The Quad-CIF Tree:
A Data Structure for Hierarchical On-Line Algorithms

Gershon Kedem
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR91
September, 1981

Abstract

In this paper we describe the quad-CIF tree data structure and its application to hierarchical on-line computer-aided design algorithms. The main idea is to overlay a tree of coordinates on top of the hierarchical representation of an integrated circuit. The coordinate tree enables one to find quickly the set of all objects that intersect a given window. We outline how one can use the data structure in order to implement hierarchical, on-line design rule checking and node extraction. We also describe some applications to plotting.

1.0 Introduction

Many algorithms in computer aided design of VLSI, such as design rule checking, node extraction, and interactive display, solve repeatedly the following problem:

Problem 1:

Find the collection of all objects that intersect a given rectangle.

We assume in this paper that the IC design is made of rectangles orthogonal to the axis (Manhattan Boxes). However everything we say can be generalized to polygons if one encloses each polygon in a minimal bounding box.

A conventional method for finding intersections is to sort all the elements that make the IC and then use a scan line method. That is, one keeps all rectangles that currently intersect the scan line in a list. Then the intersection problem is reduced to a one-dimensional problem that can be solved very quickly [see Bently et al., 1980].

The major disadvantage of the above method is that one needs to expand the cell description of the circuit so that all items could be sorted. Therefore, one cannot take advantage of the repetitive nature of VLSI circuits. Even if one uses an hierarchical algorithm the need to keep resorting elements can be quite expansive [see Whitney, 1981].

We present here an alternative way of organizing the computation. We keep the hierarchical cell structure of the IC and take advantage of it, but at the same time impose a tree of coordinates on the design. This tree of coordinates facilitates the solution of problem 1. The main idea is to impose a quad-tree coordinate system on top of the hierarchical CIF cell structure and take advantage of locality information presented by both structures.

In Section 3 we sketch how on-line hierarchical design rule checking and node extraction can be implemented. By taking advantage of the repetitive nature of the design a substantial saving in computation can be realized.

2.0 The Quad and the Quad-CIF Trees

2.1 The Quad Tree

The quad tree was originally introduced as a way to represent raster image data. See [Samet and Rosenfeld '79] and the references there. However, the quad tree can be generalized to organize any two dimensional collection of objects.

One starts with a big rectangle that contains all the objects. This rectangle is the root of the quad tree. That rectangle is divided to four equal subrectangles by dividing each of its sides into two (See figure 1). The four rectangles are the sons of the original rectangle. In turn, each of these rectangles is divided into four, and so on. Now each node in the tree has a list of objects that resides in that node.

a

b

Figure 1: A quad tree

- a) Two dimensional representation of a quad tree
- b) The corresponding tree structure

An object is put into a node if and only if it is inside the rectangle that corresponds to that node but is not inside any of its sons. Since all the objects are enclosed in the first rectangle, each object resides in a node. This node is uniquely determined by the above rule.

Since most rectangles in IC design tend to be small and uniformly distributed [Bently et al., 1980], most of the rectangles will be at the leaves of the quad tree. Therefore, the expected depth of the tree is $O(\log N)$. One more problem is left: Many small objects will be at the upper nodes of the quad tree, namely, those that intersect the node's (rectangle) division lines. Therefore, if a window intersects a division line of an upper node, all the objects that also intersect a division line of that node have to be checked. Even objects that are far away from the window have to be checked (see Figure 2).

Figure 2

In order to reduce the search effort, the items are not put in a list at the quad-tree node. Instead, they are put into a binary tree that divides either the X dividing line or the Y dividing line. The binary tree represents a successive bisection of the line segment. An object is put into a node if a dividing point is inside that node. For example, Figure 3 shows how a line segment is being divided (a) and the corresponding binary tree (b).

Figure 3: a) Division of a line segment

b) The corresponding binary tree

One final detail: The trees (the quad-tree and the binary tree) are not subdivided below a fixed size. This size was chosen to be 8λ long, since 8λ is the expected size of a rectangle (see [Bentley et al., 1980]).

2.2 The Quad-CIF tree

The quad tree structure meshes well with hierarchical representation of ICs. In the quad-CIF tree arrangement, each symbol is organized as a quad tree. The root of that quad tree is the bounding box. The items that make the symbol are stored in the quad tree. If the item is a rectangle, then it is stored as-is in the appropriate node. However, if the item is an instance of another cell, the bounding box of that cell (appropriately transformed) is stored together with a pointer to the quad tree of that cell and the transformation.

An Algorithm

The following algorithm solves problem 1. That is, given a quad-CIF tree of a cell and a window, the algorithm produces all the items (rectangles) that intersect the window. The algorithm is described by a C procedure.

This algorithm does not handle arrays of cells, since arrays are not part of CIF 2.0. However, it can easily be modified to do so. The use of arrays of cells could save a large amount of time and space and should be added into any system.

```

allIntersections(Tree,window,trans)

/* A procedure for finding all the items that intersect a given window */
QuadCIF *Tree;          /* A pointer to a quad-CIF representation of a cell */
Rectangle window ;      /* The current window */
Transformation trans ;   /* The current transformation */

{
    intersectXline (Tree -> xline, window, trans);
    intersectYline (Tree -> yline, window, trans);

/* traverse the four sons */

    if (Tree -> ULson NE NULL AND isIntersecting (Tree -> ULson.frame, window))
        allIntersections (Tree -> ULson, window, trans);

    if (Tree -> LLson NE NULL AND isIntersecting (Tree -> LLson.frame, window))
        allIntersections (Tree -> LLson, window, trans);

    if (Tree -> URson NE NULL AND isIntersecting (Tree -> URson.frame, window))
        allIntersections (Tree -> URson, window, trans);

    if (Tree -> LRson NE NULL AND isIntersecting (Tree -> LRson.frame, window))
        allIntersections (Tree -> LRson, window, trans);
}

IntersectXline(xnode,window,trans)
Btree *xnode; /* A pointer to binary tree with items */
Rectangle window ; /* The current window */
Transformation trans;
{
    Item *p ;
    Rectangle newWindow ;
    Transformation newTrans ;

    p=xnode->ItemList ;
    while(p NE NULL) /* As long as there are items in the Node */
    {
        if (isIntersecting (p -> element.Box, window)) {
            /* if the item intersects the window */
            if (p -> type EQ RECTANGLE)
                outputIt (trans, p -> element);
            /* if it is a rectangle output it */

            else {
                /* it must be a cell instance */
                inverseTransform (p -> element.transform, window, newWindow);
                /* multiply the window by the inverse of the cell transform */

                transMultiply (trans, p -> element.transform, newTrans);
                /* compute the new transform */

                allIntersections (p -> element.treePointer, newWindow, newTrans);
                /* call allIntesections with the new window and transform */
            }
            p = p -> next;
        }
    } /* end of while */
    xmiddle = (xnode -> xmin + xnode -> xmax) / 2;
    if (xmiddle GT window.xmin)

```

```
        IntersectXline (xnode -> left, window, trans);  
    if (xmiddle LT window.xmax)  
        IntersectXline (xnode -> left, window, trans);  
}
```

IntersectYline routine is similar to IntersectXline and is not included.

3. Applications

3.1 Interactive display

One of the most useful tools for IC design is an interactive graphic system. When using such a system, a great deal of time and processing is devoted to inspection of the design. The designer tends to zoom in and out and move the viewing window around quite often. The quad tree data structure reduces the amount of computation needed for these operations.

3.2 Plotting

A) Raster plots.

One of the most useful hard copy output devices is an electrostatic printer. In order to use such a printer, one has to rasterize the image. Instantiating the whole CIF file and sorting it requires a large amount of 'real time' computation. Most of the time is spent doing disk I/O. However, there is no need to instantiate the whole file at once. One can instantiate small sections of the strip at a time, sort the rectangles in memory and pipeline the result to a raster conversion program.

B) Hidden line elimination.

Multi-color pen plotters are routinely used as an output device of IC designs. Good plotting programs eliminate the annoying extra lines of overlapping rectangles. For example the two metal boxes:

should be drawn as:

In order to eliminate lines from the interior of polygons, the collection of rectangles has to be divided into a collection of equivalent classes. Each equivalent class is made of intersecting rectangles that are on the same layer. Once the rectangles are divided into the appropriate groups, hidden line elimination is quite straight forward. The quad-CIF tree enables one to take advantage of hierarchy in order to reduce the amount of work needed in forming these equivalent classes. A more detailed discussion how to use hierarchy to reduce computation in the equivalent class computation is given in section 3.4. In that section we talk about node-extraction where one solves a similar problem.

3.3 Hierarchical, on-line design rule checking

When using an interactive graphic design system for IC layout, a great deal of time and effort can be saved by an on-line, design-rule checking. It is very wasteful to check for design-rule violations 'post-mortem,' that is, after the whole chip is designed. Errors in one of the small basic cells sometimes require the redesign and realignment of major parts of the total design. Such redesign will be eliminated if design errors could be caught when cells are defined and not after the design is complete. Therefore, design rule checking should be done as part of the cell definition process. The check can be done as a background job. The designer meanwhile can continue to design other parts of the circuits.

Much time could be saved by hierarchical check. Each cell is checked only once. After that, the cell is marked checked and only interactions of the cell instances with

other objects have to be checked. Each cell interaction with other objects should be recorded and should not be repeated. Usually cells are complete functional blocks. They tend not to intersect with other cells or rectangles but rather to abut. If cells do overlap possible design rule violations have to be checked only in a window that is 3λ larger than the overlap (see figure [5]).

The quad-CIF tree structure eliminates the need for sorting and therefore simplifies the implementation of hierarchical design rule checking. For more details on hierarchical design rule checking, see [Whitney '81].

Figure 5

Library cells and cells that are generated by software like the PLA generator can be marked as being checked ahead of time. Only the interaction of these cells with others has to be checked. By introducing design methodology and structure, one can speed up design rule checking. For example the designer should be able to specify certain cells or parts of cells as being protected. After checking the cell by itself the design rule checker could report any violations of the protected area without ever looking at the content of that area.

3.4 Hierarchical node extraction

Node and parameter extraction can be done hierarchically with great savings in processing time. Again, the ability to find quickly all intersections of a given rectangle or symbol with other rectangles or symbols is what makes the algorithm practical. In order to facilitate hierarchical extraction, some restrictions are necessary.

In reality, these restrictions do not limit the designer; on the contrary they encourage better design practice. The restriction we impose is that cells are made of complete functional blocks. One is not allowed to create part of a transistor in a cell and get the full one by interaction with other cells. The other restriction is that interaction of two cells does not create new transistors. The only case we know where the above assumptions are violated is in PLA programming. The programming cell interacts with the PLA cell to form a new transistor. This case and all others that we can think of can be corrected if one defines different cells for each possible combination. In practice, the extra work needed for defining the extra cells is very small.

Node extraction is made of two tasks. The first is identifying transistors and the second is combining rectangles into electrical nodes. Identifying transistors is straightforward although messy at times. The task of combining rectangles into nodes is also simple. A rectangle is assigned to a node if it intersects another rectangle on the same layer which is assigned to that node or a contact layer on that node. Whenever a rectangle intersects two rectangles that are assigned to two different nodes, these nodes are combined.

Internally, all rectangles that make a node are linked as a linear list. In addition they all point to a special header node. The header node points to the first and the last element in the linear list. All headers of electrical nodes of a cell are also connected in a linear list (see Figure [6]).

Figure 6

Very often two different nodes are combined into a single node. At the rectangle level this is easy. The only problem is how to combine nodes of instances of cells. Clearly it is undesirable to instantiate all rectangles of a cell every time an instance of that cell is encountered.

To avoid full instantiation and to get a compact representation of the node network of a cell, we use the following scheme: Each cell instance is uniquely identified by its cell number and its transformation. Every time a node in a cell instance interacts with another object, be it another cell instance or a rectangle, a record with the node number is created. That record is put into the node list as though it were a rectangle. In addition, all the nodes of a cell instance that interact with outside objects are linked together in a linear list. This way each cell instance has a list of all its nodes that combine with other nodes. All the nodes that are not in that list are internal nodes and they are only represented once in the original cell definition.

Once the complete network is generated internally, a recursive algorithm is used to instantiate the complete node network. At each cell instance all the nodes that are part of that cell but are not part of a higher level cell are instantiated.

By now the idea of how the node extraction algorithm works should be clear. The node extraction is done bottom-up. For each symbol definition the list of all objects is scanned. Rectangles are grouped into transistors and nodes. Whenever a cell instance is encountered, it is checked against all objects that intersect it. When a node in a cell instance needs to be combined with another object (a rectangle or a node of another cell), a record that represents that node is created. That record is then treated like another rectangle in the corresponding node. In addition, all the nodes in a given cell instance that interact with other nodes in higher level cells are linked in a linear list. These nodes will not be instantiated as part of a particular cell instance, since they are part of a node in a higher level cell.

4. Conclusions

In this paper we described the quad-CIF tree representation of an integrated circuit. We briefly outlined how this representation can simplify and speed-up algorithms used in computer-aided design of integrated circuits. By keeping the cell structure of the CIF file and overlaying a two-dimensional tree of coordinates on top of the CIF structure, one is able to find, quickly, all objects that intersect a given window. This capability is the essential part of many important CAD algorithms. We outlined how the quad-CIF data structure could be used for hierarchical on-line design check and node extraction. We expect these algorithms to be much faster than nonhierarchical ones, especially in a structured design environment. We are now in the process of incorporating this representation and the above algorithms into an integrated design system.

REFERENCES

- Bently, J.L., "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. of ACM*, September 1975, Vol. 18, No. 9, 509-517.
- Bently, J.L., D. Haken, R.W. Hon, "Statistics on VLSI Designs," Department of Computer Science, CMU, CS-80-111, April 1980.
- Hon, R.W. and C.H. Sequin, "A Guide to LSI Implementation," Xerox Palo Alto Research Center, SSL-79-7, January 1980.
- Lauther, U., "Four-Dimensional Binary Search Trees as a Means to Speed Up Associative Searches in Design Rule Verification of Integrated Circuits," *Design Automation and Fault-Tolerant Computing*, 1978.
- Mead, C. and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley Publishing Company, 1980.
- O'Rourke, J., "Dynamically Quantized Spaces Applied to Motion Analysis," The Johns Hopkins University, Electrical Engineering Department, JHU-EE-81-1, January, 1981.
- Samet H. and A. Rosenfeld, "Quad Tree Structures for Region Processing," *Proceedings: Image Understanding Workshop*, November, 1979, SAI, Inc. Report No. SAI-80-974-WA.
- Whitney, T., "A Hierarchical Design-Rule Checking Algorithm," *Lambda*, 40-43, first quarter, 1981.