Coprocessor Design for Multilayer Surface-Mounted PCB Routing

Ramachandran Venkateswaran, *Student Member, IEEE*, and Pinaki Mazumder, *Member, IEEE*

***Abstract*— The printed circuit boards (PCB's) for the 1990's can be characterized by higher circuit densities, multiple routing layers, newer packaging technologies, and demand for lower manufacturing costs. The task of connecting all the traces on such a complex board will become more and more time consuming. This paper presents the issues involved in the design of a special-purpose processing array system, called HAM, which will accelerate such compute-intensive wire routing tasks. It is especially suited for double-sided surface-mounted boards which require complex three-dimensional search operations over multiple wiring planes. The novel features of the design include a hexagonal interconnection scheme to improve workload distributions during multilayer concurrent search operations and the VLSI custom design of the processors. Particular emphasis has been placed on the demands of maze routing such as in the allocation of the routing database on the multiple processors, design of buffer stores for maintaining the frontier-lists, etc. A novel scheme of cell-address propagation, which is quite different from the traditional grid-coordinate approach is discussed. This provides for rapid lookup of pertinent routing information and can be extended to any distributed memory multiprocessor system. A global pipelining scheme of cell updates and expands is discussed. Experimental results are presented relating the speedup to different criteria such as number of processors and size of the local memory for two different modes of parallel wave propagation.**

I. Introduction

The NEW generation printed circuit boards (PCB's) can be characterized by higher circuit densities, finer trace widths, multiple routing layers, stringent performance constraints, complex packaging and manufacturing technology, and demand for lower manufacturing costs. Designers must get their products to the market fast or risk losing their competitive edge. This requires an integrated design solution uniting the power and convenience of automated tools with the interactive expertise of the designer. Several cost measures are often used in the routing process. For instance, nets pertaining to analog components such as op-amps need to be routed within a certain pre-specified length. Board manufacturability and ease of update are important routing requirements. Vias have to be intelligently used to provide compact multilayer routes for all the nets. Wirelength minimization is also important for minimizing parasitic effects and is critical for high-performance designs.

Manuscript received May 4, 1992; revised August 25, 1992 and October 27, 1992. This work was supported in part by an IBM graduate fellowship, by the U. S. Army Research Office under the URI program Grant DAAL 03-87-K-0007, by the Office of Naval Research under Grant 00 014-85-K-0122, and by the National Science Foundation under Grant 9013092.

The authors are with the Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122.

IEEE Log Number 9206233.

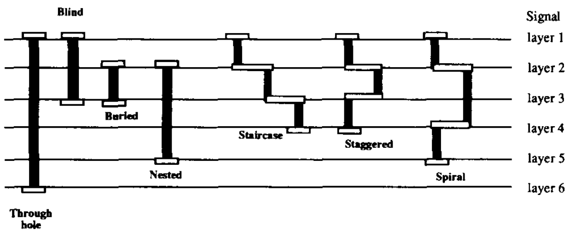


Fig. 1. Different via structures.

Also in terms of board manufacturability, the primary concern is the number of layers. Given an unlimited number of layers to play with, any router can attain 100% success rate; however, the additional layers greatly increase the manufacturing cost and so typically the number of routing layers that are actually available is limited. Most automatic routers are frequently restricted to routing between pairs of layers using vias for feedthroughs, and can only be extended to multi-layer configurations by concatenating layer pairs. Since, this does not make use of the variety of via-structures that are becoming possible in addition to the traditional through-hole vias (Fig. 1), it becomes all the more imperative that all layers be concurrently considered to achieve efficient multilayer routes. Though concurrent search can result in more compact boards, it is also more complex. Previously, routing algorithms could proceed by reaching the leads X, Y coordinates on any layer. Now it must be assumed that the terminal is available on only one surface at its X, Y location.

In addition, advances in packaging technology, such as the increasing use of *surface mounted* devices have led to double-sided mount configurations. This opens up new problems such as whether to keep closely connected components on the same side, or to divide them between the two sides so as to increase compactness. In the absence of better solutions, the approach typically taken is to generate several initial placements and route each of them separately and choose the best one. This increases the time for routing by several time folds.

One of the principal routing strategies that has been found to be capable of handling all these varied requirements in a flexible manner is the maze or flood router. First proposed in 1961 by Lee [1], this alogrithm is still the mainstay of autorouting technology for PCB's. It represents a general approach to routing (rather than a specific algorithm) and further guarantees to find a path if one exists. The maze router can be extended for multilayer cases as well. However, it is computationally extremely expensive. One simplification is to first use faster approaches such as pattern routing to complete most of the easy connections which account for about 85% of all the nets. Unfortunately, though, the last 10–15% of all traces require the most time and computing resources since they are often the most complex to route. It may also become necessary to rip-up and reroute existing connections to make way for these traces. This process typically requires three or four invocations of the Lee algorithm. Since the Lee algorithm is at its slowest when connections are not found, the speed problem becomes significant. Furthermore, it gets only more worse as this phase is often done interactively with expert designer interface and hence rapid response times are desirable.

A custom-hardware implementation for maze routing can run as much as a thousand times faster than a general purpose computer if the routing processor architecture is ingeniously designed to exploit all the intrinsic data parallelism in the search operations. Hardware costs are rapidly decreasing and with the aid of VLSI it is now possible to construct a single-board hardware accelerator that can interface to a personal computer or workstation running routing software. Furthermore, it is unlikely that the maze routing paradigm can be supplemented in the near future by any other because of its extreme flexibility, and because of the nonplanar nature of grids in complex multilayer PCB's. Thus a routing processor supporting the general maze router with flexible cost capability does not suffer the risk of obsolescence as could be the case with other algorithm-specific solutions. This paper focuses on the practical issues in the actual construction of one such system, called **H**exagonal **A**rray **M**achine and acronymed as (HAM). Since multidimensional arrays are very difficult to implement, and since the sizes of the grids are not known *a priori*, HAM maps the grid onto a smaller number of processors connected in a hexagonal wraparound topology. The hexagonal interconnection scheme has been previously shown [2], to possess the best characteristics amongst other two-dimensional topologies for multilayer concurrent searching operations.

The rest of the paper is organized as follows. In Section II, we present a motivation for our work and summarize some of the previous work done. Section III describes the overall architecture and some issues pertaining to maze routing. A new and more practical scheme for address computation during wavefront propagation is presented. Section IV describes the VLSI design issues for the construction of the individual HAM processors. Issues such as memory and buffer storage organization, datapath and microprogrammed control are discussed. The timing and instruction flow is described along with a critical path analysis. In Section V, we summarize the results of system level simulations which evaluate the effect of internal storage size, number of processors and mode of wavefront expansion (one wavefront or multiple wavefronts at a time) on overall performance. It may be noted that the distributed nature of computation proposed here is also applicable to other implementations as well.

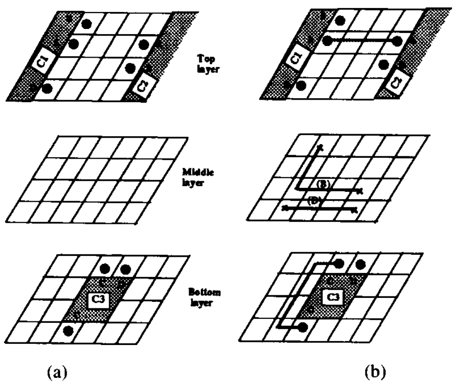


Fig. 2. (a) A small double-sided SMT routing problem (b) The three-layer routing solution.

II. Multilayer Concurrent Maze-Routing

A. Maze Routing

Maze routing is usually the only practical solution to do multilayer routing with double-sided *surface mounted module* placement. Maze routing consists of three main operations: the first prepares the board for routing by partitioning it into hundreds or thousands of cells. The size of the grid cells is determined by the pad spacing and other design rules being employed. All the structures on the board such as pads, copper areas, traces, tooling holes, etc. are marked in the cells to which they belong. After creating the grid, the maze router begins an "expansion" stage. The router examines all the grid points in larger and larger concentric rings around the source pad till the destination is reached. Simultaneously, the router assigns cost values to each cell in accordance to some metric. These metrics are based on a variety of routing variables such as heading toward the target, adding a via, making a corner, or using preferred routing layers. In fact, the popularity of the maze router stems from the capability to tailer the cost functions to meet almost any routing requirements. Once the target pad is reached a "backtrace" is performed to the origin pad from the target point along one of the lowest cost paths.

B. Need for Concurrent Multilayer Search

One reason for multilayer search as was mentioned earlier is to minimize the total number of layers, thereby reducing the manufacturing costs of the board. The problem is more severe in double-sided surface-mounted PCB's with multiple signal layers. For example, consider a three-layer SMT board with three modules $C1, C2$, and $C3$ as shown in Fig. 2(a). Nets A and B represent interconnections between terminals on one side of the board while net C interconnects two terminals on the other side of the board. Net D, on the other hand, connects two pins on opposite sides. An optimal shortest path routing solution is shown in Fig. 2(b). Note the buried vias for net B, which allows net C to be routed beneath its terminals, could be readily found only by using concurrent search on all layers.

TABLE I
SOME EXISTING ROUTING ACCELERATORS

Accelerator	Architecture		Routing Model	Comments
	Interconnection Network	PE design		
Wire Routing Machine [7]	32 × 32 array with endaround row/column wraparounds	General purpose Z80 μproc + 15Kb mem/node	Maze routing for detailed/global wiring	1–2 layer grids with variable grid weighting
Distributed Array Processor [8]	64 × 64 array with global row/col buses for global movement	Bit-serial proc with memory	Maze routing for detailed/global wiring	2-layers unit cost grids; DAP is a commercial machine
Toroidal Machine [9]	Prototype 8 × 8 array with twisted torus wraparounds	NEC μPD7800 PE + 8Kb ROM and RAM	Maze routing for detailed/global wiring	1–2 layers weighted grids; support for interactive rip-up and reroute
MANURE2 [4]	SISD microcode machine	Custom designed bitmap + address + mark/cost processors	Maze routing for detailed wiring	Multilayer support by reconfiguring via-bits in bitmap; support for diagonal routing; staged expansions

C. Previous Accelerators

Routing accelerators can be broadly categorized as either SISD (single instruction single data) or SIMD (single instruction multiple data) machines. The first category [4]–[6] consists of a conventional processor aided by special-purpose support hardware to speed up some of the computations involved such as address computations, frontier-list managment, etc. However, they do not capture the parallelism inherent within the algorithm. Instead, speedup is obtained by the elimination of operating system overheads and by efficiently performing some of the common operations in hardware.

The SIMD systems account for the intrinsic data parallelism in maze routing. The primary idea is to use an $N \times N$ array of identical processing elements that have a one-to-one correspondence with the $N \times N$ grid plane and so achieve a linear runtime for finding a path. A major disadvantage with such "full-grid" machines is that they need $O(N^2)$ PE's despite the fact that almost all of them are not utilized at the same time. Moreover, they cannot handle problem sizes which are bigger than the physical size of the processing array. This is solved by allowing for wraparound connections and making each PE in the array to be responsible for maintaining the status of several grid cells. Some of the designs which fall in this category are the Wire Routing Machine [7], the Distributed Array Processor [8], and the Toroidal Machine [9]. A brief comparison of these routers is provided in Table I.

The HAM approach is also SIMD based. It improves on existing approaches in three main aspects. a) The individual compute elements have been custom designed keeping in mind the nature of data retrieval and manipulation operations required for maze routing. Careful consideration has been given to the interprocessor communication demands which is often the bottleneck for previous routers. b) The second reason has to do with the mapping used to assign grid cells to processors so that the workload gets uniformly distributed. The mapping provides the maximum *interprocessor cycle period,* i.e., the minimum distance between two occurrences of the same processing element along any straight line (including diagonal lines) [2]. c) The hexagonal interconnection which supports concurrent search in multiple layers needed in complex board routing.

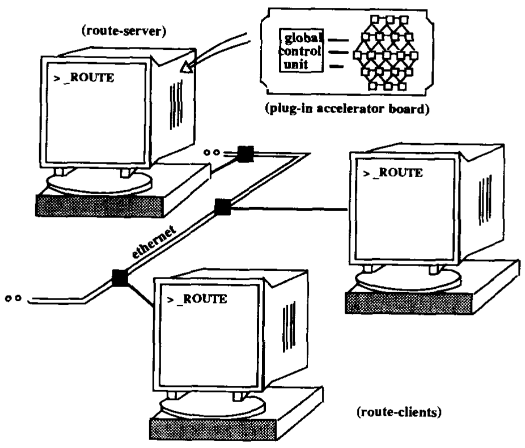


Fig. 3. Overview of the HAM routing system.

III. HAM System Architecture

A. General Organization

In this section, we present the overall system organization. There are three levels of interfaces involved. The first deals with the external interface between the workstation and the accelerator. The second deals with the interface between the accelerator controller and the processors in the array and the third level is the one between the processors themselves.

External Interface: Fig. 3 shows the HAM system organization in a distributed CAD system running on a network of workstations. It is conceived of as a single board system that can be plugged into an expansion slot of any conventional workstation running CAD software. The layout software can therefore address the accelerator as a device whenever it needs to perform a maze routing operaton. Processors are organized in a two-dimensional lattice. The processor array operates under the supervision of a *global control unit* (GCU) which is responsible for interfacing with the host workstation, for performing the sequential parts of the algorithm and for issuing the commands which are performed by all processors in a lock-step fashion.

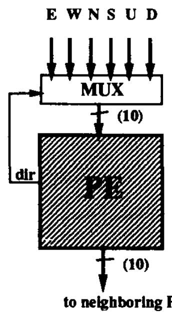


Fig. 4. Interprocessor communication interface.

GCU/Array Interface: The HAM system is based on an SIMD model, wherein each processor basically executes the same instruction in the same clock cycle on its local data set. In each cycle, the GCU broadcasts an address to all processors which corresponds to a particular instruction stored in the processor's control memory. The processors themselves lack any decision making capability. Sample instructions include $expand\,in\,direction\,d$, $backtrace$, $start\,a\,new\,wavefront$ and so on. In addition, the GCU has access to the input and output ports of the processors so as to be able to perform initialization and obtain results in the end.

Interprocessor Interface: Maze routing is highly communication intensive, but the communications follow a near-neighbor pattern. Consequently, each processor is directly connected to six other nodes in the array. Communication is allowed only on these links. In particular, there is no message-passing mechanism between any two arbitrary processors in the array. This model therefore eliminates the need for any hardware message router. Also, the proposed system is based on a distributed memory model. Any changes to the memory contents is to be accomplished by message passing alone. This eliminates the need for complex data consistency and data coherency control present in a shared-memory system. In an ideal system, each processor will have six parallel ports to communicate with the six neighbors; but in our implementation we have opted for a single-port time-multiplexed scheme, wherein, in any one clock cycle, all processors in the array talk to their neighbor in one of the six directions. A multiplexer is placed at the input pins to select one of the 6 neighbor data, as shown in Fig. 4. This selection is based on the *dir* control bits generated by the processor.

B. Maze Routing Requirements

In the HAM system, as presented above, each processor has access only to its local memory, called grid memory, which stores the information pertaining to all the cells that are mapped onto that processor. In this paper, the per-cell storage format used is shown in Fig. 5. The cost field stores the least cost path discovered to that cell from the source cell. The directional mask field is used for backtracing the net from the target cell to the source. The content of the status field is as shown in the figure and is used to control the wave propagation and backtrace phases.

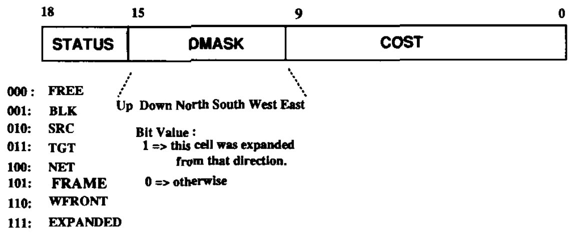


Fig. 5. Information stored in grid-memory for each grid cell.

During the maze algorithm, the processors constantly have to access their local stores and exchange information between themselves through message-passing. This raises several interesting questions that become critical in determining performance.

1) How does one processor inform the other as to the identity of the cell being expanded?
2) What is the message overhead during wave-propagation?
3) How do wavefronts proceed?
4) How do the processors keep track of the frontier list (i.e., cells that have been reached during wave propagation but which have not yet been expanded out)?

1) Next-Cell Address Computation: Suppose two neighboring grid cells c_1 and c_2 are mapped to processors p_1 and p_2 and their information is stored at addresses m_1 and m_2 in the respective local memories. Then, in the traditional *grid-coordinate transfer* scheme, p_1 communicates the $\langle x, y, z\rangle$ grid coordinates of c_2 to p_2; which lacking other information has to search, possibly its entire memory, trying to determine the location (m_2) where data for c_2 is to be stored. In [2] we had alluded to this problem and had suggested that it would be much faster to have p_1 communicate m_2 directly to p_2. This is the basis of our *address-transfer* scheme. The message size for the address-transfer scheme is only $\lceil \log[(kG_xG_y)/N]\rceil$ bits for a k-layer $G_x \times G_y$ grid mapped onto an N-processor system. As opposed to this, transferring coordinates needs $\lceil(\log k + \log G_x + \log G_y)\rceil$. Thus there is also a saving in the message traffic.

The problem, therefore, is reduced to one of each processor determining the address to transmit to their neighbors in all six directions. An *index-based* mapping scheme was given an earlier paper [2] and is briefly summarized below. "Let $c_1, c_2, \cdots, c_k$ be the ordered list of cells to which processor p is assigned. The ordering is by a row-major traversal of cells one layer at a time. Then, we define INDEX$[c_i] = i$. Once, all the INDEX values are known, each processing element can calculate the difference Δ_d, between its INDEX value and the INDEX values of its neighbor in direction d. Grid boundaries can be handled using a dummy value X." Furthermore, it was shown that for the hexagonal mapping on N processors, the maximum absolute value of Δ_d is $\lceil \max(G_xG_y)/N\rceil$, independent of the number of layers k.

Thus in this scheme, the entry for cell (x, y, z) is stored in the local memory of the mapped processing element at address $\langle z_k \cdots z_0, b_g \cdots b_0\rangle$ where $z_k \cdots z_0$ is the binary representation of z and $b_g \cdots b_0$ is the binary representation of INDEX

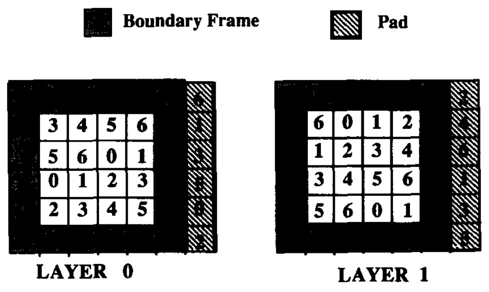


Fig. 6. Padding scheme for a 2-layer 4 × 4 grid.

$[x, y, z]$. Then if m_a is the address of the cell currently being expanded, the information passed to the neighboring processing element in direction d is the value $(m_a + \Delta_d)$, where Δ_d is the difference stored at m_a for direction d and so can be trivially computed.

Padding Approach: The main problem with the above approach is the additional amount of memory needed to store the Δ values. In fact for a grid of size $100 \times 100 \times 4$ and $N = 61, |\Delta_d| \leq 2$ and thus $3 * 6 = 18$ bits are needed for every cell. This is almost equal to the size of the information stored for the cell and is therefore not acceptable.

Instead, we use a simpler modification which we call the *padding* approach. First a 1-cell rectangular frame is padded to the original grid and all these cells are marked blocked. These serve to delineate the boundaries of the grid and play the role of the dummy X INDEX values. Subsequently, the grid is padded in the X-dimension with as few dummy cells as are needed to make G_x a multiple, say r of N. Note that addition of dummy cells do not affect the routing as they are simply treated as blocked cells. This is shown in Fig. 6 for a small 4×4 2-layer grid using a 7-node system. After the addition of the boundary frame, an additional column is needed to make $G_x = r * 7, r = 1$.

For the hexagonal mapping, each row has exactly r occurrences of each PE. Visit the processors proceeding from layer 0 to the last layer and going row-by-row within each layer. Then, if a cell c corresponds to the ith occurrence of the PE, information pertaining to it will be stored at address i in the grid-memory. From this, it can be concluded that the north, south, up and down neighbors of c will be stored at address $i - r, i + r, i - rG_y$, and $i + rG_y$, respectively, in the grid-memory of the respective PE's. Moreover, the east and west cells will be stored at the same address i of the east/west neighbor. The exception is for the processor assigned to the leftmost boundary of each row when $r > 1$. This processor needs to send address $i - 1$ to and *add*1 to address received from its west neighbor. However, this requires only 1 bit of additional information to be stored with each cell. Also, since the grid size is known at the outset, the offsets can be precomputed reducing the address calculation and memory retrieval to very simple operations.

2) Buffer Store Design: Each processor needs to maintain a list of frontier cells which have to be expanded in subsequent cycles. We refer to the unit maintaining the address of frontier cells as the *buffer store.* The simplest implementation of the buffer store is as a hardware stack. Other implementations include a queue structure. Stacks are preferable for synchronous expansion while queues are better for asynchronous expansion as explained below.

3) Expansion Style: A multiprocessor system running Lee's algorithm can be built on two possible approaches for wavefront propagation.

Synchronous: Here, the entire current wavefront is expanded before the next one is considered for expansion. It is possible here, due to multiple cell assignments, that certain processors which have cells on the new wavefront are forcibly kept idle till the expansion of the previous wavefront is completed.

Asynchronous: In this mode of operation, at any cycle, any processor that has a cell that is yet to be expanded is allowed to do so. The concept of a wavefront has now to be interpreted as a collection of cells that have been reached from the source but have not yet been expanded.

From the implementation point of view, the asynchronous mode is simpler; for the processors can simply inspect their buffer stores and if they find a cell start expanding it. The GCU only has to be informed when the target cell is reached. On the other hand, in the synchronous mode, each processor has to inform the GCU if it has any cells left on the current wavefront that are still to be expanded. This effect can be realized by maintaining two stacks, one for the current wavefront cells and one for the new wavefront cells. A special instruction has also to be added to the instruction set of the GCU to initiate a new wavefront.

From the algorithmic point, when the wavefronts are allowed to proceed asynchronously, it is possible that one message may go racing out ahead of the others and cause one or more grid cells to be expanded incorrectly. This has the implication that when the target is first reached, the cost $c(s, t)$, may not represent the shortest path from the source. This race effect can be minimized by adopting the policy of always expanding a frontier cell with the lowest cost or alternately adopting a queue data structure instead of a stack for the buffer store.

Note, both the queue and two-stack structures can be realized using a single RAM module and two sets of counters. In case of the latter, one stack proceeds from the top down while the other proceeds from the bottom up. The counters denote the top of the two stacks and their roles can be interchanged at the start of a new wavefront. For the queue, the two counters mark the head and tail, respectively.

IV. Processor Element Design

There are several engineering issues such as chip and board area, power consumption, timing, control mechanism, wireability, memory organization and so on that are crucial in determinig the viability of the accelerator. Furthermore, the desire to meet all these criteria at reasonable expense mandates that the individual processors of the accelerator array and the

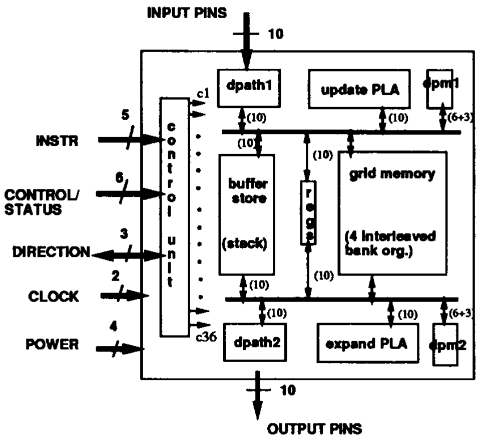


Fig. 7. Block-level diagram of a processor.

array controller be custom designed and not just built with general-purpose or off-the-shelf components. Since, our goal is ultimately to accelerate maze routing, this customization will be largely influenced by the typical operations to be performed therein. Within this framework, we wish to incorporate as much flexibility as possible so as to allow for different cost functions and expansion criteria to match the fabrication technology requirements.

A. General Design Overview

We consider the following four issues here:

- designing the datapath to include hardware support for commonly used operations and data-structure handling such as conversion between cell coordinates and memory address, computation of next cell address, maintaining frontier-list, and so on;
- designing the grid memory configuration so as to optimize access to the grid-related information;
- designing an appropriate instruction set;
- the global strategy for control of the processors.

Fig. 7 shows a block-level diagram of the processor showing the major components.

Control: We use a microprogrammable control-based design. Each PE has a 32-word deep microstore that generates 36 bits to control the datapath. A 5-bit address select one microinstruction each cycle. The control unit is pipelined so that while one instruction is being fetched, the previous one is being executed. It also allows for separate testing of the control and data sections of the PE by allowing the user to either read the contents of the microinstruction memory or to test the operation of the datapath under direct microcontrol. Since, the microstore can be downloaded at runtime, special instructions that make the full use of the parallelism afforded by the datapath can be designed and used for different algorithms. This approach makes it possible to run various versions of maze algorithms (and possibly other similar approaches) on the same hardware simply by reprogramming the control memory. This was considered important at least in the prototype version.

Datapath: The main datapath is 10-bit wide and includes the input and the output units, a register file, and two PLA blocks called the *Update* unit and the *Expand* unit. As the names suggest, the update unit helps to update the status of the grid cell which is being expanded into; while the expand unit does the appropriate cost and status processing needed to expand a cell on the wavefront to the neighbors. With a small modification, the same logic can handle backtracing as well. There is a separate 6-bit datapath for handling the directional mask information and a 3-bit datapath for cell status. The operation of the datapath is controlled by the microcode bits generated in the control section.

Memory: The grid-memory stores the data regarding each grid cell that has been assigned by the hexagonal map to the processor. It is organized into 4 equal banks to increase simultaneous access. This is useful for the grid-clearance phase for instance. During expansion the higher order two bits of the address are used to select the appropriate bank. A separate *buffer-store*, implemented using a 1K RAM and up–down counters is used to maintain the current frontier list at that processor, i.e., all grid-cells that have been mapped to that processor and which are in the current wavefront.

B. Communication with Other PE's

Each processor needs to communicate information to its 6 physical neighbors. This data (address and cost information) is assumed to be 10-bit wide. In the prototype version, each processor is implemented in a separate chip with only one 10-bit wide parallel input port and a separate 10-bit parallel output port. External switches are, therefore, needed to select the data from and to an appropriate neighbor during any clock cycle. The selection is based on the direction dir bits from the processor. In the current SIMD version, the dir bits are the same for all processors; consequently in one clock cycle all processors communicate to say their *east* neighbor (dir = 0) or *north* neighbor (dir = 2) and so on.

This design, though slightly more complex, is adopted for the following reasons: (1) It reduces the pin count problem from 120 pins to 20. (2) Even if a cell is expanded and the information propagated to all six neighbors in parallel, the receiving processor has to sequentially process the data and update its grid store. (3) Parallel expansion in all six directions require that the next address and cost computation circuitry be replicated.

The latter two problems are due to the fact that the expansion phase needs to access memory only once; whereas the receiving processor can receive messages from six neighbors in one cycle for six different cells; thus requiring six memory reads and writes. Serial communciation between processors could also solve the pin-count problem but not the algorithmic asymmetry in the update and expand operations. The typical scenario of computation and communication is shown in Fig. 9.

C. Timing

Timing is a critical issue especially in the design of a multiprocessor SIMD system operating under a global clock(s). A

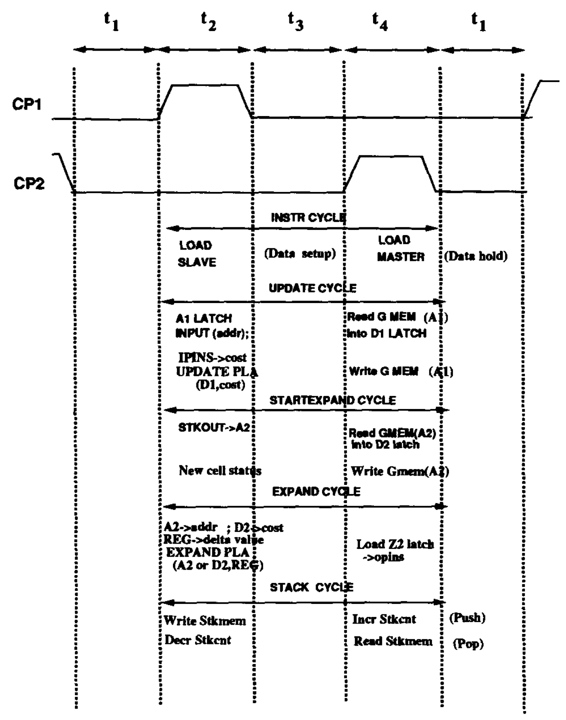


Fig. 8. Typical instruction modes.

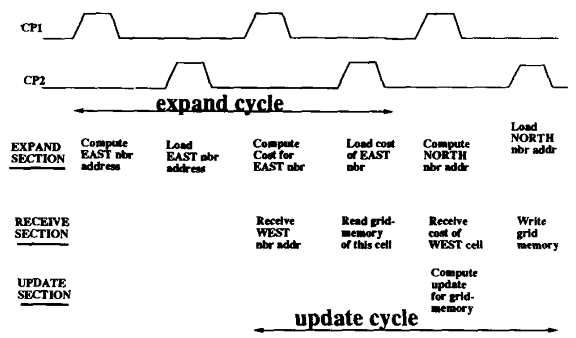


Fig. 9. Flow of parallel expands and updates across processors.

highly complex multiphase scheme can be counterproductive because of possible skews and other overhead. For maze routing, we found that a simple two-phase nonoverlapping clock strategy, as shown in Fig. 8, is adequate. The instruction issue is as follows. The memory address (external instruction) is maintained stable from the end of CP1 to the end of CP2. At this time, the data is read into the master. During CP1, this data gets transferred from the master to the slave register in the control memory. Simultaneously, the memory address circuitry is precharged during CP1. The control signals controlling the operation of the processor are decoded from the output of the slave and thus remain stable from the start of one CP1 cycle to the next.

The rest of the processor performs three main operations.

StartExpand: This cycle is initiated at the start and it serves to select a cell on the current wavefront for expansion. The information pertaining to the cell chosen is obtained from the grid memory. Each processor maintains a list of addresses pertaining to its set of current frontier cells in a buffer store. The *StartExpand* cycle takes two CP1–CP2 cycles. During the first CP1 cycle, the address at the top of the store is popped and is used to load the A2 latch. In the following CP2 cycle, the cost and expansion status is read from the grid memory and latched in the D2 register. The status of the grid memory is then updated (changed from *wavefront* to *expanded*) during the next CP1–CP2 cycle.

During expansion operations, sometimes a given processor has no current cells on the wavefront. In such a case, the processor sends an address of 0. By making memory address 0 a reserved word, the receiving processor essentially performs noops on this data. Address 0 is not to be pushed into the buffer store; hence preventing it from being used in future expansions.

Expand: Expanding a cell c on the current wavefront consists of computing the cost and address information to propagate to the neighboring processors. As mentioned previously, the address information identifies the expanded cell to the receiving processor. In the HAM system address computation is done using the padding scheme described before. The *expand* cycle takes two CP1–CP2 cycles. The Expand PLA computes the next-cell address during CP1 and latches it onto the output ports during CP2; in the next CP1–CP2 cycle, it does the same using the cost information instead. This address and cost information is received and processed by the update unit of the neighboring processor as described below.

Update: This consists of receiving information (address and cost) of a cell, say c being expanded from the input ports, accessing the grid memory for the current status of c and updating the information as dictated by the maze algorithm. The A/D latches serve the role of address and data registers for this purpose. The whole operation is designed to complete in two CP1–CP2 cycles and is referred to as one *update cycle*. During the first CP1 on-period, the A latch gets loaded with the address for cell c. This value is held constant for the rest of the update cycle. The contents of the grid memory for that address is read and is available during the following CP2 cycle at which time it is latched into the D latch. During the second CP1 cycle, the latched cost information for c is compared with the new cost information received from the input port by the Update PLA and a new value (cost and status) is determined and is latched in the Z1 register. Next, during CP2, the new updated information gets written into the grid memory. This completes the update cycle and the processor then proceeds to receiving a new set of ⟨address, cost⟩ information pertaining to other cells being added to the wavefront.

Fig. 9 summarizes the above activities for one expand and one update cycle. One way to view this is as a global pipelining of expand and update operations taking place across the processor array. In the steady state, each cell expand takes 14 clock cycles: 2 to get the next cell on the wavefront; 2 for expanding to one neighbor (for six neighbors). A read or write is performed on the memory whenever CP2 is on. The backtrace phase operates along the same lines as the propagation phase with the exception that there is no longer any need for the cost information. Backtrace proceeds by passing the address of the next cell, if it is to be included in the final net-route; 0, otherwise to the neighboring PE. The backtrace logic is considered as part of the Expand unit and is discussed later. Backtrace needs 8 clock cycles.

D. Input and Next-Cell Units

The input and next-cell units store cost information from the neighboring PE's and from the memory. The *Input* unit is made up of two 10-bit latches A1 and D1. The two also serve as the address and data-registers for the grid memory. Their function is controlled by 3 microcode bits: *ald1, dld1*, sand *a1bus*. The last is used to determine which of the latches actually place data on the A1 bus. The *Next-cell* unit comprises of latches A2 and D2 which serve a similar role.

E. Update Unit

The main function of the Update unit is to determine the new contents of the grid-memory during the expansion phase. It has 4 modes of operation controlled by the microcode bits *fu1* and *fu2*.

fu1	*fu2*	function
0	0	SEL A
0	1	SEL B
1	0	INC A
1	1	MAZE

Maze Operation: This serves to perform the Update algorithm and is implemented as a PLA. The Update unit uses two sets of inputs. One set pertains to the grid-memory contents of the cell *ao* being updated. This has three components ⟨*co, mo, so*⟩ which denote the current lowest cost to reach the cell, the directions from which the cell has been reached so far and the status of the cell. The second set of inputs has two components ⟨*cn, mn*⟩ where *cn* is the current cost to reach the cell; *mn* represents the direction of the sending processor w.r.t. this PE (i.e., east, or west neighbor, etc).

The Update algorithm first checks if this cell is the target. If so a special *end* signal is activated which is caught by the GCU and used to terminate the wavefront expansion. Otherwise, if it is a new cell, then the new cost is used and the cell status is changed to *wavefront*. It is also possible for the same cell to be reached from more than one neighboring directions. If the new cost is less than the current lowest cost path, then it becomes the new lowest cost and the direction information is updated accordingly; if it is the same then only the direction part gets affected; otherwise the old memory contents remain unaffected.

Update Algorithm

```
Begin Update
 If old status (so)is FREE or TARGET)
    /*this is the first time this cell
          has been hit*/
    or (status is WAVEFRONT OR EXPANDED
          and co > cn)
    /*the new path is the least cost path,
          so accept it*/
 Then   CostOut = cn
        MaskOut = mn
        StatOut = WAVEFRONT
 Else
        CostOut = co
        StatOut = so
        If (status is WAVEFRONT or EX-
           PANDED and co = cn)
           /*alternate path of same cost*/
           MaskOut = mo|mn /*bitwise or*/
          Else
           MaskOut = mo
          Endif
  Endif
End Update
```

The directional mask information serves two main purposes: (1) it can be used in a flexible manner during the backtrace phase to retrace a path to the source, (2) it can be used during expansion to prevent spurious messages being sent. For instance, say processor 1 expands cell c_1 and propagates the information to processor 2. Then processor 2 marks in its memory that it has received this information from processor 1. Consequently, when processor 2 is expanding c_2 (the cell adjacent to c_1), it need not propagate the message back to c_1.

The output of the Update logic is latched into Z1 based on microcode bit *zld1*. There is another bit *exp* which if set to 1 will force StatOut to EXPANDED during wave-propagate phase and to NET during Backtrace phase. The INC A mode of the Update logic is mainly used during the initialization process to step through the memory addresses in a sequential fashion; the SEL A and SEL B modes are used in the backtrace phase when the update algorithm is to be bypassed. Also, they allow flexibility in performing other computations if so needed.

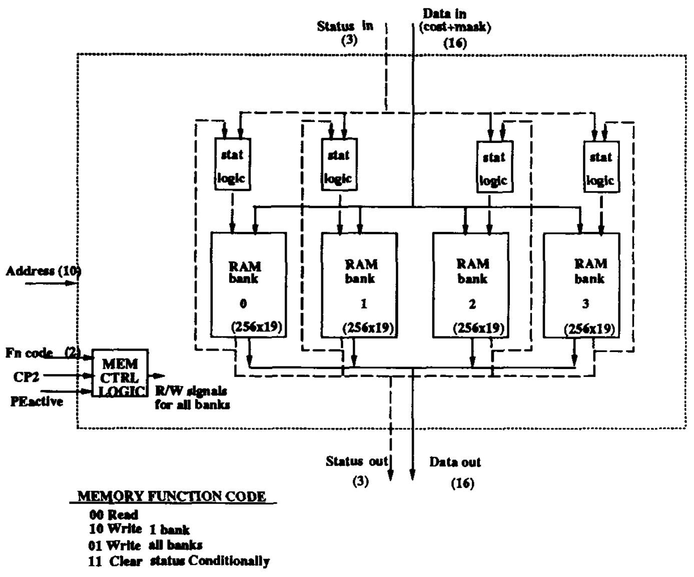


Fig. 10. Local-memory organization.

F. Expand Unit

The Expand unit does the appropriate cost and mask processing needed to expand a cell on the wavefront to the neighbors. It also has logic to account for backtracing and has 4 modes of operation controlled by the microcode bits *fe1* and *fe2*.

fe1	*fe2*	function
0	0	SEL A
0	1	SEL B
1	0	A + B
1	1	A − B

Usually the *B* input consists of data from the register file. This could be the additive factors for the address calculation or the incremental cost to propagate in a certain direction. Note, the output is *zerod* out if the *PeActive* control signal is not *True*.

In addition there is a *backtrace logic* which is operative during the backtrace phase. If it is determined that the cell under consideration has been labeled from multiple directions, then the direction chosen for the backtrace to proceed is the one that causes the fewest number of bends and layers changes. This is done using a priority encoder circuit which compares the direction from which the cell had been expanded and the direction from which the backtrace operation has reached the cell.

G. Local-Memory Organization

This serves to maintain information pertaining to each grid-point that is mapped onto that PE. The memory is implemented using a static RAM of size 1024 × 19. The memory needs to be accessed in every instruction cycle of the propagation phase. Also, during the initial set-up phase and during the cleargrid phase of the algorithm, the status fields have to be updated. To improve performance, it was decided to interleave the memory into 4 banks of 256 × 19 RAM's. The two higher order bits of the address is used to select the appropriate bank. This permits the same location of all banks to be simultaneously written into in one memory cycle.

Fig. 10 shows the grid-memory organization in more detail. The 2-bit *fncode* determines the memory operation. All read/writes take place during CP2. Address and input data are available at the end of CP1 and are held constant during CP2. The *StatusLogic* block is used to selectively change all *Expanded* or *Wavefront* status fields to *Free* when the *ClearStatus* command is issued. This essentially clears the grid and is to be performed upon completion of routing the given net and prior to starting the maze search for the next net. Thus the "clearphase" can be performed by all PE's simultaneously in 256 cycles. The *Memory Control Logic* is responsible for activating the appropriate read or write signals. If the processor is inactive, no write is performed and the *fncode* is in essence disregarded.

H. Buffer Store Organization

During the wavepropagation, while the PE is expanding one cell of the old wavefront, it may receive up to 6 new addresses for other cells that map onto it and have been newly inducted into the wavefront. The PE has to keep track of these since all of them will have to be eventually expanded. The buffer store is needed as just updating the *Status* fields in the grid-memory is insufficient to identify wavefront-cells without undertaking a sequential search.

The buffer store can be organized as either a pair of LIFO stacks or as a FIFO queue. This has been implemented using a 1024 × 10 RAM and a pair of up–down counters. During the wave propagation phase, as each address is received it gets pushed onto the top of the store. An unexpanded wavefront-cell can then simply be recovered by popping the top of the appropriate stack or queue. The counter is assumed to point to the next free location. Thus for PUSH, the RAM is written into during CP1; the counter is incremented in CP2. On the other hand, for POP operation, we decrement the counter in CP1 and access the memory in CP2.

There is one additional complication, however. A particular grid cell can be reached from more than one direction. Consequently, if the address of such a cell is already in the store, then it should not be pushed in. The solution to this problem is to validate each PUSH. The control unit raises a *pushvalid* signal during CP2 if the address corresponds to a new cell. This information is determined by reading in a previous clock phase the corresponding *Status* field in the grid-memory. The counter is incremented only if *pushvalid* is true.

I. Control Unit

The control unit generates 36 microcode bits that are used to control the datapath and the memory units. Currently, the microinstruction memory is implemented as a 32 × 36 static RAM. This means that at any point in time 32 different instructions can be stored. This was felt to be sufficient for the maze-routing algorithms. An initial set consists of 6 wave-expand, 1 wave-receive instruction (needs 3–4 microinstructions), 4 microinstructions for backtrace, 1 for clearing the grid, 3 for resetting various elements, 4 for initializing the status fields of the memory, and 10 for miscellaneous operations. The use of a static memory provides the ability to interrupt clocks between instruction definition and execution.

A 5-bit address (instruction) selects one 36-bit microinstruction every cycle. This gets loaded onto the master register in CP2 and then into the slave during CP1 from where it is decoded appropriately and connected to the different control points. Data can be shifted in and out of the master/slave registers by setting SH. To load a new instruction into the memory, the data is serially shifted in (SI port) for 36 cycles and then WRT is activated to store the contents into the memory. The *Csel* or chipsel input is used to disable a particular PE. This is useful for the initial setup of the grid-memory and final result gathering operations. When *Csel* is enabled the slave register is loaded with a microinstruction implementing the *no-op* operation. The control-block is combinational in nature and is used to activate the various control signals in the appropriate clock phase and also decode some fields of the microinstruction. (See Fig. 11.)

J. Testing

The chip has been designed keeping in mind the testing requirements. Testing can be done in two parts. In the first part the control section can be tested out by shifting in data into the pipeline register and observing the output at the PSO port. Once the pipeline register is verified, it can be used to test the microinstruction memory by storing data at specific locations and then loading them back into the pipeline register and then shifting the data out. Subsequently simple instructions can be loaded into the RAM to test the functionality of the data path. The contents of all the datapath elements are observable at the output port by activation of the appropriate control signals. For instance, the operation of the A1 latch can be tested by loading test data from the input pins and then enabling the connection between the two R buses, the same data can be observed and verified at the output. Once this operation is verified, the A1 latch can be made to store a grid memory address and test data can be written into and subsequently read out from the memory.

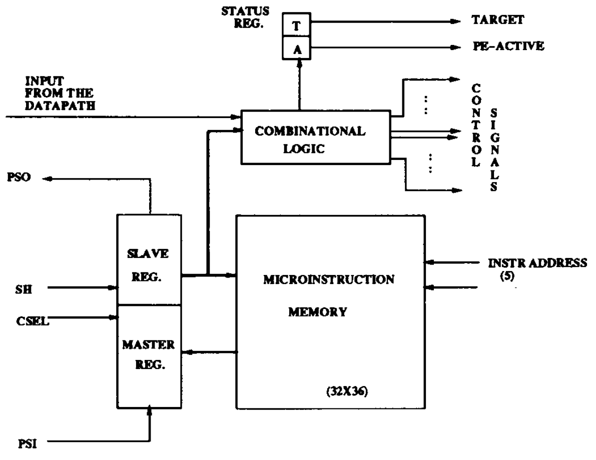


Fig. 11. Control unit organization.

TABLE II
Area for the Major PE Blocks

Unit Name	Area (100 sq. mils)	Unit Name	Area (100 sq. mils)
Input unit	1.79 (0.6%)	Control unit	27.47 (9.1%)
Next-cell unit	1.79 (0.6%)	Datapath	53.30 (17.6%)
Update unit	4.52 (1.5%)	Buffer store	65.36 (21.6%)
Expand Unit	3.78 (1.25%)	Grid Memory	144.32 (47.7%)

K. Statistics

A single processing element has been laid out in a 40-pin package with a die size of 200 by 220 mil (see Fig. 12) using the Chipcrafter[1] package. This includes 10 kbit of buffer-store memory and 19 kbit of grid memory which is sufficient for routing grids with as many as 64K grid cells on a five-dimensional processor array comprising of 61 processors. A 1-μm 2-metal 1-poly technology from National Semiconductors was employed. Table II shows the area in units of 100 sq. mils for the major components with the percentage of total chip area shown in parentheses.

The processor runs at a clock frequency of 16 MHz. An expand cycle takes 0.84 μs while a backtrace cycle takes 0.48 μs. Thus the HAM system is capable of sustaining a little over 1 million expansions every second. These figures do not include the time for the initialization and communication costs with the host. However, such costs can be amortized over

[1]Chipcrafter is a trademark of Seattle Silicon Corp.

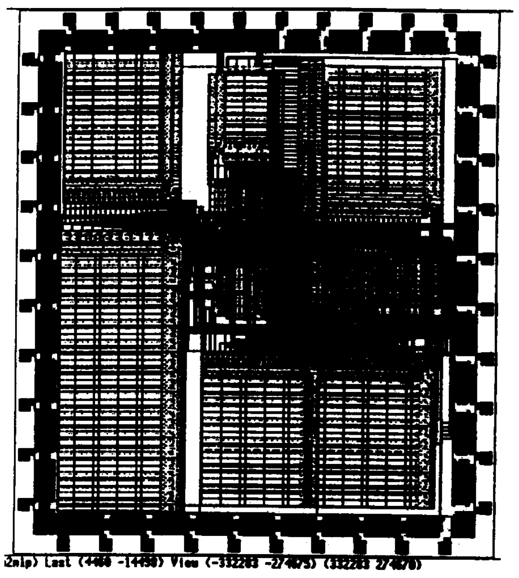


Fig. 12. Layout of a single HAM processor.

several nets and so HAM will continue to offer significant speedup over any uniprocessor solution.

Clock Frequency: This is determined by the following considerations.

1) The propagation delays of the *Update* and *Expand* units (denoted by t_{du} and t_{de}, respectively).
2) The setup and hold times of the various RAM components (grid memory, buffer memory , control memory) which are denoted by t_{su} and t_{hd}, respectively.
3) The access time (time before valid data is available at the output for a read operation or the minimum time for which the write pulse has to be activated to write in new data) which is denoted by t_{ac}.
4) The time for the incrementing and decrementing of the counters which are part of the buffer store address circuitry for implementing the push and pop operations which is denoted as t_{ct}.
5) Interprocessor (t_{s1}) and intraprocessor (t_{s2}) signal skews.

The inputs to the update unit (A1 and B1 bus) change at the start of each CP1 and the output forms the new data which is to be written during the CP2 on-period in the grid memory. Similarly, for the expand unit the inputs (A2 and B2 bus) are available at the start of CP1 and the output is latched onto the output Z register during CP2. These requirements give rise to the following set of constraints:

$$\begin{aligned} t_2 + t_3 &\geq t_{du} + t_{su} \\ t_2 + t_3 &\geq t_{de} \\ t_4 &\geq t_{ac} \\ t_1 &\geq t_{hd}. \end{aligned}$$

In case of the buffer store, a *pop* operation consists of decrementing the counter during CP1 and reading the corresponding memory address during CP2. For the *push* the reverse is performed, i.e., a write is performed during CP1 and the counter is incremented during CP2. Thus the counter always points to the next free location. Note that the terms increment/decrement are interchanged for the bottom stack in the two-stack synchronous mode of wavefront expansion. This leads to the following constraints:

$$\begin{aligned} t_2 + t_3 &\geq t_{ct} + t_{su}\,(pop) \\ t_4 + t_1 &\geq t_{ct} + t_{su} \\ t_2 &\geq t_{ac}; t_3 \geq t_{hd}\,(push) \\ t_4 &\geq t_{ac}; t_1 \geq t_{hd}. \end{aligned}$$

The case of the control memory and grid memory which are only read/written during CP2 is quite simple, viz., $t_3 \geq t_{su}; t_4 \geq t_{ac}; t_1 \geq t_{hd}$. The interprocessor communication delays are accounted for by t_1 since data are latched on to the output ports during CP2 and that data are received by the neighboring processor during the following CP1. Hence, it is sufficient that $t_1 \geq t_{s1}$. The value of t_{s2} is determined by the manner of satisfying the above inequalities.

The measured values for the above parameters were: $t_{ac} = 12.5$ ns$, t_{su} = 10$ ns$, t_{hd} = 5$ ns$, t_{de} = 23.8$ ns$, t_{du} = 15$ ns, $t_{ct} = 10$ ns. Our simulations was performed setting $t_i = 15$ ns$, i = 1, \cdots, 4$. These satisfy all the above criteria and can tolerate a signal skew of 8% of the total clock cycle in the datapath and 4% for the memory control. Fig. 13 shows a part of the actual Quicksim[2] trace for the processor. The signals *ald1* to *dld2* are control signals controlling the loading of the *A/D* latches and signals *bus1, bus2* determine the A bus gets sourced by the *A*\ or *D* latch. The nets *a1bus, b1bus* are the *A* and *B* inputs of the Update unit; while the signals *mold, min, camin, alu1out,* and *mout* correspond to *mo, mn, so, CostOut* and *MaskOut* of the Update algorithm respectively. Similarly the nets *a2bus* and *b2bus* are the *A* and *B* inputs of the Expand unit and its output is latched onto the output pins. For illustration purposes, at the start of the trace, all grid memory cells are initialized to zero except cell 2: ⟨ *cost* = 20; *dmask* = 1 ⟩ and cell 3:⟨ *cost* = 16; *dmask* = 0 ⟩. The buffer store has one cell address, viz. 3. The StartExpand cycle starts at time 1440. Thus in the expand cycles, subsequent to this, the processor outputs neighbor address of (3 + *offset*) for that direction as per the padding scheme; and cost of 16 + 1 = 17. Concurrently the Update unit receives address and cost data from its neighbors and proceeds to modify the grid memory as per the Update algorithm.

V. Performance Analysis

In this section, we present simulation results pertaining to the performance of the overall HAM system using custom designed processors, working in both the synchronous (SYNC) and asynchronous (ASYNC) expansion modes. All figures are computed assuming that the *shortest path* to the target is desired and not just any path. This assumption could have a significant impact especially on the ASYNC mode results. We have used three main criteria for our evaluations: execution time (τ), speed-up (s), and processor usage efficiency (η) and study how they change with N, the number of processors used.

[2]Quicksim is a trademark of Mentor Graphics.

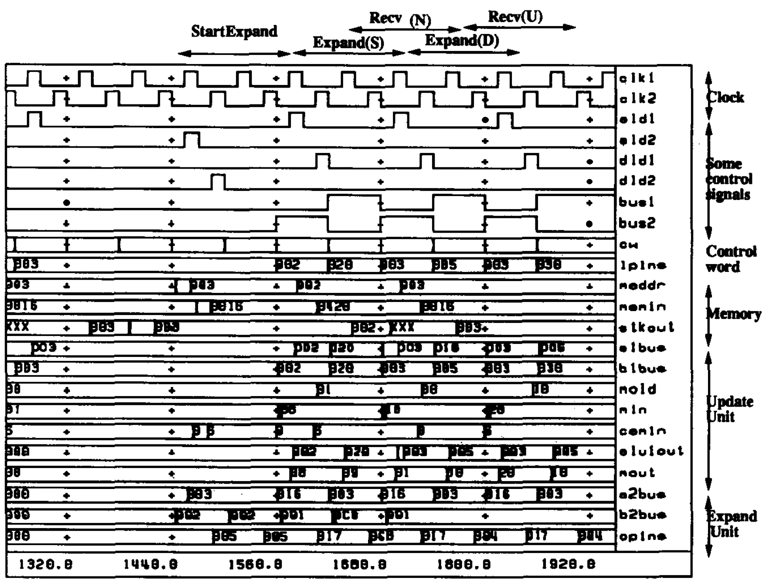


Fig. 13. Simulation trace for the HAM processor.

All plots reflect the average obtained by running the system simulation on 25 randomly generated nets on a grid of size 100 × 100 × 4. We also consider the effects of framing, i.e., restricting the grid-space to be searched for a connection to a rectangular box formed by the source and target; and the effect of blockages caused by prior nets.

A. Execution Time

We have chosen to characterize the execution time in terms of the number of *atomic* cycles (expand or update cycle) required to complete expansion. This makes the results to be readily applicable to other possibly faster implementations. For our processor implementation, an atomic cycle corresponds to 14 × 60 ns = 0.84μs. These results are shown in Fig. 14. The asynchronous mode takes more time than the synchronous mode for empty grids. The reason for this is that in the current implementation, the processor merely picks out the first entry in its buffer store which could very well be a cell leading away from the target. This leads to the domino effect wherein a message pertaining to a higher cost message can propagate first to the target. Subsequently, when the correct update message with a lower cost is received by the processor, the expansion in a sense gets repeated. This process leads to many more messages being sent back and forth which increases the total time. The solution to get around this problem is to have the processors make an intelligent choice as to the next cell to expand but this would involve more complex hardware. Note that the two modes yield similar results when framing is used or in congested grids with lots of cell blocks. This is because the chances of first proceeding in the wrong direction are considerably reduced here. In fact, it is possible that the ASYNC mode may be the faster of the two in such cases. Also it is clear that if the length of a net is less than N then no advantage can be gained by increasing the number of processors. This is reflected in the graphs where it can be seen that the curves tend to flatten out as N becomes larger.

B. Speedup

The speedup is measured with respect to the corresponding time taken by a uniprocessor which is directly proportional to the total number of cells expanded. Thus

$$s = \frac{\begin{array}{c}\text{total number of cells expanded}\\ \text{on a uniprocessor}\end{array}}{\begin{array}{c}\text{number of atomic cycles taken by}\\ \text{the multiprocessor}\end{array}}.$$

Note that this speedup value is a lower bound as it does not include consideration for the smaller expand cycle time of the HAM processor. So, in absolute terms, the expected speedup will be much more. The results are shown in Fig. 15. Again the lower speedup for the ASYNC mode for empty grids without framing is a direct consequence of the increased time taken in this mode to find connections.

C. Usage Efficiency

The efficiency is defined as the overall processor usage measured over the whole of the program's execution. It is calculated as follows:

$$\eta = \sum \begin{pmatrix}\text{total number of active processors}\\ \text{per expand cycle}\end{pmatrix} \cdot (N * \tau)^{-1}.$$

An active processor in this context is one which either receives at least one message from one of its neighbors or sends a message to its neighbors. The high efficiency figure for the asynchronous mode is a direct consequence of the routing policy of allowing the processor to expand any cell in its buffer store. The results are plotted in Fig. 16.

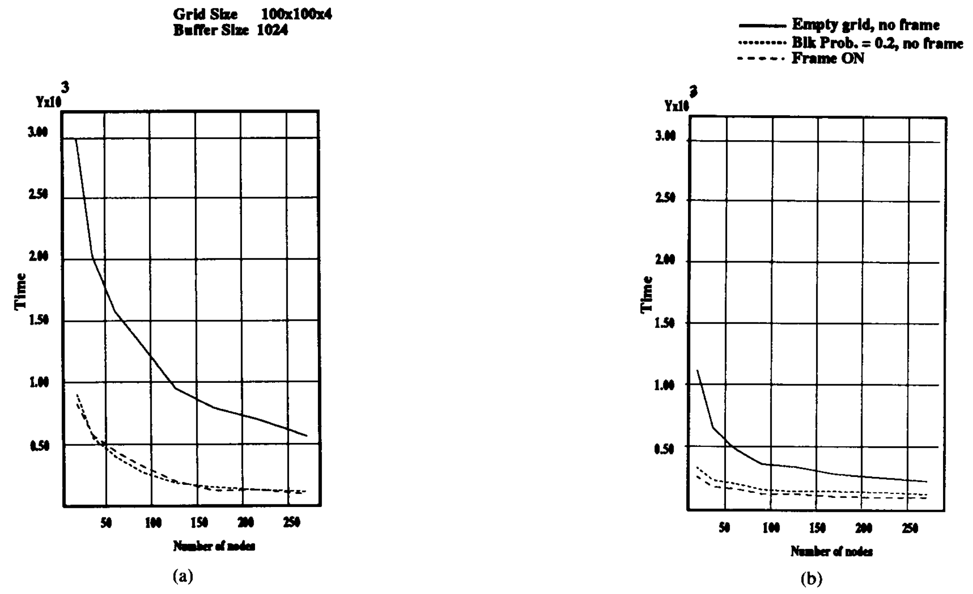


Fig. 14. Effect of number of processors on total time. (a) ASYNC mode. (b) SYNC mode.

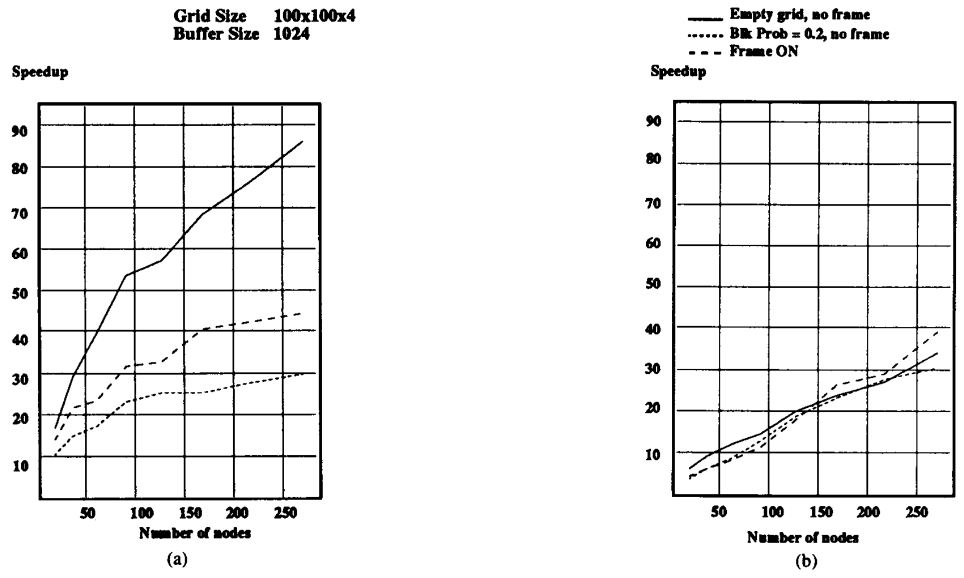
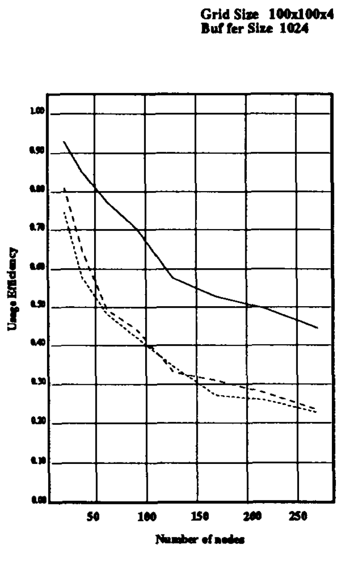


Fig. 15. Effect of number of processors on speedup. (a) SYNC. (b) ASYNC.

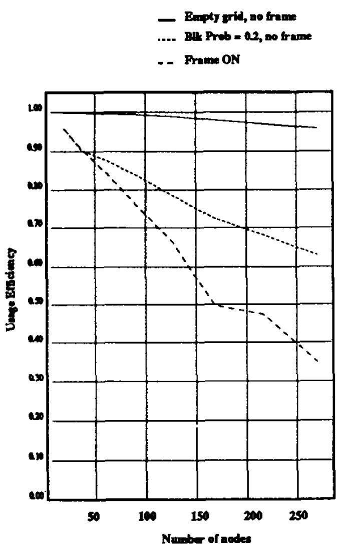
D. Size of the Buffer Stores

It is clear that the maximum size needed for a buffer store is equal to the maximum number of cells that are mapped to the processor. However, by considering the manner in which wavefronts propagate, it was felt to be highly unlikely that all cells could be simultaneously part of the current wavefront. In fact simulations have led us to believe that the maximum number of elements present in the buffer at any given time is less than 10% (25%) of the total number of cells mapped to the processor for the SYNC (ASYNC) modes. This can be seen in Fig. 17 where the maximum buffer sizes used while routing on an empty 100 × 100 4-layer grid are shown. The theoretical maximum for a k-layer $G_x \times G_y$ grid is given by $\lceil k * G_x * G_y/N \rceil$. Generally, the asynchronous mode requires 3–4 times larger buffer stores since there are more messages being transmitted.

These results suggest that a significant area saving can be realized by simply using a smaller buffer store. Alternatively, the area can be used to build a larger grid memory that allows the mapping of even larger routing grids. Also, because there is some intrinsic redundancy in the expansion, such as each cell receiving information from more than one direction (pro-



(a)



(b)

Fig. 16. Effect of number of processors on efficiency. (a) SYNC. (b) ASYNC.

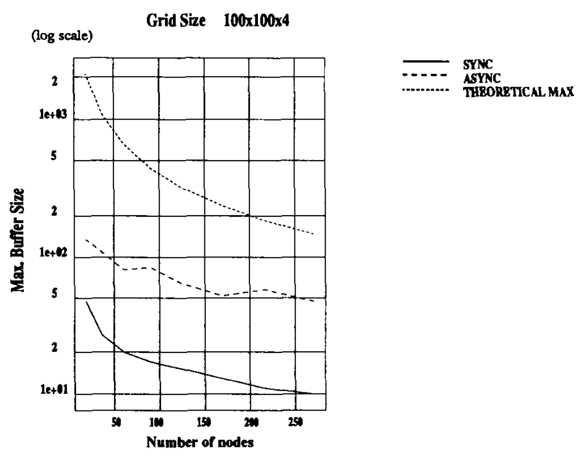


Fig. 17. Determination of maximum buffer store size needed.

cessor), routing is not much affected even if a few messages are lost because the processor receiving it has no place to store it in its buffer.

E. Choice of Mode

From the above experiments it can be concluded that unless framing techniques are used or the grid is congested, the synchronous mode is better. The amount of buffer store needed for synchronous mode is also less since we only expand a wavefront at a time which leads to a more uniform distribution of grid cells to processors. Framing techniques are also used in software methods to restrict the amount of grid space that gets searched by the wave propagation, but seem to have a different implication in the multiprocessor mode. Though not currently implemented, framing could be realized in the HAM system by marking the cells lying on the frame boundary to *Block* before starting routing the net and restoring their original state upon completion of the route. However, this is more complicated to compute in the distributed memory map since a transformation will have to be made between frame coordinates and memory addresses. Consequently, our solution (without framing) is to employ synchronous mode initially and then resort to asynchronous mode as the grid gets more congested. Both models can be supported using the two-counter model discussed earlier.

VI. Conclusion

The HAM system derives its speedup over conventional solutions in two ways: (1) Since the processing elements are custom-designed, the per cell computation time is significantly reduced for both wave-expansion and backtrace operations. This time includes the time to fetch the status of the cell from memory, perform cost calculations and add it to the new wavefront list (in our case propagate to the six adjacent neighbors). This paper has identified the hardware requirements that are most cost-effective in building such custom processors. (2) The other speedup results from the manner in which the processing elements cooperate with one other during routing. Expand and update operations are pipelined across the processors which are connected in a hexagonal wraparound fashion. Such a topology has been previously shown to be optimal for concurrent multilayer search operations in three dimensions. This meets our goal of using the HAM system for double-sided surface-mounted board routing. However, the processor design is independent of the interconnection topology used; rather the processors can be interconnected in any manner desired and run with suitable microprograms.

A multiprocessor system solution for maze routing poses several problems not encountered with uniprocessors. One factor of significant import in a distributed memory model is that each processor has only limited information of the overall grid, in particular, it only stores the status of cells which are assigned to it. This has consequences for the mode of interprocessor information exchange, control mechanism, and synchronization scheme employed. In this paper, we have identified these issues and proposed some practical solutions. In particular, we have suggested the use of memory-address rather than the traditional grid-coordinate based message transfer between processors during expansion as a means to reduce both message traffic as well as speed up the memory search times. The design of the buffer store as either a stack or a queue to support either a synchronous or asynchronous mode of expansion has also been shown to be critical in achieving good performance.

REFERENCES

[1] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Trans. Electron. Comput.*, vol. pp. 346–365, 1961.

[2] R. Venkateswaran and P. Mazumder, "A hexagonal array machine for multi-layer wire routing," *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 1096–1112, Oct. 1990.

[3] J. Soukup, "Fast maze router," in *Proc. 15th Design Automation Conf.*, pp. 100–102, June 1978.

[4] D. A. Edwards, "MANURE2—A second generation accelerator for PCB routing," in *CAD Accelerators*, pp. 219–233, 1989.

[5] S. Sahni and Y. Won, "A hardware accelerator for maze routing," in *Proc. Design Automation Conf.*, pp. 800–806, 1987.

[6] R. A. Rutenbar and D. E. Atkins, "Systolic routing hardware: Performance evaluation and optimization," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 397–410, Mar. 1988.

[7] R. Nair, S. J. Hong, S. Liter, and R. Villani, "Global wiring on a wire routing machine," in *Proc. Design Automation Conf.*, pp. 224–231, June 1982.

[8] H. G. Adshead, "Employing a distributed array processor in a dedicated gate-array layout system." in *Proc. ICCC*, pp. 411–414, Oct. 1982.

[9] K. Suzuki, Y. Matsunaga, M. Tachibana, and T. Ohtsuki, "A hardware maze router with application to interactive rip-up and reroute," *IEEE Trans. Computer-Aided Design*, vol. 5, pp. 466–476, Oct. 1986.

[10] T. Blank, M. Stefik and W. van Cleemput, "A parallel bit map processor architecture for DA algorithms," in *Proc. Design Automation Conf.*, pp. 837–845, 1981.

[11] J. Cooper and D. Chyan, "Autorouting today's high density PCB's," *Printed circuit design*, pp. 36–46, Oct. 1988.

[12] A. Iosupovici, "A class of array architectures for hardware grid routers," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 245–255, Apr., 1986.

[13] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Reading, MA: Addison Wesley, 1986.

[14] R. Goering, "Design automation," *High Performance Syst.*, pp. 20–40, Dec. 1989.

[15] P. Lund, *PCB Precision Artwork Generation and Manufacturing Methods*. Bishop Graphics, Inc., 1986.

[16] J. Soukup, "Circuit layout," *Proc. IEEE*, vol. 69, pp. 1281–1304, Oct. 1981.

[17] H. Schutzman, "A behind-the-scenes look at autorouting," *Printed circuit design*, pp. 34–40, Dec. 1988.

[18] T. Blank, "A survey of hardware accelerators used in CAD," *IEEE Design and Test*, pp. 21–39, Aug. 1984.

[19] D. Hicks and D. Roach, "Implementing a parallel router with RISC technology," *High Performance Syst.*, pp. 61–64, Mar. 1990.

Ramachandran Venkateswaran (S'89) received the B.Tech. degree in computer science from the Indian Institute of Technology, Bombay, in 1988, and the M.S. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1992. He is currently working toward the Ph.D. degree at the same university.

His areas of interest include design automation with particular emphasis on wire layout problems and VLSI system design. Other research interests include parallel architectures, fault tolerant computing and neural networks. In the summer of 1991, he worked at the Thomas J. Watson Center, IBM, Yorktown Heights, NY, on hierarchical compaction.

Mr. Venkateswaran has received the IBM Graduate Fellowship in Computer Science during 1991–1993. He is a member of ACM SIGDA.

Pinaki Mazumder (S'84–M'87) received the B.Sc. degree in physics from Gauhati University, India, the B.S.E.E. degree from the Indian Institute of Science, Bangalore, and the M.Sc. degree in computer science from the University of Alberta, Canada, in 1985, and the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1987.

He has worked over six yeras as a Senior Design Engineer at Bharat Electronics Ltd., India (a collaborator of RCA-GE) in its integrated circuits design-and-application laboratory. During the summers of 1985 and 1986 he was a Member of the Technical Staff at AT&T Bell Laboratories. Currently, he is an Associate Professor in the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. His research interests include VLSI testing, computer-aided design, and parallel architecture.

Dr. Mazumder has received Digital's Incentives for Excellence Award, NSF Research Initiation Award, and Bell Northern Research Laboratory Faculty Award. He is a member of Phi Kappa Phi and ACM SIGDA.