A Class of Cellular Architectures to Support Physical Design Automation

ROB A. RUTENBAR, STUDENT MEMBER, IEEE, TREVOR N. MUDGE, MEMBER, IEEE, AND DANIEL E. ATKINS, MEMBER, IEEE

Abstract—Special-purpose hardware has been proposed as a solution to several increasingly complex problems in design automation. This paper examines a class of cellular architectures called *raster pipeline subarrays–RPS* architectures–applicable to problems in physical DA that are (1) representable on a cellular grid, and (2) characterized by local functional dependencies among grid cells. Machines with this architecture first evolved in conventional cellular applications that exhibit similarities to grid-based DA problems. To analyze the properties of the RPS organization in context, machines designed for cellular applications are reviewed, and it is shown that many DA machines proposed/constructed for grid-based problems fit naturally into a taxonomy of cellular machines.

The implementation of DA algorithms on RPS hardware is partitioned into *local* issues that involve the processing of individual cell neighborhoods, and *global* issues that involve strategies for handling complete grids in a pipeline environment. Design rule checking and routing algorithms are examined in an RPS environment with respect to these issues. Experimental measurements for such algorithms running on an existing RPS machine exhibit significant speedups.

From these studies are derived the necessary performance characteristics of RPS hardware optimized specifically for grid-based DA. Finally, the practical merits of such an architecture are evaluated.

Manuscript received February 20, 1983; revised February 28, 1984. This work was supported in part by the National Science Foundation under Grant MCS-8009315 and under Grant MCS-8007298.

The authors are with the Department of Electrical Engineering and Computer Science, and the Computing Research Laboratory, University of Michigan, Ann Arbor, MI 48109.

I. INTRODUCTION

THE successful implementation of increasingly complex integrated systems has been made possible only because of the existence of increasingly sophisticated DA tools. Traditional DA research–for example, mathematical analysis of DA algorithms and data-structures, application of software structuring techniques to chip layout, and use of databases to manage the design process–has produced software tools running on conventional serial computers. These tools are limited in three fundamental ways: by the inherent complexity of the problem, by the efficiency of the coded implementation, and by the resources of the machine on which the code runs. To overcome these three limitations recent attention has focused

0278-0070/84/1000-0264$01.00

on special-purpose hardware for DA problems [1]-[20]. The strategy is to structure the machine architecture to exploit the problem's inherent parallelism, replace software with hardware, and include precisely those resources critical to the problem's solution.

This paper examines a class of architectures suitable for physical design problems that are well represented on a fixed cellular grid, and characterized by local functional dependencies among cell neighborhoods. Problems such as design rule checking (DRC), device extraction, placement, and routing have been solved in this framework. An immediate candidate for such problems is a cellular array, and indeed, advances in technology have heralded a renaissance for arrays in many applications, including DA. However, traditional two-dimensional arrays are not the only machine organization capable of efficient solution of grid-based DA problems. Architectures for solving grid-based problems have been studied extensively in fields such as image processing, pattern recognition, and mesh-based numerical analysis. Useful parallels may be drawn between architectures for cellular processing and for grid-based physical DA: each is characterized by how storage and processing power are allocated to cells in the problem.

This paper describes cellular architectures called *raster pipeline subarrays* (RPS). An RPS machine is a pipeline of subarray stages that processes a large grid in a serial cell-stream. We analyze grid-based DRC and maze-routing on an RPS architecture. As part of our examination we report the results of DA experiments performed with a cytocomputer, a research prototype in a family of existing RPS machines designed for geometric image processing [21]-[23]. This disussion sets in context the feasibility studies in [15] and expands on the benchmarks in [18]. Results presented here indicate that with respect to hardware expandability, admissible problem-size, speed, and range of application, the RPS organization is a cost-effective approach applicable to an important class of DA tasks.

The paper is organized as follows. Section II enlarges the analogy with conventional cellular applications to show how a taxonomy of cellular processors effectively categories the diverse hardware proposed/constructed to solve grid-based DA problems. The central features of RPS architectures and cytocomputers are characterized. Next, Sections III and IV analyze grid-based DRC and maze-routing in an RPS environment. Concrete implementations and performance statistics are given for DRC and routing systems functioning on the hardware. A principle goal of the experimental work is to identify performance bottlenecks, separating those generic to RPS systems from these endemic to the current hardware. Section V outlines an optimized DA machine with an RPS organization based on experience with those experimental systems. Included is an evaluation of the practical strengths of the RPS organization for these DA problems. Section VI presents concluding remarks.

II. Cellular Architectures

Research in special-purpose hardware for cellular applications spans more than two decades and has accelerated with recent advances in technology. We focus on machines that support image-processing and pattern-recognition [24]-[29]. The analogy between these problems and grid-based DA is conceptually useful, but must not be taken too far. Some issues critical to these applications and potentially influential to architectures supporting them are wholly absent from DA. From the narrow perspective of grid-based DA we construct a taxonomy of these processors emphasizing:

(1) how storage is allocated to the cells of a grid being processed,

(2) how processing power is applied to individual cells or groups of cells,

(3) how processing elements and storage elements are interconnected.

It will be shown that many grid-based DA architectures fit naturally into this scheme. In particular, the place of RPS architectures is described in this scheme.

2.1. DA Architectures as Cellular Architectures

A central problem for a cellular architecture that manipulates large grids is how to distribute all grid cells over a numerically smaller set of processors. In image processor architecture this problem has been referred to as *windowing* [27]. Because the grids representing real images span the range 10^2-10^5 cells on a side, it is generally impossible to allocate a unique physical processor to each cell; rather the grid must be manipulated in subsections or windows. For our purposes, the shapes of these discrete sections, their path to and from processing elements, and the amount of parallelism in data-movement and data-manipulation define the architecture.

Fig. 1 shows a taxonomy emphasizing these features. It has three salient points. First, because the objective here is to classify DA architectures, this scheme is just large enough to contain most of the interesting grid-based DA architectures of which we are aware; cellular architectures that have no close analogue in current DA machines (e.g., pyramid machines) are simply omitted. Second, it is explicitly a hierarchical classification in contrast to other schemes [24], [27]. Specifying a machine by its parents in the hierarchy gives its concise relationship to other machines emphasizing critical similarities and differences. Third, it places RPS machines in this scheme to show their natural relationship with other array organizations.

At the first level the hierarchy divides into two basic machine organizations. As noted in [24] there are machines whose architectures are dominated by a central bus structure or, more generally, by an Interconnection-Network (ICN) structure. The other basic organization is, as expected, the array structure. By array structure we mean specifically the existence of one or more spatially distributed arrays of interconnected processor/storage elements and the machinery to move data through these elements. Each of these two basic organizations is subdivided into two classes.

ICN structured machines are classified as using either a single bus or a routing-network. For example, PICAP II [30] employs a single high-speed bus to connect image memories,

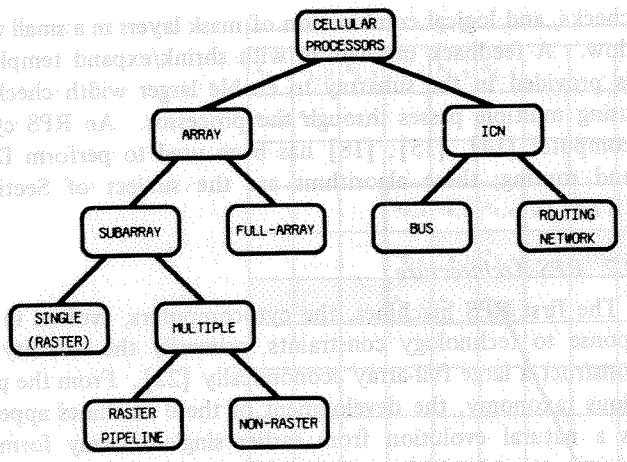


Fig. 1. Taxonomy of cellular processors.

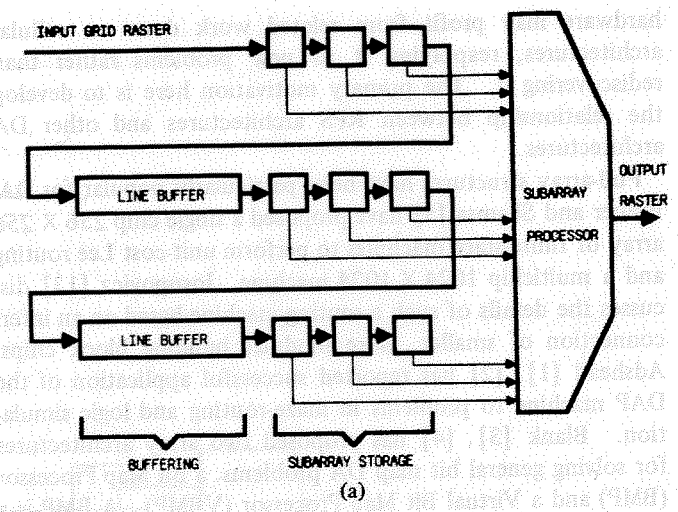


Fig. 2. Raster pipeline subarray organization. (a) Single subarray stage. (b) Pipeline of stages.

a neighborhood processor, and a filter processor. The proposed PASM architecture [31] employs multipath routing-networks to connect a set of processor/memory subsystems.

The class of array-structured machines is also divided into two subclasses. Adopting the terminology of [27], array structured machines are classified as being *subarrays* or *full-arrays*. The distinction here requires clarification, as it depends not only on structural differences, but also on the size of the array involved. The full-array would be labeled a traditional cellular array: a matrix of processor/memory pairs each connected locally to its neighbors. Modern machines in this class are typically large square arrays of simple bit-processors with storage at each node. Examples include CLIP4 [32], a 96 X 96 array with 32 bits/node; the Distributed Array Processor (DAP) [33], a 64 X 64 array with 4K bits/node; the Massively Parallel Processor (MPP) [34], 128 X 128 with 1K bit/node; and the Adaptive Array Processor (AAP) [35], based on a single chip 8 X 8 array with 96 bits/node. Some machines with large memories at each node, e.g., MPP and DAP, incorporate a notion of grid-folding; grids larger than the physical array are folded into several planes and mapped onto the storage available at each node. Some also provide for limited global communication, e.g., DAP includes an additional bus for each row and column, enabling complete row-vectors and column-vectors to be moved around the array.

The subarray class is further subdivided, and is characterized by the range of subarray sizes and the connections between distinct subarrays. A subarray is an array much smaller than the entire grid to be processed; it is a processing window. The smallest subarray is a single neighborhood, 3 X 3 on a square grid, while the largest is generally between 16 X 16 and 32 X 32. The simplest subarray organization is the class of raster single-subarrays (see Fig. 2(a)). The idea is to process the entire cell grid in a serial stream (raster order) as it passes by a subarray processor. To do this, shift-registers are introduced as buffers for a few rows of the grid. As the stream passes through the buffers and the processor, enough of the grid is present to insure that each subarray of cells eventually arrives at the subarray processor to be processed. The GLOPR machine [36] is an early example of this.

The subarray class is not restricted to a single subarray processing element; the second subclass contains the multiple-subarray machines. It too is subdivided. Because the single subarrays just discussed can output a data stream with format identical to the input stream, it is possible to connect several in a pipeline. Here the individual processors are called stages, and the entire machine is a raster pipeline subarray (see Fig. 2). This is the organization of the cytocomputer family [21], [22]. The RPS organization and cytocomputers are the subject of Section 2.2.

Multiple subarrays are not restricted to a raster input format. Nonraster organizations use several interconnected subarray memories and subarray processors to concurrently process several pieces of a complete grid. The major difference between these machines and the apparently similar ICN based machines is a matter of emphasis. Of primary interest in multiple subarrays is the physical design of the subarray buffers and processors, with their interconnections of secondary interest. In ICN structures much of the architecture is subordinated to the interconnection scheme. An example is the Preston-Herron Processor (PHP) [37], in which three cell memories communicate with 16 table-driven processors.

The purpose of this concise overview of cellular architectures is to classify the diverse set of proposed and contructed DA machines. It will be shown that many grid-based DA machines are related through the previous taxonomy. The existence of some superficial architectural similarities between these two classes of machines is not surprising given the similarities between geometric tasks like pattern recognition, and grid-based DA tasks like DRC. However, it is significant that for most of these DA machines there is a precise analogue in the cellular-computing world. This implies that future DA

hardware may profit from related work done on cellular architectures, reapplying it to new problems rather than rediscovering it. The primary motivation here is to develop the relationship between RPS architectures and other DA architectures.

Full-array structures have been particularly popular for DA. Breuer and Shamsa [5] have proposed a single chip 256 × 256 array of finite-state machines to perform unit-cost Lee routing and a multichip 1024 × 1024 machine. Iosupovicz [13] discusses the details of such a routing machine based on an interconnection of smaller, more modular building block chips. Adshead [1], [2] has reported successful application of the DAP machine to problems in maze-routing and logic simulation. Blank [3], [4] has proposed two array architectures for solving general bit map DA problems: a Bit Map Processor (BMP) and a Virtual Bit Map Processor (VBMP). A BMP is a standard array of 1024 × 1024 simple processors each with memory. A VBMP is a much smaller array, e.g., 32 × 32, with special hardware to fold a larger virtual grid onto the physical array. Large memories reside at each array node, and the edge and corner nodes include special mechanisms for dealing with border effects. Each cell in the array can also be individually addressed via row and column lines to provide some global communication. Simulations for DRC and simple maze-routing have been constructed, and a 4 × 4 TTL prototype is being fabricated. Hong *et al.* [10], [11] describe a Wire Routing Machine based on an array of commercial microprocessors, which also incorporates provisions for folding large problems onto the array. An 8 × 8 prototype with 15K bytes/node is operational, and claims are made that a 32 × 32 structure would likely suffice for all real problems. Sophisticated global-routing algorithms have been implemented and run on modest test grids [16].

Placement algorithms have also been considered in a full-array environment. The basic idea is to peform many concurrent pairwise interchanges among adjacent modules until total wire length is minimized. The restriction to adjacent interchanges enables each node to compute the change in wire length from one interchange; the array structure enables concurrent interchanges. Ueda *et al.* [20] and Chyan and Breuer [7] describe similar array machines for placement. (Outside the area of grid-based DA, Kane and Sahni [12] describe a systolic array organization for DRC using polygon edges as the basic data element.)

Bus structured machines have also been constructed. Damm *et al.* [8] have built a Lee-routing engine by modifying a commercial minicomputer. A special cell-memory, a hardware "kernel" of routing operations, and an interconnecting bus were added to optimize performance. Successful operation with PC boards has been reported. (Outside the area of grid-based DA, the Yorktown Simulation Engineer (YSE) [17], a compiled logic-simulator, is an ICN structured machine. Up to 256 logic processors, each storing and updating logic elements, communicate over a cross-bar switch.)

Subarray architectures also appear. Seiler [19] has developed a hardware implementation of Baker's raster DRC [38] using a raster single-subarray. The processing section uses a few custom PLA-based chips to perform width checks, edge checks, and logical combination of mask layers in a small window. A feedback mechanism with shrink/expand templates is provided in the subarray to enable larger width checking using multiple passes through the processor. An RPS cytocomputer [14], [15], [18] has been used to perform DRC and routing; these algorithms are the subject of Sections III and IV.

2.2. RPS Architecture

The first RPS machines, the cytocomputers, evolved in response to technology constraints, primarily the inability to construct a large full-array economically [22]. From the previous taxonomy, the development of these machines appears as a natural evolution from earlier single-subarray forms.[1] We propose the RPS class as the appropriate abstraction of the novel features introduced by the cytocomputers in image-processing work. This section characterizes the inherent parallelism and functional limitations of the class, and describes the cytocomputers.

2.2.1. Local and Global Issues in RPS Architecture

Fig. 2 illustrates the central feature of the RPS organization: the processing of a cellular grid as a stream of cells moving through a pipeline of subarray processors. RPS hardware/software design partitions into two issues: *local* issues that concern the processing of individual cells in each stage, and *global* issues that concern the movement of entire grids through the pipeline of subarray stages.

Local issues pertain to subarray stage design. The three basic components of one stage are (1) the line buffering scheme, (2) the subarray storage, and (3) the subarray processor. The buffering insures that each subarray of data arrives at the subarray processor as the cells pass through the stage. Subarray storage size and subarray processor function are arbitrary. The sequence of operations performed by one subarray processor is called one *subarray-computation;* a typical sequence consists of examining the cells in the subarray storage, computing one or more new cell values, and relocating the new cell values. Relocation is accomplished by retaining cells internally as temporary variables, injecting them into the line buffering scheme to replace cells in the input grid, or injecting them into the output stream to form the input grid for the next stage.

Global issues pertain to the physical properties of the pipeline, as distinct from the functional properties of a stage. They include pipeline rate, pipeline length, and pipeline control. Two properties of a stage are defined globally: the stage cycle time, t_{stage}, which is the time between the output of *consecutive* cells from a streaming stage, and the stage latency, t_{lat}, which is the time required for a *single* cell to pass completely through the stage. Note that t_{stage} is a design parameter for the stage, while t_{lat} is a function of the grid size being processed and the subarray/buffering scheme. Fig. 3 illustrates this relationship assuming a 3 × 3 subarray. Each stage completes one subarray-computation in t_{stage} time units. The

[1] Taxonomies have been proposed in which the cytocomputers are inappropriately categorized as completely disjoint from all other cellular organizations [24], [27].

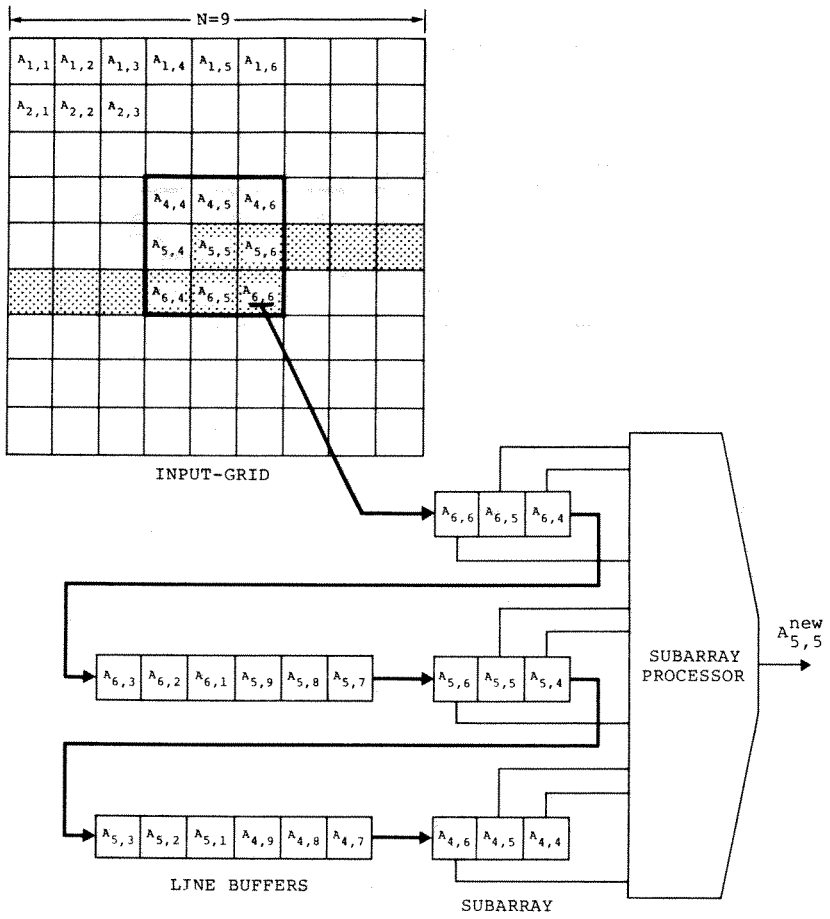


Fig. 3. Grid moving through RPS Stage.

movement of the grid through the stage can be visualized as a 3 X 3 subarray window moving across the grid [22]. After cell $A_{6,6}$ in the cell stream of Fig. 3 has entered the stage, the complete 3 X 3 neighborhood of cell $A_{5,5}$ is stored in the stage subarray. The subarray processor then performs a single computation and, in this example, simply computes the new value $A_{5,5}^{\text{new}}$ and injects it into the output stream. All this activity occurs in one t_{stage} cycle. On the next cycle, $A_{6,7}$ enters the stage and the computed value $A_{5,6}^{\text{new}}$ is output. In this example the latency is the number of cycles between $A_{i,j}$ entering and $A_{i,j}^{\text{new}}$ leaving and is precisely the number of cells in the cell stream between $A_{5,5}$ and $A_{6,6}$ (shaded in Fig. 3). For an image N cells wide $t_{\text{lat}} = (N+2)\ t_{\text{stage}}$. A pipeline can be viewed as a series of 3 X 3 stage windows following each other across the image, each processing the previous stage's output.

Pipeline length affects the decomposition of algorithms into individual steps performed in each stage. Consider an algorithm composed of many repetitions of one processing step. If this step can be realized as K subarray-computations, then a KS-stage pipe performs S of these steps on each grid pass. If the pipeline is too short, $S < K$, then $\lceil K/S \rceil$ passes are required to realize the step, and each stage performs a different function on each pass. Long pipelines are generally desirable, but have a longer total latency, which is defined as the time until the first result appears at the last stage of the pipe. However, this is typically a minor effect. For example, the time, t_{pass}, required to pass a grid with N columns and M rows through an S-stage pipe of 3 X 3 subarray stages is

$$t_{\text{pass}} = S(N+2)t_{\text{stage}} + MNt_{\text{stage}}$$
$$= \text{pipe\_length} \times t_{\text{lat}} + \text{grid\_size} \times t_{\text{stage}}. \quad (1)$$

The complete processing time is the sum of the total latency and the time to flush all grid cells through the pipe. Total latency is the sum of all stage latencies. For large grids, latency is roughly the small fraction $S/(S+M)$ of total processing time.

The principle issue in pipeline control is the implementation-dependent overhead associated with managing pipeline data movement and stage programming. As in most full-array machines, the existence of some global controller is assumed, the task of which is to synchronize pipeline data movement, stage programming, and the interface to the data source.

2.2.2. *Cytocomputer Architecture*

Cytocomputer stage design is motivated by the model of computation embodied in a full-array of state-machines. One pass of a cellular grid through one subarray stage is intended to emulate one state-transition in the full-array. In one step, all nodes in the full-array examine their neighbors and all simultaneously change state; no information ever moves fur-

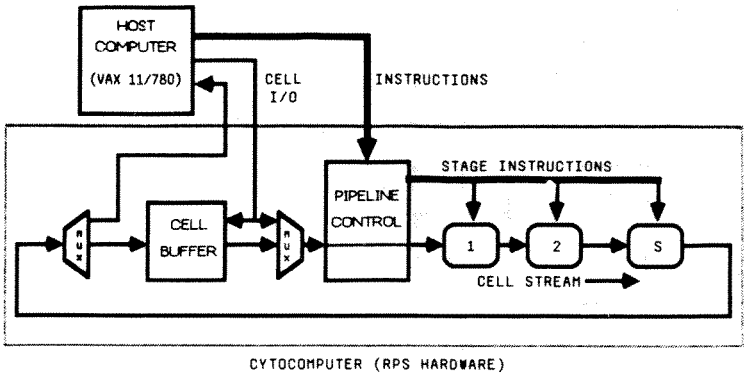


Fig. 4. Experimental cytocomputer environment.

ther in one step than the immediate neighbors of one code. Similarly, each cytocomputer stage is *designed* so that no information ever moves further in one subarray computation than the diameter of a single subarray.

Each stage uses the 3 X 3 subarray just discussed (Fig. 2(a)) and processes a stream of 8-bit cells. The following steps outline the processing that occurs in one stage in one t_{stage} clock period (see [21] for details):

Step 1: A cell enters the stage, is biased (normalized) or has some of its bits masked.

Step 2: The 9 cells of the stored subarray are transformed into a 9-bit vector. Each bit is a true/false decision about each cell, the result of a threshold comparison with an arbitrary constant. This vector is an address into a table.

Step 3: A new cell value is selected from the following: the old center value in the subarray, the largest value in the subarray, or the value in the table addressed by the 9-bit address from (2).

Step 4: The cell from (3) is unbiased and unmasked, i.e., Step 1 is selectively undone. It is possible to alter only a single bit in this cell as a function of all the bits of the subarray.

Step 5: The cell from (4) addresses another table to produce a modified cell value. This table is used typically for Boolean operations on bits or for further arithmetic biasing.

Step 6: The cell from (5) is injected into the output stream. It occupies the same location in the output grid as the center cell of the current subarray.

Consistent with the model of one state-transition, the computed value of Step 6 is never retained internally for further computation. This property of the cytocomputer stage will be referred to as *statelessness*. Statelessness is a design choice, and is *not* fundamental to RPS organizations. The bias-values, masks, constants, and tables in Steps 1-6 are the instructions for a single stage. Because the stage depends heavily upon table look-up the perceived format of the bits in each cell is arbitrary.

Cytocomputers exist in MSI and LSI implementations. Existing MSI versions have between one and ten stages. LSI stages have been fabricated with t_{stage} ranging from 100 ns to 2 μs. Fig. 4 shows the global configuration used for the experiments reported in this paper. Table I summarizes the performance characteristics of this system.

TABLE I
HOST/CYTOCOMPUTER CHARACTERISTICS

Component	Description
RPS Machine	ERIM Cytocomputer II
Pipe Length	3 stages (expandable)
Subarray Stage	3X3 8-bit cells
t_{stage}	2μs
Cell-buffer	256K-byte
Host	DEC Vax 11/780 + Unix

The system is configured as a host with an attached cytocomputer. The host sends instructions to the global controller, a microprogrammed unit that programs the stages and manages the pipeline. The controller provides a low-level instruction set (that is, one not dedicated to a specific application) to process grids through the pipeline. In a typical processing step, the host deposits a grid in the cytocomputer cell-buffer, instructs the controller to send it repeatedly through the pipeline and back into the buffer, then retrieves some portion of the processed grid from the buffer.

The ideal pipeline processing time for 3 X 3 stages given by t_{pass} in (1) is affected by the system configuration. The actual time to process an M row by N column grid K times through an S-stage pipe (i.e., KS computation steps) can be modeled as

$$T_{\text{sys}} \approx t_{\text{pi}} + St_{\text{si}} + Kt_{\text{pass}}. \tag{2}$$

Kt_{pass} is the time for K pipeline passes. The other terms are lumped delays: t_{pi} is the time to initialize the pipeline and St_{si} the time to set up all S stages. These terms arise because of the nonneglibible time to dispatch low-level instructions from the host to the controller. Although small, these delays are always larger than t_{stage}, e.g., 15 ms versus 2 μs on our hardware, yielding an effective cost of several thousand subarray-computations for each controller instruction.

A complete implementation of DRC or routing includes the software running on the host and the instructions in each stage. Earlier DA studies were performed in an interactive image-processing environment [15] that was neither flexible enough nor fast enough for large, production DA work. Algorithms reported in this paper run in a new environment developed for RPS DA studies.

III. Design Rule Checking

The design rules are a set of geometrical constraints that the masks of the wafer fabrication process must satisfy. The two

general approaches to the implementation of DRC's reflect the data-structure chosen for the IC mask. Geometric-shapes checkers perform checks on masks represented as sets of intersecting polygons or rectangles [39]. Grid-based checkers work with a mask represented as a grid whose cells are labeled according to the presence or absence of particular mask layers. Both nonuniform grids (the chips are dissected into contiguous rectangles of arbitrary size [40]) and uniform grids (the cells are squares) have been used. Raster-scan approaches have been developed [38], [41] that access a uniform grid in raster order and check *local* design rules; the idea is to pass a small window over the grid and identify the local violation-patterns appearing in the window. This latter approach motivates a DRC on RPS hardware.

Roughly speaking, a design rule checker performs the following on mask features:

Connectivity Resolution: Merge discrete shapes on the same layer into a single larger shape if they overlap; connectivity is similarly assessed across several layers, e.g., across contact windows.

Layer Combination: Create new layers from Boolean combinations of existing layers, e.g., the intersection of several layers.

Tolerance Checks: Determine whether a local group of shapes on one or more layers satisfies some spatial constraint, e.g., corner/edge separation, incursion, inclusion, exclusion, size, area, perimeter.

When a mask is represented as a grid, local connectivity and layer combination are easily computed. Overlapping shapes automatically become a single entity as the cells within the shapes are labeled as belonging to a particular layer, and Boolean combinations performed globally across several layers are simply performed on each cell in the mask. Global connectivity is harder to resolve because it involves propagating nodal connectivity information around the cells of the grid; such global data movement is inefficient if we are restricted to local processing of cells. Tolerance checks are more interesting since they require not just cell by cell processing but also pattern recognition operations on spatially distributed group of cells.

Accordingly, this section describes tolerance checks implemented on a cytocomputer. The checks are applicable to the NMOS design rules of [42]; we employ a uniform grid of $\lambda \times \lambda$ cells where λ is the basic length unit used to express design rules.

3.1. Formalism for DRC Algorithms

We outline a formalism [43], [44] that introduces useful operators applicable to binary-images, and hence to masks, and also introduces an algebraic framework in which to manipulate them. The approach provides a convenient notational tool for DRC algorithms. It treats a binary image as a set of points (the opaque points on a transparent mask) where a point is an ordered pair of coordinates in the grid.

If A and B are masks, and hence sets, the usual mask-to-mask Boolean operators appear in set-theoretic form as intersections, unions, etc. Next, define the *translation* of a set A by a point p to be $A_p = \{a + p \mid a \in A\}$; if A is regarded as a geometric shape, A_p is A with its local origin moved to p. With translation define the two primitives of *dilation* $\oplus$, and *erosion* $\ominus$ as follows:

$$A \oplus B = \bigcup_{b \in B} A_b, \quad A \ominus B = \{p \mid B_p \subseteq A\}.$$

The dilation $A \oplus B$ is the union of translations of A by points from B. The erosion $A \ominus B$ is the set of points to which we can translate B and still have it contained in A. Loosely, dilation and erosion are formal generalizations of the intuitive ideas of expanding and shrinking. However, erosions and dilations are defined for arbitrarily complex sets A and B, whereas expands and shrinks are usually specified with simple patterns. This formalism is useful because these operators are *local*, and can be reduced to a sequence of subarray-computations.

In an expression such as $A \ominus B$, A is typically a complete mask, and B is a small figure such as a circle or rectangle. If B fits inside one subarray, then $A \ominus B$ is implemented in one subarray-computation as grid A streams through one RPS stage. Operations with a larger more complex B will be decomposed into a sequence of smaller operations each suitable for execution in one stage. The algebra includes identities which permit simplifications similar to those done in Boolean algebra. For example, to compute $A \ominus B$ when B is a dilation of subarray-size figures $B = B_1 \oplus B_2 \oplus B_3$, it suffices to compute $(((A \ominus B_1) \ominus B_2) \ominus B_3)$ which can be done directly in three stages.

Two operators defined as compositions of the dilation and erosion primitives are also useful. These are called *opening* and *closing*. If X and S are sets, X opened by S is $X_S = (X \ominus S) \oplus S$, and X closed by S is $X^S = (X \oplus S) \ominus S$. Again interpret sets X and S as geometric figures. Then X opened by S is the set of points in X touched by S as S slides around inside X. Closing has a similar interpretation for the complement of X. Fig. 5 demonstrates all these operations.

3.2. DRC Algorithms

We illustrate a cellular DRC by constructing an algorithm for a width-3λ tolerance check on an orthogonal mask. This check identifies regions of a mask less than 3λ wide. A single mask is a binary image occupying one bit in each 8-bit cell of the input grid. The algorithm produces another binary image, stored one bit per cell, indicating the locations of width violations. In a complete DRC all masks are stored in these parallel bit-planes. The 8-bit cytocomputer datapath allows up to eight masks to be processed simultaneously.

The algorithm is based on the simple observation illustrated in Fig. 6, which shows a mask on which we wish to perform a width-W check. Slide a disk of diameter W around inside the mask to all possible locations at which it may be completely contained (Fig. 6(b)). While it slides, note those points covered by the disk and trace the path of its center. It is clear the disk should not pass through regions which are too narrow, i.e., regions which fail the width test. Except for some square-corner effects, those regions left uncovered all violate the width test (Fig. 6(c)). Note also that the region traced out by

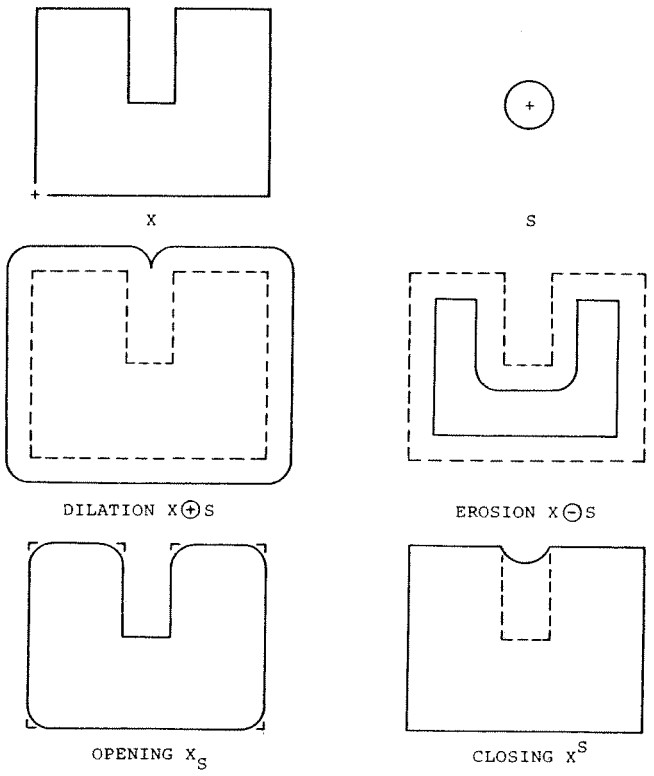


Fig. 5. Formal mask-image operators. Origin in shapes X and S marked "+".

the center of the disk is not connected across the diagonal neck of the mask.

With these observations one can construct an executable width-3λ algorithm. First define the following geometric shapes. Let M be the mask-image to be checked. Because a real mask has square corners, replace the diameter-3 disk with $S(3)$, a 3 X 3 square. The set of points covered by $S(3)$ as it slides in M is precisely the opening $M_{s(3)}$. The region traced out by the center of $S(3)$ is the erosion $M \ominus S(3)$; call this C. Define $Q(3)$ to be a cellular approximation of a radius-3 quarter-circle, the first quadrant of a radius-3 circle centered at the origin. Breaks in C are characterized by diagonally adjacent corners of components of C separated by no more than 3λ. Mark one set of corners, T_{NE}, the northeast corners, and then search nearby each for an unconnected southwest corner; each region searched takes the shape of $Q(3)$. The algorithm can be outlined as follows:

Width-3λ Test:

Step 1: Open M with $S(3)$. Areas of M not in the opening are errors.

Step 2: Erode M by $S(3)$ to get C. This is the path traced by the center of $S(3)$.

Step 3: Tag the northeast corner of each component of C; call these points T_{NE}. This prepares to identify the regions of M which restrict the passage of $S(3)$ by finding the breaks in C.

Step 4: Dilate T_{NE} by $Q(3)$ over M. This dilation intersects the southwest corner of another region of C if and only if a break has occurred along a northeast/southwest axis. Mark the points in this intersection as errors; they indicate a diagonal width violation.

Steps 5, 6: Similar to Steps 3, 4 to find errors along the northwest/southeast axis.

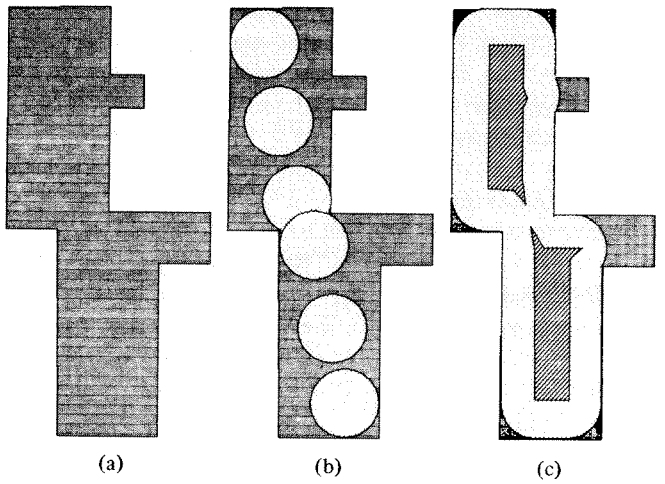


Fig. 6. Geometric basis for width check. (a) Mask feature. (b) Width-W disk slides through feature. (c) Regions covered by disk (dotted) and traced by center (striped).

This algorithm tags any region smaller than 3λ X 3λ, and tags the *north* side of each pinched-neck diagonal width violation. It is generic because replacing $S(3)$, $Q(3)$ by $S(W)$, $Q(W)$ gives a width-Wλ check. Note in the case $W = 3\lambda$ a radius-3λ quarter-circle is approximated as a 3 X 3 square.

This algorithm illustrates the utility of the formalism presented in the previous section. Algorithms are designed as sequences of operators working on geometric objects; altering the size of these objects does not alter the basic algorithm. These operators are formally decomposed into a set of concrete subarray-computations for the hardware. This generally frees algorithm design from the tedious detail of pattern specification inherent in pattern matching approaches.

3.3. DRC Experimental Results

The width-3λ check requires 8 subarray-computations to run. Fig. 7 shows the results of running the algorithm on a 64 X 64 test pattern. All errors are correctly detected. On a 3-stage cytocomputer this test required about 0.5 s to run. As a more realistic test, we stacked this figure 64 times to yield a 4096 X 64 grid, representing a slice of a real chip. This check required 5.1 s to run: 2.3 s of pipeline processing and 2.8 s to transfer the grid between host memory and cell-buffer. Depending on the sophistication of the checking needed and extra processing required to put meaningful labels on errors, a DRC for this class of rules on five NMOS mask layers will take between 150 and 250 subarray-computations. Table II estimates the time to run such a DRC on a 4096λ X 4096λ chip for different pipeline lengths. We assume processing is done in contiguous vertical strips, each 64 cells wide and overlapped by 4 cells to avoid errors, and sum all strip times. We use the transfer time for the single strip test, and assume that the host generates these strips as fast as the pipeline requires, i.e., with negligible delay between strips.

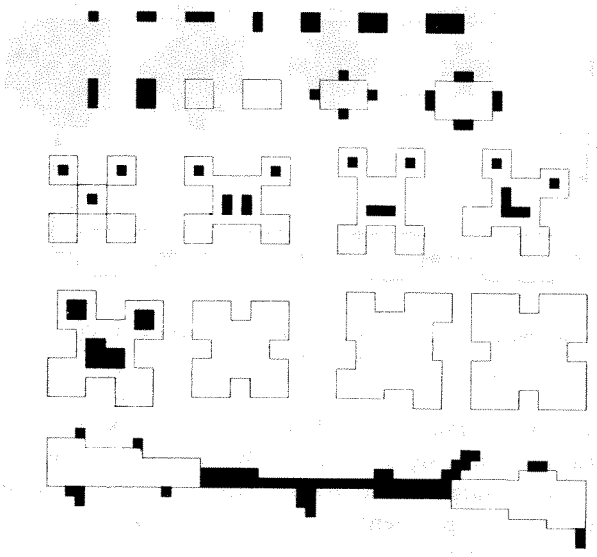


Fig. 7. Result of width-3λ test. Errors superimposed in black over mask features.

TABLE II
ESTIMATED DRC TIME, 4096λ × 4096λ IC

Pipeline Length	Estimated DRC Time in Seconds	
	150-Step-DRC	250-Step-DRC
1	5621.	9239.
10	737.2	1099.
50	305.8	380.9
100	266.9	304.0
250	230.7	231.7

3.4. *Enlarging the Scope of Application*

It is clear from Table II that with a modest pipeline (10-100 stages) a chip represented as a grid with a several million cells can be checked against rules equal in complexity to [42] in 5-20 min. However, two potential difficulties arise if this methodology is extended to very large chips: the *size* of a large layout, and the *quality* of a DRC for such a layout.

Consider a large chip drawn on a cellular grid. If f is the minimum feature size of the given layout, some commercial microprocessors drawn on a grid of $f \times f$ squares require 10 to 15 million cells [45]; for reference, a 300 mil × 300 mil chip drawn on a 1-μm grid has about 60 million cells. Given that a complete DRC for an entire chip or just one strip will require several passes through a practical pipeline (10-100 stages) the question is how to generate, store or regenerate the required grid. We suggest two solutions:

1) Dedicate a large disk and a large cell-buffer to the pipeline. We can then generate the complete chip grid, store it on the disk, and process either the whole chip or contiguous strips. The key constraints are the mask-generation rate of the host, the I/O speed of the disk, and the size of the cell-buffer. As an example, consider a system with the parameters of Table III. A DRC using contiguous strips then requires roughly 1500 s for the data-transfer and t_{pass} pipe-processing to make the grid initially, and 69 overlapped strips at 6 s per disk buffer-pipeline-buffer-disk DRC cycle. About 66 percent of this is the time needed for the host to make the mask-image. Expected overhead will likely increase this by at least 50 percent.

2) Dedicate special hardware to the task of mask-image generation. The preceding analysis assumes that the mask-image is fully instantiated on the disk with no encoding. If some pre- and post-processing is available at each end of the pipe, a simple run-length coding of each line will reduce storage requirements and hence data transfer time. The single-chip polygon-to-raster converter discussed in [19] would enable a polygon-based mask to be rapidly streamed through an RPS machine. More complex schemes [46], [47] may also be possible.

TABLE III
EXAMPLE DRC SYSTEM PARAMETERS

System Parameter	Description
Mask-Image Size	64M-bytes (8096 × 8096)
Host Processing Rate	64K cells/sec
Disk Transfer Rate	1M-byte/sec (average)
Pipeline Length	64 stages
t_{stage}	1μs
Cell-Buffer Size	1M-byte
Complete DRC Length	256 steps

We conclude that some combination of dedicated storage and special hardware is sufficient to manage the size problem for large chips.

The second and perhaps more serious problem is the issue of quality: a cellular DRC imposes restrictions on the layout of a chip and on the geometry-rules to be checked. Layouts represented with a polygon data-structure may contain features of arbitrary shape and arbitrary size; polygon checkers can usually resolve electrical connectivity and use this information in tolerance checks. Cellular checkers restrict the layout to a uniform grid, restricting all features to orthogonal boundaries (no oblique lines) and all distances to multiples of the unit cell size. Electrical connectivity is not usually available during tolerance checking, resulting in nuisance errors. These issues are addressed below.

1) Many layouts do not require features of arbitrary size. The popularity of simplified design rules in several technologies [48], [49] suggests that grid-based may be equally acceptable.

2) Some oblique lines are representable on cellular grids at the expense of increased storage or processing; see [19], [40]. Most layouts are primarily orthogonal. It has been argued that obliques are a questionable luxury that may become too expensive to check in the face of VLSI complexity [50].

3) Lack of electrical connectivity information is not unique to cellular checkers. Other recent university systems [51], [52] have not implemented obliques and/or connectivity in order to focus on other research directions. Connectivity extraction is not impossible here but the overhead to store a node label in each cell is expensive. Only rules based explicitly on electrical information, e.g., fanout rules, are compromised here. Complex geometrical rules can always be checked on a cellular grid; the only drawback is the possibility for false errors from connectivity pathologies.

Given the existence of large designs representable on cellular grids, the strong demand for a comprehensive chip DRC, the long execution time for a typical software DRC (often measured in days on a mainframe [53]), we conclude a reasonable quality DRC for such designs executing on RPS hardware in 10 min to 1 h is a potentially useful DA tool.

IV. Routing

Maze-routing is a natural application for a cellular architecture. The continuing viability of Lee-type routers in both PC board and LSI applications is indicated by recent surveys [54], [55]. Much research has focused on modifications of the basic Lee algorithm [56] to improve efficiency [57]-[59]. This section describes experimental one-layer and two-layer routers running on a cytocomputer. These are complete implementations; the host accepts a net-list and produces a grid with embedded wires. The cytocomputer is the inner-loop: its only job is to find a path between points. The host handles unroutable nets, etc.

4.1. Routing Algorithms

The routing of a single source-to-target path has three phases:

Wavefront-Expansion: Iteratively expand from the source a wavefront of labeled cells; cells on one labeled frontier begin a path to the source and are equidistant from the source. Continue until the target cell is reached. This can be realized in parallel across all cells. Each cell depends only on immediate neighbors.

Back-Trace: Trace a path from the target back to the source along labeled cells. This is wholly sequential, best done on the host.

Clean-Up: Remove extraneous labeled cells and relabeled the new path as a future obstacle. This is also parallel.

Elementary wavefront-expansion in an RPS pipe is shown in Fig. 8. Assuming the labeling of one cell is one subarray-computation, the key idea is that each stage adds one layer of cells to the wavefront as the grid passes by. Thus the L expansions needed to find any path of length L require $\lceil L/S \rceil$ passes with an S-stage pipe.[2] This addition of one layer of cells to a wavefront is one *wave-expand step.* The cytocomputer's 8-bit datapath precludes labeling with weights or penalties. We treat the implementation of unit-cost routers.

The one-layer router is based on the cell encoding of [5] where an expanding wavefront is labeled with source-pointing arrows. The activity of each grid cell is encoded in the alphabet {*source, target, free, blocked,* $\leftarrow, \uparrow, \downarrow, \rightarrow, T\leftarrow, T\uparrow, T\downarrow, T\rightarrow$}.

The wave-expansion phase places an arrow in each *free* cell if it is bordered by an active wavefront; the new arrow points to the active cell. Any label in {$\leftarrow, \uparrow, \downarrow, \rightarrow$, *source*} can be on a wavefront. When more than one labeling may be chosen, we choose directions in the order $\uparrow, \leftarrow, \rightarrow, \downarrow$. This is implemented in one cytocomputer stage, i.e., an S-stage pipe adds S layers to a wavefront. Ideally, this step is repeated just until the *target* cell is labeled with an arrow. However, *target* may actually be reached in the middle of the pipeline and we cannot simply stop the raster stream. Hence, *target* will likely be overrun and a few extraneous cells will be labeled. The issue is how to determine when *target* is reached. On the current hardware the best solution is to add this operation to the pipeline controller's microcode, allowing it to check the buffer after each pass and signal the host. We use a slower approach and return the target cell to the host after L expansions, where L is the expected path length. The back-trace phase traces the source-pointing arrows from the target back to the source, attaching a T to each cell on the unique path traced. The path is thus encoded $\{T\leftarrow, T\uparrow, T\downarrow, T\rightarrow\}$. The host performs this trace by returning some of the grid; this operation also should be added to the controller's microcode. The clean-up phase labels the new path as an obstacle and removes all other cells labeled during expansion; it requires one cytocomputer stage.

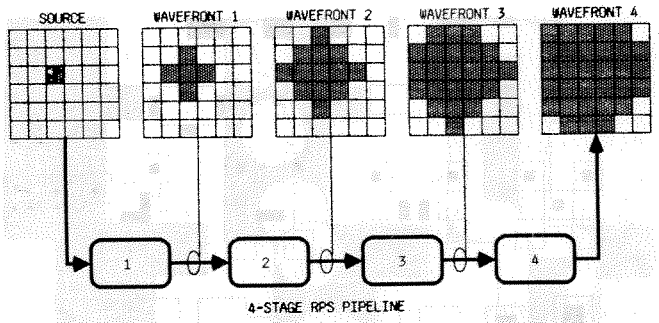


Fig. 8. Elementary RPS wave-expansion. One stage adds one layer to wavefront.

The two-layer router is similar, but uses a larger alphabet. It assumes preferred horizontal and vertical layers connected by vias. To reduce vias, small jogs are allowed in the nonpreferred direction. To reduce complexity, a strict rule is imposed: one wire segment can jog just once, and must change layers to jog again; vertical segments can jog 1 unit, horizontal segments 2 units. This permits a small 3-stage wave-expand step. Back-trace is done as before. Clean-up now also performs via-exclusion, labeling cells as blocked to avoid illegal via adjacencies; it requires 2 stages.

One additional global consideration must be addressed. It is clearly inefficient to process the entire grid for each wire because most cells will be inactive. Instead, we expand incrementally in a sequence of increasing frames bounding the active wavefront. We expand until we reach the boundaries of a frame, increase the frame-size, and repeat. This avoids processing inactive cells; [18], [60] discuss optimal framing sequences. A complete 2-point router has this outline:

```
ROUTE:
    estimate spatial extent of net;
    compose framing sequence;
    while( more frames ) {
        expand in frame;
        if ( target reached ) {
            return frame to host;
            back-trace;
            return frame to hardware;
            clean-up;
            STOP.
        }
        else next frame;
    }
```

Multipoint nets are handled by relabeling each net-segment as a source and expanding again.

[2] This results from a stateless stage. If labeled cells can also *replace* old cells in the stage buffering then one pass through one stage could find entire north-to-south and west-to-east net segments [60]. Also, in practice unused stages (L mods $\neq 0$) are disabled and pass the grid with negligible delay.

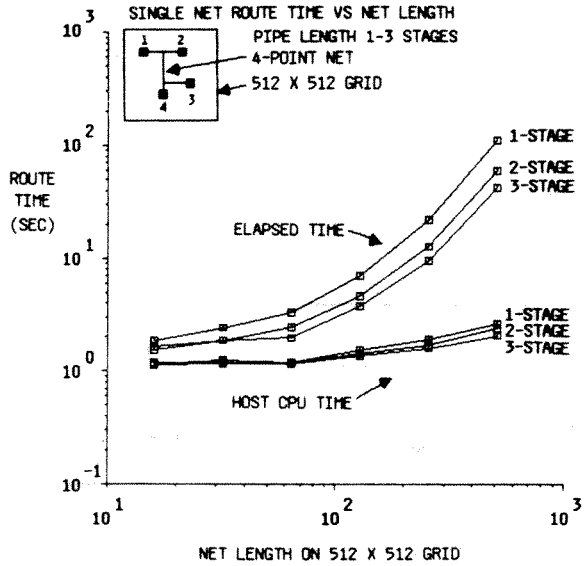


Fig. 9. 1-layer routing time versus pipe length and net length. 4-point nets of identical shape are placed in 512 X 512 grid.

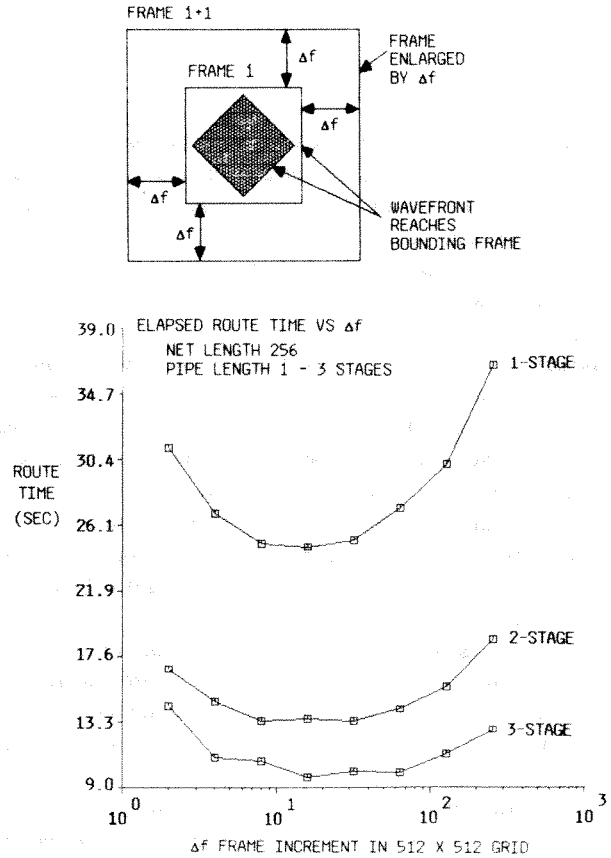


Fig. 10. Optimal framing for 1-layer router. Plot shows time to route 2-point net of length 256 on 512 X 512 grid versus frame increment Δf. Δf is roughly how many wave expands are done per frame. Δf too small creates overhead to manage frames; Δf too large expands mostly inactive cells. Suboptimal framing increases routing times.

4.2. Routing Experimental Results

Single layer routing time versus pipe length and net length is shown in Fig. 9 for identically shaped 4-point nets. As expected, time increases with net length, but decreases roughly linearly with pipe length. The test used the results of Fig. 10 which shows the effects of varying the frame increment for one net. As a more realistic test, we compared our one-layer router against the benchmark and software maze-router of [4] for several machines. The resulting time (Table IV) is superior to all but the large mainframe. Table V gives the result of routing a PC board with the two-layer router.

TABLE IV
1-LAYER ROUTER BENCHMARK. ROUTE 200 2-POINT NETS, AVG. LENGTH 170, ON A 512 × 512 GRID. CPU TIME FOR HOST + HARDWARE REFERS TO HOST TIME; FOR FAIRNESS, ALL MACHINES ARE LIGHTLY LOADED

Machine	mips (approx)	Elapsed Time (seconds)	CPU Time (seconds)
Amdahl 5860	12	847.0	321.9
Vax 11/780 + 3 Cyto stages	-	1939.7	235.3
Vax 11/780	1	5006.2	3985.0
Vax 11/750	0.6	8587.3	8253.0
Apollo DN 600	0.5	8994.4	8910.2

TABLE V
LAYER ROUTER BENCHMARK. ROUTE 2-POINT NETS IN 2-LAYERS ON PCB

Parameter	Value
PCB Size	200×240 cells (10×12 inches)
Nets Tried	412
Nets Completed	372 (91%)
Total Wire Length	23069 cells
Host CPU Time	335.2 sec
Elapsed Time	2039.8 sec

4.3. Enlarging the Scope of the Application

With a modest pipeline (10–100 stages) the current machine can quickly route many PC boards and gate-arrays (up to 1000 X 1000 grid). Assuming times decrease linearly with stages as in Fig. 9 (i.e., total t_{pass} time decreases), a 32-stage machine with the same host overhead, about 1.5 s/net, can place 1000 4-points nets of length 128 in about 30 min. If overhead were reduced to 100 ms/net, e.g., by doing back-trace in the RPS hardware to avoid returning the grid to the host, then a 32-stage machine with $t_{stage} = 1$ μs executes the same task in roughly 3 min. Extensions to larger grids can be accommodated with a larger cell-buffer. Extension to more complex routing schemes is more difficult. Although the restriction on two-layer jogs can be removed at the cost of more stages, more complex schemes require a different stage design.

Alternatives such as channel-routing have in many applications supplanted maze-routers. Although maze-routers offer a wide range of routing performance their slow execution rate restricts them to those final connections unroutable by other means. However, an RPS maze-router removes this time penalty and makes this an efficient scheme for all connections.

V. RPS ARCHITECTURES FOR DA

Cytocomputers are RPS machines designed as image-processors; they are not optimal for these DA problems. This section suggests a design for an RPS stage more closely matched to the DA applications previously described. We also discuss the merits of RPS systems with respect to practical considerations for selecting DA hardware.

5.1. An Optimized DA Architecture

There are three essential characteristics of grid-base DA problems: a wide range of grid sizes-up to 10^8-10^9 cells, a

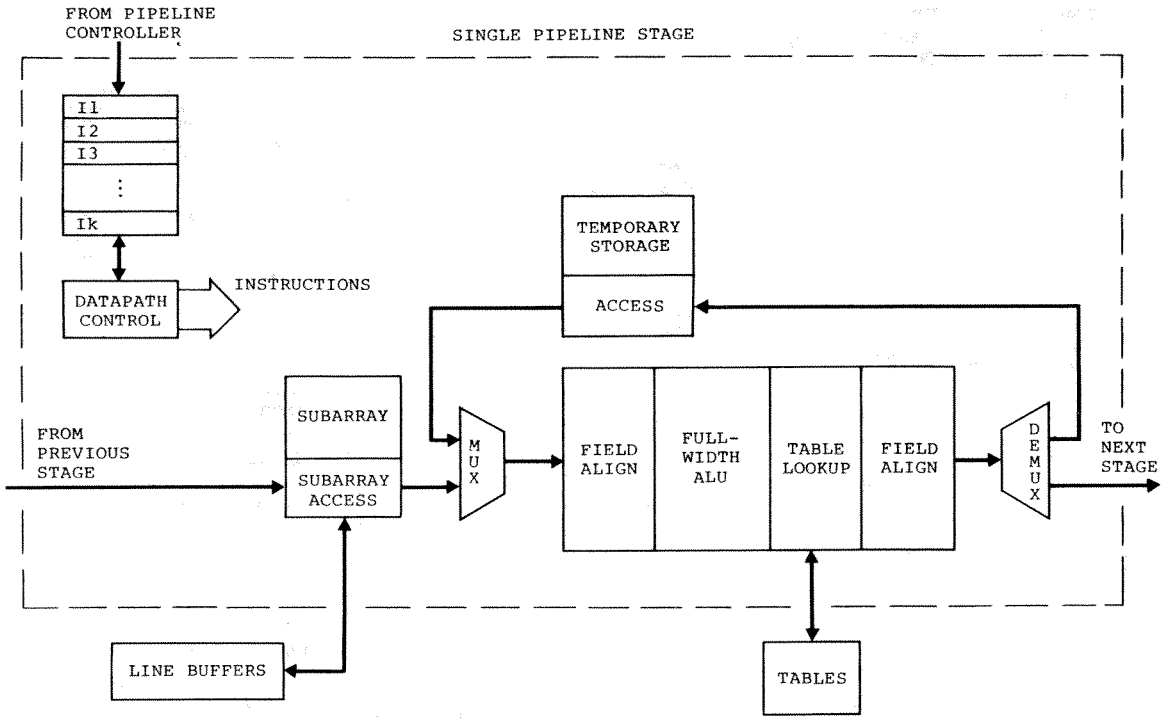


Fig. 11. RPS stage for DA.

wide range of data—bits, fields, integers—required in each cell, and a wide range of processing—pattern recognition, bit-manipulation, integer arithmetic—required on each subarray of cells. The 8-bit table-driven structure of current cytocomputers is insufficient for these problems. Moreover, the table-driven model does not simply scale up to wider data-paths. Table look-up is impractical for more than 12 or 13 bits, and hence, subarray-computation based only on direct table look-up is impossible for wider data. Arithmetic capabilities are limited in current cytocomputers. This section suggests a stage structure to handle these problems.

An algorithm is realized on RPS hardware as a sequence of stage operations. The number of pipeline passes to implement the algorithm dominates its execution time. To minimize this time, it is desirable to incorporate as much hardware in each stage as is necessary to perform each algorithm step in one stage. For example, in a DRC it should be possible to perform pattern recognition steps on several independent mask-planes simultaneously in a single stage. Also, a router should perform one wave-expansion step with arbitrary integer-weights/pen-alites in one stage.

Fig. 11 shows the structure of such a stage. It resembles a cytocomputer stage in that there is subarray storage, line-buffering, and a datapath using table look-up. However, the following new features are incorporated:

Wide Datapath: 24-32 bits wide in all storage and processing sections to support several data formats in each cell.

Subarray Access: with a 32-bit datapath the subarray is configured as a 3 X 3 X 32 array of bits, accessible as nine 32-bit words and thirty two 9-bit mask-planes. This supports pattern processing steps on independent mask-planes.

ALU, Field Manipulation, Table Look-Up: The datapath has a full-width ALU with complete arithmetic capabilities. Table look-up is still provided but only for the low-order bits of the datapath; 12-13 bit look-up is practical. To line up data for the table, barrel-shifts in 2-4 bit increments are provided at both ends of the datapath. Integers, multibit fields, and bit-planes can coexist in a single cell; arithmetic, logic, and table substitution can be performed on any of these formats. Temporary storage similar to the subarray is provided for stage-intermediate results.

Datapath Instructions: Explicit control of the flow through the datapath of a stage is provided by a controller with its own instruction-set. Each instruction operates on one *minor-cycle* of the stage clock (similar but less flexible minor-cycles exist in current cytocomputers). Several instructions are stored in a stage and executed in order. Each instruction determines the source, processing and destination of datapath operands. If storage permits, literal operands could be injected into the datapath.

Note that this structure resembles that of a microprogrammed bit-slice machine. The primary departures are the subarray access mechanisms, the explicit support for tables and fields, and the need to fit everything on a few chips to allow long pipes.

This structure minimizes the number of stages required to implement DA algorithms. Consider a DRC application: several independent mask-planes are processed on successive minor-cycles by accessing different bit-planes in the subarray and operating on each with transformations stored in the datapath table. A more general wave-expand step is done in one stage: four cycles to determine the bordering cell with minimum/maximum weight, one cycle to add/subtract this from the central cell, and one cycle to update any flags. Table VI gives the performance goals for such a stage. Several trade-offs are apparent. Datapath width affects the complexity of the ALU, subarray and temporary storage, and instruction storage. Pipeline rate impacts the number of feasible minor-

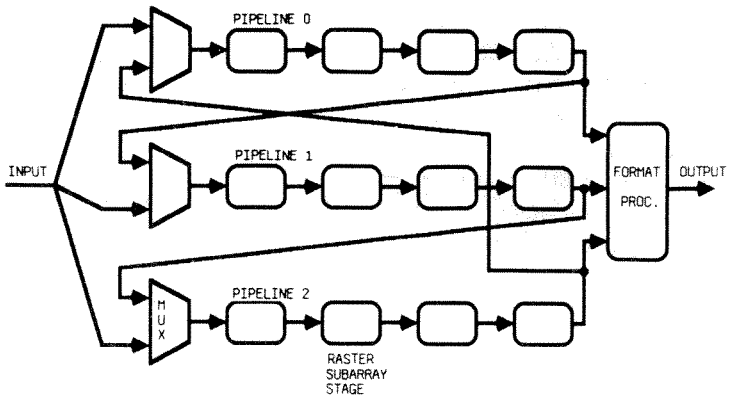


Fig. 12. RPS pipeline for DA.

TABLE VI
PERFORMANCE GOALS FOR PIPELINE STAGE

Parameter	Value
Datapath Width	24-32 bits
Line-Buffer Length	4K-16K
Stage Cycle t_{stage}	$< 1\ \mu s$
Minor-cycles (instructions)	10-12, $\approx$ 100 ns/cycle
Table Look-Up	4K-8K words
Field-Alignment	Barrel Shift, 2-4 bit increments
Temporary Storage	4-9 full-width words

cycles, and line-buffer and table access times. With a semi-custom implementation and 64K-bit memories a single stage will require about 10 chips. With a custom implementation, 256K-bit memories and relaxed pin constraints a 3-chip stage is possible: one chip each for the line-buffers, the stage-processor, and the tables.

We have not yet addressed the appropriate length for a pipeline of these stages. Most applications argue for very long pipes. However, it is usually not the case that all stages are required at all steps of the algorithm, for example, global decisions may be necessary after short processing sequences, in which situations a long pipeline is underutilized. The solution shown in Fig. 12 employs multiple shorter pipelines. Each pipeline can be connected to the adjacent one to form longer pipes. More importantly, several short pipes can concurrently perform different tasks: short DRC steps, independent path connections, for example. (Conceivably, several independent users can have a short dedicated pipe if appropriate multiple I/O channels are available.) This organization requires only the addition of switching multiplexors at the front of each pipe, and a small *format-processor* to choose which bits of which streams are placed in the single final output stream. The complexity of such a system is not excessive; assuming a 3-chip stage, a 1-chip switch, and a 10-chip format-processor a subsystem with four 32-stage pipelines will require about 400 chips.

5.2. Practical Considerations for DA Architectures

Several criteria are available to evaluate the merits of proposed special-purpose machines [1] including practical trade-offs among cost, speed, expandability, and range of application. RPS machines have these advantages:

Expandability: A machine based on a pipeline of homogeneous stages is inherently modular. Adding stages is straightforward and practical. In addition, the loose coupling of major system components—disk, cell-buffer, controller, pipeline—permits independent component upgrading.

Cost/Performance Range: Both cost and performance are proportional to pipeline length and cell-buffer size. A low-end system will have only a single short pipe and small buffer. A high-end system will have several long pipes, a large buffer, and a dedicated disk.

Direct Accommodation of Large Problems: Grid-size is limited only by the length of the stage line-buffers.

Application Range: Clearly a variety of DRC and routing tasks can be performed. Any problem represented on a cellular grid characterized by strongly local cell dependencies is a candidate.

The primary weakness of these architectures is the restriction on global and conditional data-manipulation imposed by the pipeline structure. Pipeline inertia means that a decision based on the complete state of the grid or the movement of complex grid sections must often be postponed until the processed grid is available at the end of the pipe. It is difficult for a state change in one cell to influence globally all subsequent pipeline stages. In our experiments, neither of these problems appears serious enough to warrant abandoning RPS machines.

It is useful to compare RPS machines with full-arrays on some of these points. The pipeline structure accommodates additional processing stages. Arrays are generally not designed to accommodate additional processors. Arrays with thousands of processors are usually restricted to simple but fast bit-sequential processors; algorithms may be lengthy because of this bit-level processing but overall speed can be significant due to the enormous number of processors. Pipelines with 10–100 processors can afford more complex stages; the goal is to incorporate as much processing power in each stage as necessary to minimize pipeline passes. Arrays with large memories at each node and pipelines with long line buffers can deal directly with large problems. However, arrays are limited by the total storage available across all nodes, whereas pipelines are limited by the length of the line buffers. Consider, for example, that both a 64 X 64 array with 4K-bits per node and a 64-stage 32-bit wide pipeline with 4K-cell line buffers require 16M-bits of storage. A 704 X 704 X 32-bit grid can

be folded directly onto the array effectively filling up all storage; any larger image must paged in and out of this storage. A 4K × 4K × 32-bit image can be directly streamed through the pipeline. Both arrays and pipelines benefit uniformly from improvements in device density and speed: incorporating more stages (processors) onto a chip allows the construction of larger pipelines (arrays). There will inevitably be some point at which chip count for a large pipeline system matches that of a large array. In this situation the particular structure of the problems at hand will determine the choice of hardware.

VI. Conclusions

The RPS class can effectively support several grid-based DA applications, the principal strength being the wide cost/performance range achievable with a modular pipeline structure. Results from experimental DA algorithms running on small cytocomputers are encouraging; some of these systems are already superior to their software counterparts. A more optimal RPS design resulting from these experiments will further improve execution times and permit more complex DRC and routing applications.

Acknowledgment

The Environmental Research Institute of Michigan provided early access to prototype cytocomputers, and the authors thank Robert Lougheed and David McCubbrey in particular for this support. They are also grateful to Tom Blank for the benchmark and router of [4], and to Paul Killey for system support.

References

[1] H. G. Adshead, "Towards VLSI complexity: The DA algorithm scaling problem: Can special DA hardware help?" in *Proc. 19th Design Automation Conf.*, pp. 339-344, June 1982.

[2] —, "Employing a distributed array processor in a dedicated gate-array layout system," in *Proc. ICCC*, pp. 411-414, Oct. 1982.

[3] T. Blank, M. Stefik and W. van Cleemput, "A parallel bit map processor architecture for DA algorithms," in *Proc. 18th Design Automation Conf.*, pp. 837-845, June/July 1981.

[4] T. Blank, "A bit map architecture and algorithms for design automation," Ph.D. dissertation, Dept. of EE, Standord Univ., Stanford, CA, Sept. 1982.

[5] M. A. Breuer and K. Shamsa, "A hardware router," *J. Digital Systems*, vol. IV, issue 4, pp. 393-408, 1981.

[6] C. R. Carroll, "A smart memory array processor for two layer path finding," in *Proc. 2nd Caltech Conf. on Very Large Scale Integration*, Jan. 1981.

[7] D. Chyan and M. A. Breuer, "A placement algorithm for array processors," in *Proc. 20th Design Automation Conf.*, pp. 182-188, June 1983.

[8] E. Damm and H. Gethoeffer, "Hardware support for automatic routing," in *Proc. 19th Design Automation Conf.*, pp. 219-223, June 1982.

[9] M. M. Denneau, "The Yorktown simulation engine," in *Proc. 19th Design Automation Conf.*, pp. 55-59, June 1982.

[10] S. J. Hong, R. Nair, and E. Shapiro, "A physical design machine," in *VLSI 81*, J. P. Gray, Ed., London: Academic, pp. 346-365, 1981.

[11] S. J. Hong and R. Nair, "Wire routing machines—New tools for VLSI physical design," *Proc. IEEE*, vol. 71, pp. 57-65, Jan. 1983.

[12] R. Kane and S. Sahni, "A systolic design rule checker," TR-83-13, Computer Science Dept., University of Minnesota, July 1983.

[13] A. Iosupovicz, "Design of an iterative array maze router," in *Proc. ICCC*, pp. 908-911, 1980.

[14] T. N. Mudge, R. M. Lougheed and W. B. Teel, "Cellular image processing techniques for checking VLSI circuit layouts," in *Proc. 1981 Conf. on Information Sciences and Systems*, The Johns Hopkins University, pp. 315-320, Mar. 1981.

[15] T. N. Mudge, R. A. Rutenbar, R. M. Lougheed and D. E. Atkins, "Cellular image processing techniques for VLSI circuit layout validation and routing," *Proc. 19th Design Automation Conf.*, pp. 537-543, June 1982.

[16] R. Nair, S. J. Hong, S. Liles and R. Villani, "Global wiring on a wire routing machine," *Proc. 19th Design Automation Conf.*, pp. 224-231, June 1982.

[17] G. F. Pfister, "The Yorktown simulation engine: Introduction," in *Proc. 19th Design Automation Conf.*, pp. 51-54, June 1982.

[18] R. A. Rutenbar, T. N. Mudge and D. E. Atkins, "Wire routing experiments on a raster pipeline subarray machine," in *Dig. Papers, IEEE International Conf. on CAD*, pp. 135-136, Sept. 1983.

[19] L. Seiler, "A hardware assisted design rule check architecture," *Proc. 19th Design Automation Conf.*, pp. 232-238, June 1982.

[20] K. Ueda, T. Komatsubara and T. Hosaka, "A parallel processing approach for logic module placement," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, pp. 39-47, Jan. 1983.

[21] R. M. Lougheed, D. L. McCubbrey and S. R. Sternberg, "Cytocomputers: Architectures for parallel image processing," in *Proc. IEEE Workshop on Picture Data Description and Management*, Aug. 1980.

[22] R. M. Lougheed and D. L. McCubbrey, "The cytocomputer: A practical image processor," *Proc. 7th Annual International Symp. on Computer Architecture*, pp. 271-277, May 1980.

[23] S. R. Sternberg, "Language and architecture for parallel image processing," in *Pattern Recognition in Practice*, E. S. Gelsema and L. N. Kanal, Eds., Amsterdam: North Holland, 1980.

[24] P. E. Danielsson and S. Levialdi, "Computer architectures for pictorial information systems," *Computer*, vol. 14, no. 11, pp. 53-67, Nov. 1981.

[25] *Languages and Architectures for Image Processing*, (M. Duff and S. Levialdi, Eds.), London, England: Academic, 1981.

[26] M. Kidode, "Image processing machines in Japan," *Computer*, vol. 16, no. 1, pp. 68-80, Jan. 1983.

[27] K. Preston, "Cellular logic computers for pattern recognition," *Computer*, vol. 16, no. 1, pp. 36-47, Jan. 1983.

[28] K. Preston, M. J. B. Duff, S. Levialdi, P. Norgren, and J. Toriwaki, "Basics of cellular logic with some applications in medical image processing," *Proc. IEEE*, vol. 67, pp. 826-856, May 1979.

[29] *Multicomputers and Image Processing: Algorithms and Programs*. K. Preston and L. Uhr, Eds., New York: Academic, 1982.

[30] D. Antonsson *et al.*, "PICAP—A system approach to image processing," *IEEE Trans. Computers*, vol. C-31, no. 10, pp. 997-1000, Oct. 1982.

[31] H. J. Siegel, *et. al.*, "PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Computers*, vol. C-30, pp. 934-947, Dec. 1981.

[32] M. J. B. Duff, "Review of the CLIP image processing system," in *Proc. National Computer Conf.*, pp. 1055-1060, 1978.

[33] J. K. Iliffe, *Advanced Computer Design*, London, England: Prentice Hall, ch. 12, 1982.

[34] K. E. Batcher, "Architecture of a massively parallel processor," in *Proc. 7th Annual Symp. on Computer Architecture*, pp. 168-174, 1980.

[35] M. Aoki *et al.*, "An LSI adaptive array processor," *Proc. ISSCC*, pp. 122-123, Feb. 1982.

[36] K. Preston and P. E. Norgren, "Interactive image processor speeds pattern recognition," *Electronics*, vol. 45, p. 89, 1972.

[37] J. M. Herron, J. Farley, K. Preston and H. Sellner, "A general-purpose high-speed logical transform image processor," *IEEE Trans. Computer*, vol. C-31, no. 8, pp. 795-800, Aug. 1982.

[38] C. M. Baker, "Artwork analysis tools for VLSI circuits," M.S. thesis, MIT, Cambridge, MA, 1980.

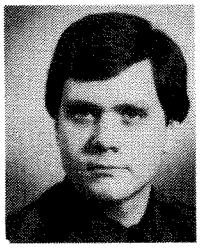
[39] H. S. Baird, "Fast algorithms for LSI artwork analysis," *J. Design Automation and Fault Tolerant Computing*, vol. 2, no. 2, May 1978, pp. 179-209.

[40] P. Losleben and K. Thompson, "Topological analysis for VLSI circuits," in *Proc. 16th Design Automation Conf.*, pp. 461-473, June 1979.

[41] R. Eustace and A. Mukhopadhyay, "A deterministic finite auto-

maton approach to design rule checking for VLSI," in *Proc. 19th Design Automation Conf.*, pp. 712-717, June 1982.
[42] C. Mead and L. Conway, *Introduction to VlSI Systems.* Reading, MA: Addison-Wesley, 1980.
[43] G. Matheron, *Random Sets and Integral Geometry.* New York: Wiley, 1975.
[44] J. Serra, *Mathematical Morphology and Image Processing.* London, England: Academic, 1981.
[45] E. H. Frank and R. F. Sproull, "Testing and debugging custom integrated circuits," *Computing Surveys*, vol. 13, no. 4, pp. 425-452, Dec. 1981.
[46] M. Marek-Sadowska and W. Maly, "A hierarchical layout description for artwork analysis of VLSI IC," *Proc. ICCC-82*, pp. 419-422, Oct. 1982.
[47] J. Wilmore, "The use of bit maps in designing efficient data bases for integrated circuit layout systems," *J. Digital Systems*, vol. IV, issue 1, pp. 71-95, 1980.
[48] R. F. Lyon, "Simplified Design Rules for VLSI Layouts," Lambda, vol. II, no. 1, pp. 54-59, 1st Quarter, 1981.
[49] T. W. Griswold, "Portable design rules for bulk CMOS," *VLSI Design*, vol. III, no. 5, pp. 62-67, Sept./Oct. 1982.
[50] P. Losleben, "Computer aided design for VLSI," in *Very Large Scale Intergration VLSI: Fundamentals and Applications*, D. F. Barbe, Ed., Springer-Verlag, 1980.
[51] J. L. Bentley, D. Haken and R. W. Hon, "Fast geometric algorithms for VLSI tasks," in *Proc. COMPCON-80*, pp. 88-92, 1980.
[52] M. H. Arnold and J. K. Ousterhout, "Lyra: A new approach to geometric layout rule checking," in *Proc. 19th Design Automation Conf.*, pp. 530-536, June 1982.
[53] S. E. Bello, J. L. Hoffman, R. I. McMillan and J. A. Ludwig, "VLSI hierarchical design verification," *Proc. ICCC-82*, pp. 530-533, Oct. 1982.
[54] M. A. Breuer, A. D. Friedman and A. Iosupovicz, "A survey of the state of the art in design automation," *Computer*, vol. 14, no. 10, pp. 58-75, Oct. 1981.
[55] J. Soukup, "Circuit layout," *Proc. IEEE*, vol. 69, pp. 1281-1304, Oct. 1981.
[56] C. Y. Lee, "An Algorithm for Path Connections and Its Applications," *IRE Trans. Electronic Computers*, vol. EC-10, Sept. 1961, pp. 346-358.
[57] F. Rubin, "The Lee path connection algorithm," *IEEE Trans. Computer*, vol. C-23, pp. 907-914, Sept. 1974.
[58] J. H. Hoel, "Some variations of Lee's algorithm," *IEEE Trans. Comput.*, vol. C-25, pp. 19-24, Jan. 1976.
[59] J. Soukup, "Fast maze router," in *Proc. 15th Design Automation Conf.*, pp. 100-101, June 1978.
[60] R. A. Rutenbar, Ph.D. dissertation, Univ. Michigan, in preparation.

*

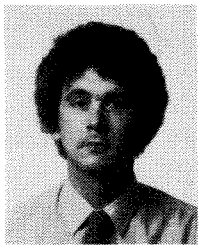


Rob A. Rutenbar (S'78) received the B.S. degree in electrical and computer engineering from Wayne State University, Detroit, MI, in 1978, and the M.S. degree in computer, information and control engineering (CICE) from the University of Michigan, Ann Arbor, in 1979. He is currently a doctoral candidate at the University of Michigan, where his thesis research concerns computer architectures for CAD problems.

While studying at the University of Michigan he has consulted in the areas of LSI CAD software and worked on hardware design for parallel processors. His research interests include VLSI design automation, CAD hardware, computer architecture, and artificial intelligence.

Mr. Rutenbar is a member of ACM and Eta Kappa Nu.

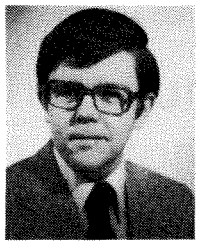
*



Trevor N. Mudge (S'74-M'77) received the B.Sc. degree in cybernetics from the University of Reading, England, in 1969, and the M.S. and Ph.D. degrees in Computer Science from the University of Illinois, Urbana, in 1973 and 1977, respectively.

He has been with the Department of Electrical and Computer Engineering at the University of Michigan since 1977 and currently holds the rank of Associate Professor. His research interests include computer architecture, operating systems, VLSI circuit design, computer vision and robotics.

*



Daniel E. Atkins (S'68-M'70) received the B.S. degree in electrical engineering from Bucknell University, Lewisburg, PA, in 1965, the M.S. in electrical engineering and Ph.D. in computer science from the University of Illinois, Urbana, in 1967 and 1970, respectively.

He is a Professor of Electrical and Computer Engineering at The University of Michigan, Ann Arbor, where he is teaching and conducting research in the areas of computer arithmetic, parallel computer architecture, and digital design methodology. His is codirector of the Computing Research Laboratory and a member of the Robotics Laboratory. He is past chairman of the Technical Committee of Computer Architecture of the IEEE Computer Society and has been co-organizer of several symposia on computer arithmetic. He is a member of the ACM and SIGMICRO, SIGARCH, and SIGCSE. He was Program Committee Chairman for the 1980 International Symposium on Computer Architecture.

He participated in the design of Illiac III and has also been a member of the faculty at Bucknell University and The University of Maryland. Through university projects and consulting, he has participated in the design and implementation of eight different high speed parallel processors.