
M. L. Yu and W. J. Kubitz, "A VLSI Cell Synthesiser with structural constraint considerations", *ICCAD Digest 1985*, pp 58-61.

[newt81]
R. Newton, D. Pederson, A. Sangiovanni-Vincentelli and C. H. Sequin, "Design Aids for VLSI: The Berkeley Perspective", *IEEE Transactions on Circuits and Systems*, vol CAS-28, no 7, July 1981, pp 666-680.

[ohts79]
T. Ohtsuki, H. Mori, E. S. Kuh, T. Kashiwabara, and T. Fujisawa, "One dimensional logic gate assignment and interval graphs," *IEEE Trans. Circuits Syst*, pp 675-684, Sept 1979.

[park79]
A. C. Parker, D. E. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Lieve and J. Kim, "The CMU Design Automation System", in ACM IEEE 16th Design Automation Conference Proceedings 1979.

[pati75]
S. Patil, "An Asynchronous Logic Array", *Project MAC Tech. Memo TM-62*,May 1975.

[pati79]
S. Patil and T. Welch, "A Programmable Logic Approach to VLSI", *IEEE Trans. on Computers*, vol C-28 Sept. 1979, pp 594-601.

[pati85]
S. S. Patil, "Private Communication".

[rome84]
F. Romeo, and A. Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications,*ERL College of Engineering, University of California, Memorandum No. UCB/ERL M84*,13 March 1984

[rome85]
F. Romeo, and A. Sangiovanni-Vincentelli, "Probabilistic Hill Climbing Algorithms: Properties and Applications,*H Fuchs ed. 1985 Chapel Hill Conf. on VLSI*, May 1985.

[rude85a]
R. Rudell, "ESPRESSO-IIC Users Manual", *U. C. Berkeley Internal Report*, July 1985.

[rude85b]
R. Rudell and A. Sangiovanni-Vincentelli, "ESPRESSO-MV: Algorithms for

Multiple Valued Boolean Minimization". *Proceedings of the Custom Integrated Circuits Conference*, May 1985.

[sech84]
C. Sechen and A. Sangiovanni-Vincentelli."The TimberWolf Placement and Routing Package" *Proc. 1984 Custom Integrated Circuit Conference, Rochester*.May 1984

[sech85]
C. Sechen and A. Sangiovanni-Vincentelli."The TimberWolf Placement and Routing Package" *Journal of Solid State Circuits*, May 1985.

[smit82]
K. F. Smith, T. M. Carter and C. E. Hunt, "Structured Logic Design of Integrated Circuits using the Storage/Logic Array (SLA)," *IEEE Journal of Solid State Circuits vol SC-17, No 2*, pp 395-406, April 1982.

[souk81]
J. Soukup, "Circuit Layout", *Proc. of the IEEE*, Vol 69, no. 10, October 1981, pp 1281-1304.

[vecc83]
M.P. Vecchi and S. Kirkpatrick, "Global Wiring using Simulated Annealing", *IEEE Trans. on CAD*, Oct. 1983.

[wein67]
A. Weinberger."Large Scale Integration of MOS Complex Logic: A Layout Method" *IEEE Journal of Solid-State Circuits*, vol SC-2.No 4, pp 182-190, December 1967.

[wong85]
D. F. Wong, H. W. Leong and C. L. Liu, "A Simulated Annealing Channel Router", *International Conference on Computer-Aided Design of Integrated Circuits and Systems*, ICCAD-86 Digest, Santa Clara, Nov. 1985.

[wong86]
D. F. Wong, H. W. Leong and C. L. Liu, "Multiple PLA folding using the method of Simulated Annealing", *Proc. of Custom Integrated Circuit Conference*, May 1986.

[wing82]
O. Wing."Automated Gate Matrix Layout", *Proc. 1982 IEEE Int. Symp. on Circ. and Sys*, Rome, Italy, pp 681-685.

[wing85]
O. Wing, S. Huang, and R. Wang."Gate Matrix Layout", *IEEE Transactions on Computer-Aided Design*,Vol.CAD-4,No. 3,July 1985

[wood79]
R. A. Wood."A High Density Programmable Logic Array Chip",*IEEE Trans. on Computers*, vol C-28,No 9. pp. 602-608,September 1979

[yosh75]
H. Yoshizawa. H. Kawanishi. and K. Kani. "A Heuristic procedure for ordering MOS arrays",*Proc. 12th Design Automation Conference*, 1975,pp 384-89

NAME

genie — A Generalized Array Optimizer

SYNOPSIS

genie [options] inputfile [> outputfile]

DESCRIPTION

Genie is a program that performs topological compaction of arrays using a combinatorial optimization technique called simulated annealing. The primary goal is to optimize the silicon area occupied by the array. The kinds of arrays which can be compacted are PLAs, Weinberger arrays, Gate matrices, Multi-level matrices and SLAs. Simple/Multiple constrained/unconstrained folding is supported.

Genie is run in a batch mode. *Genie* reads the symbolic representation of the array to be compacted and the constraints under which the compaction is desired from an input file. *Genie* also requires a *technology* file which contains information about the sizes of the constituent cells in the array. *Genie* produces a compacted array in a symbolic format folded under the various constraints. *Genie* can be run for long or short runs depending on the amount of cpu time the user wishes to spend.

Genie needs a unfolded connectivity matrix description as its input. It produces a simply/multiply folded matrix compatible with the *Tinker*(1) input format. Both simply-folded and multiply-folded MLMs can be assembled by the program *Tinker*. Tilers for SLAs, Weinberger arrays and Gate matrices will be available in the near future with *Genie* output compatibility.

INPUT/OUTPUT

The input to *Genie* is a matrix with symbolic characters. The characters can be anything. However, each character size must be specified in the *genie.tech* file in syntax that will be described. Some reserved characters like the . (dot) have special meanings. The . character implies a connection and no cell at that particular location. Presently, *Genie* accepts only unfolded matrices as input, and nets in *Genie* are implicit. All cells (characters other than a .) on the same row are assumed to be connected by a horizontal net and all cells on a column are assumed to be connected by a vertical net. The sizes of the cells has to be specified in the tech file in the working directory in the simple syntax of

character width length

on separate lines for all the different characters except the . character. An example tech file is shown below. The technology file should reside in the current working directory and be called genie.tech.

```
# parallel transistor
p 1 1
P 1 1
c 1 1
# series transistor
s 1 1
S 1 1
# output device
o 1 1
O 1 1
# flip flop
F 5 5
f 5 5
# inverter
I 5 5
i 5 5
# schottky diodes
```

```
0 1 1
1 1 1
# bus through
R 1 1
r 1 1
+ 1 1
A 1 1
~ 1 1
```

OPTIONS

-octr reads the OCT logic view, from the cell called cellName. The view read is optlogic. This view is normally written by *MIS(1)*.

-octw writes into the OCT symbolic view, to the cell called cellName. This view is read by the module generation programs *ELECTRA(1)* and PLA generators. The view consists of OCT bags attached to a GENIESTRUCTURE bag.

-gread reads a gate net-list from a text file. The gate net-list is described in a lisp-based syntax. Each line in the file describes a gate in the syntax shown below.

(o out (s (p sig1 sig2) (p sig3 sig4)));

represents a two level static CMOS gate with output signal out and input signals sig1 thru sig4. s denotes series connections between transistors or branches and p denotes a parallel connection. This description is converted internally into a connectivity matrix for a static CMOS layout style being used by the GEM module generator. This connectivity matrix is transparent to the user.

-m restricts the row folding to being simple. By default, multiple row and column folding is assumed.

-l restricts the column folding to being simple. By default, multiple row and column folding is assumed.

-pla is an option which translates an ESPRESSO file into the *Genie* format.

-d is the debug option and is useful only to someone who knows the program.

-r restricts the folding to be row folding only, no column folding is performed. This is used for Gate Matrix and Weinberger Array structures.

-c restricts the folding to be column folding only, no row folding is performed. This is used for Gate Matrix and Weinberger Array structures.

-v takes into account the varying sizes of the cells during topological compaction. This is an expensive option, cpu time wise and should be used only when the cell sizes vary widely.

-p Assumes an AND-OR PLA structure, and performs simple/multiple column folding of the PLA, and constrains the input and the complement to lie adjacent to each other in the AND plane. No row folding is performed since an AND-OR structure is assumed and the two planes are not merged.

-s Identical to the -p option except that the input and complement are not constrained to lie adjacent to each other.

-w is used to control the aspect ratio of the final folded matrix. This implies column folding is preferable to row folding. Note that this is different from the -c option in that it allows row folding, simple or multiple. The default is that row folding is preferable to column folding. However, using the specify -i option more control on the aspect ratio can be obtained.

-o is used for Domino MLM structures. Domino MLM structures have extra buffer circuitry. (see input file format).

-i is the specify option. This can be used to control the CPU time required for the annealing process, by specifying the number of states per fundamental unit per temperature point. It also allows specification of aspect ratio requirements by the row and column parameters which are proportional to the preferences associated with row and column folding. The five parameters asked for if this option is used are States_begin, States_end, Row_param, Col_param and Start_temp. States_begin is the number of states generated per fundamental unit per temperature point in the beginning of the annealing process and it gradually is changed to States_end which should always be greater than or equal to States_begin. Row_param and Col_param are aspect ratio control parameters and should not differ by more than a factor of 2 or be changed from the default values by more than a factor of 2. Start_temp is the specified starting temperature which is calculated internally but can be changed by the user. The default values for the first four parameters are 1, 1, 10 and 10 respectively. A good choice for obtaining better results without great increase in cpu time for States_begin and States_end are 1 and 5 respectively. A good rule of thumb for changing Start_temp while changing States_begin and States_end is to increase it over the default temperature by the factor of increase in States_end over its default value.

INPUT FILE FORMAT

An example input file is shown below.

```
new 42 22
                SRC1s<2>          ..............ss....
                readRFaccessA1    o.....................
                  3               so.....................
               CPIPE1s<7>         s.s..s....ssss.....s..
                DSTvalid*         .p.p...................
                pbusDtoINA        .p.p..............p...
                SRC1equalDST2*    .p.....................
               CPIPE1s<7>*        .p.pp...s..............
                readRFaccessB1    ..o....................
                  11              ..so...................
                SRC2equalDST2*    ...p...................
                SRC2equal16       ...p...................
                 AIzero1          ....o..................
                  18              ....po.................
                  21              .....so................
                SRC1s<2>*         ......s.......s.......
                  22              ......so......s.ss....
                SRC1s<1>*         ......s.......s.ss....
                SRC1s<0>*         ......s..........s....
                SRC1s<3>*         .......s..............
                SRC1s<4>          .......s..............
                pbusDtoINA*       .......s.......s.....s
                 AIzeroforce      ........o.............
                 busDtobusAa      .........o............
                preadSWPtoA       .........po...........
                 preadPCtoA       .........p...o........
                 preadTBtoA       .........p.o..........
                  35              .........p..o....p...
                  42              .........s......o....
```

```
          43          ..........s....o.....
          44          ..........s..o......
          45          ..........so.......
    SRC1s<0>          ..........s.s.....
SRC1equalDST2         ..........s.......
     DSTvalid         ..........s.....s
     opc2load*        ..........s.....s
   DSTtobusDa2        ..........o...
   pForwardtoINB      ..........po..
          66          ..........so.
          68          ..........so
SRC2equalDST2         ..........s.
    SRC2equal16*      ..........s.
```

left CPIPE1s<7>* DSTvalid* SRC2equalDST2* 42
right pForwardtoINB SRC1s<2> 42
ordered left CPIPE1s<7>* DSTvalid*
ordered right 42 SRC1s<2>

The unfolded input matrix shown represents a multi-level combinational circuit. The tiles s and p refer to series and parallel transistors, the tile o represents a output tile. *Genie* however, is only concerned with the existence of a tile and its size but not its function. The constraints on the signals are expressed using a left-right notation. Note that a signal can be constrained to both sides of the array, like 42 in the example. Ordering constraints can be placed on a set of signals. For example, in the array shown above, CPIPE1s<7>* has been constrained to lie above DSTvalid* on the left hand side of the array. Ordering constraints can be placed only on signals which have been constrained to the left or right of the array using the left and right declarations. The corresponding left and right declarations must precede the order constraint declaration. For Domino MLM's (-o option) an extra buffer row is required at the top of matrix which is not counted in the number of rows specified in the beginning of the file. This extra row indicates the type of buffer s or p for the gate occupying the column. An example domino MLM is shown below with the extra buffer row. Domino MLM's can be multiply row and simply column folded.

new 10 5

```
                spspp
sig_01          ....o
sig_02          o....
sig_03          so...
sig_04          s.s..
sig_05          .p.p.
sig_06          .p.p.
sig_07          .p..p
sig_08          .p.pp
sig_09          ..o..
sig_10          ..so.
```

For static CMOS circuits, the p aand n parts of the matrix may need to remain separate. The -s option constrains them to be so. Given a static CMOS MLM, a third number has to specified in the new declaration which is the demarcation between the n and p parts of the matrix (which is the row number corresponding to the last p channel signal). Only row folding is allowed in this case and the signals on opposite sides of the demarcation never intermix with each other.

new 10 5 5

sig_01_n	p.o.
sig_02_n	op..
sig_03_n	p.p.
sig_04_n	.o.s
sig_05_n	p.po
sig_01_p	s.o.
sig_02_p	os..
sig_03_p	s.s.
sig_04_p	.o.s
sig_05_p	s.so

The new declaration's associated numbers are the number of rows, number of columns and the last n column, namely row 5. For pla folding genie directly takes a PLA input file and converts it into its own format using the -pla option. Thus to fold a PLA in a AND-OR fashion constraining the input and the complement to lie next to each other in the folded array one can use
genie -pla inp.pla | genie -p > inp.folded
If constraints are to be specified, the file generated by genie -pla has to contain the constraint information, the input PLA file cannot contain the constraint information. An example input pla file is shown below

```
.i 3
.o 4
.p 4
.ilb in1 in2 in3
.ob out1 out2 out3
101 1101
-00 0101
1-0 0010
001 1110
.e
```

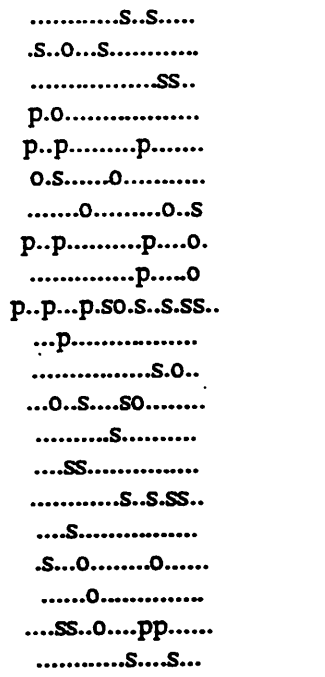
This is converted into the *Genie* input format and folded. The .ilb and .ob declarations serve to name the input and output signals, and all inputs/outputs have to have distinct names. If the .ilb and .ob declarations dont appear, distinct default names are generated within *Genie* for them. The conversion to the *Genie* input format entails expansion of the AND plane of the PLA so each input is converted into a signal and its complement. If a signal or its complement does not feed into any product terms, the signal/complement is dropped.

OUTPUT FILE FORMAT

An example output file is shown below.

new 29 21

```
p.....o.....
.....o...p.o...s.
..s...s.ss.s....s..ss
.o.....s.....
....ss....s...p.o....
.....s.....
.....s..s..
.s.....p...o.....
```



(row 0 SRC2equal16 0 DSTtobusDa2 13)
(row 1 pForwardtoINB 0 45 15)
(row 2 CPIPE1s<7> 0)
(row 3 66 0)
(row 4 pbusDtoINA* 0 preadTBtoA 14)
(row 5 SRC1s<4> 0)
(row 6 SRC1s<0> 0)
(row 7 SRC2equal16* 0 18 7)
(row 8 SRC1s<2>* 0)
(row 9 SRC2equalDST2 0 44 4)
(row 10 SRC1s<2> 0)
(row 11 SRC2equalDST2* 0 readRFaccessB1 2)
(row 12 pbusDtoINA 0)
(row 13 11 0 Alzeroforce 9)
(row 14 Alzero1 0 42 17)
(row 15 DSTvalid* 0 preadPCtoA 14)
(row 16 preadSWPtoA 0)
(row 17 CPIPE1s<7>* 0 22 10)
(row 18 SRC1equalDST2* 0)
(row 19 43 0)
(row 20 3 0 21 11)
(row 21 SRC1s<3>* 0)
(row 22 DSTvalid 0)
(row 23 SRC1s<1>* 0)
(row 24 SRC1equalDST2 0)
(row 25 68 0 busDtobusAa 14)
(row 26 readRFaccessA1 0)
(row 27 opc2load* 0 35 8)
(row 28 SRC1s<0>* 0)
(column 9 0 13)

The folded version of the matrix is shown above. Multiple folding of horizontal signals and simple column folding was done. The fold information is represented below the matrix in row and col declarations. A row declaration has the form

(row rowNum sigName1 startingColNum1 <sigName2> <startingColNum2>..)

where startingColNum1 is the starting column of the sigName1. We thus have signal sigName1 beginning at startingColNum1, and sigName2 beginning at startingColNum2. The columns are numbered from 0 to numCols - 1. The col declarations have no signal names associated with them, we have

(col colNum startingRowNum1 <startingRowNum2> ...)

which implies that column colNum has a vertical signal starting at startingRowNum1 and another vertical signal starting at startingRowNum2 and so on. The rows are numbered from 0 to numRows -1. In the above output file all the rows including the unfolded ones have been described with the signal names specified. For example, row 0 has SRC2equal16 starting at column 0 and DSTtobusDa2 beginning at column 13. Column 9 has been folded with the first vertical signal starting at row 0 and the second starting at row 13. *Genie's* output file contains all the columns in the col declarations but here, for brevity, the unfolded columns have been omitted.

DIAGNOSTICS

The input routine gives out error messages in case of wrong specification of the input matrix rows and columns and exits.

SEE ALSO

mkarray(1) electra(1) mis(1) espresso(1)

AUTHOR

Srinivas Devadas (devadas@ic.berkeley.edu)

BUGS

The -v option is not fully debugged.