HEX : A Hierarchical Circuit Extractor

By

Yen-Jen Oyang

Technical Report 5139

Department of Computer Science

California Institute of Technology

Pasadena, California

June 1984

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science

Supported by
Defense Advance Research Projects Agency
ARPA Order Number 3771
Monitored by the
Office of Naval Research
Contract Number N00014-79-C-0597

ACKNOWLEDGEMENT

I am gratefully indebted to my thesis advisor, Professor Randal Bryant, for his support and guidance. This thesis would not be as readable as it is without the hard work of Andy Matsuda who turned my draft into grammatically correct, smoothly flowing English. Tak-Kwong Ng, John Ngai, and Wen-King Su also provided helpful suggestions about grammar and style.

ABSTRACT

This report describes the algorithm, implementation, and performance of a hierarchical circuit extractor, HEX, for Metal-Oxide Semiconductor(MOS) layout designs at Caltech. The input to HEX is a layout design in Caltech Intermediate Form(CIF), a hierarchical layout description language, and the output is a hierarchical netlist describing the circuit. HEX avoids redundant work by finding out the repetitive cells in the input CIF file. To handle overlapping instances, HEX modifies the hierarchy in the CIF file to generate a new one without overlapping instances. HEX then traverses the resulting hierarchical structure, calls a flat extractor to extract leaf cells and composes cells bottom up to get the circuit information of the whole chip.

TABLE OF CONTENT

CHAPTER 1

INTRODUCTION

1.1. BACKGROUND

1.1.1. Why We Need a Extractor

Aside from fabrication problems, there are many other factors that contribute to faulty VLSI chips. Planners may make mistakes in algorithm, logic, circuit, or layout design. To verify their plan at each level, designers use specific verification tools. Functional or register transfer simulators are used for algorithm designs, while logic and switch level simulators check the logic. A number of items, i.e., timing simulators, design rule checkers, etc, verify circuit and layout designs. However, to demonstrate that a circuit is valid, the plan must be presented in some specific form to the verification tools. To facilitate this need, extractors withdraw the desired circuit information from the layout description files. In general, the extracted information includes (1) all the transistors along with their connectivity (2) electrical characteristics, such as resistance and capacitance, of the elements on a chip.

1.1.2. Motivation

The artwork of integrated circuit design is usually available in the form of a hierarchical specification, in which each symbol is made up of primitive geometry (e.g. rectangles and polygons) and references to other symbols. The typical approach to circuit extraction involves fully instantiating the hierarchical structure in layout description. The extractor then traverses this flat representation and carries out its function. There are several objections to the full instantiation approach of circuit analysis. The most obvious of these is time efficiency. A flat extractor usually takes several hours of CPU time just to extract a chip containing a few thousand devices. In a few years it will be feasible to fabricate VLSI chips containing more than one million devices [3,4]. Even if we assume that the execution time of extractors grows linearly in time with respect to the number of devices in a chip, we can expect flat extractors to take several days for VLSI circuit.

Since integrated circuit designs often exhibit considerable degrees of regularity and hierarchy, determining which parts are identical can eliminate redundant work. Recognizing identical groups of geometry in a fully instantiated description is difficult at best. However, a hierarchical description of the structure makes this task possible. Instead of trying to determine if a given group of geometry matches some other group, a hierarchical extractor need only decide if the groups are instances of the same symbol.

There is another advantage of using hierarchical approach in implementing extractors. Since this may be used in most VLSI CAD tools in the future, it is desirable that extractors retain the hierarchical structure embedded in layout description of circuit for other verification tools.

1.2. OVERVIEW

In this report, HEX, a hierarchical circuit extractor is described. Hex takes in layout descriptions specified in Caltech Intermediate Form (CIF) [4] and exploits the regularity of chips based on the hierarchical structure in CIF file. The output of HEX is a hierarchical description of the electrical circuit denoted by the layout. The extracted circuit information includes (1) all transistors together with their connectivity (2) capacitance of the wires and transistors on the chip.

This report consists of five chapters. Chapter 2 illustrates what a extractor should do and how a flat extractor carries out its function. The algorithm proposed by C.M. Baker is described in this chapter.

Chapter 3 discusses the problems that must be resolved by hierarchical extractors when they exploit the hierarchy and regularity of cells. The problems and proposed solutions are discussed in general.

Chapter 4 describes the detail algorithms and data structures used in implementing HEX. The output format generated by HEX is also described in this chapter.

Chapter 5 discusses the complexity analysis of hierarchical algorithm. It also contains some execution time data of HEX and the flat extractor. These data are obtained by running the two programs on VAX-780 system.

Some results obtained by running HEX on various IC design are presented in appendix.

CHAPTER 2

BASIC ALGORITHM FOR FLAT EXTRACTORS

As the name implies, extractors extract information about all the transistors and their connectivity from the layout description. All it needs to do is simply

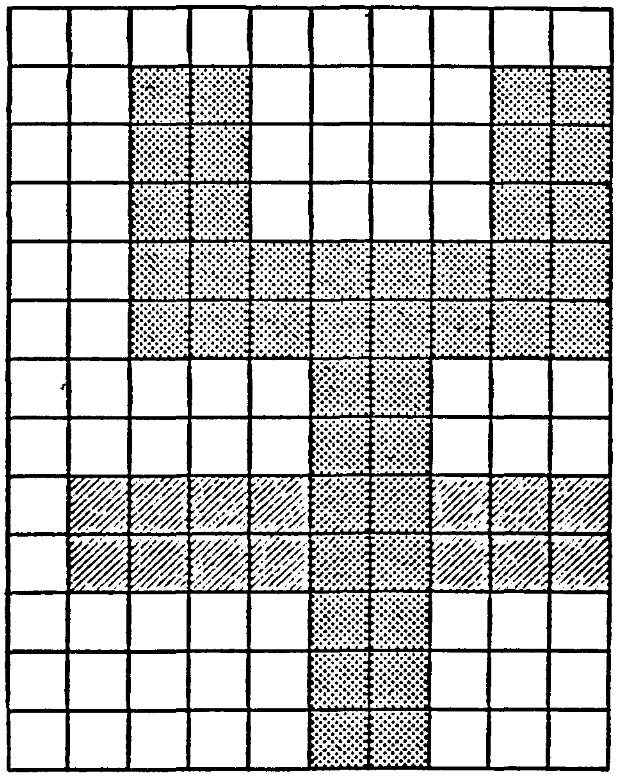
(1) Follow the connectivity of the wires.

(2) Find transistors.

In 1980, C.M. Baker proposed a algorithm for flat extractor. Baker's algorithm consists of two stages. In the first stage, the extractor rasterizes the geometry to obtain a bit map. Every pixel in the bit map contains one bit for each layer which indicates the existence of the layer. Figure 2.1 shows a typical example of rasterization procedure. In the second stage, the extractor scans the bit map to carry out the two tasks mentioned above.

2.1. PONDS AND ISLANDS PROBLEM The basic algorithm for following connected regions in a bit map comes from the classic ponds and islands problem which is defined as follows. Given a two dimensional array of zeros and ones, where the zeros represent water and ones represent land, write a program that counts the number of land masses and prints out the area for each one. Assume that land must connect horizontally and vertically but not diagonally. Baker's algorithm scans the bit map from left to right, and top to bottom, with one scan line buffered in memory (see figure 2.2).[1] At each bit, the access requires three bits of information: the current bit, the bit to the

Really, portions of two scan lines are buffered. The total space used in equivalent to that of a single scan line.



Bits Maps for Diffusion and Polysilicon Layers

```
0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 0 0 0 1 1
0 0 1 1 0 0 0 0 1 1
0 0 1 1 0 0 0 0 1 1
0 0 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1 1
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 1 1 0 0 0
```

Diffusion Layer

```
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1
0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
```

Polysilicon Layer

Figure 2.1

left, and the bit above. Since there are three bits of information, there are eight cases to consider. Four of them can be handled at once. (1-4) If the current bit is water, there is nothing to do. (5) If the current bit is land and the other bits are water, then the upper left corner of a new piece of land has been found. This information is remembered in an array as big as the scan line, and the new piece of land is assigned a unique number. (6) If the current is land and the bit to the left is land and the bit above is water, then this is the top of a horizontal strip and the number is the same as the number to the left. (7) If the current is land and the bit to the left is water and the bit above is land, then this is the left edge of a vertical strip and the number is the same as the number above. (8) The interesting situation occur when all three bits are land. (8a) If the number above is the same as the numbers to the left, then that must be the number for the current bit. (8b) If the numbers are different, the lower right inside corner of some shape has

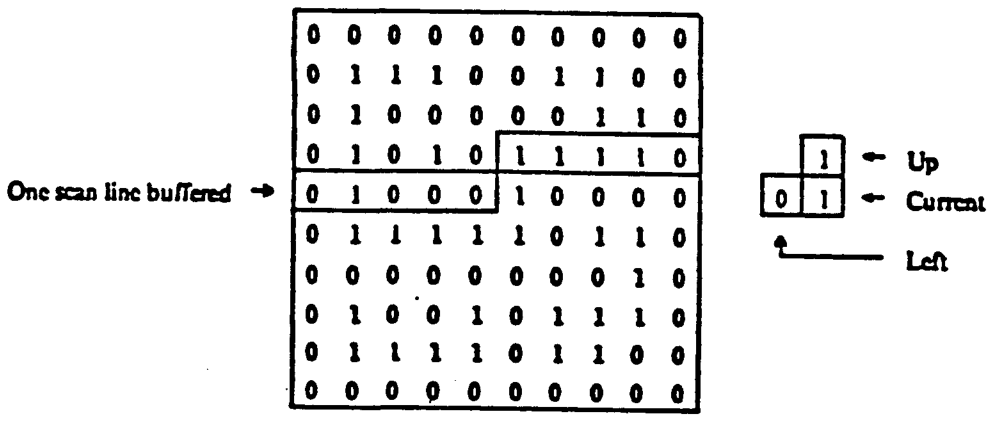


Figure 2.2

been found, and two pieces of land which previously seemed distinct are found to be part of the same mass (see figure 2.3). Some bookkeeping is needed to update the counts of one land mass number with those of the other land mass.

2.2. HOW TO FOLLOW THE CONNECTIVITY OF THE WIRES The same algorithm that work with land and water will also work with poly, diffusion, and *metal*. For the purpose of following wires and finding transistors, four derived layers will be used. Metal (M) and poly (P) correspond to the mask layers by the same name. In the extractor, diffusion (D) will be the diffusion mask minus the poly mask and transistor (T) will be the diffusion mask intersected with the poly mask.

The basic algorithm for following connectivity given a raster scan version of the masks can be thought of as a ponds and islands search on four

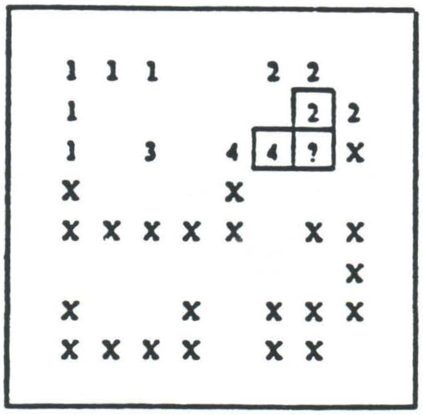


Figure 2.3

layers at once. The electrical properties of contact cuts can be added by checking to see if the current pixel contains P, M, and C. If it does, merge the number of the poly layer with the number of metal layer. Another kind of cut for D and M is handled similarly.

2.3. HOW TO FIND TRANSISTORS

By checking the current pixel for T, the extractor can find pieces of transistor. From the number of P, the extractor knows the number of its gate node. However, edges of the transistor lay contains useful information, namely the node number of its source/drain nodes. During the ponds and islands processing, the transistor finder looks for one of four cases: (1) current pixel is transistor and left is diffusion, (2) current pixel is diffusion and left is transistor, (3) current pixel is transistor and up is diffusion, (4) current pixel is diffusion and up is transistor. For each match, some pointers between the records for node and records for transistor are set to remember the relation found.

CHAPTER 3

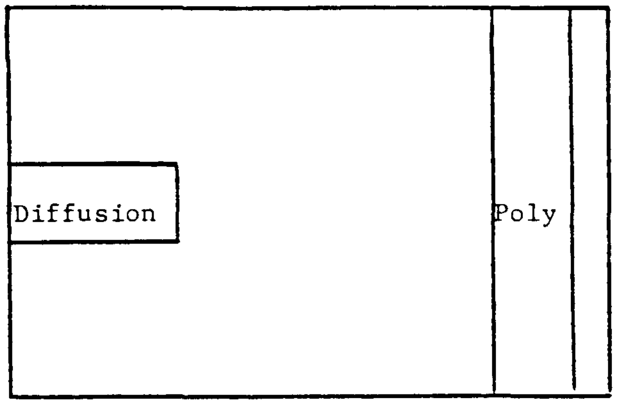
PROBLEMS IN HIERARCHICAL APPROACH

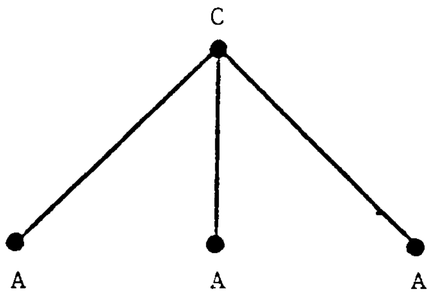
3.1. OVERVIEW OF HIERARCHICAL EXTRACTORS

The basic idea of a hierarchical extractor is to avoid redundant analysis work using the hierarchy in layout description. One straight forward approach is to implement a recursive depth-first program which extracts the layout description from the lowest level symbols. This is where the symbols contain only primitive geometry to build up the whole chip. It is followed by the symbol-calling hierarchy and extraction of each symbol only once. There are two problems needs to be considered. The first one is about the overlapping instances in layout description. The second one is how to build up the whole chip.

3.2. OVERLAPPING INSTANCES

Since CIF allows instances physically overlap on the chip surface, many strange cases may happen. Overlapping instances may create new transistors that were not in original symbols, or remove transistors that previously were there. A conductive path might get broken by an overlap, or it might get shorted to another path. Figure 3.1 shows a typical example. In figure 3.1, symbol C contains three instances of symbol A. In the overlapping area of the adjacent instances of symbol A, new transistors are generated. That makes it difficulty to obtain the circuit information of symbol C from the circuit information of symbol A.





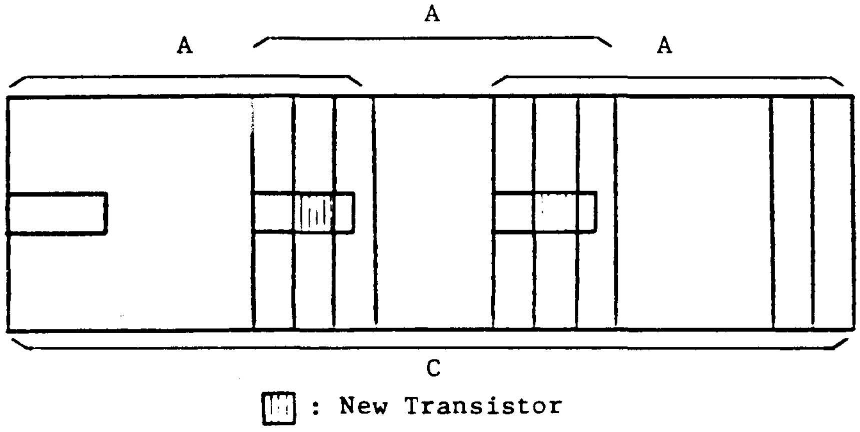


Figure 3.1

3.2.1. Disjoint Transformation

This problem was addressed in several articles [2,5,6,7]. In 1982, M. Newell and D. Fitzpatrick proposed a solution, called disjoint transformation [2,5]. Disjoint transformation uses a preprocessor to transform the hierarchical description specified in CIF into another hierarchical description with no overlapping cells[1]. The non-overlapping hierarchical description satisfies another property: only the cells at lowest level contain primitive geometry. The cells at higher level contain only references to other cells. Figure 3.2 illustrates the function of disjoint transformation. The hierarchical description of cell C in figure 3.2a is transformed into the non-overlapping hierarchical description shown in figure 3.2b.

3.2.2. Strategy of Disjoint Transformation

Of course, there are many approaches to carry out disjoint transformation. A satisfactory approach should be able to fulfill two requirements.

(1) Preserve the repetition of instances so that repeating the same analysis work can be avoided.

(2) Reduce the overhead due to hierarchical approach in case chips containing very few repetitions are extracted.

One straight forward strategy for disjoint transformation is to divide a cell into several non-overlapping subcells according to the boundary of the instances in this cell. However, this approach fails to fulfill (1) in many cases. For example, a transformation of the hierarchical description in figure 3.1 to the non-overlapping hierarchical description in 3.3a is rather poor since the hierarchical extractor can not take advantage of the similarity of A_1, A_3, and

[1]In non-overlapping hierarchical description we use cell instead of instance to avoid ambiguity.

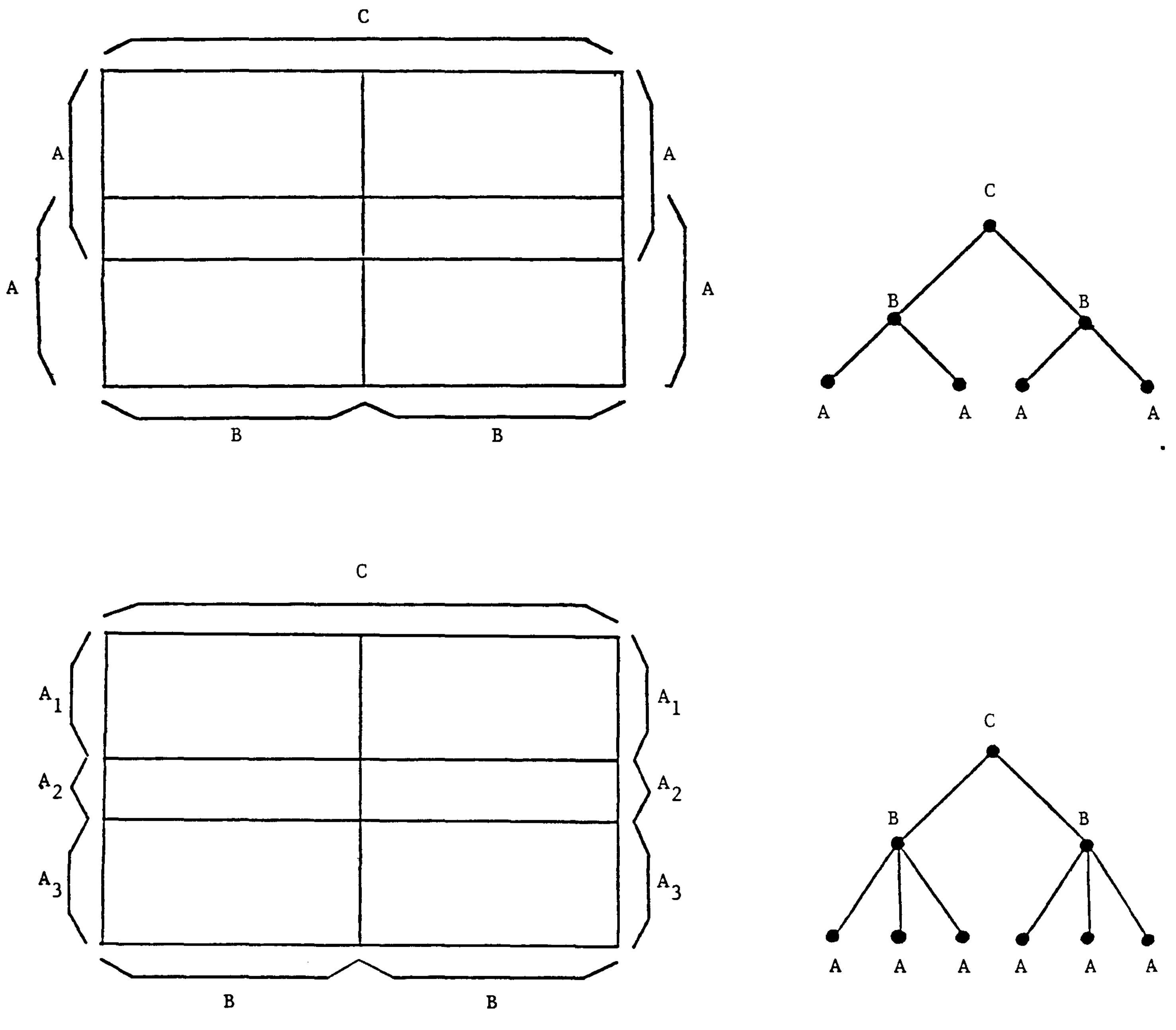


Figure 3.2

A_4. Similar cases occur quite often in VLSI design, so it behooves one to seek an amelioration. A better algorithm should find out the similarity between A_1, A_3, and A_4, then further divide A_1 and A_4 as shown in figure 3.3b. That makes the total area that needs extracted very much reduced.

If a chip contains very few repetitions, then the hierarchical extractor might take more time than an ordinary flat extractor. The time due to disjoint transformation is not significant because it is not a time-consuming job and the hierarchical extractor need do it only once. On the other hand, the hierarchical extractor needs to compose cells at each level. Assume that at each level the average overhead due to the composition of cells is about 5% compared to the extraction time, and a chip has about 7 to 9 levels in hierarchical description. Then, the total overhead is $((1.05)^7-1)$ to $((1.05)^9-1)$, or about 40%-50%, compared to the extraction time. One possible solution for this problem is to use a post-processor to modify the non-overlapping hierarchical description. The post-processor traverses the whole structure and determines which cells are good for hierarchical approach. If a cell is unsatisfactory, then the post-processor will remove the underlying substructure. That produces the flatness necessary for the hierarchical extractor. Figure 3.4 illustrates the function of the post-processor. In figure 3.4a, even though cell B contains two different subcells B_1 and B_2, a smart post-processor should find that B is good for hierarchical approach since a majority of the subcells contained in B_1 are also contained in B_2. On the other hand, cell C is obviously not good for hierarchical approach. The post-processor will remove the substructure below C. The resulting configuration is shown in figure 3.4b.

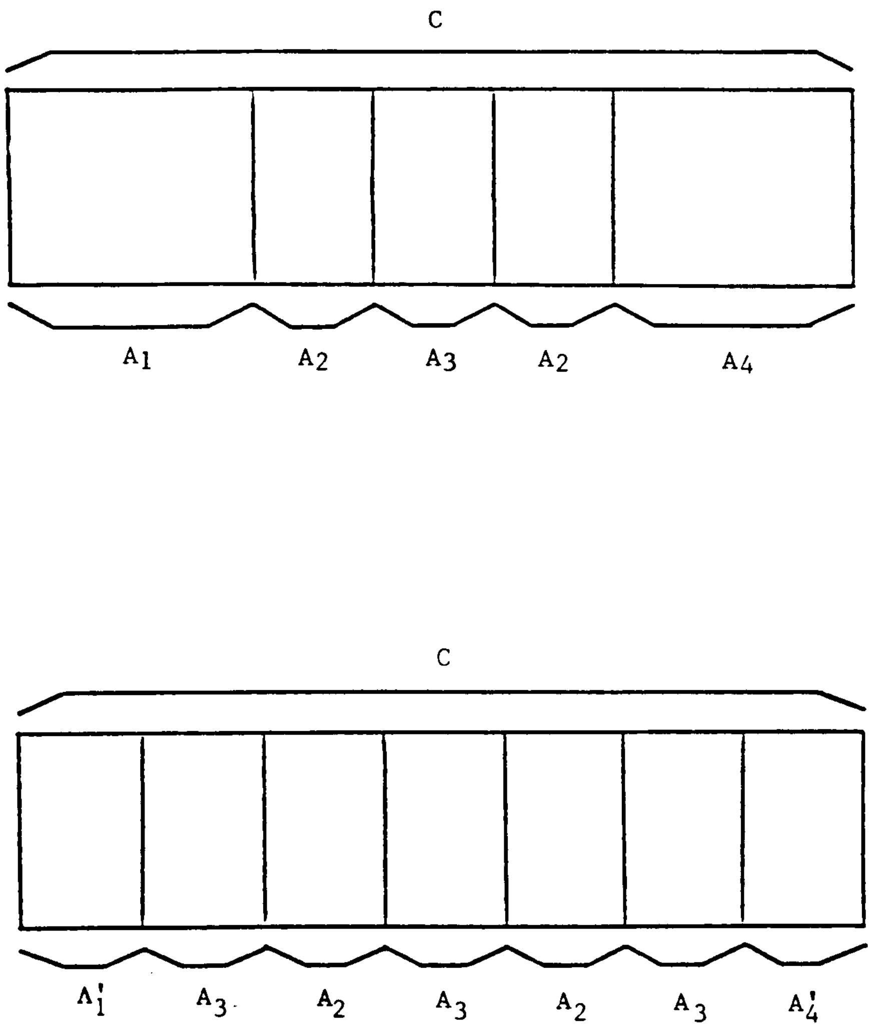


Figure 3.3

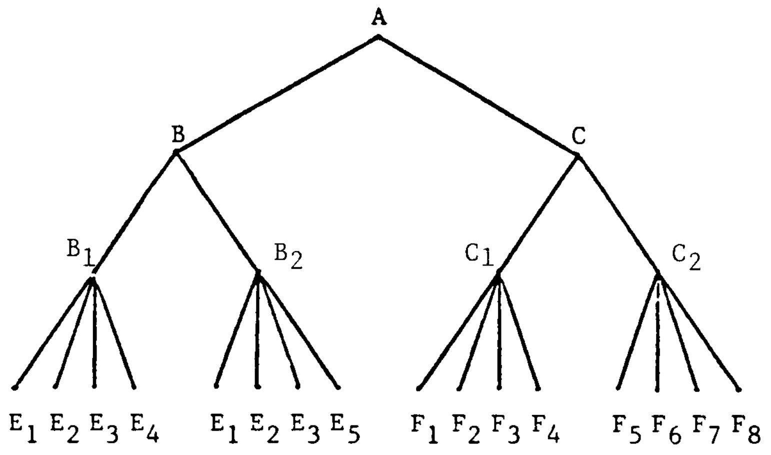


Figure 3.4a

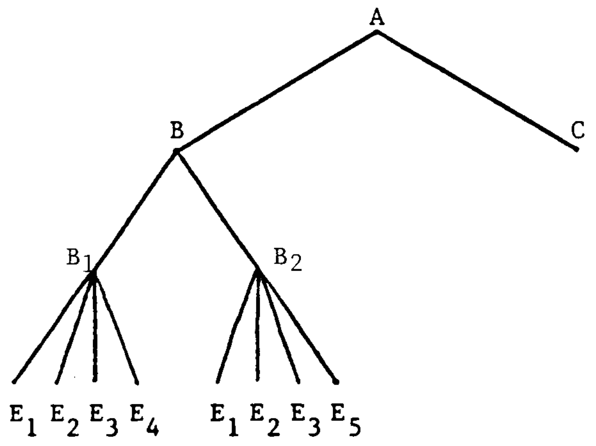


Figure 3.4b

3.3. HOW TO BUILD UP THE WHOLE CHIP

Once the non-overlapping hierarchical description is available, the hierarchical extractor starts extracting the cells at lowest level.[2] Then, it composes the cells at lower level to obtain the cells at higher level. The hierarchical extractor continues this activity, called a composition task, until it reaches the root of the non-overlapping hierarchical description.

What the hierarchical extractor needs to do in the composition task stage depends on what kind of output format is desired, since there is a choice between two: flat format and hierarchical format. No matter which selection is made, the hierarchical extractor faces the following two problems:

(1) nodes and transistors may cross the boundary of cells.

(2) The gate node and source/drain nodes of a transistor may lay on different cells.

3.3.1. Boundary Nodes and Bondary Transistors

To handle the two situations above, the nodes and transistors in a cell *are* partitioned into two groups. One group consists of the nodes and transistors that touch the boundary of this cell, called boundary nodes and boundary transistors respectively. Another group contains the nodes and transistors that don't touch the boundary of this cell, called local nodes/transistors. When composing cells, the hierarchical extractor uses the locations where the boundary nodes and boundary transistors touch the border to determine connectivity of boundary nodes and boundary transistors in different cells.

[2] Actually, the hierarchical extractor calls a flat extractor to extract leaf cells.

3.3.2. Flat Output Format and Hierarchical Output Format

In general, the flat output format consists of a list of nodes, a list of transistors, and a connection description. The nodes and transistors list record all of the nodes and transistors in the chip. As one might guess, the connection description specifies the connectivity relation between nodes and transistors. The hierarchical output format usually consists of a list of cell definitions. Each cell definition contains three parts. The first part contains a list of references to other cells, denoted as calling commands. The second part describes some circuit in this cell, which is known as circuit commands. The third part describes the connectivity of the boundary nodes and boundary transistors of this particular cell's subcells, called connection commands. The cell-calling relation denotes a hierarchical structure, which is usually isomorphic to the hierarchical structure of the non-overlapping description.

3.3.3. Generating Flat Output Format and Hierarchical Output Format

To generate the flat output format, the hierarchical extractor must combine the circuit information of the cells at a lower level to obtain the circuit information of cells at a higher level. Therefore, when composing cells the hierarchical extractor has to obtain the circuit information of each cell. For the cells that have previously been extracted, the hierarchical extractor only needs to retrieve their circuit information. For cells yet to be processed, the hierarchical extractor need only extract them. But, no matter how the hierarchical extractor obtains circuit information, the time complexity of composition is approximately proportional to the total area of these cells.

On the other hand, if the hierarchical extractor generates hierarchical output format it doesn't need to obtain all the circuit information of the cells being composed. It generates calling commands by just following the hierarchy in the non-overlapping description and generates connection commands by checking the connectivity of the boundary nodes and boundary transistors of these cells. Hence, the time complexity of composition is approximately proportional to the total length of these cells.

The argument above shows that generating hierarchical output format usually takes less time than generating flat output format. Hence, the hierarchical output format is prefered.

CHAPTER 4

IMPLEMENTATION OF HEX

4.1. BASIC ALGORITHMS OF HEX

HEX contains essentially two subprocedures. The first one, called transformer, is responsible for disjoint transformation. The second one, called H-extractor, is responsible for extracting circuit information. H-extractor is a top-down depth-first recursive procedure that walks through the non-overlapping hierarchical structure, calls flat extractor to extract leaf cells, and composes cells bottom-up to build up the whole chip.

Below is the basic algorithm of H-extractor written by Algol-like language. The detail algorithm and data structure used to implement HEX are described later.

```
procedure H-extractor(cell P);
begin
      for each subcell Qi of P do
      begin
      generate a reference command for Qi;
        if Qi has been extracted
            then retrieve the information about the
              boundary nodes and boundary transistors
              of Qi ;
        else if Qi is a leaf cell
              then call flat extractor to extract Qi and
              store the information about the
              boundary nodes and boundary transistors
              of Qi ;
        else call H-extractor(Qi) and store the information
              about the boundary nodes and boundary transistors
              of Qi ;
        compose Qi to P;
      end;
end;
```

4.2. OUTPUT FORMAT OF HEX

The output format generated by HEX must be specified before details of the data structure and algorithm are described. The hierarchical output format generated by HEX, called HXTR format, consists of a list of cell definitions. Each cell definition contains three parts.

(1) a list of arguments:

Every argument contains a list of node names that are present in this cell.

(2) a list of references to other cells:

Each reference to another cell, called a calling command, specifies the particular cell called and two sets of parameters which are passed to the called cell. Each parameter in the first set passes a character string to the called cell. These character strings are used in handling the node names defined by users. The second set of parameters pass node names to the called cell. By passing node names to the subcells, some relation between the node names of the calling and called cell is defined. In HXTR format, the node names in different cells that denote the same node are specified in this way.

(3) a list of nodes, a list of transistors, and a connection description:

The node list and the transistor list define[1] some local nodes and some local transistors of the cell[2]. In node definition commands, every node name is associated with a decimal number. When this cell is called, the number specifies which character string is passed from the calling cell through the first set of parameters then put before the node name.

[1] By "define", we mean the information about the node (or transistor) is presented.

[2] Some local nodes and local transistors may be defined in this cell's subcells. Actually, in the HXTR file generated by HEX, a node (or transistor) is defined in the smallest cell in which the

The formal definition of the format is given below. The standard notation proposed by Niklaus Wirth is used: production rules use equal = to relate an identifier to expressions, a vertical bar | for or, double quotes " " to surround terminal characters, curly brackets {} to indicate repetition any number of times, (including zero), square brackets [] to indicate optional factors, and parentheses () for grouping.

```
output-file        = {{blank}[cell-definition] semi}{blank}"EOF".
cell-definition    = "CELL" cellname semi {callingCommand}
                          {arguments} {nodeCommand}
                          {transistorCommand} "ENDC".
callingCommand     = {"CALL" cellname "(" prefixes ")" parameters semi}.
prefixes           = {characterstring comma}.
parameters         = {nodename semi}.
arguments          = {nodelist semi}.
nodelist           = nodename " " {nodename " "}.
nodename           = decimalnumber characterstring.
nodeCommand        = {polyarea polygatearea diffusionarea metalarea
                          {nodename} semi}.
transistorCommand= {transistortype transistorarea
                          gateNodeName contactNodeName
                          contactNodeName semi}.
semi               = ";".
comma              = ",".
```

4.3. DETAIL ALGORITHM AND DATA STRUCTURE

4.3.1. Disjoint Transformation

The transformer procedure carries out disjoint transformation by dividing a cell into several non-overlapping subcells recursively. At each level, it follows the following steps.

(1) Does this cell contain only primitive geometry ? If so, return.

node (or transistor) is a local node (or local transistor).

(2) Expand the cell one level. See figure 4.1a.

(3) Slice the cell into several subcells, using the boundary of the instances in cell as guidance. See figure 4.1b.

(4) Compare each newly generated cell with the cells that were generated previously. If an identical cell is found, jolly good show. It is not necessary to extract this newly generated cell. Otherwise, (bloody hell!) try to further divide this cell so that some of its subcells are identical to some of the cells that were generated previously.

(5) Apply steps (1) - (4) to the newly generated cells that have no identical cell.

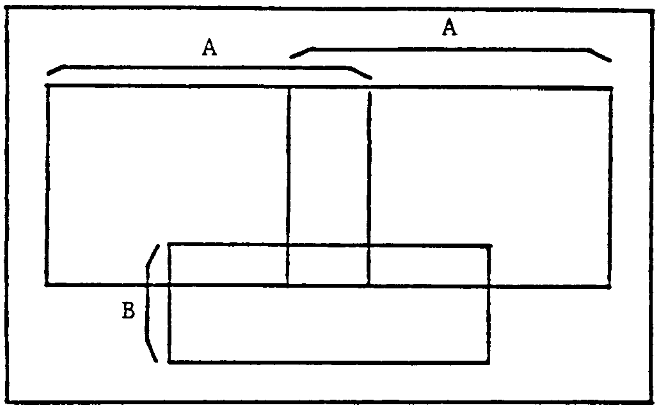
The purpose of involving step 4 is to obtain a non-overlapping description that preserves repetition better. Of course, this strategy fails in some cases. However, for most cases in VLSI chip it works quite well. For example, the typical case in VLSI chip like that one in figure 3.3a can be solved by this approach.

4.3.2. Algorithm and Data Structure for Flat Extractor

The flat extractor is responsible for extracting the lowest level cells. The implementation of the flat extractor is based on Baker's algorithm [1], which is described in Chapter 2. The data structure used in implementing the flat extractor is described below.

(1) node records : The information of a node is saved in a node record.

(2) transistor records : The information of a transistor is saved in a transistor record. Each transistor record contains a pointer to its gate node and another pointer to the list of its source/drain nodes.



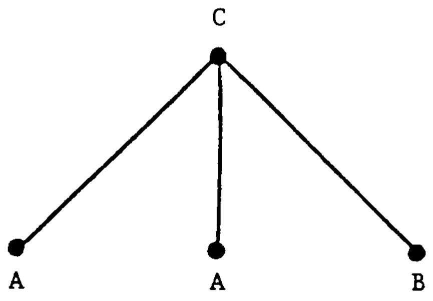


Figure 4.1a

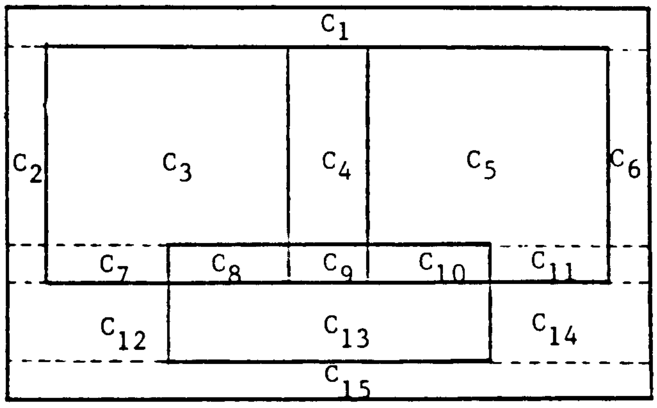


Figure 4.1b

Actually, both node and transistor records contain one more pointer, called realnode. This pointer is used in merging two nodes or two transistors. When merging two nodes or transistors, the extractor, (in addition to combining the information of two nodes or transistors), will make the realnode pointer of one node or transistor point to another node or transistor.

[3] A queue of pointers : The size of the queue is equal to the number of columns in the bit map. Each entry in the queue has one pointer to transistor record and three to node record. The three pointers to node record correspond to the nodes on the derived layer D, P, and M. To retrieve the information of a pixel, the extractor must be able to obtain the pointers of the nodes and transistor on the pixel. During the ponds and islands processing, the transistor and node pointers of the pixels that were scanned most recently are saved in the queue. Keep in mind that the first entry of the queue keeps pointers for the pixel above the current picture element the last entry keeps pointers for the pixel left of current.

[4] a list of node records: The list contains the records of the local nodes in the cell.

[5] a list of transistor records: The list contains the records of the local transistors in the cell.

The data structure above is enough for flat extractor. However, it is not enough for hierarchical extractor. As it is mentioned in Chapter 3, in hierarchical extractor the information of the boundary nodes and boundary transistors needs to be saved. Hence, in HEX, each cell has a array of pointers for this purpose.

[6] a array of pointers: The node and transistor pointers of the pixels on the boundary are saved in the array. Each entry in this array is mapped to a pixel on the boundary and the index of the array reflects the location of its corresponding pixel. When given a location on the boundary, the hierarchical extractor can use this configuration to find out the nodes and transistor that touch the boundary at the location.

Figure 4.2 shows the configuration of the data structure after the extractor has finished extracting the cell.

4.3.3. Algorithm of Composition

In composition, HEX generates calling commands according to the structure in non-overlapping hierarchical description. To handle the boundary nodes and boundary transistors, HEX examines every common border of the cells being composed pixel by pixel. From the array of pointers for boundary pixels, HEX can easily determine the connectivity relation of the boundary nodes and boundary transistors in the cells being composed. The connectivity relation can be definitely described by passing node names through the cell-calling mechanism. In addition to generating calling commands, HEX carries out the two functions below.

(1) Merge the boundary nodes/transistors that are connected.

(2) Establishes the transistor-source/drain relation of every pair of boundary node/transistor that are connected.

After (1) is done, some boundary nodes (or boundary transistors) of the cells being composed may still be the boundary nodes (or boundary transistors) of the cell newly formed. For this kind of nodes, HEX assigns one argument to each of them. For the boundary nodes/transistors that become local

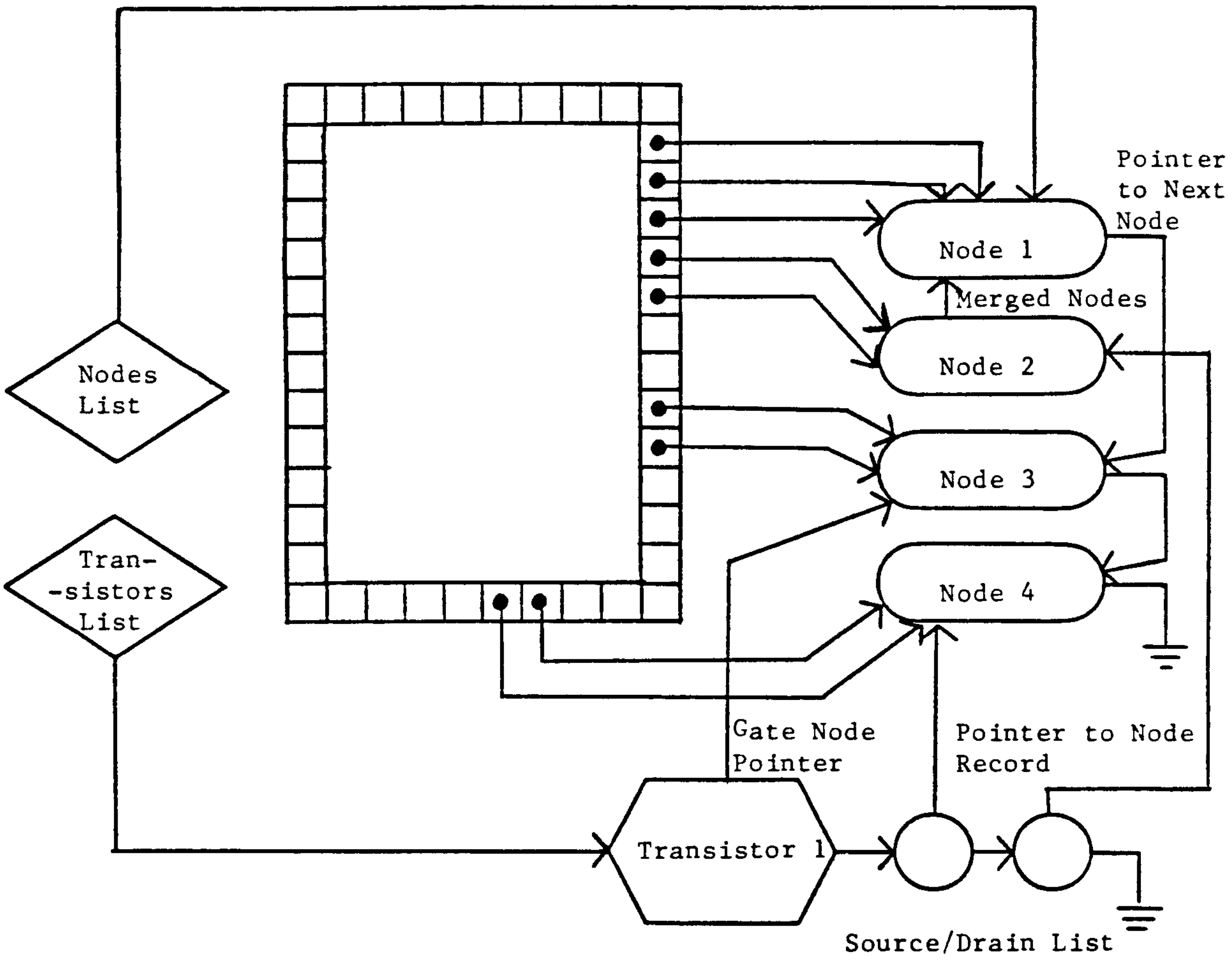

Pointer to Next Node
Node 1
Merged Nodes
Node 2
Node 3
Node 4
Nodes List
Tran- -sistors List
Gate Node Pointer
Pointer to Node Record
Transistor 1
Source/Drain List

Figure 4.2

nodes/transistors) of the newly formed cell, HEX prints their definition commands.

CHAPTER 5

COMPLEXITY AND PERFORMANCE

5.1. COMPLEXITY

To analyze the complexity of hierarchical extractors let us consider the following two cases.

(1) Each symbol contains m identical instances except those at lowest level.

(2) Each symbol contains m distinguish instances except those at lowest level.

Case 1 is the best configuration for hierarchical extractors and case 2 is the worst configuration for hierarchical extractors. The upper bound and lower bound of the time complexity of hierarchical extractors can be obtained by analyzing these two cases.

At first, define

T(A) : time to extract a cell with area A hierarchically. Actually, the area of a cell is in unit of number of pixels in it.

a * l: time to compose two cells with common boundary length l. Actually, length is in unit of number of pixels on the boundary.

b * A: time for the flat extractor extracting a cell with area A.

For case 1, the recursive equation of time complexity is of the form :

(5.1) $T(A) \sim T(A/m) + (m-1)*a*A^{0.5} + d^1$

Because the extractor needs to extract only one of the m discells, the coefficient of the term T(A/m) is 1. The term $(m-1)*a*A^{0.5}$ is the time to compose these m discells.(the length of common boundary of two discells is about of the order of square root of A.) The last term d is the time to divide this cell and is almost a constant. The approximate solution for recursive equation (5.1) is

$$(5.2) \qquad T(A) \sim m*a*A^{0.5} + d*\log_m A + b.$$

For case 2, the recursive equation of time complexity is of the form :

$$(5.3) \qquad T(A) \sim m*T(A/m) + (m-1)*a*A^{0.5} + d.$$

Because these m discells are distinguish, the extractor needs to extract each discells separately. So, the coefficient of T(A/m) is m. The terms $(m-1)*a*A^{0.5}$ and d are the same as that in 5.1. The approximate solution for this recursive equation is

$$(5.4) \qquad T(A) \sim b*A + m*a*A^{0.5} + d*\log_m A.$$

The complexity analysis above shows that in the best case hierarchical approach reduces the the time complexity from O(A) down to $O(A^{0.5})$. In the worst case the the hierarchical approach doesn't increase the order of time complexity but adds two more terms of smaller order.

---

[1]the symbol "~" means "approximately equal to"

5.2. PERFORMANCE

The run time data shows that for the cells with mid-size to large size[2] the time to retrieve circuit information of the cell and compose the cell with other cells is usually less than 5% of the time to extract the cell flatly. This means that potentially the speedup of hierarchical approach is at least 20. However, except the chips such as memory and systoic array the repetition of 4 to 16 times is more often. The table below shows the run time data for some regular chip.[3]

Other running time data shows that if a chip containing very few repetitions is extracted then HEX may take 20%-50% more time than the flat extractor.

chip	# of repetitions of identical cells	time used by HEX/ time used by flat extractors
communication ring	16	0.186
supermesh	16	0.299
supermesh	64	0.103
pipelined multiplier	8	0.352

[2] a cell with 10,000 lambda square is considered as a mid-size cell.

[3] these data are obtained by executing HEX and the flat extractor on VAX-UNIX system.

CHAPTER 6

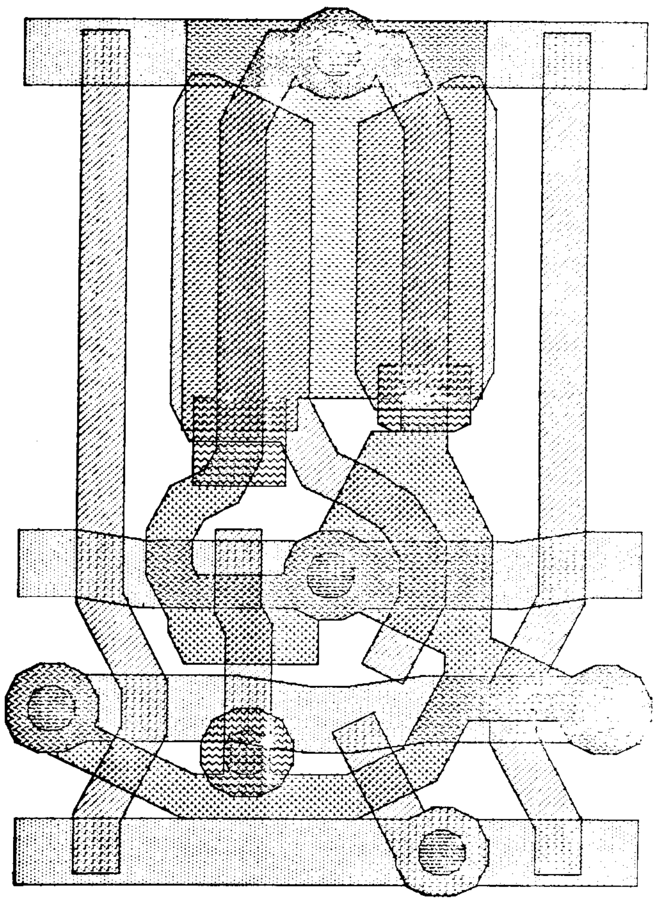
CONCLUSION AND FUTURE WORK

The performance of HEX demonstrates that hierarchical approach is one possible way to handle the more and more complicated VLSI chip. The runtime data shows that for regular designs the speedup is about 3-10 times faster. While for some design, HEX runs a little slower than the flat extractor, (primarily due to the overhead of disjoint transformation and composing subcells). The performance of HEX can be further improved by using more intelligent strategy in disjoint transformation. This work is still under research. Another thing to be done is making extractors be able to extract more electrical information; there is still no good algorithm for extracting resistance information.

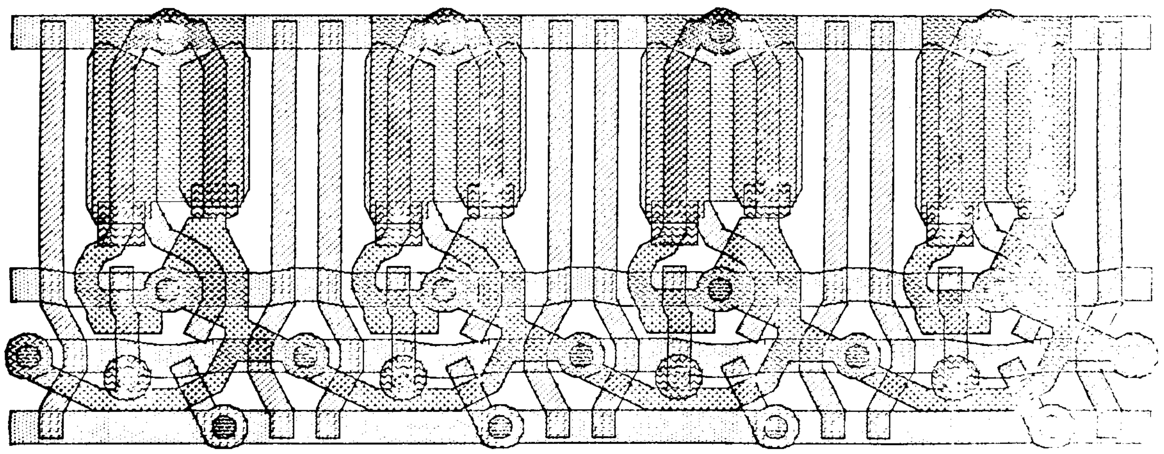
REFERENCE

1. C.M. Baker, "Artwork Analysis Tools for VLSI Circuits", Technique Report-239 MIT Laboratory for Computer Science, Cambridge, Massachusetts, 1980.
2. R. Hon "The Hierarchical Analysis of VLSI Design", PhD Thesis Proposal, VLSI Memo V073, Carnegie Mellon University, Pittsburgh, 1981.
3. W.Lattin "VLSI Design Methodology : The Problem of the 80's for Microprocessor Design" Proc. Caltech Conf. on Very Large Scale Integration, pp 248-252, Caltech, 1979.
4. C. Mead and L. Conway, Introduction to VLSI System, Addison-Wesley, 1980.
5. M. Newell and D. Fitzpatrick "Exploiting Structure in Integrated Circuit Design Analysis", Proc. 1982 Conf. on Advanced Research in VLSI, MIT, Cambridge, pp 84-92, 1982.
6. M. Tucker and L. Scheffer "A Constrained Design Methodology for VLSI", VLSI Design, May/June, 1982.
7. T. Whitney " A Hierarchical Design Analysis Front End", Proc. VLSI Intl. Conf. on Very Large Scale Integration, Edinburgh, pp 217-225, 1981.

APPENDIX



1-Bit Register



4 repetitions

HXTR FILE GENERATED BY HEX
(for the 4-repetition slice)

```
h 100 t.cif regslice NMOS;
CELL 1 0 0;
#1 = _NODE1 ;
#2 = _NODE2 ;
#3 = _NODE3 ;
#4 = _NODE4 ;
#5 = _NODE5 ;
#6 = _NODE6 ;
ENDC;

CELL 2 2 5;
#1 = _NODE14 ;
#2 = _NODE18 ;
#3 = _NODE19 ;
#4 = _NODE7 ;
#5 = _NODE20 ;
#6 = _NODE21 ;
#7 = _NODE22 ;
#8 = _NODE15 _NODE16 _NODE17 ;
#9 = _NODE8 _NODE9 ;
#10 = _NODE11 _NODE12 ;
s 492 182 55 28 106 24 13 0 0 0 _NODE10;
s 1 0 1 1 0 0 0 0 0 0 _NODE13;
d 98 25 5 7 _NODE15  _NODE15 0 4 0 0 _NODE11 2 0 0 3;
n 71 16 10 3 _NODE10  _NODE15 0 1 16 7 _NODE8 10 5 0 1;
n 14 5 4 3 _NODE14  _NODE15 0 0 3 1 _NODE7 3 0 0 0;
d 111 28 5 7 _NODE10  _NODE11 0 0 2 3 _NODE10 0 4 0 0;
n 31 8 5 3 _NODE7  _NODE8 0 0 8 1 _NODE10 8 1 0 0;
ENDC;

CELL 3 0 0;
    CALL 2() 10;
      E#1 = _NODE23;
      E#3 = _NODE24;
      E#4 = _NODE25;
      E#5 = _NODE26;
      E#6 = _NODE27;
      E#9 = _NODE28;
      E#10 = _NODE29;
      E#2 = _NODE30;
      E#7 = _NODE31;
      E#8 = _NODE32;
#1 = _NODE23 ;
#2 = _NODE30 ;
#3 = _NODE24 ;
#4 = _NODE25 ;
#5 = _NODE26 ;
```

```
#6 = _NODE27 ;
#7 = _NODE31 ;
#8 = _NODE32 ;
#9 = _NODE28 ;
#10 = _NODE29 ;
ENDC;

CELL 4 0 0;
    CALL 1() 6;
      E#1 = _NODE33;
      E#2 = _NODE34;
      E#3 = _NODE35;
      E#4 = _NODE36;
      E#5 = _NODE37;
      E#6 = _NODE38;
    CALL 3() 10;
      E#1 = _NODE39;
      E#3 = _NODE40;
      E#4 = _NODE41;
      E#5 = _NODE42;
      E#6 = _NODE43;
      E#9 = _NODE44;
      E#10 = _NODE45;
      E#2 = _NODE46;
      E#7 = _NODE47;
      E#8 = _NODE48;
#1 = _NODE33 _NODE39 ;
#2 = _NODE46 ;
#3 = _NODE35 _NODE42 _NODE43 ;
#4 = _NODE47 ;
#5 = _NODE48 ;
#6 = _NODE37 _NODE44 ;
#7 = _NODE38 _NODE45 ;
s 309 16 136 8 0 0 0 0 0 0 _NODE36 _NODE34 _NODE40
   ;
s 85 31 22 5 93 10 19 0 0 0 _NODE41 _0#reg;
n 16 5 5 3 _NODE36  _NODE41 0 1 3 0 _NODE35 3 0 0 1;
ENDC;

CELL 5 0 0;
#1 = _NODE49 ;
#2 = _NODE50 ;
#3 = _NODE51 ;
#4 = _NODE52 ;
#5 = _NODE53 ;
#6 = _NODE54 ;
#7 = _NODE55 ;
#8 = _NODE56 ;
ENDC;

CELL 6 0 0;
#1 = _NODE57 ;
#2 = _NODE58 ;
```

```
#3 = _NODE59 ;
#4 = _NODE60 ;
#5 = _NODE61 ;
#6 = _NODE62 ;
ENDC;

CELL 7 0 0;
    CALL 2() 10;
      E#1 = _NODE63;
      E#3 = _NODE64;
      E#4 = _NODE65;
      E#5 = _NODE66;
      E#6 = _NODE67;
      E#9 = _NODE68;
      E#10 = _NODE69;
      E#2 = _NODE70;
      E#7 = _NODE71;
      E#8 = _NODE72;
    CALL 6() 6;
      E#1 = _NODE73;
      E#2 = _NODE74;
      E#3 = _NODE75;
      E#4 = _NODE76;
      E#5 = _NODE77;
      E#6 = _NODE78;
#1 = _NODE63 _NODE73 ;
#2 = _NODE64 ;
#3 = _NODE65 ;
#4 = _NODE66 ;
#5 = _NODE71 _NODE67 _NODE75 ;
#6 = _NODE68 _NODE77 ;
#7 = _NODE69 _NODE78 ;
s 308 25 143 6 0 0 0 0 0 0 _NODE70 _NODE74 _NODE76
   ;
s 332 98 30 12 189 38 18 0 0 0 _NODE72;
n 25 7 5 3 _NODE70  _NODE71 0 1 3 2 _NODE72 7 0 0 2;
ENDC;

CELL 8 0 0;
    CALL 4(R0) 7;
      E#7 = _NODE79;
      E#1 = _NODE80;
      E#2 = _NODE81;
      E#3 = _NODE82;
      E#4 = _NODE83;
      E#5 = _NODE84;
      E#6 = _NODE85;
    CALL 5(R1,R0) 8;
      E#1 = _NODE86;
      E#3 = _NODE87;
      E#4 = _NODE88;
      E#5 = _NODE89;
      E#7 = _NODE90;
    E#8 = _NODE91;
    E#2 = _NODE92;
    E#6 = _NODE93;
  CALL 2(R1) 10;
    E#1 = _NODE94;
    E#3 = _NODE95;
    E#4 = _NODE96;
    E#5 = _NODE97;
    E#6 = _NODE98;
    E#9 = _NODE99;
    E#10 = _NODE100;
    E#2 = _NODE101;
    E#7 = _NODE102;
    E#8 = _NODE103;
  CALL 5(R2,R1) 8;
    E#1 = _NODE104;
    E#3 = _NODE105;
    E#4 = _NODE106;
    E#5 = _NODE107;
    E#7 = _NODE108;
    E#8 = _NODE109;
    E#2 = _NODE110;
    E#6 = _NODE111;
  CALL 2(R2) 10;
    E#1 = _NODE112;
    E#3 = _NODE113;
    E#4 = _NODE114;
    E#5 = _NODE115;
    E#6 = _NODE116;
    E#9 = _NODE117;
    E#10 = _NODE118;
    E#2 = _NODE119;
    E#7 = _NODE120;
    E#8 = _NODE121;
  CALL 5(R3,R2) 8;
    E#1 = _NODE122;
    E#3 = _NODE123;
    E#4 = _NODE124;
    E#5 = _NODE125;
    E#7 = _NODE126;
    E#8 = _NODE127;
    E#2 = _NODE128;
    E#6 = _NODE129;
  CALL 7(R3) 7;
    E#1 = _NODE130;
    E#2 = _NODE131;
    E#3 = _NODE132;
    E#4 = _NODE133;
    E#5 = _NODE134;
    E#6 = _NODE135;
    E#7 = _NODE136;
s 380 56 68 13 0 0 0 1220 62 39 _NODE80 _NODE86 _NODE94
  _NODE104 _NODE112 _NODE122 _NODE130;
```

```
s 0 0 0 0 380 48 16 1220 62 39 _NODE79 _NODE91 _NODE100
   _NODE109 _NODE118 _NODE127 _NODE136;
s 0 0 0 0 352 80 9 1240 76 39 _NODE120 _NODE102 _NODE83
   _NODE82 _NODE88 _NODE97 _NODE98 _NODE106
   _NODE115 _NODE116 _NODE124 _NODE133 _NODE134
   ;
s 309 16 136 8 0 0 0 0 0 0 _NODE129 _NODE128 _NODE131
   ;
s 0 0 0 0 404 64 12 1232 74 39 _NODE85 _NODE90 _NODE99
   _NODE108 _NODE117 _NODE126 _NODE135;
s 85 31 22 5 93 10 19 0 0 0 _NODE132 _0#R3.reg;
s 308 25 143 6 0 0 0 0 0 0 _NODE119 _NODE123 _NODE125
   ;
s 332 98 30 12 189 38 18 0 0 0 _NODE121;
s 309 16 136 8 0 0 0 0 0 0 _NODE111 _NODE110 _NODE113
   ;
s 85 31 22 5 93 10 19 0 0 0 _NODE114 _0#R2.reg;
s 308 25 143 6 0 0 0 0 0 0 _NODE101 _NODE105 _NODE107
   ;
s 332 98 30 12 189 38 18 0 0 0 _NODE103;
s 309 16 136 8 0 0 0 0 0 0 _NODE93 _NODE92 _NODE95
   ;
s 85 31 22 5 93 10 19 0 0 0 _NODE96 _0#R1.reg;
s 308 25 143 6 0 0 0 0 0 0 _NODE81 _NODE87 _NODE89
   ;
s 332 98 30 12 189 38 18 0 0 0 _NODE84;
n 16 5 5 3 _NODE129  _NODE132 0 1 3 0 _NODE120 3 0 0 1;
n 25 7 5 3 _NODE119  _NODE120 0 1 3 2 _NODE121 7 0 0 2;
n 16 5 5 3 _NODE111  _NODE114 0 1 3 0 _NODE120 3 0 0 1;
n 25 7 5 3 _NODE101  _NODE120 0 1 3 2 _NODE103 7 0 0 2;
n 16 5 5 3 _NODE93  _NODE96 0 1 3 0 _NODE120 3 0 0 1;
n 25 7 5 3 _NODE81  _NODE120 0 1 3 2 _NODE84 7 0 0 2;
ENDC;

ENDC
```

XTR FILE GENERATED BY FLATOR

```
h 100 t.cif regslice NMOS;
s 492 182 55 28 106 24 13 0 0 0 _NODE-7;
s 1 0 1 1 0 0 0 0 0 0 _NODE-8;
s 309 16 136 8 0 0 0 0 0 0 _NODE-9;
s 85 31 22 5 93 10 19 0 0 0 _NODE-10 R0.reg;
s 492 182 55 28 106 24 13 0 0 0 _NODE-11;
s 1 0 1 1 0 0 0 0 0 0 _NODE-12;
s 492 182 55 28 106 24 13 0 0 0 _NODE-13;
s 1 0 1 1 0 0 0 0 0 0 _NODE-14;
s 492 182 55 28 106 24 13 0 0 0 _NODE-15;
s 1 0 1 1 0 0 0 0 0 0 _NODE-16;
s 308 25 143 6 0 0 0 0 0 0 _NODE-17;
s 332 98 30 12 189 38 18 0 0 0 _NODE-18;
s 380 56 68 13 0 0 0 1220 62 39 _NODE130 _NODE122 _NODE112
   _NODE104 _NODE94 _NODE86 _NODE80;
s 0 0 0 0 380 48 16 1220 62 39 _NODE136 _NODE127 _NODE118
   _NODE109 _NODE100 _NODE91 _NODE79;
s 0 0 0 0 352 80 9 1240 76 39 _NODE134 _NODE133 _NODE124
   _NODE116 _NODE115 _NODE106 _NODE98 _NODE97
   _NODE88 _NODE82 _NODE83 _NODE102 _NODE120
   ;
s 309 16 136 8 0 0 0 0 0 0 _NODE131 _NODE128 _NODE129
   ;
s 0 0 0 0 404 64 12 1232 74 39 _NODE135 _NODE126 _NODE117
   _NODE108 _NODE99 _NODE90 _NODE85;
s 85 31 22 5 93 10 19 0 0 0 _NODE132 R3.reg;
s 308 25 143 6 0 0 0 0 0 0 _NODE125 _NODE123 _NODE119
   ;
s 332 98 30 12 189 38 18 0 0 0 _NODE121;
s 309 16 136 8 0 0 0 0 0 0 _NODE113 _NODE110 _NODE111
   ;
s 85 31 22 5 93 10 19 0 0 0 _NODE114 R2.reg;
s 308 25 143 6 0 0 0 0 0 0 _NODE107 _NODE105 _NODE101
   ;
s 332 98 30 12 189 38 18 0 0 0 _NODE103;
s 309 16 136 8 0 0 0 0 0 0 _NODE95 _NODE92 _NODE93
   ;
s 85 31 22 5 93 10 19 0 0 0 _NODE96 R1.reg;
s 308 25 143 6 0 0 0 0 0 0 _NODE89 _NODE87 _NODE81
   ;
s 332 98 30 12 189 38 18 0 0 0 _NODE84;
d98 25 5 7 _NODE84  _NODE84 0 4 0 0 _NODE79 2 0 3 0;
n71 16 10 3 _NODE-7  _NODE84 0 1 7 16 _NODE85 10 5 1 0;
n14 5 4 3 _NODE80  _NODE84 0 0 1 3 _NODE-10 3 0 0 0;
d111 28 5 7 _NODE-7  _NODE79 0 0 3 2 _NODE-7 0 4 0 0;
n31 8 5 3 _NODE-10  _NODE85 0 0 1 8 _NODE-7 8 1 0 0;
n16 5 5 3 _NODE-9  _NODE-10 0 1 0 3 _NODE82 3 0 1 0;
d98 25 5 7 _NODE103  _NODE103 0 4 0 0 _NODE100 2 0 3 0;
```

```
n71 16 10 3 _NODE-11  _NODE103 0 1 7 16 _NODE99 10 5 1 0;
n14 5 4 3 _NODE94  _NODE103 0 0 1 3 _NODE96 3 0 0 0;
d111 28 5 7 _NODE-11  _NODE100 0 0 3 2 _NODE-11 0 4 0 0;
n31 8 5 3 _NODE96  _NODE99 0 0 1 8 _NODE-11 8 1 0 0;
d98 25 5 7 _NODE121  _NODE121 0 4 0 0 _NODE118 2 0 3 0;
n71 16 10 3 _NODE-13  _NODE121 0 1 7 16 _NODE117 10 5 1 0;
n14 5 4 3 _NODE112  _NODE121 0 0 1 3 _NODE114 3 0 0 0;
d111 28 5 7 _NODE-13  _NODE118 0 0 3 2 _NODE-13 0 4 0 0;
n31 8 5 3 _NODE114  _NODE117 0 0 1 8 _NODE-13 8 1 0 0;
d98 25 5 7 _NODE-18  _NODE136 2 0 3 0 _NODE-18 0 4 0 0;
n71 16 10 3 _NODE-15  _NODE135 10 5 1 0 _NODE-18 0 1 7 16;
n14 5 4 3 _NODE130  _NODE132 3 0 0 0 _NODE-18 0 0 1 3;
d111 28 5 7 _NODE-15  _NODE-15 0 4 0 0 _NODE136 0 0 3 2;
n31 8 5 3 _NODE132  _NODE-15 8 1 0 0 _NODE135 0 0 1 8;
n25 7 5 3 _NODE-17  _NODE134 0 1 2 3 _NODE-18 7 0 2 0;
n16 5 5 3 _NODE129  _NODE120 3 0 1 0 _NODE132 0 1 0 3;
n25 7 5 3 _NODE119  _NODE121 7 0 2 0 _NODE120 0 1 2 3;
n16 5 5 3 _NODE111  _NODE120 3 0 1 0 _NODE114 0 1 0 3;
n25 7 5 3 _NODE101  _NODE103 7 0 2 0 _NODE120 0 1 2 3;
n16 5 5 3 _NODE93  _NODE120 3 0 1 0 _NODE96 0 1 0 3;
n25 7 5 3 _NODE81  _NODE84 7 0 2 0 _NODE120 0 1 2 3;
```