A STRUCTURED APPROACH TO VLSI LAYOUT DESIGN

M.S.KRISHNAN

XEROX Corporation
LSI Development A3-74
701 S.Aviation Blvd.
El Segundo, CA 90245

ABSTRACT

A new approach to the VLSI layout problem is proposed that produces a structured floor plan for an arbitrary network of interconnected processing elements. It is based on extracting a minimum spanning tree from a given representation of a computation network and using an efficient, structured layout scheme for this minimum spanning tree. Techniques to lay out trees as arrays of layout slices are presented. It is assumed that the nodes of a network are identical in their layout size and connectivity. This method is valid at any level of a VLSI design since these nodes may represent gates, cells or complex macros. An application of this approach to modified tree networks is described. Other useful applications of the method are mentioned.

1. INTRODUCTION

A significant portion of a VLSI chip of any reasonable complexity is consumed by the communication paths among the various macros comprising the chip. This situation is further aggravated by several factors:

a) Decreasing feature sizes, e.g. transistors and wires, which cause communication delays to decrease non-proportionately with gate delays.
b) Presence of random logic with the attendant interconnection that is also irregular.
c) Lack of adequate design aids that guide a designer along a structured, hierarchical design sequence that is also streamlined to provide masks within a short time.

Traditionally, the two major phases of digital system design, namely logic design and physical, or more appropriately geometric design, have been treated as separate, sequential operations. This methodology was generally adequate before the LSI revolution. With the enormous computing power and hence complexity present in a network of processors in one chip, the separation of these two tasks tends to overlook the effects of one on the other. It is essential for a VLSI designer to be able to evaluate the effects of his logic design on the layout and vice versa early enough to incorporate them into his design. Yet there is a lack of adequate design aids for evaluating the potentials of alternative chip designs carried through the logic design to the floor plan of the chip.

The major goal of the work described in this paper is the development of design aids that will produce structured chip layouts that are efficient in area and are easy to generate. The problem of regular and/or area-efficient layouts for VLSI has generated considerable interest recently [7]. The Bristle Blocks approach [6] attempts to develop cells that have built-in stretching points so that neighboring cells may be made to conform to the same pitch. Leiserson [2] describes a divide-and-conquer approach to the layout problem wherein any planar graph that satisfies the conditions of the separator theorem of Tarjan and Lipton may be recursively bisected by removing edges until subgraphs realizable as rectangular layouts with the desired aspect ratio are obtained. These are then recursively connected by restoring the deleted edges. Brent and Kung [1] proposed a regular layout for a carry-lookahead addition scheme.

The present work attacks the problem through the creation of regular "layout slices" for a few commonly found computation structures -- tree, cube, hexagonal array etc. and treating a given computation network as a composition of instances of these structures. Thus the tasks of logic design and geometric design are being absorbed into one wherein the impact of one on the other can be handled systematically.

Section 2 describes some regular layout schemes for trees proposed in the literature. Section 3 presents some new layout schemes that are implementable as arrays with useful properties. These schemes are applied to modified tree networks in Section 4. Section 5 describes an algorithm to generate the floor plan for an arbitrary computation network of interconnected processors. Some potential applications of this method are listed in Section 6.

2. LAYOUT SCHEMES FOR TREES

A structure that has a wide range of applications from multiplexors, decoders etc. to multiprocessors is a tree network. A regular layout for the carry chain computation in an adder has been proposed [1]. It is actually a set of trees with common inputs and is illustrated in Fig. 1. Its area is O(nlog n) where n is the number of leaf nodes.

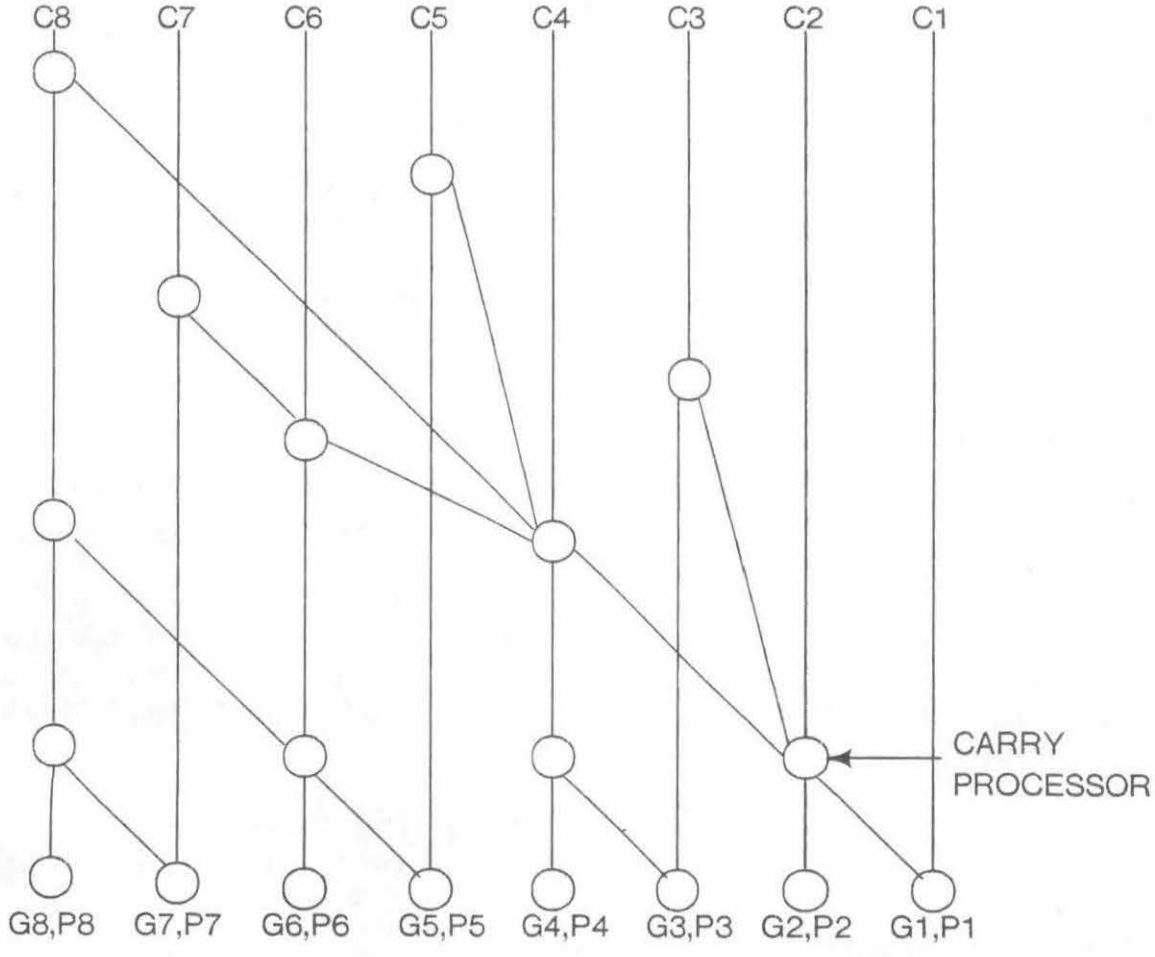


Fig. 1. An O(nlogn) layout for a carry lookahead adder tree

An H-tree layout for binary trees has been proposed [9] that is more space-efficient. The H-tree requires O(n) area and is shown in Fig. 2.

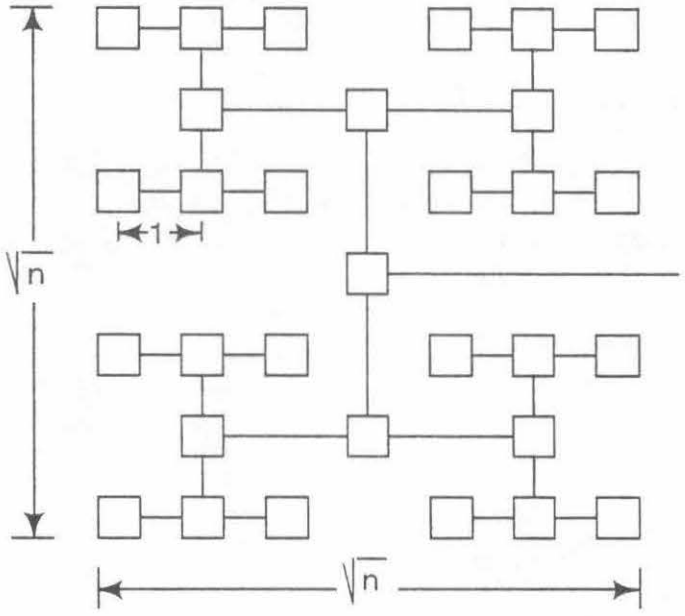


Fig. 2. The H-Tree Layout for a Complete Binary Tree

Several observations can be made on these layout schemes :

a) The tree layout scheme of Fig. 1 has all the leaf nodes at one end and the output nodes at the other. Thus data travels in one direction only and a total distance proportional to log n.

b) The H-tree scheme has leaf nodes spread throughout the interior as well as the periphery of the square area. The data flow alternates between the two directions from level to level. The total distance traversed by data from the leaf nodes to the output, using the unit shown in Fig.2, is :

$$T_d = \begin{cases} 1.5\ 2^k & \text{for } n = 2^{2k} \text{ leaf nodes} \\ 2^k + 1 & \text{for } n = 2^{2k+1} \text{ leaf nodes} \end{cases}$$

This delay is proportional to sqrt n.

c) Both schemes grow in both directions as the tree expands. In both, the number of nodes in either direction is not constant and varies across the layout. Therefore neither scheme is suitable for realization of a tree as a one-dimensional array.

3. LAYOUT SLICE SCHEMES FOR TREES

We develop, in this paper, a structured layout scheme as a one-dimensional array of "layout slices". The new layout technique places the leaf nodes along the edges for ease of routing, minimizes the data propagation time so that the latency time of the tree as a segment in a pipeline is minimized. The ease of access to the leaf nodes is critical in applications where not only the root but also the leaf nodes of the tree communicate with other macros on the chip.

3.1 BINARY TREES

An algorithm to generate the layout for a complete binary tree is given below. The numbering notation used in this paper for levels of nodes is shown in Fig. 3.

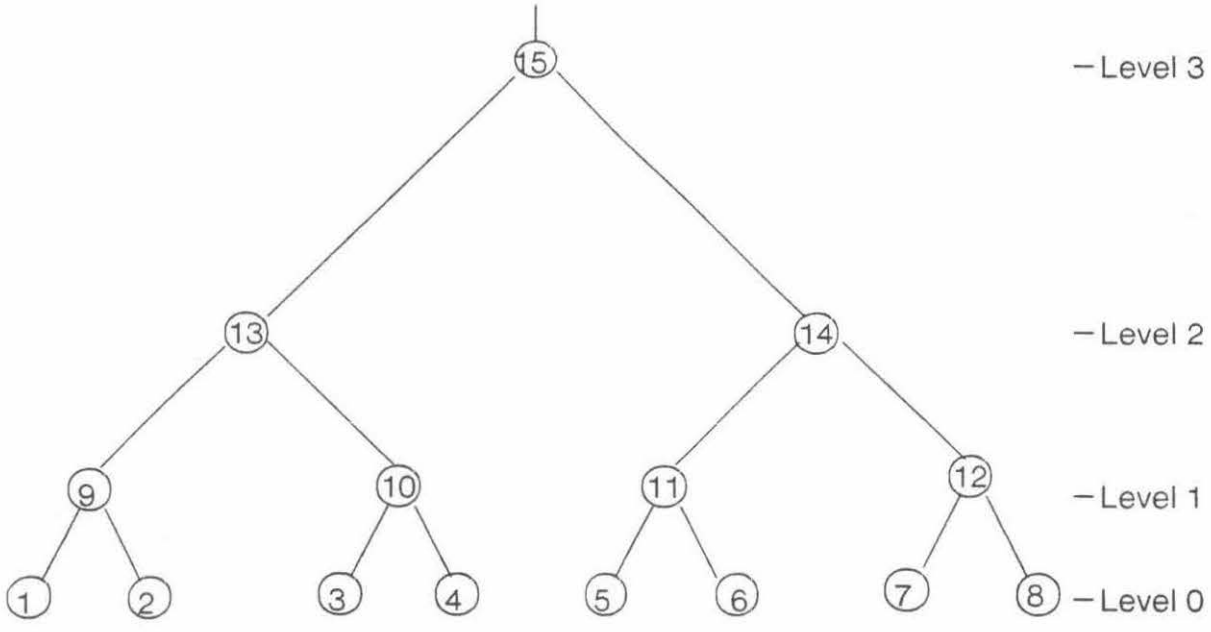


Fig. 3. A Logical Binary Tree with Eight Leaf Nodes

ALGORITHM 1 (Algorithm for the layout of a binary tree) :

Let the binary tree have $n = 2^k$ leaf nodes for some integer k. The two main tasks in the layout process are placement of the nodes and interconnections among them.

1. PLACEMENT

   a) Traverse the tree in order.
   b) Group the nodes in the traversal into pairs.
   c) Assign every pair obtained above to a layout slice.

2. INTERCONNECTION

   a) The connections to the leaf nodes (level 0) are straightforward since they receive external inputs.
   b) For nodes at higher levels in the tree, the level number of a node in a given slice can be determined in a simple manner. For nodes at level 1, the inputs are from within the same slice and the slice immediately to its right. For nodes at level i, $i > 1$, the inputs are from the slices 2^{i-2} positions to the left and 2^{i-2} positions to the right. □

The algorithm is illustrated for a tree with $n = 8$ in Fig. 4. Step 1 of the algorithm yields (1,9) (2,13) (3,10) (4,15) (5,11) (6,14) (7,12) (8). Step 2 can be observed from Fig.4 which shows the realization of the tree.

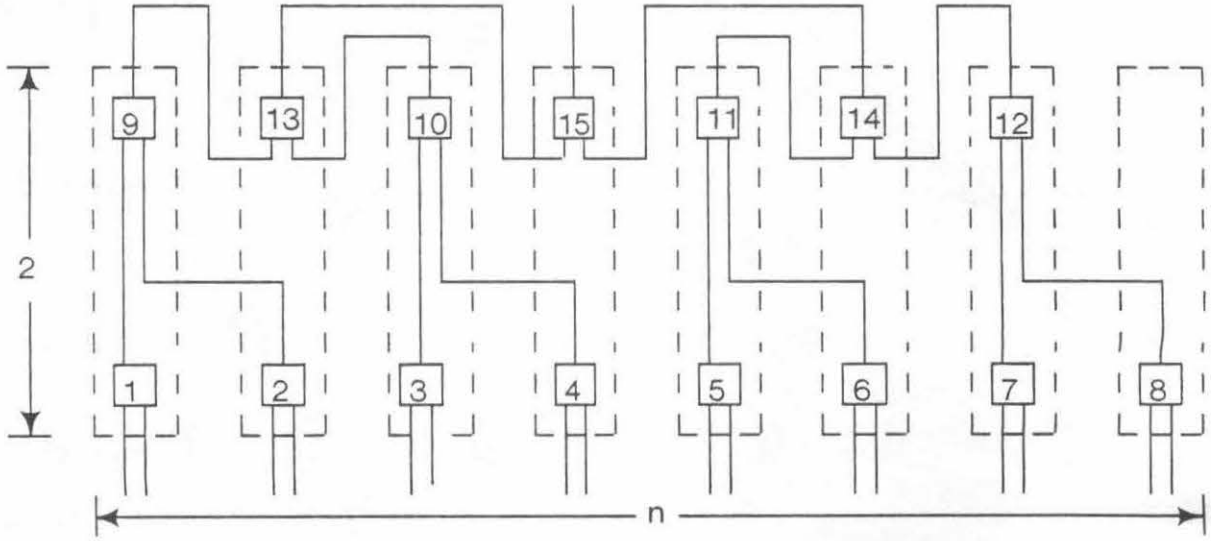


Fig. 4. Realization of a tree as an array of "layout slices"

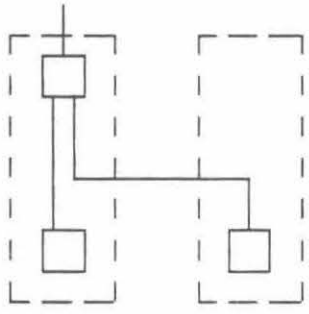
The following lemma determines the number of slices required for such a layout and the bounds on the degree and eccentricity of a slice.

LEMMA 1 :

A complete binary tree with n leaf nodes can be realized in area O(n) as an array of n layout slices where each slice contains exactly two nodes with a) at most four wires connected to it and b) at most log n · 2 wires passing around it without connection, where a wire is an interconnect between two nodes in the tree.

Proof:

We prove the first part of the lemma by induction on the number of leaf nodes. For a complete tree, $n = 2^k$ for some integer k. For $n = 2$, we need two slices as shown below:



Consider a tree with 2^m leaf nodes.

The total number of nodes in the tree $= 2^m + 2^{m-1} + \ldots + 2^1 + 2^0$

$= 2^{m+1} \cdot 1$

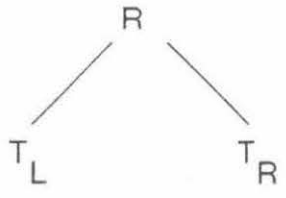
$= 2(2^m) \cdot 1$

$= 2(n) \cdot 1$

This shows that the nodes of the tree can be assigned to the 2^m slices, two nodes to a slice, such that exactly one half of one of the slices is unused.

We now apply induction to a tree with 2^{m+1} leaf nodes. Let

$T_{m+1} =$



be a complete tree with 2^{m+1} leaves where R is the root node and T_L and T_R are the left and right subtrees of T_{m+1} with 2^m nodes each. By the induction

hypothesis, both T_L and T_R have structured layouts with 2^m slices each. But we have shown above that the layout for T_L has a slice, say S, that has an unused node. Let R be assigned to this unused slot and connected to the roots of T_L and T_R. This results in a layout for T_{m+1} as an array of 2^{m+1} slices with each slice containing two nodes. An interesting property of the placement strategy is that every slice has exactly one leaf node and one node from a higher level. This follows from the in-order traversal of the tree. Thus the area of the layout, using any choice of units is O(n). The maximum propagation length from the leaf nodes to the root is n/2. The maximum number of wires across any vertical cross section of the layout is log n.

b) Let the levels of the tree be numbered with the leaf nodes at level 0 and let l(P) denote the level of node P. As mentioned above, one of the two nodes in a slice is from some level > 0. Let P be such a node. There can be no wires passing around the slice containing P that connect to to a node at level l(P) or less since, by construction, none of the left sons of node P are to the right of P. The maximum number of wires passing around a slice occurs for the case l(P) = 1 for which there are log n - 1 levels above it. However, the root node is not connected to any other slice and hence there are at most log n - 2 wires passing around a slice. The number of wires connected to a slice is maximum when l(P) ≠ 1 and can be seen to be four. □

The number of wires passing around a slice was treated specifically above to bound the width of the routing channels although these wires may be run through the cells. Only one half of one slice is unused. Note in Fig. 4 that there are at most log 8 - 1 i.e. 2 wires passing around a slice. The output from a node at level i passes around $2^{(i-1)} - 1$ slices using this arrangement of nodes.

3.2 TREES WITH LARGER FANOUT

The above scheme for binary trees can be generalized to any k-ary tree as stated in the following lemma.

LEMMA 2 :

Any k-ary tree where each node has k inputs and one output, with n leaf nodes can be realized in area O(n) as an array of n layout slices where each slice contains at most two nodes with a) at most k + 2 wires connected to it and b) at most log n - 2 wires passing around it without connection.

Proof:

The construction of a k-ary tree layout is similar to that of a binary tree. At each level i, a node is placed in the same slice as its $\lceil k/2 \rceil$ th input for i = 1 and in the slice k^{i-2} positions to the right of its $\lceil k/2 \rceil$ th input for i>1 so as to place at most two nodes in every slice. The properties of this layout follow from arguments similar to those for Lemma 1. □

The placement criterion stated above is illustrated for a ternary tree with n = 9 in Fig. 5. Note that this criterion is also true for a binary tree.

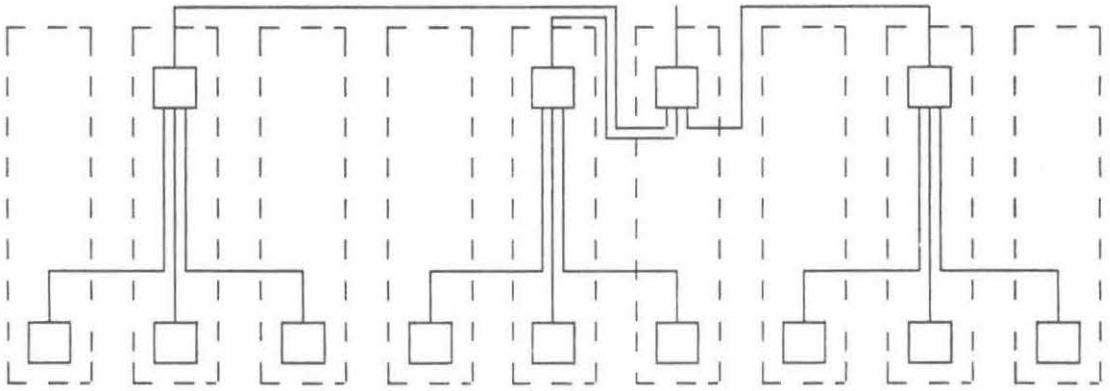
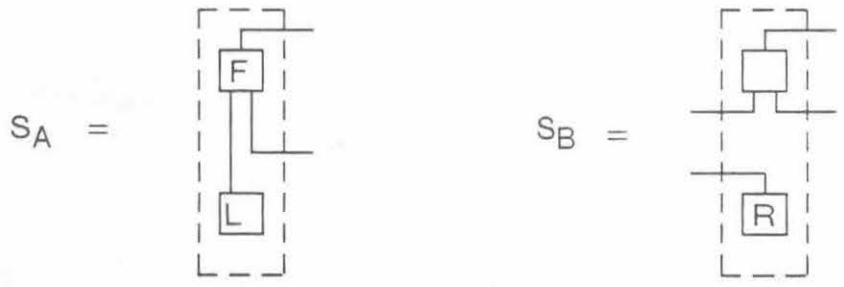


Fig. 5. Ternary tree layout as array of slices with two nodes each

The above layout schemes have two noteworthy properties:

a) There are two distinct types of slices, characterized by the I/O connections of their nodes. These can be represented for the binary tree as



Although S_A can be obtained from S_B, we choose to distinguish them in the following. These two slice types alternate in the array.

b) The number of nodes in a slice is not restricted to 2 as described in Lemmas 1 and 2, but can be any power of 2.

Some useful implications of these properties are stated in the following lemma.

LEMMA 3 :

A tree network can be realized as an array of two distinct types of slices, denoted by S_A and S_B, where S_A consists of a left subtree and its parent node, say F, and S_B consists of the right subtree of F and an ancestor of F. The array realization is an alternating sequence of these two slice types. There are log n + 1 different realizations of the tree, characterized by the number of nodes in a slice, where n is the number of leaf nodes.

Proof:

The basic unit in a binary tree is a node along with its two children. Out of the three ways of assigning these three nodes to two adjacent slices, the ones that yield the minimum inter-slice communication assign the parent node and one of its subtrees to be in the same slice as its parent. We have arbitrarily chosen the left subtree to be in the same slice as its parent. The sequence S_A,S_B by this definition, expands the left subtree contained in S_A to the next higher level while the sequence S_B,S_A inserts the right subtree of the parent node contained in S_B. The result follows from an inductive argument on the sequence of slice types. With $n = 2^m$ leaf nodes, there are $2^{m+1}-1$ nodes in the tree. A slice may contain 2^i nodes, $1 \leq i \leq m+1$. Each of these produces a distinct realization and hence there are $m+1$, or equivalently, log n + 1 different realizations of the tree.

It should be pointed out that the layout slice arrangement is amenable to a gate array in which a "gate" may be a layout slice and the number of interconnecting channels may be bounded as above.

3.3 AN ALTERNATIVE REALIZATION OF A BINARY TREE

Another layout slice scheme for complete binary trees that uses a single slice type is briefly described below. Each slice contains a basic unit of the tree.

LEMMA 4 :

A complete binary tree with n leaf nodes can be realized as an array of $\lceil (2n-1)/3 \rceil$ identical layout slices where each slice contains a parent node and its two children, with a) at most five wires connected to a slice and b) at most log n - 2 wires passing around it without connection.

Proof:

a) As shown in Lemma 1, the total number of nodes in a tree with n leaf nodes is 2n-1. Assigning a parent node and its two children to a slice results in $\lceil 2n-1/3 \rceil$ slices and at most five wires connected to a slice.
b) The bound on the number of wires passing around a slice can be proved by induction on the number of leaf nodes. □

This scheme also has the leaf nodes at the edges of the layout and is illustrated in Fig. 6 for n = 4 and n = 8. It can be observed that the slices are fully used when m is odd and there are exactly two unused slots when m is even, where $n = 2^m$.

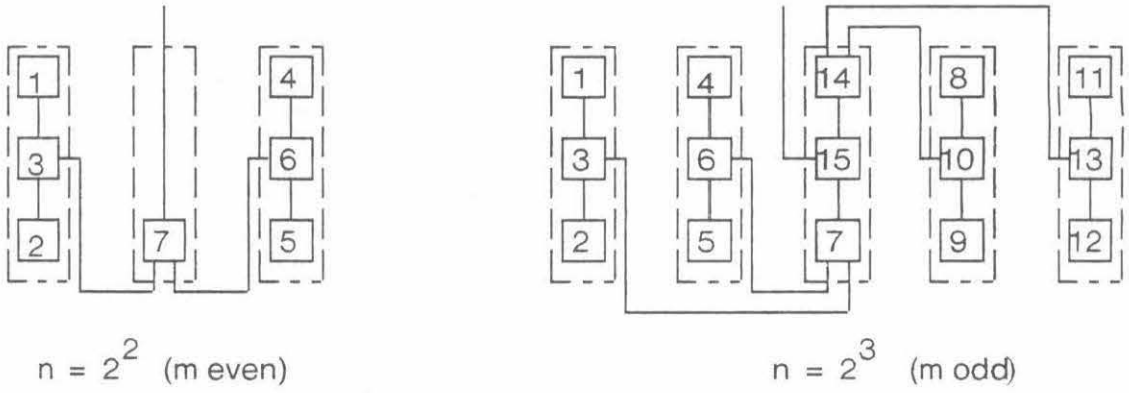


Fig. 6. An alternative array realization of the tree with one slice type.

4. MODIFIED TREE NETWORKS

Let us consider the applicability of the array-realizable layout schemes to modified tree networks. Such a network of considerable interest is a carry-save adder that adds a set of n-bit numbers using standard full/half adder cells with the carry propagation deferred to the very last stage. The number of levels in this tree is determined by the type of basic adder cells used in reducing the given h n-bit numbers to two rows of bits before performing a carry propagation. For instance, using full/half adder cells for each node of the network one would require approximately $\log_s h$ levels in the tree excluding carry propagation, where $s \simeq 1.5$ [10]. It suffices to say that the number of levels in the tree for a given type of adder cell can always be bounded.

An assignment of full/adder cells to add six 3-bit numbers is shown in Fig. 7. The

horizontal lines separate the successive levels in the tree. The numbers within circles are the unit numbers of the adder cells and the numbers within the adder cells (boxes) represent the outputs of other adder cells from previous levels. There are four levels in the tree and the carry propagation is done at the fourth level.

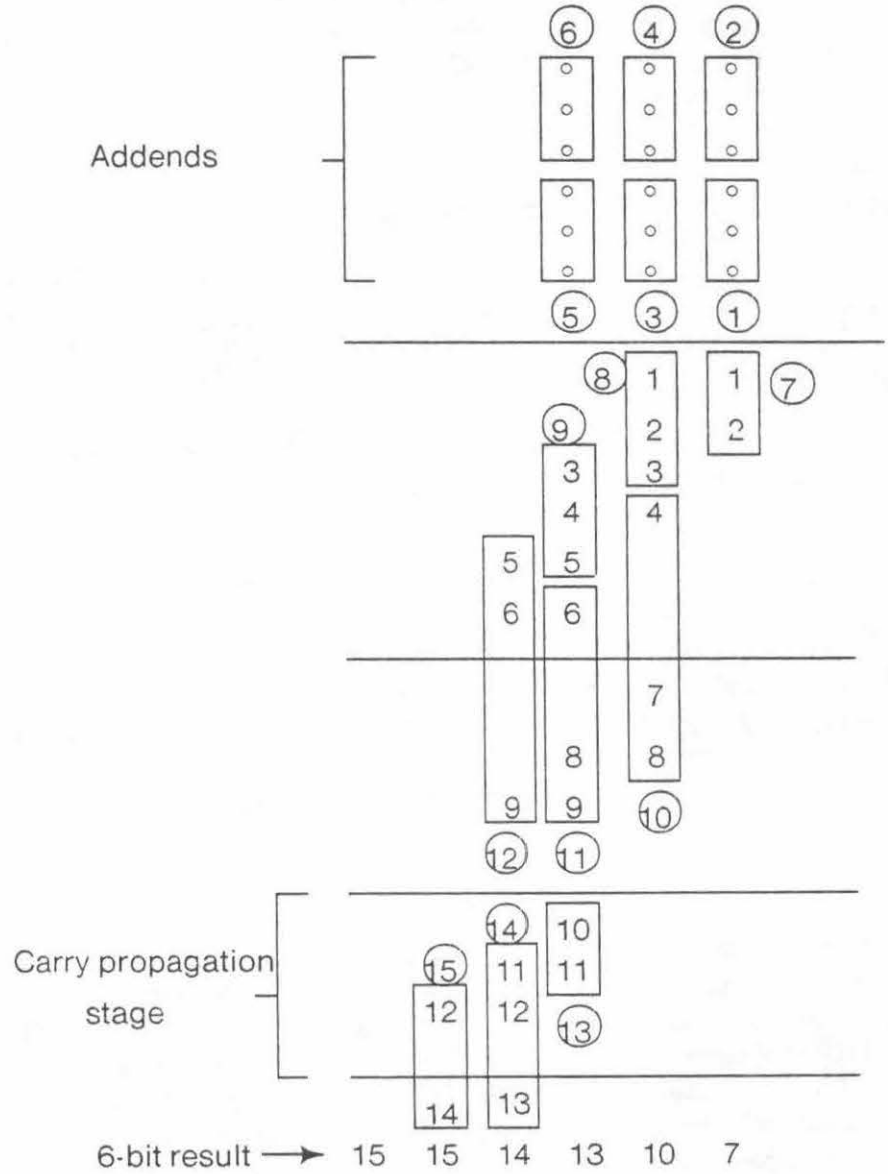


Fig. 7. Carry-save addition of six 3-bit numbers using full/half adders

The tree network for this adder scheme is shown in Fig. 8. Let us assign the nodes in the tree to layout slices as follows:

Define a slice for each leaf node. Assign a non-leaf node P to the same slice as its middle input if P is a full adder and to the same slice as its right input if P is a half adder. Note that there is at most one node from any given level in a slice. The placement of the carry propagation stage cells is explained below. The effect of this assignment strategy is that in any slice there is at most one node from any given level. A more rigorous method for obtaining the layout slices for general networks will be decribed in Section 5. We can now state the following for such an adder tree implementation using layout slices:

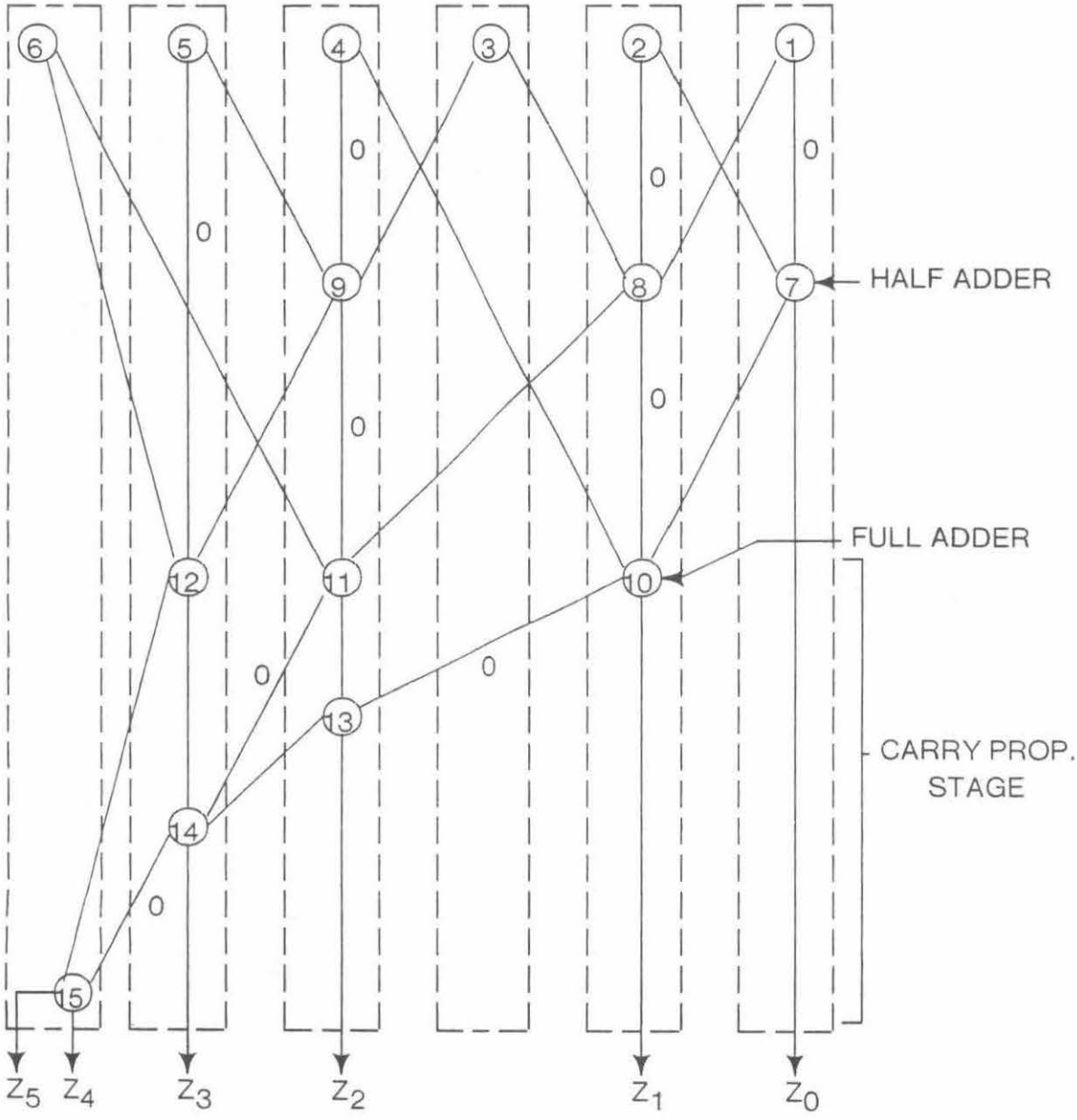


Fig. 8. Tree Network for Carry-save Addition using Full/Half Adders

LEMMA 5 :

An adder tree network that adds h n-bit words using carry-save addition with full/half adder cells can be realized in area $O(n\log_s h)$ as an array of $n + \lceil h/2 \rceil$ slices with each slice containing at most $\lceil \log_s h \rceil + 1$ nodes and at most $5 \lceil \log_s h \rceil + 4$ wires connected to it, where $s \simeq 1.5$.

Proof:

As discussed above, the upper bound on the number of levels in the tree using full adders is $\log_s h$. For the carry propagation, we need at most $(n-1) + \lceil h/2 \rceil$ more levels where the term $\lceil h/2 \rceil$ accounts for the fact that the sum of a pair of n-bit numbers will need an additional bit for overflow and so the sum of the h n-bit numbers may have up to $n + \lceil h/2 \rceil$ bits. For simplicity, we allow these additional $\lceil h/2 \rceil$ bits of the sum to have separate slices for the carry propagation stage. Thus we need at most $\lceil \log_s h \rceil + 1$ nodes in each slice including the carry propagation stage. Each of the nodes in a slice has two or three inputs and two outputs. However, from the assignment of the nodes to slices, we are guaranteed that at least one node in every slice has at least one input from within the same slice. Thus there are at most $5(\lceil \log_s h \rceil + 1) - 1$, i.e. $5\lceil \log_s h \rceil + 4$ wires connected to any slice. The area of the layout is therefore $O(n\log_s h)$. □

The dotted lines in Fig. 8 indicate the slices used in the realization of the tree.

5. A FLOOR PLAN GENERATION APPROACH

In this section an approach to develop the floor plan of an arbitrary computation network of interconnected processors is outlined. The task of generating a floor plan is to lay out the individual nodes and the interconnections among them in a rectangular area satisfying the specified design constraints like line length, width and number of crossovers. The availability of alternative layout schemes is bound to suggest alternative logic realizations at any level of a hierarchical design. The layout schemes described above are not limited to any particular logic level, e.g. transistors, gates etc. Thus various multiprocessor architectures as well as different multiplication schemes may be tried with the same abstraction. The elementary nodes themselves may in turn be laid out in detail at a lower level to the desired degree of optimization.

5.1 DESCRIPTION OF THE METHOD

The approach consists of the following steps:

a) Obtain a weighted network of processing elements with appropriate weights assigned to the edges. These weights may represent the required degree of proximity of the two nodes connected by that edge.

b) Extract a minimum spanning tree for the network i.e. a tree that spans all nodes of the network and has the minimum total weight for any tree. The designer can force critical interconnections into this tree by assigning appropriate weights to these edges in the original network. The resulting minimum spanning tree will contain the part of the original network that is critical in terms of topology constraints.

c) Lay out the minimum spanning tree obtained using the regular tree layout schemes described above.

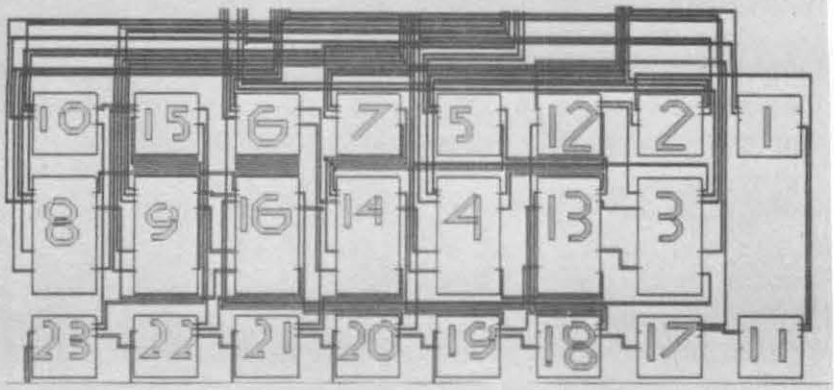
d) Realize the original network by restoring the remaining edges.

The implementation of the above method is illustrated in Fig. 8. The leaf nodes are at one end and each leaf node defines a slice. The middle input to a full adder and the right input to a half adder have been assigned weight 0. The effect of this criterion on the assignment of nodes to slices is evident although the actual strategy for the assignment of weights to the edges is not relevant to the proposed floor plan method.

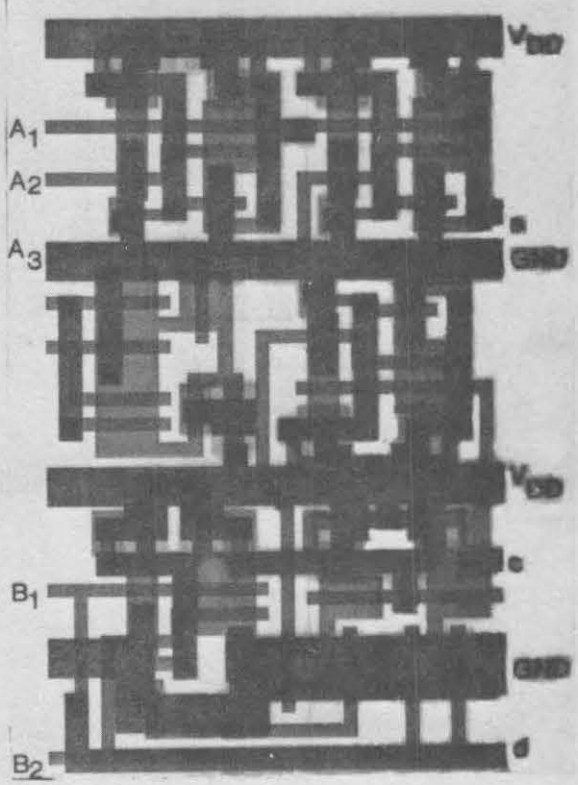
The layout of a bigger version of a carry-save adder macro generated using the above method is shown in the accompanying plate which shows the overall floor plan of the macro and the internal layout of an adder cell. There are five 6-bit operands and the sum is truncated to 6 bits. The leaf nodes are in the top row and the output nodes in the bottom row. Note that the middle row consists of one type of adder cell while the top and bottom rows contain a smaller adder cell although the above method was developed primarily for identical cells. The inputs and outputs of a cell are on opposite faces of a cell. A cell layout or its mirror image may be used in a slice.

5.2 INCOMPLETE TREES

Although the layout schemes described above have assumed complete or nearly complete trees, incomplete or unbalanced trees may still bring useful layout structure to the original network. Fig. 9 illustrates this with incomplete trees laid



OVERALL FLOOR PLAN



LAYOUT OF ADDER CELL
LAYOUT OF ADDER MACRO

out as arrays. The interconnections among nodes are weighted, with 0 indicating a closer required proximity than 1. External inputs are shown unweighted and undirected. Note that in (b), node 3 is not a conventional leaf node.

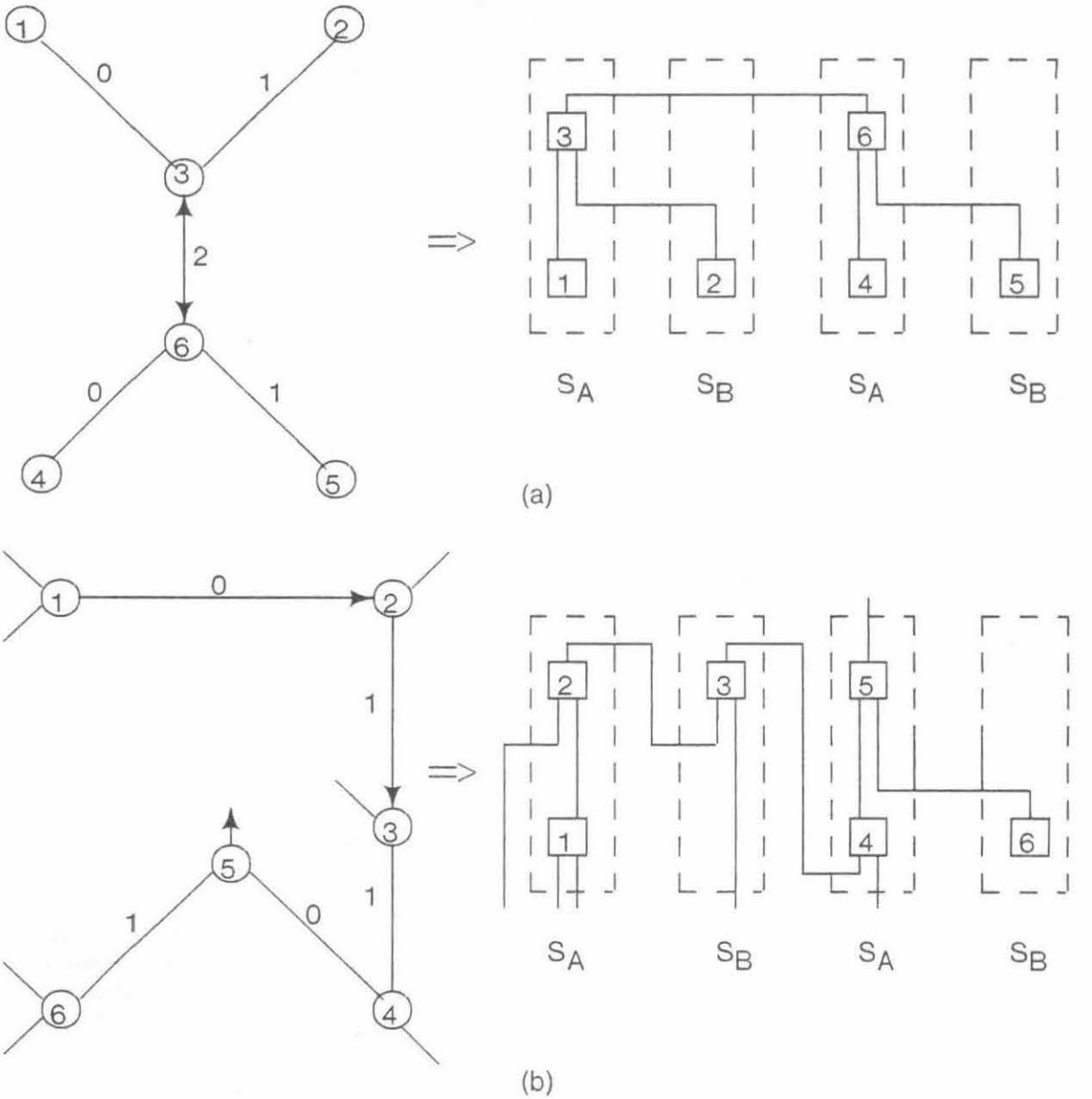


Fig. 9. Examples of layouts for incomplete trees

5.3 PROPERTIES OF THE PROPOSED FLOOR PLAN APPROACH

1) It is more general than the divide-and-conquer method underlying Leiserson's scheme in that any network, not necessarily planar, can be handled as a minimum spanning tree problem.

2) It yields a regular, array-realizable layout with known bounds for the number of processors in a slice and the number of wires beween slices which provide for uniform spacing for routing purposes.

3) There are efficient, polynomial-time algorithms to extract minimum spanning trees [12]. More importantly, alternative minimum spanning trees can be easily obtained using cyclic interchange methods [11] making it possible to systematically generate alternative floor plans. A useful implication of this is that the design hierarchy may be reevaluated in the light of the floor plans generated, resulting in a modified computation network. Thus the processes of logic design and physical design can be integrated to simplify the time-consuming and often error-prone task of a detailed layout for VLSI chips.

4) An apparent disadvantage of this approach is that for an unbalanced tree the slices are not utilized efficiently. However, as mentioned in the previous section, regularity in layout can still be imposed on unbalanced trees. Also, at the mask generation stage, the unused portion of a slice may be eliminated so that the unused part of a slice does not consume any power.

6. CONCLUSIONS

Some layout schemes for tree networks and a possible solution to the floor plan generation problem using such schemes were proposed above. Possible directions for pursuing this approach are mentioned in this section:

1) Efficient layout slices for other commonly used structures e.g. cube, hexagonal array etc. may be developed such that different types of layout slices are compatible in terms of number of wires and/or number of processing elements *in* a slice. The goal here is similar to that of the Bristle Blocks project [6]. Thus a given network may be decomposed as a set of these structures which can then be laid out individually using efficient layout slices for each of these structures and interconnected. This compares favorably with the arbitrary division approach used in the divide-and-conquer method. For instance, the slice concept applied to a cube network is demonstrated in Fig. 10.

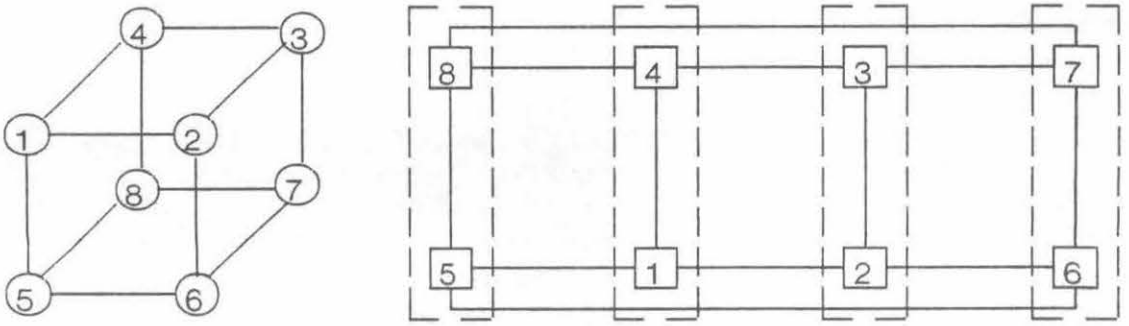


Fig. 10. Cube Interconnection realized as a regular array of slices

Note that each slice for the cube also contains two nodes similar to a tree slice and there are two slice types.

2) There are no efficient methods to extract minimum spanning trees with special constraints such as a bound on the degree of a node etc. This problem can be viewed differently : Are there useful computation networks that are modifications of a balanced tree and are realizable as arrays of slices within the bounds discussed above? The adder trees of Section 4 are examples of such networks.

3) A natural extension of the layout schemes described above would be to tree structures where the nodes are not identical in their sizes or connectivities. The selection of a basic slice or slice types would be critical to an array realization.

4) Since the processing elements within a slice may be connected internally, specific optimizations both in layout and in logic design are possible. For example, for a layout slice where a leaf node communicates its carry signal to the node within the slice, e.g. slice S_A above, it is possible to use a complemented carry signal thereby eliminating two inverters and saving their area and power. However, such optimization is meaningful only in situations where it does not cause a proliferation of layout slice types. This may be treated as a problem of characterizing the interconnection pattern among the slices.

ACKNOWLEDGEMENT

The author is thankful to the Xerox Corporation for their support during this research.

REFERENCES

1) Brent, R.P. and Kung, H.T., "A Regular Layout for Parallel Adders", Tech. Report, CMU-CS-79-131, Dept. of Computer Science, Carnegie Mellon University, June 1979.
2) Leiserson, C.E., "Area-Efficient Layouts for VLSI", Tech. Report, Dept. of Computer Science, Carnegie-Mellon University, August 1979.
3) Bentley, J.E. "Multidimensional Divide-and-Conquer", Comm. of the ACM, Vol. 23, No.4, April 1980, pp 214-229.
4) Rowson, J.A. "understanding Hierarchical Design", Ph.D thesis, Dept. of Computer Science, Caltech, April 1980.
5) Browning, S.A., "A Tree Machine", Lambda, Second Quarter, 1980, pp 32-36.
6) Johannsen, D., "Bristle Blocks: A Silicon Compiler", Caltech Conf. on VLSI, Jan 1979, pp 303-310.
7) Marshall, M., "VLSI pushes super-CAD techniques", Electronics, July 31, 1980, pp 73-80.
8) Mead, C.A. and Conway, L.A., "Introduction to VLSI Systems", Addison-Wesley, Mass. 1980.
9) Rem, M., "Mathematical Aspects of VLSI Design", Caltech Conf. on VLSI, Jan 1979, pp 55-63.
10) Stenzel, W.J. et al, "A Compact High-Speed Multiplication Scheme", IEEE TC, Vol. C-26, No. 10, Oct 1977, pp 948-957.
11) Deo, N., "Graph Theory with Applications to Engineering and Computer Science", Prentice-Hall, Englewood Cliffs, N.J., 1974.
12) Cheriton, D. and Tarjan, R.E., "Finding Minimum Spanning Trees", SIAM J. of Computing, Vol. 5, No. 4, Dec 1976, pp 724-742.