A Hierarchical Design Rule Checker

Telle Whitney

Computer Science Department
California Institute of Technology

4320:TR:81

CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science Department

Silicon Structures Project

Technical Report 4320

A Hierarchical Design Rule Checker

by

Telle Whitney

Submitted in Partial Fulfillment

of the Requirements for the degree

of Master of Science

May 19, 1981

Abstract

This thesis describes a new approach to the problem of Geometrical Design Rule Checking (DRC). Previous DRC implementations have dealt with fully instantiated geometrical artwork. As the complexity of VLSI increases, it becomes infeasible to analyze the vast amounts of information present in a fully instantiated design. The DRC algorithm presented here introduces an approach that exploits the structural hierarchy of a design in order to reduce the computational complexity of the geometrical tests that need to be made. The technique described is also applicable to other types of design checking such as circuit extraction, functional verification and electrical rule verification.

A new DRC algorithm has been developed that, by making use of the structure inherent in a hierarchical design, eliminates many redundant design rule checks. In this approach there are two places where possible design rule violations may occur. The first is within a symbol definition. The second is the area where two symbols interact. The algorithm checks a given definition only once, and then examines how interactions within each new environment where the definition is placed modify the original definition. A note is made after each interaction has been scrutinized, so that a duplicate situation will not be rechecked.

An implementation of the hierarchical DRC algorithm has been written at Caltech. This implementation extracts a minimal number of pairwise geometrical comparisons needed to check the entire design. The program accepts as input a design description in the Caltech Intermediate Form (CIF). The output of the program is currently a fully instantiated version of those portions of the geometry that need to be checked in order to check the entire design.

A means of expressing the designer's intent through the design description is required. Current DRC's deal with geometrical artwork exclusively. Most of the difficult design rules are involved in the checking of devices. Rather than restricting the designer to the use of geometry, the idea of a primitive element is introduced. A primitive element is defined to be anything that cannot be broken down into sub-elements. A design defined using primitive elements conveys more of the functional structure than a purely geometric definition.

Acknowledgements

I would like to express my gratitude to all the students, faculty and industrial sponsors at Caltech who gave me their support and encouragement during the course of this project. In particular I wish to thank Bob Sproull who originally developed the ideas on which this thesis is based and Ivan Sutherland who introduced me to the ideas. Also special thanks goes to the "geometry group", a group which met once a week and contributed many ideas to this project. This group was comprised of Ivan Sutherland, Ed McGrath, Don Oestreicher, Ricky Mosteller and Eric Barton. I owe special thanks to Dick Lang whose constant support helped me enormously. This project was supported by the Silicon Structures Project at Caltech.

Table of Contents

List of Figures and Tables

Figure

Chapter I

Introduction

1.1 Background

Hierarchical design is a design methodology which allows IC designers to cope with the complexity of a VLSI design. VLSI technologies are currently capable of fabricating chips of remarkable complexity. Chips are under development which contain 100,000 transistors. Chips containing over a million transistors can be visualized, and the technology will support fabrication of such chips in the near future. The magnitude of the design effort required for such a chip would overwhelm any designer if the current design methods were used. Techniques need to be adopted which reduce the complexity of a design. One such design methodology is hierarchical design. A hierarchical approach to complex systems is an important concept for reducing the design and computational complexity[22].

A hierarchical design is a design which is defined as the composition of other smaller designs. A hierarchical design style is based on a methodology that assumes that a problem can be broken down into a set of smaller problems. This division of the problem continues until a simple solution for each of the small problems is feasible. Then the primitive solutions are combined together to form the larger solution. This approach to problem solving is a hierarchical approach.

The approach of design tools should change to accommodate and exploit changes in the design methodology used. Current design tools view an LSI design as a single entity. This view of the chip as a single nonhierarchical entity no longer makes sense. Design tools should use the hierarchy present in a design produced using a hierarchical design method. Design tools which exploit the hierarchy may be the only ones that can cope with the computational complexity of a million transistor chip.

A Geometrical Design Rule Checker (DRC[1]) is one of the design tools traditionally available to IC designers. A Geometrical Design Rule Checker is a piece of software that accepts a geometrical description of an integrated circuit design as input, and

lists the possible geometrical design errors implied by a set of design rules. These errors are violations of rules dictated by the technology in which the integrated circuit design is to be implemented. Current DRC programs accept a description of the geometric artwork of the entire design as input. These programs then determine all polygonal interactions and note any design rule violations.

Recent DRC programs reduce the number of comparisons performed by implementing algorithms that incorporate various sorting and windowing schemes[1]. In the worst case, the number of comparisons performed between polygons is proportional to the square of the number of polygons. Sorting and windowing ensure that comparisons between two pieces of geometry occur only when they are within a specified distance of each other. This approach reduces the amount of information the DRC program must process at any given time. However the design is still regarded as a single, monolithic entity.

A hierarchical design typically contains many uses of a particular collection of geometry, each use in a different context. A simple example of this is a memory array. A memory array is a memory cell repeated many times. There are two types of repetitious structures which occur: 1) the same memory cell is repeated many times, 2) the placement of two cells together is repeated many times.

If a design contains repetitive structures, a DRC need not test many redundant comparisons. The redundancy is clearly apparent in a memory array. The DRC must check the memory cell once. Then, it must determine if the placement of two cells next to each other introduces any new violations. In an array, however, the identical placement of cells occurs many times. A comparison between two cells need be done only once for each unique placement.

A hierarchical design rule checker can reduce dramatically the number of comparisons needed for designs described in a structured and regular manner. The DRC can accomplish this by checking a defined interaction only once, though the interaction may occur many times throughout the design.

[1] The acronym DRC has come to be associated with a Geometrical Design Rule Checker. This abbreviation is somewhat misleading since any program that checks any part of a design could be considered a design rule checker. For the purposes of this thesis however, the term DRC is used to refer to a geometrical design rule checker.

1.2 Overview

This thesis describes a new approach to design rule checking. The approach described exploits the hierarchy of a VLSI design. The subjects covered include the algorithm, an example implementation and experimental results.

The first section describes the three part hierarchical design rule checking algorithm. The first part of the algorithm looks at a symbol definition. The second part finds comparisons between elements. The third part performs polygonal checks to reveal possible design rule violations.

The second section describes a SIMULA implementation of a hierarchical design rule filter. The program was written and tested at Caltech. The filter takes a design specified in the Caltech Intermediate Form (CIF) and removes many redundant design rule checks. The program produces a fully instantiated design description which can then be used as input to a traditional design rule checker.

The third section contains a description of some of the results obtained by running various IC designs through the filter. The designs used as input included Caltech student designs and Digital Equipment Corporation (DEC) designs.

The two appendices describe features of the hierarchical design rule filter implementation. Appendix I presents some CIF user extensions supported by the filter. Appendix II is a user's manual for the SIMULA version of the filter.

The results of testing the filter on some large designs have been encouraging. The filter works very well on "structured" designs. Structured design, in this thesis, refers to a design in which there is both a minimal number of overlapping symbols and minimal global wiring. The filter works well where the locality of a symbol definition is preserved regardless of the placement of the symbol. Use of the filter has demonstrated the possibility of decreasing the program run time of design rule checking by a factor of five. This decrease is realized by introducing an awareness of hierarchy into design analysis tools.

The filter uses bounding boxes to represent all its data elements. The implementation of the filter places no restrictions on how the design is defined, but there is an underlying philosophy inherent in this data representation. The bounding box is a rectangle which bounds all of the element's internal points. A piece of geometry may contain any arbitrary angle. But the bounding box represents the entire piece of geometry. A bounding box also represents a symbol instance. A rectangle, versus a polygon, was found to be an adequate representation of a symbol.

Chapter II

A Description of the Approach to Design Rule Checking

There are several reasons for introducing the notion of hierarchy into a Design Rule Checker. An obvious reason is to save computer run time and memory usage by avoiding redundant design rule checks. A less obvious reason is to reduce the number of false errors generated by a DRC program. A design rule violation is flagged once, since each unique design rule violation is a unique interaction.

The DRC must examine two different types of situations in order to ensure completeness. The first situation is a symbol definition. The second situation is a given interaction between two elements. Redundant design rule checks are eliminated by not making the same check more than once.

The following terms are used throughout this thesis. A symbol is a collection of elements. The entire design definition is one example of a single symbol. An element is either a primitive element or an instance of a symbol. An instance of a symbol is a reference to a symbol with an associated transformation which provides placement and orientation information for the symbol. Primitive elements include pieces of geometry, such as wires, boxes or polygons, and any other elements that may not be subdivided.

The DRC checks both a symbol definition and an interaction between two elements once. A definition is checked the first time the algorithm encounters it in the design. This original check requires looking at all possible interactions of the symbol's internal elements. When the definition is instantiated, the surrounding elements' effect on the original definition is checked. A note is then made indicating that these interactions have been checked.

The DRC passes through two distinct phases while checking each symbol definition. The first phase finds and checks all unchecked definitions referenced by the current symbol. The second phase finds and checks all pairs of elements which interact.

2.1 Data Structures

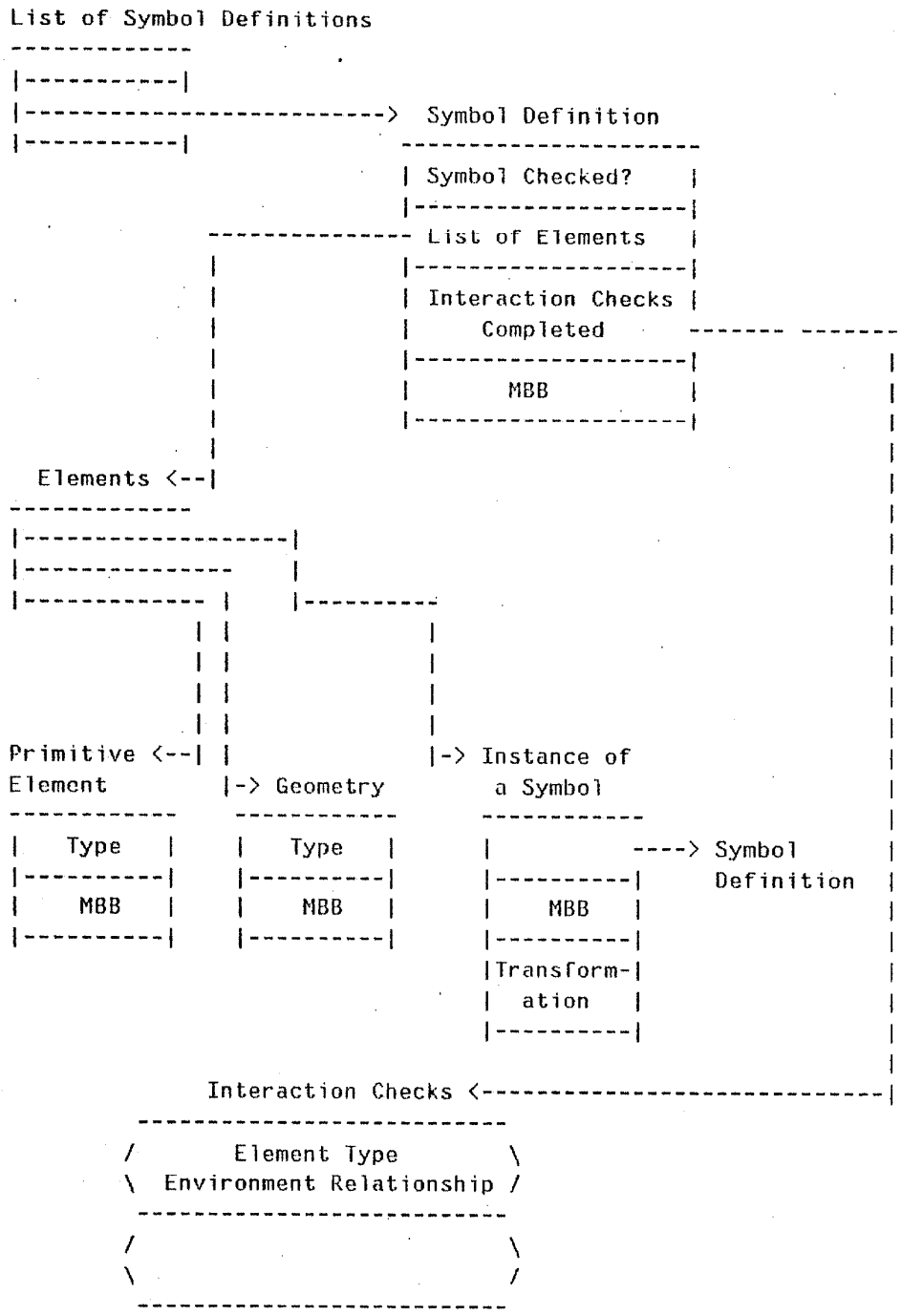
The symbol definitions are the basis of the DRC's data structure. Figure 2.1 summarizes the contents of the data structure. Design rule checking is performed on the top level symbol definition. Each lower level symbol is considered a separate design, and treated no differently than the top level symbol. Associated with each symbol definition are four pieces of information. These are: 1) a list of the symbol's elements, 2) a list of the symbol's completed interaction checks, 3) a boolean flag indicating whether or not the symbol has been checked, and 4) the symbol's Minimum Bounding Box (MBB).

An element may be one of two types. The first type is a symbol instance. A symbol instance has three components. These are: 1) a pointer to the associated symbol definition, 2) the transformation describing the placement of the definition, and 3) the transformed MBB of the symbol. The second type is a primitive element, which may be either a piece of geometry or a predefined primitive. Geometry is defined by three pieces of data. These are: 1) its type, e.g. wire, box or polygon, 2) it's MBB in the symbol's environment, and 3) a pointer to coordinate data. Primitive elements are defined by two pieces of information. These are: 1) a type and 2) a MBB. An example of a possible primitive element is a transistor.

The interaction list associated with each symbol definition describes all interaction checks previously performed involving this definition. This list is associated with the symbol definition and not the symbol instance. An interaction check is described by two pieces of information. These are: 1) the other element and 2) an environment relationship description.

The Boolean flag associated with the symbol is an indication of whether or not the symbol definition has been checked. A symbol is checked the first time it is encountered in the design. Checking a symbol involves checking the symbols it references.

Figure 2.1 - Data Structures



The minimum bounding box is one of the most important attributes of the entire data structure. It is important that all elements and definitions have a MBB. The MBB represents all elements until more detailed information is necessary. At the appropriate time, the MBB is peeled back to display the contents of the element.

The MBB represents all elements throughout the DRC process. The simplicity of performing operations between rectangles versus the difficulty of performing operations between polygons was the reason for this decision. It is relatively simple to detect two rectangles which overlap. This situation flags an interaction between two elements. It is possible, and highly probable, for two element's MBBs to overlap where there is not a true interaction. If this occurs, the algorithm displays the contents of one of the elements, and performs further interaction checks. Thus bounding box checks will occur more frequently than bounding polygon checks. However, the computational complexity of a bounding polygon check is far greater.

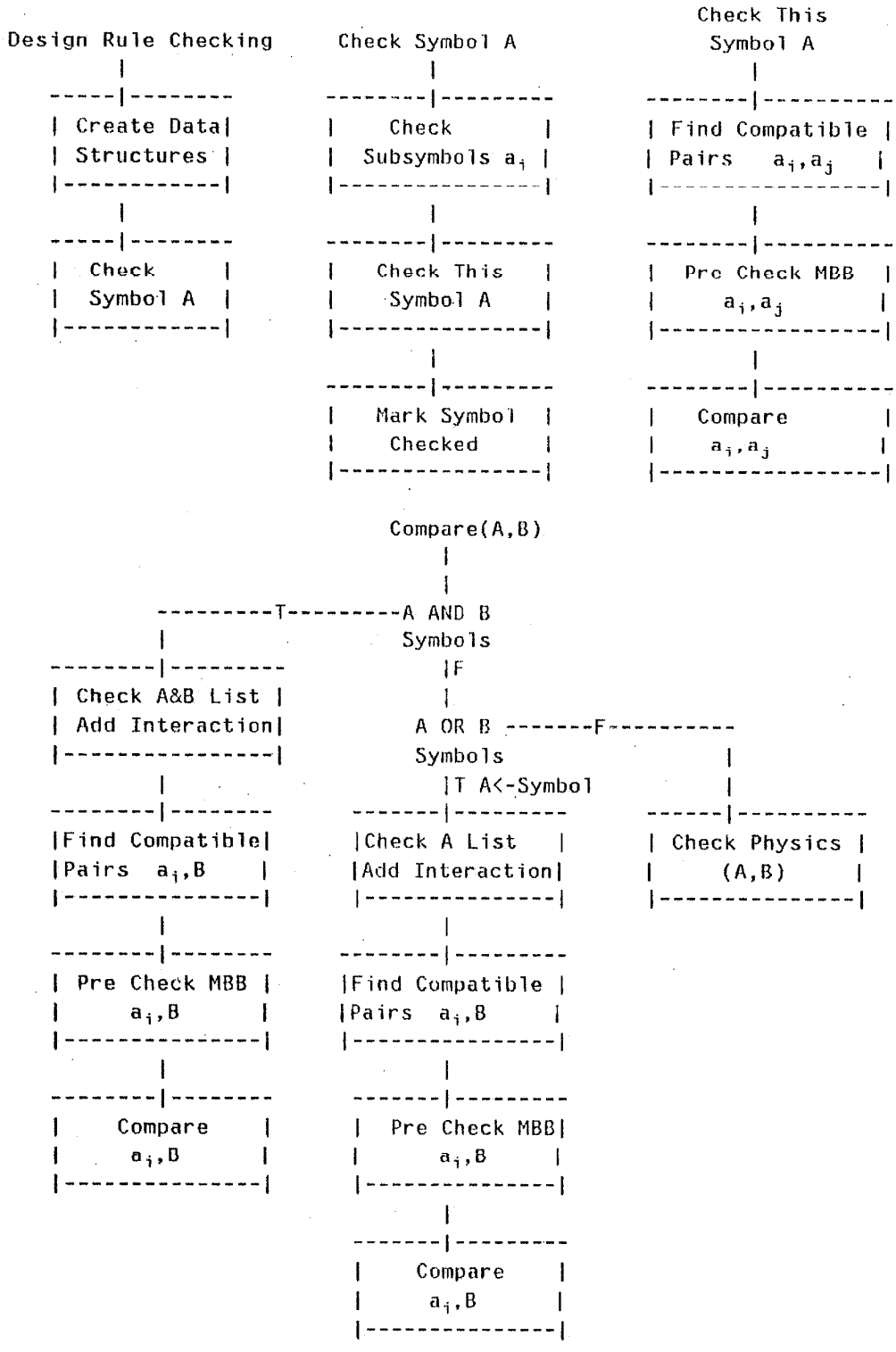
2.2 Description Of The Algorithm

The hierarchical DRC extracts all unique elemental interactions by operating at two distinct levels. First, it operates within a single symbol definition. The algorithm extracts pairs of elements inside the symbol which might interact. It then compares these elements to determine possible design rule violations. Second, the DRC operates within the area common to two symbol instances. The algorithm processes two separate symbol definitions each placed in a distinct environment. The DRC determines the area common to both environments. This area is where additional design rule violations might occur. Figure 2.2 is a block diagram illustrating the basic flow of the algorithm; a detailed description follows.

2.2.1 Design Rule Checking

The DRC takes an unchecked design definition, which is also a symbol definition, and checks it. The input to the DRC is a set of symbol definitions. The top level symbol is checked, and in the process the entire design is checked recursively. At each level in the hierarchy, the DRC treats symbol definitions as complete design descriptions. After checking a definition, the DRC determines how outside influences affect the definition.

Figure 2.2 - Algorithm Flow



2.2.2 Check Symbol

A symbol is checked by examining all interactions between the symbol's elements. All unchecked referenced symbols are recursively checked before interactions are examined. The DRC determines interactions, by looking at where the elements' bloated MBBs overlap. The MBB is bloated by the largest applicable spacing rule. The next step is to check all elemental interactions, and then mark the symbol checked.

2.2.3 Check This Symbol

The DRC determines what pairs of elements have a possibility of interference. All pairs of elements whose bloated MBB's overlap interact by definition. One obvious implementation of this search for pairs is to compare each element's MBB against the MBB's of all other elements. A sorting algorithm may be used to improve the efficiency of this operation. The intention is to find all the pairs of elements that may interact. It is then possible to do a quick bounding box check to see if the two elements' bloated bounding boxes do indeed interfere. If the bounding boxes overlap, then it becomes necessary to examine the two elements further to determine where violations might occur.

2.2.4 Compare

Given two elements whose bloated bounding boxes overlap, the compare operation determines if a design rule violation is present. The bounding box is the only information known about the element up to this point. Now more information is necessary. Each element is described in a separate environment. It is now time to find how the two elements' environments interact. There are three separate cases. In the first case, both elements are symbol instances. In the second case, one element is a symbol instance and the other a primitive element. In the third case, both elements are primitive elements.

When two symbol instances interact, it is necessary to find how each symbol definition is modified by the other. The DRC knows that each definition has been checked separately, but the interaction may introduce new violations. If this interaction has been checked previously, there is no need to check it again. Otherwise, since the MBB's have insufficient information to determine whether a

violation is present, more information is needed. The DRC introduces this information by peeling back the MBB of one of the symbols. Now there is a set of elements, previously defined as one of the symbols, and the original second symbol instance. A bounding box check determines which elements interact with the second symbol instance, and then the compare operation is repeated recursively.

When only one of the elements is a symbol, the DRC uses a simplied version of the symbol to symbol interaction test. As before, if the interaction has not previously been checked, the process must continue. This is done by peeling back the symbol's bounding box. The DRC introduces the symbol's elements, and the process begins again, recursively.

When the two elements are both primitive elements, the algorithm has reduced the problem to the traditional DRC problem. It is now possible to perform geometrical manipulations on the elements to determine if design rule violations exist.

2.2.5 Summary

The hierarchical design rule checker uses three techniques to reduce the number of design rule checks made to check a VLSI design. The first is to check a symbol definition once. This is done by checking all the symbol's elemental interactions. The second is to check an interaction between two elements once. This is done by recording the interactions that have previously been checked. The third is to use sorting of elements and bounding box checks to minimize the number of elemental comparisons.

Chapter III

SIMULA Implementation

A hierarchical design rule filter at Caltech implements the hierarchical design rule checker algorithm. The filter takes a design description in the Caltech Intermediate Form (CIF) and filters out elements that are redundant. The program produces a fully instantiated geometry file which can then be used as input to a traditional geometric Design Rule Checker (DRC). The program contains little specification of geometrical properties and uses a minimal specification of the design rules associated with the implementation technology.

The program is implemented in SIMULA[8], an object oriented language. In SIMULA, data structures are defined using a CLASS construct. A CLASS has two types of information. The first type is data, such as arrays, pointers, Booleans and so forth. The second type is procedures. Using these procedures, a data structure not only contains information about itself, but also performs modifications upon itself. The CLASS construct allows a clear association between the data declarations and the code which performs the data manipulations.

The design rule filter consists of four SIMULA modules defined exclusively for the filter, and two already existing modules. The first four modules are data structure descriptions, and the last two modules contain the code. A module dependency graph is shown in figure 3.1. The following is a description of the function of each of the modules:

Things - primitive data structure declarations[10].

Cif201 - the CIF2.0 parser[6].

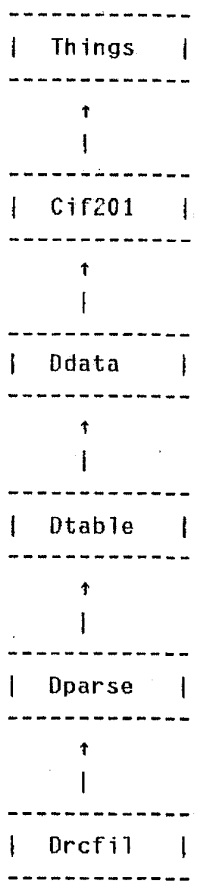


Figure 3.1
SIMULA Module Dependency

Ddata - the data structure declarations for the filter.

Dtable - the table declaration which will contain the information about what the filter has done.

Dparse - the module which performs the parsing of the design description, creating the necessary data structures.

Drcfil - The hierarchical design rule filter. This module performs the functions described in the DRC algorithm description.

3.1 Structured CIF2.0

CIF is a general purpose geometric language[3] which is not entirely suitable for the hierarchical design rule filter. CIF is a high level interchange language for pattern generators. There are many design tools available which generate CIF. Rather than design a new language, CIF was used as the input language to the filter. However, some restrictions had to be placed on the format of the CIF design description in order to make the design description compatible with the DRC approach.

The following is a brief description of each of the statements available in CIF2.0:

Symbol Manipulation

DS n - start of symbol definition n.
DF - end of symbol definition.
DD n - delete symbol definitions >=n.

Geometric Primitives

C n - call symbol n (create a symbol instance).
L - layer command - all following geometry declarations will be on this layer.
P - polygon
B - box
W - wire

R - round flash
digit - user extension (defined by a user).

The design rule filter does not support some of the statements in the CIF language. It does not support the round flash statement. It does not support the delete definition statement, since this statement disrupts the assumed integrity of a symbol definition throughout the design. The filter does not support arbitrary user extensions. Appendix I describes some user extensions used to provide further information to the filter. The filter ignores any other user extensions and issues a warning. Traditionally, CIF user extensions are used for labeling, so this usually is not a problem.

The filter restricts the structure of the CIF file in order to maintain the integrity of the algorithm at all levels of the hierarchy of the design. A call to the top level symbol is the only command allowed outside of a definition. This final call is a pointer to the top level symbol definition and therefore cannot have a transformation associated with it. It is easy to see that checking the design is equivalent to checking a symbol definition with this type of structure enforced. Symbols not referenced are not checked.

3.2 SIMULA Data Structures

There are three types of data structures. The first type is used for definitions which will be used many times. This type includes geometry and symbol definitions. The second type is a table which records what work the filter has previously completed. The interaction list is of this type. The third type is used for global structures, which once created are accessed only for information. This type includes the dictionary of symbols and the design rules.

3.2.1 Basic Definitions

"Geometry" is the initial CLASS of which all geometrical primitives are a subclass. A subclass inherits all the attributes of the original CLASS plus any additional attributes defined within the new CLASS. The following is the SIMULA declaration

of the CLASS geometry:

```
!*********************************************************************;
!   class GEOMETRY.  superclass of all elements used in the
!                    design description.
!*********************************************************************;


    thing CLASS geometry(geometric_name,clayer); VALUE geometric_name;
    TEXT geometric_name;INTEGER clayer;

    VIRTUAL:
    BOOLEAN PROCEDURE geom_okay;          !geometry meets design requirements?;
    PROCEDURE compute_min_bb;             !saves minimum bb of geometry;
    TEXT PROCEDURE cif_name;              !returns cif type;
    REF(rectangle)PROCEDURE bloat_mbb;    !returns mbb bloated with max rule;
    PROCEDURE write_cif;                  !writes geometry out as cif;
    REF(geometry)PROCEDURE transf_geom;   !returns the geometry transformed
                                           ! according to a specified transform;

    BEGIN
        REF(rectangle)min_bb;             !minimum bounding box;
        REAL hiy,hix,loy,lox;              !coordinates of bb;
        INTEGER lineno;                    !cif line number geometry occured on;
    END of geometry;
```

A new object of type geometry requires two parameters at the time of creation. The first parameter is an optional name associated with the geometry. The second parameter is the layer on which the geometry is to be placed. A number represents this layer, even though the CIF layer command contains a text string. The parser automatically translates a text layer specification into a unique number.

VIRTUAL procedures define all possible procedures of geometry. A virtual procedure declares that a sub-class of this class may have a procedure by this name. The SIMULA runtime system ties the actual procedure to the virtual procedure according to which subclass of geometry the object is. For example, geometry has four sub-classes (figure 3.2). Each of these subclasses has a procedure declaration corresponding to these virtual procedure declarations. The program may access these procedures from an object of type geometry without knowing which type of geometry the object is. The binding occurs at runtime.

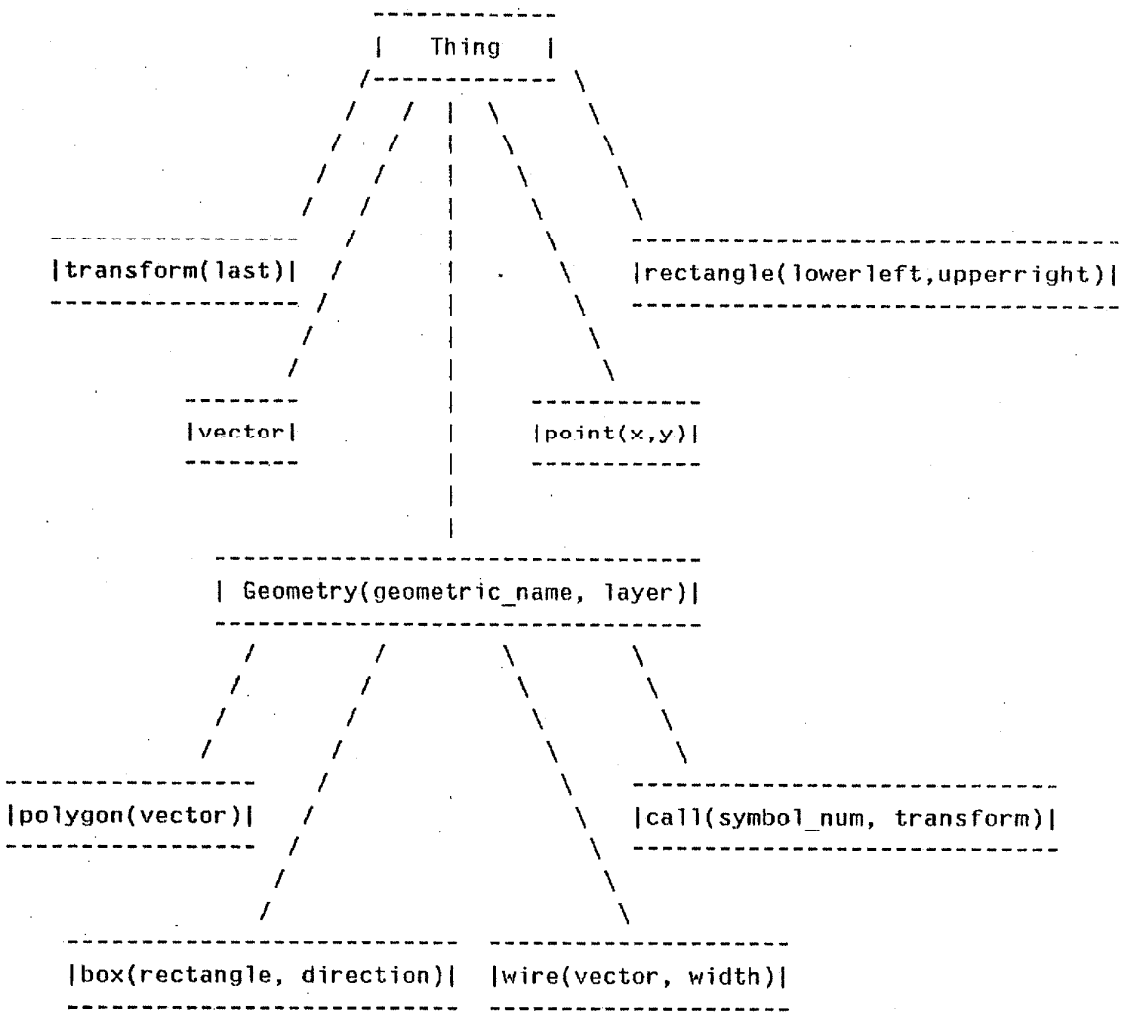


Figure 3.2 Primitive Data Dependency Graph

The pieces of geometry are the basic data structures of the design rule filter. The parser translates every geometric primitive defined in the CIF file to an object corresponding to the CIF type. There are four possible types of geometry. These are: 1) a box, 2) a wire, 3) a polygon or 4) a call.

The filter uses four lower level data structures as part of the definitions of its geometrical primitives. These lower level data structures are all subclasses of a superclass called a "thing". A thing can be anything. For the purposes of this discussion, a thing may be: 1) a vector, 2) a point, 3) a rectangle, or 4) a transform. A vector is a list of things, which is capable of growing dynamically as more things are added to the list. A point consists of an x and y coordinate. A rectangle consists of a lower left and an upper right point. A transform consists of six real numbers which define the rotational and translational components of a graphical transformation matrix[7]. The relationship of all the sub-classes to each other is shown in figure 3.2.

The filter creates and uses the appropriate data primitives to define the four geometrical primitives. These data primitives become attributes of the geometry. A box has two attributes: 1) a rectangle and 2) a direction point. The direction point denotes a vector which specifies the direction of the new positive x axis of the box after rotation. This vector defines the rotational components of a transform which can then be applied to the rectangle. A polygon has a single attribute which is a vector of points. There is an implied polygonal edge between each pair of consecutive points, and between the last point and the first point. A wire has two attributes, which are: 1) a path and 2) a width. The path is a vector of points. The width is constant along the path, and the endpoints are capped by an arc with a radius of half the width. A call has two attributes which are: 1) a symbol number and 2) a transform. The symbol number allows the program to access the associated symbol definition. The transform describes the placement of the symbol definition in the case of this symbol instance.

The "symbol" is the next important data structure used by the filter. The symbol CLASS definition contains all important attributes of a CIF symbol definition. The symbol definition has two parameters which are: 1) a symbol number and 2) the CIF scaling factor applied to all of the geometric points[3]. Internally the symbol has several other attributes. These are: 1) a checked flag, 2) three special symbol type

flags, 3) a list of its elements, 4) its name and 5) the symbol's minimum bounding box. The three special symbol type flags provide further information about the symbol to the filter and are described in detail in appendix I. The three type are: 1) primitive, 2) prechecked, and 3) leaf. The list of elements is a vector of geometrical primitives.

3.2.2 Table Definitions

The interaction list is a linear list kept in a disk file which represents an n by n upper diagonal matrix, where n is the number of symbol definitions defined in the CIF file. The filter remembers interactions only when they occur between two symbol instances. The interaction list contains n(n+1)/2 records, given n symbols. These are ordered as (1,1), (1,2),...,(1,n),(2,2),..(2,n),..(n-1,n),(n,n) on the disk. The (i,j) entry contains transformations describing the previously checked interactions between symbol i and symbol j. The filter accesses this record using the following formula:

x=symbol1 y:=symbol2 x<=y

n(n+1)/2=final record in the interaction file

n+1-x((n+1-x)+1)/2=offset from the final record to the start of x's list of interactions

y+1-x=y's offset into x's list of interactions

record needed= $n(n+1)/2-(n+1-x)(n+1-x+1)/2+(y+1-x)$

$= (n^2+n-(n+1-x)(n+2-x))/2+(y+1-x)$

$= (n+n-(n^2-2nx+3n-3x+x^2+2))/2+(y+1-x)$

$= (2nx-2n+3x-x^2-2)/2+(y+1-x)$

$= 2(nx-n+x(3-x)/2-1)/2+(y+1-x)$

$= nx-n+x(3-x)/2-1+(y+1-x)$

$= nx-n+x(1-x)/2+y$

Each record in the interaction list consists of a set of transformations. Each transformation represents a previously checked interaction between the two symbols used to index into the file. The transformation describes the placement of the second symbol instance with respect to the first symbol instance. A maximum of four entries are contained in each record. An overflow file contains all interactions greater than four that occur between two symbols. The overflow file is constructed using a linked list technique.

3.2.3 Global Definitions

The filter uses two global data structures. These are: 1) the dictionary of symbol definitions, and 2) the design rules. The symbol dictionary is a vector containing all symbol definitions. The design rules definition consists of several arrays containing information about the design rules of the process the current design is to be implemented in.

The design rule filter knows some minimal information about the process design rules. The filter knows enough to determine all potential design rule violations, but not enough to find the violations. The design rules describing all potential errors are expressed in terms of either minimum spacing between two layers or width of a single layer. The design rule data structure contains the information the filter needs.

The design rule definition consists of two one dimensional arrays, two matrices and one bloat factor. The first array holds the layer names used in the process. The second array contains the minimum width for each layer. The first matrix is a boolean array indicating whether a design rule exists between two layers. The second matrix is a real array indicating the minimum spacing between two layers. The final piece of information is a real bloat factor. This number indicates the largest minimum spacing between any two layers. The bloat factor is used for bloating symbol instance MBBs.

3.3 SIMULA Design Rule Filter Implementation

The design rule filter implements the algorithm described in the previous chapter with the following three differences. First, the module Check__Physics, the actual design rule checker, is implemented by writing the two pieces of geometry out to a file. Second, the interaction list structure is modified, primarily because of the memory limitations of the DEC-20. Third, the sorting algorithms used are a quicksort in the y direction and a bucket sort along the x axis.

The module Check__Physics determines whether two primitive elements represent a possible design rule violation. A primitive element, in the filter, is a piece of

geometry, i.e. a box, wire, or polygon. Check_Physics determines whether a rule exists between the two pieces of geometry. If a rule exists, then the module writes the pieces of geometry out to a CIF file. This file will eventually contain all the geometry needed to determine design rule violations.

The structure of the interaction list in the filter changed from the description in the previous chapter. First, the list contains only those interactions involving two symbol instances. Initially the intention was to remember all interactions in which a symbol was involved. The list initially included interactions involving a primitive element and a symbol instance. The primitive element to symbol instance interactions took up a great deal of memory and saved very few interaction checks. Second, the disk array interaction list construct previously described was developed. There is one entry for each symbol to symbol interaction rather than a separate entry associated with each symbol definition. This construct is possible since only symbol to symbol interactions are remembered.

A quicksort in the y direction, and a bucket sort along the x axis implements the sorting algorithm used for sorting a symbol's elements. A symbol's elements are initially sorted linearly according to the lower y coordinates of the elements' bounding boxes. The x axis is then divided up into the square root of n number of buckets, where n is the number of elements. John Bentley suggested this number of buckets [13]. Each element is placed in the applicable buckets starting with the first element in the sorted y list. The program then compares the element to the elements already contained in the buckets it is placed in. The filter removes elements from a bucket as soon as the high y of the bounding box is less than the next element's low y. This sorting approach greatly reduces the number of comparisons that are needed.

The following Algol-like description describes how the SIMULA version of the design rule filter is implemented:

```
!Top Level Definition;
BEGIN
    scan_cif_noting_all_symbol_definitions;
    check_symbol(top_level_symbol);
END;
```

```
FUNCTION check_symbol(symbol);
BEGIN
    IF prechecked(symbol) THEN note_bounding_box
    ELSE IF leaf(symbol) THEN BEGIN
        write_out_all_geometry(symbol);
        note_bounding_box;
    END ELSE BEGIN
        FOR each symbol_element DO BEGIN
            IF symbol(symbol_element) THEN BEGIN
                IF NOT checked
                    THEN check_symbol(symbol_element);
                    note_bounding_box;
            END ELSE note_bounding_box;
       END;
       check_interactions_between_symbol_elements(symbol);
       mark_symbol_checked;
    END;
END;


FUNCTION check_interactions_between_symbol_elements(symbol);
BEGIN
    sort_elements_in_x_and_y;
    FOR each a,b which_may_interact DO BEGIN
        IF
        bloated bounding box overlap(symbol_element(a), symbol_element(b))
        THEN BEGIN
            IF (symbol(symbol_element(a)) OR symbol(symbol_element(b)) THEN
                check_element_interactions
                (symbol_element(a),symbol_element(b))

            ELSE
                design_rule_check(symbol_element(a),symbol_element(b));
        END;
    END;
END;

FUNCTION check_element_interactions(element1, element2);
BEGIN

    IF symbol(element1) OR symbol(element2) THEN BEGIN
        element1:-which_element_is_symbol;
        elemnt2:-other_element;
         FOR each element1_element DO
```

```
            IF bloated_bounding_box_overlap(element1_element, element2)

            THEN  check_element_interactions(element1_element, element2)

    END ELSE design_rule_check(element1, element2);
END;

FUNCTION design_rule_check(geometry1, geometry2);
BEGIN
    IF design_rule_exists(geometry1, geometry2) THEN BEGIN
        write_out_cif(geometry1);
        write_out_cif(geometry2);
    END;
END;
```

The above description contains references to leaf and prechecked symbols. These are CIF user extensions supported by the filter. For further information, refer to Appendix I.

The limited address space available on the DEC-20 was the limiting factor in the development of the design rule filter. A large amount of memory management was written into the program in order to test the filter on any reasonable sized design. This was the most time consuming part of the project. The size of the design which the filter can accept is currently limited by the largest symbol definition in the design. The filter assumes that one symbol definition may be held in memory. The filter was eventually able to process some reasonably large designs. The next chapter describes the results of these experiments.

Chapter IV

Experimental Results

The hierarchical design rule filter has processed several NMOS designs, both Caltech student designs and Digital Equipment Corporation (DEC) designs. This chapter presents two Caltech designs, used as input to the filter. A pictorial representation shows the initial design's geometry and the remaining geometry after the filter processed the design. The first design shown is the design of a Least Recently Used (LRU) implementation of Content Addressable Memory (CAM), done by the author as a project for the LSI design course at Caltech. When the filter processed the design, it did not contain the pads or the routing to the pads included in the final project. The second Caltech design shown is a self-timed FIFO. Eric Barton[19] did this project as a student project for the LSI course at Caltech. A Floating Point Processor (FPP) experimental design, and the Memory Management Unit (MMU) for the LSI 11/23 were the two designs used as benchmarks at DEC. Pictures of these designs are not available, but the statistics of the filter's processing of the designs are contained in this chapter.

4.1 LRU CAM Example

The LRU CAM is a highly regular design. It consists of two arrays, four bits wide by eight rows deep, and two columns of logic between the arrays. Symbol instances overlap minimally. The filter took 8 minutes and 29 seconds to process the LRU CAM on a DEC-2060. The LRU CAM design is shown in Figure 4.1. A picture of the geometry remaining after the filter processed the design is shown in Figure 4.2. The CIF file of fully instantiated geometry used to create this plot could be used as input to a traditional DRC.

Table 4.3 is the statistics gathered by the filter for the LRU CAM example. The following is a detailed discussion of the various statistical quantities gathered. The titles to the various sections correspond to the labels of the filter's statistics. The statistics generated by the design rule filter are not totally self-explanatory therefore the following explanation is included. Table 4.6, which was generated by the filter, follows the same format.

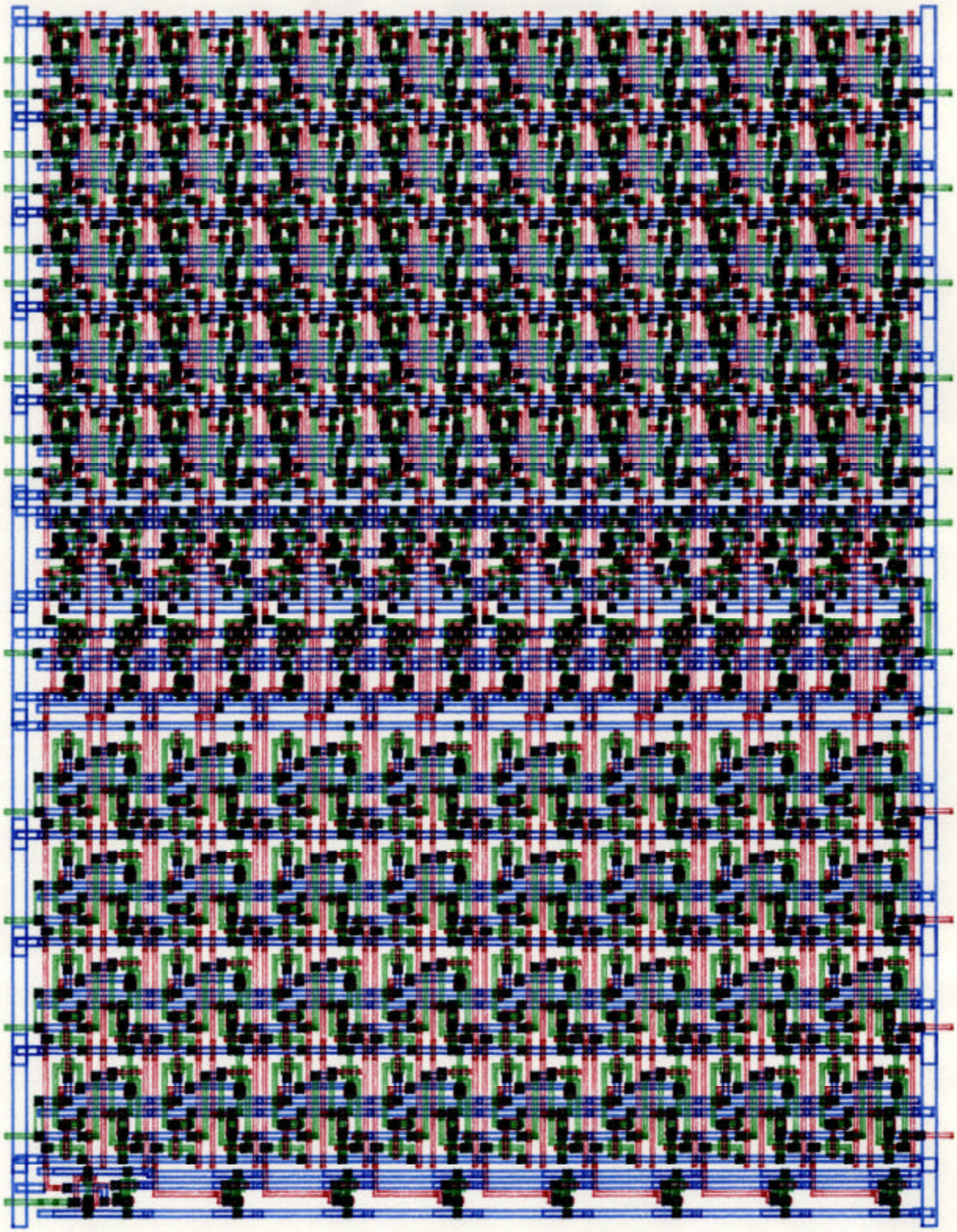


Figure 4.1 - LRU CAM

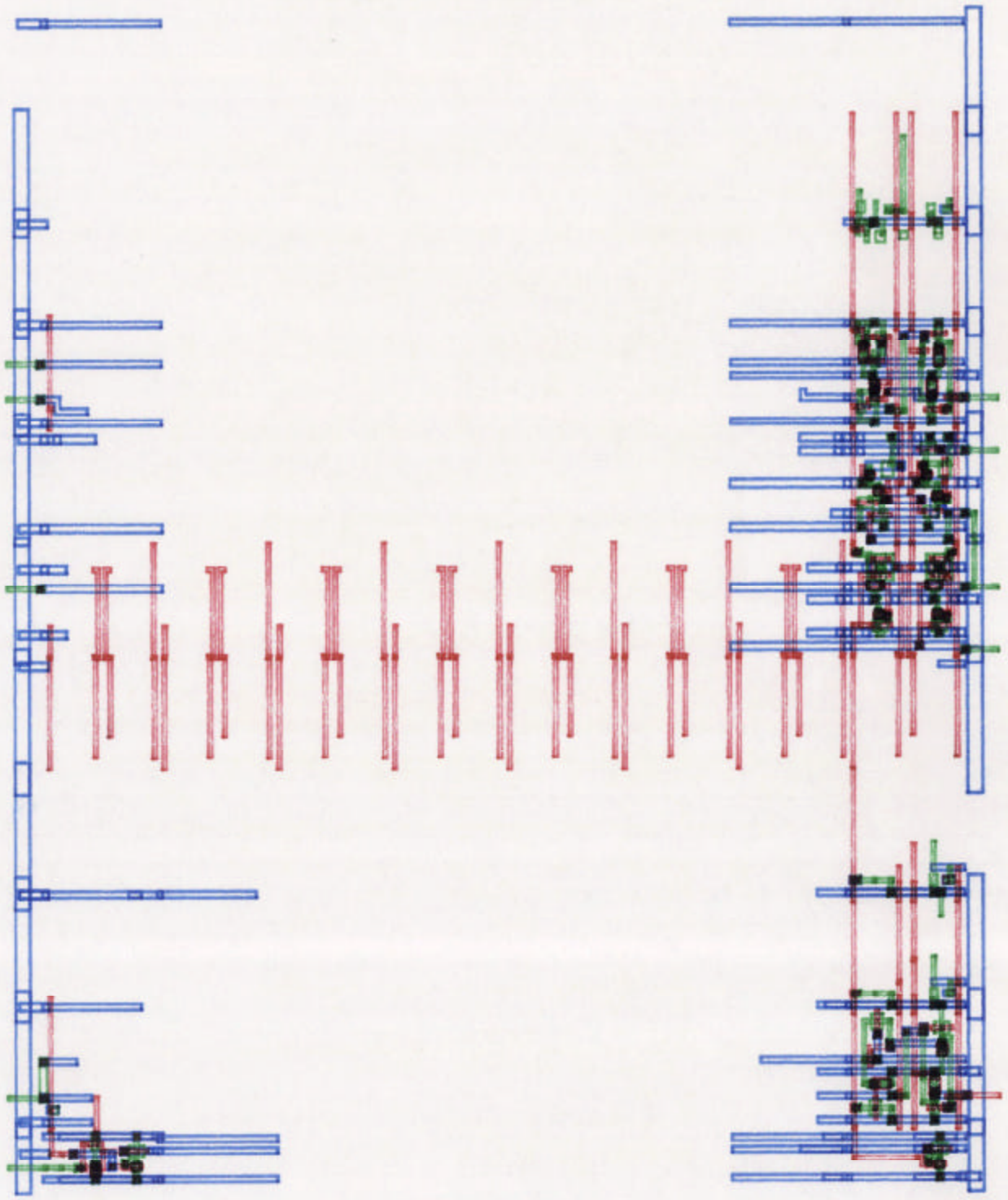


Figure 4.2 - LRU CAM Filtered Geometry

Summary of Input - gives an indication of the regularity of the design. There were 47 symbol definitions within the file. The number of polygons, wires, boxes and calls, in all the symbol definitions is 585. If the design was fully instantiated, the resulting file would contain 12,505 pieces of geometry, e.g. polygons, boxes or wires. The minimum number of elements in a symbol is 2. The maximum number of elements in a symbol is 81.

Check Symbol

Find Compatible Pairs - The filter compared 1,205 pairs of elements while checking the symbol definitions, using the quicksort and the bucket sort. Assuming that a worst case algorithm would make $n(n-1)/2$ number of comparisons for each symbol definition, where n is the number of elements contained in a definition, the total number of comparisons would have been 11,308. Therefore the quicksort and bucket sort saved the filter 10,103 comparisons.

Precheck BB - After the filter extracts a pair of elements, it compares their bloated bounding boxes to determine if an interaction really exists. In this example 924 of the 1,205 pairs revealed overlap between bounding boxes.

Bucket Sort Information - The filter placed a total of 530 elements in buckets. The average number of buckets that referenced an element was 2.37. The minimum number of buckets that referenced a given element was 1 and the maximum number was 9.

BB Discard - When two elements appear in the same bucket, the filter compares their bloated Bounding Boxes to determine whether they overlap. After the elements were sorted, the filter discarded 281 comparisons because their bounding boxes did not overlap in the x direction.

Compare - The routine which compares two elements interactions, shown in figure 2.2, received a total of 4,211 interaction checks. 435 of these interaction checks were between two symbols, 1,223 were between a symbol and a primitive element, and 2,553 checks were between two primitive elements.

2 Symbols

Interactions - 242 of the 435 interaction checks between two symbols could be discarded because they had previously occurred.

Pre Check BB - The remaining 193 symbol interactions required expansion, such that one of the symbols was broken apart into it's components parts. The program compared the element's of the symbols bounding boxes to the other symbol's bounding box. Only 667 of the 2,505 elements MBBs overlapped a symbol's BB.

BB Discard - Of the 1,838 of the above interactions whose BB's did not overlap, 1,292 did not overlap in the x direction, and 546 did not overlap in the y direction.

1 Symbol

Interactions - Since previous interactions are not kept for symbol to element interactions, the program examines all interactions further.

Pre Check BB - After the filter expands the symbol into it's component elements, of the 35,215 interactions which were checked, 32,595 interactions were discarded because the bounding boxes did not overlap.

BB Discard - Of the interactions which were discarded, 28,146 did not overlap in the x direction, and 4,449 did not overlap in the y direction.

Check Physics - There were a total of 2,553 elemental interactions. Of these interactions, 1,003 were trivial, i.e. there was no design rule between the two pieces of geometry.

Interaction List - The average length of the interaction list throughout the entire process was 1.72.

Table 4.3

LRU CAM Statistics

STATISTICS FOR DRC

Input File is GUTS.CIF

Created by the DRC on 1980-10-22 at 11:58:31

Summary of Input

47 Symbol Definitions with 585 Elements defined in Symbols
12505 flattened geometries
Min elements/symbol = 2 Max elements/symbol = 81

Check Symbol

Find Compatible Pairs	Compatible	Ignored	Total
	1205	10103	11308
Precheck BB	Intersected	Discarded	Total
	924	281	1205

Bucket Sort Information

Num Elements	Avg Num Buckets	Min Buckets	Max Buckets
530	2.37	1	9
BB Discard	X Discard	Y Discard	Total Discard
	281	0	281

Compare

2 Symbols	1 Symbol	Check Physics	Total
435	1223	2553	4211

2 Symbols

Interactions	Need to Expand	On List: Discard	Total
	193	242	435
Pre Check BB	Intersected	Discarded	Total
	667	1838	2505
BB Discard	X Discard	Y Discard	Total Discard
	1292	546	1838

1 Symbol

Interactions	Need to Expand	On List: Discard	Total
	1223	0	1223
Pre Check BB	Intersected	Discarded	Total
	2620	32595	35215
BB Discard	X Discard	Y Discard	Total Discard
	28146	4449	32595

Check Physics

Check Further	Connected	Trivial	Space OK
1550	0	1003	0

TOTAL
2553

Average Length of Interaction List
1.72

4.2 FIFO Example

The second Caltech design shown is a FIFO, which is also a regular design. There is minimal overlap in the inner part of the design but the design has an interesting structure. One symbol definition contains the pad definitions and a second symbol definition contains the internal part of the design. Therefore, the two symbol instances of these definitions entirely overlap. The algorithm checks both symbol definitions separately, and then checks the interactions between the two. This creates a large amount of overhead as the two symbol instances are compared. This design is small enough that the overhead does not drastically increase the runtime. However, a larger design, containing the same structure, would create disproportionally more overhead. The filter run time, while processing this design on a DEC 2060, was 25 minutes and 22 seconds. A comparison of the FIFO results to the LRU CAM results shows that the FIFO processing run time was longer whereas the design was smaller. This can be attributed to the structure described above.

Figure 4.4 shows the FIFO design. Figure 4.5 shows the fully instantiated geometry of the FIFO after the filter has finished processing it. Table 4.6 contains the statistics generated by the filter for the FIFO.

4.3 FPP and MMU Examples

The filter processed two DEC designs at Digital Equipment Corporation (DEC) in the summer of 1980. One design was an experimental floating point processor (FPP) and the second was the production chip LSI 11/23 Memory Management Unit (MMU). The MMU is considered irregular. A DEC team assembled the FPP chip using a design system which enforced strict rules of composition[15,16], i.e., a symbol's MBB was not allowed to overlap any other symbol's MBB.

Figure 4.7 contains the results of the comparison. We ran the test on a DEC-2020. The regularity factor is computed as the ratio of drawn transistors to placed transistors[20]. The regularity factor is one measure of the repetitiveness of a design and can give a clue to the possible effectiveness of the filter.

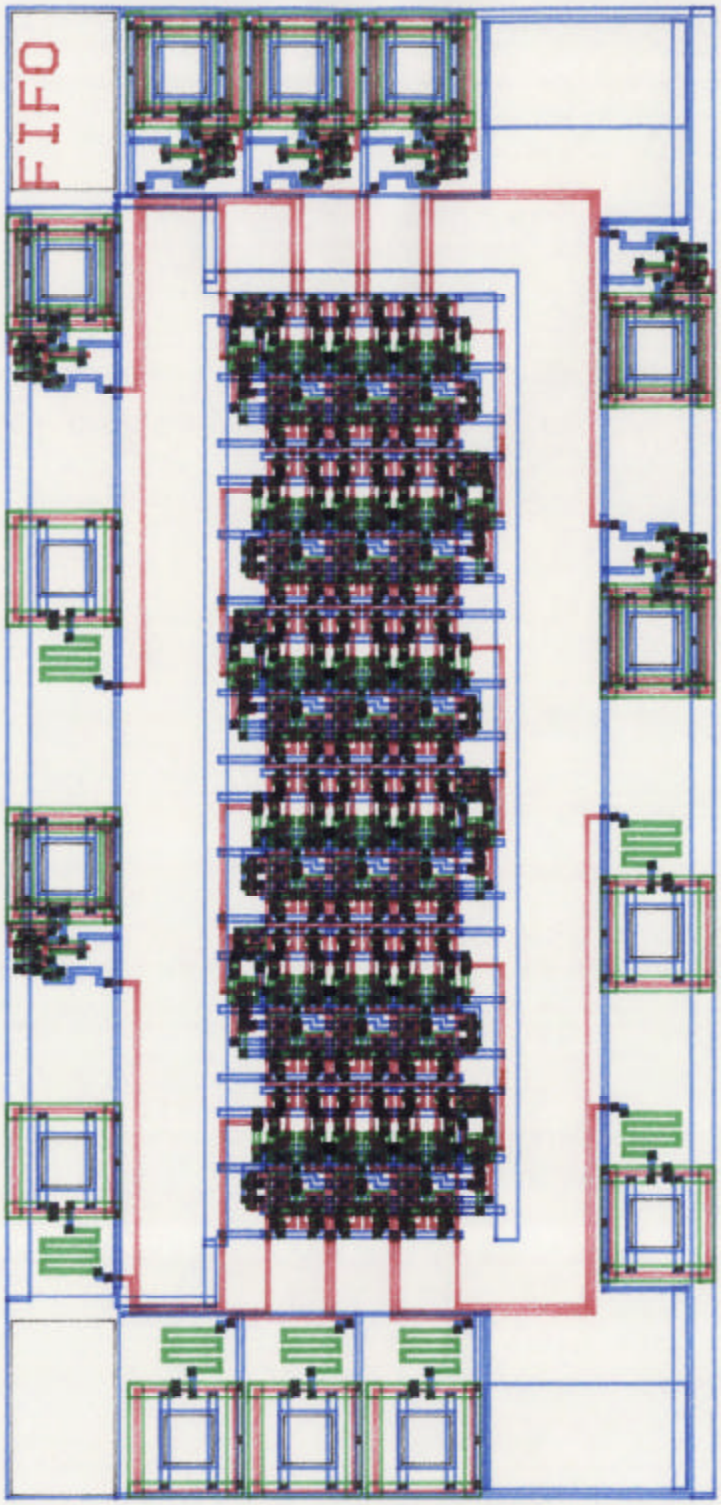


Figure 4.4 - FIFO

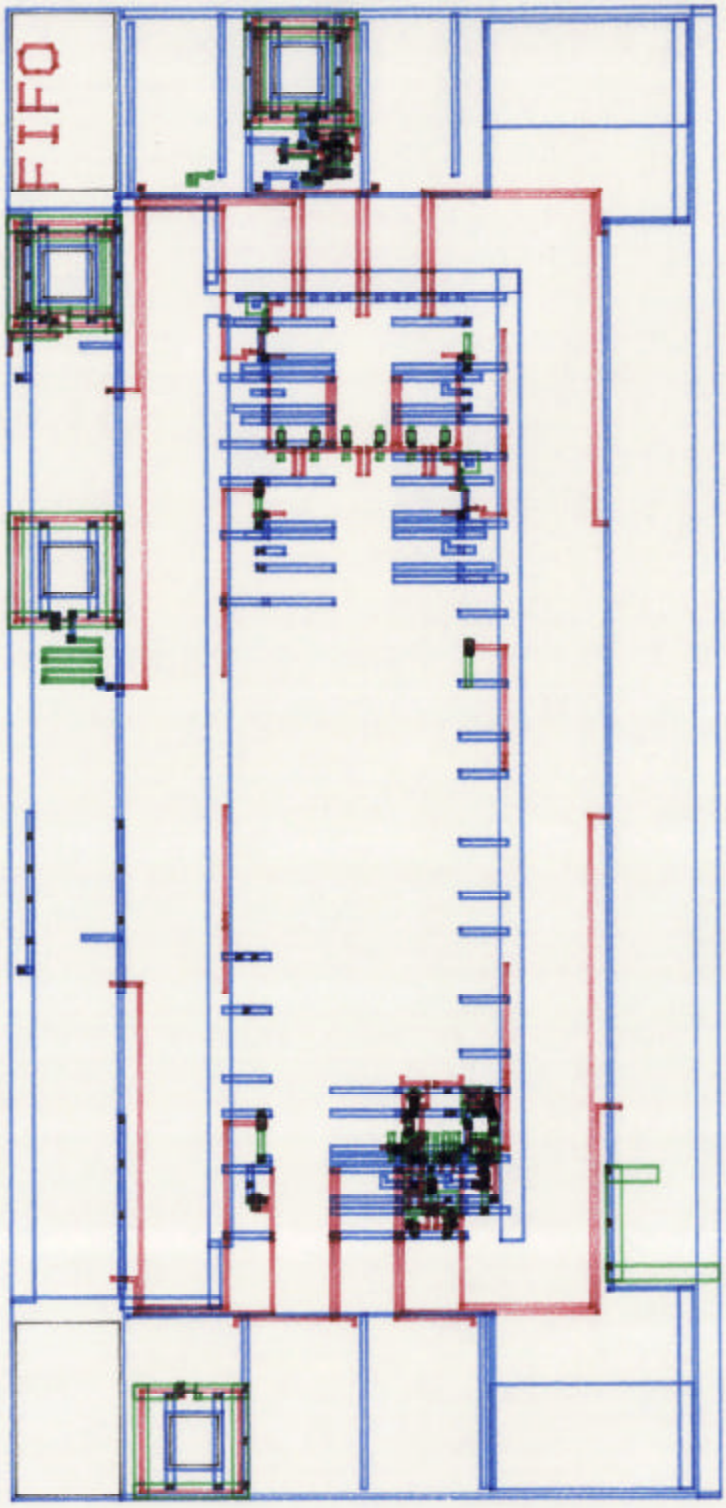


Figure 4.5 - FIFO Filtered Geometry

Table 4.6

FIFO Statistics

STATISTICS FOR DRC

Input File is FIFO.CIF

Created by the DRC on 1980-10-17 at 14:50:08

Summary of Input

42 Symbol Definitions with 546 Elements defined in Symbols
5990 flattened geometries
Min elements/symbol = 2 Max elements/symbol = 39

Check Symbol

Find Compatible Pairs	Compatible	Ignored	Total
	1215	3782	4997
Precheck BB	**Intersected**	**Discarded**	**Total**
	842	373	1215

Bucket Sort Information

Num Elements	Avg Num Buckets	Min Buckets	Max Buckets
410	1.76	1	6
BB Discard	**X Discard**	**Y Discard**	**Total Discard**
	373	0	373

Compare

2 Symbols	1 Symbol	Check Physics	Total
1399	6075	2564	10038

2 Symbols

Interactions	Need to Expand	On List: Discard	Total
	1341	58	1399
Pre Check BB	Intersected	Discarded	Total
	6484	961	7445
BB Discard	X Discard	Y Discard	Total Discard
	503	458	961

1 Symbol

Interactions	Need to Expand	On List: Discard	Total
	6075	0	6075
Pre Check BB	Intersected	Discarded	Total
	2712	204598	207310
BB Discard	X Discard	Y Discard	Total Discard
	192392	12206	204598

Check Physics

Check Further	Connected	Trivial	Space OK
1390	0	1174	0

TOTAL
2564

Average Length of Interaction List
4.06

There is a factor of 17.5 more errors, and a factor of 5.15 greater runtime between the nonfiltered versus the filtered designs. When a cell is repeated n times, and there is an error, the same error is called out n times. The filter helps to get rid of redundant error messages.

All errors caught in the nonfiltered version were also caught in the filtered version. Given that an error exists, the implication is that two pieces of geometry are too close together. Therefore, the bounding boxes of the elements, once the boxes have been bloated half the maximum spacing rule, must overlap. The filter will either write out these two pieces of geometry, or decide that the same elemental interaction has occured elsewhere. Thus the algorithm ensures that all spacing violations are detected by the filter without checking the fully instantiated geometry. Width errors are caught as soon as a piece of geometry is encountered in the CIF file.

FPP and MMU comparison:

	transistor count	Filter CPU time
FPP	(chip1) 22,560	1 hr. 44 min.
regularity factor 77K/1.04K=74 (24 without PLA/RAM/ROM)		
LSI 11/23 MMU	10, 367	>72 hr.
regularity factor 10.4K/3.5K=3 (1.3 without RAM)		

A portion of the FPP chip was run through the filter, and then the filtered geometry was given to the NCA DRC package. The design, without any filtering, was also given to the NCA DRC package.

The following are the results:

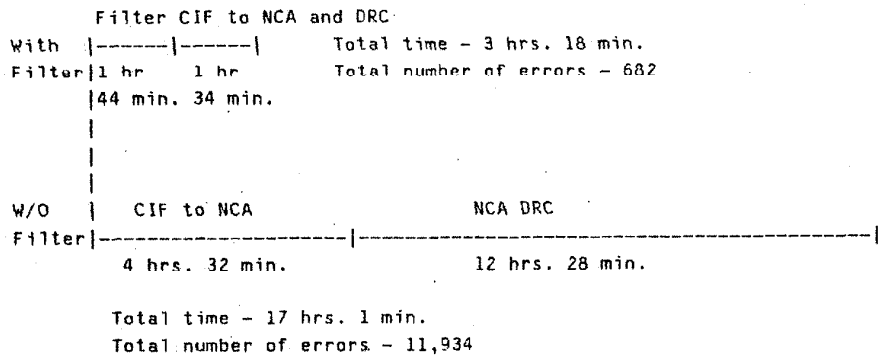


Figure 4.7 - FPP Results

Chapter V

Conclusions

5.1 Conclusions from the Hierarchical Filter

The Design Rule Filter processed numerous designs, and the results were encouraging. In all cases, when the program finished the processing, the remaining geometry was substantially less than the original fully instantiated geometry. The implication of this is that a design analysis tool may use the hierarchy of a VLSI design to greatly reduce the work performed.

The Design Rule Filter is a general purpose design analysis tool. It accepts any design, regardless of the structure. The experimental results show that any design could be processed given infinite time, but the design must contain a certain amount of structure to make the filtering process worthwhile. As an example, a standard DRC could process the MMU chip in less than twelve hours. The filter introduced additional checking, which would not have been encountered in a fully instantiated design description. With this additional overhead, the runtime of the filter was greater than 72 hours.

The amount of effort placed into writing memory management for the filter was way out of proportion to the effort. The addressing limitations on memory imposed by the SIMULA system on the DEC-20 were particularly severe when processing large designs. A hierarchical design rule checker becomes necessary only when designs become large. A great deal more investigative work could have been done if the program was written on a virtual memory system with a larger address space.

The design rule checker tried to avoid performing the same design rule check twice. This implies that the program must record whether the check has been performed previously. This also implies the program must decide what pieces of information are worth saving, such that the data will have a maximum possibility of being helpful in the future. The interactions between two symbol instances is one case where redundant checks were avoided. There are several cases where the

filter repeated an obviously redundant check. As the filter moves down the hierarchy, it must determine which symbol definitions to break apart. The filter may lose information by breaking apart one symbol prior to discovering the interaction is redundant. This phenomena occured in the case of the filtered version of the LRU CAM (figure 4.2). The poly wires running the length of the design remained after filtering. An additional piece of information worth saving, which could reduce the number of interaction checks, is the interactions between primitive elements and symbols.

If the filter knew more information about the design system used for a design, or if some restrictions were placed on the design, the filter could incorporate many optimizations. An example of such a design is the FPP chip designed at DEC. DEC people designed this experimental chip in a restricted design environment[15,16]. The filter processed the design rapidly. In general the filter could be optimized by exploiting the specifics of a particular design environment.

The design rule filter is a fruitful first step in learning about analysis tools which exploit the design hierarchy. Difficulties became more apparent as the filter was developed. By not placing restrictions on the design structure, we learned what restrictions would be the most beneficial both to the designer and to the design tools.

The hierarchical design rule filter, an analysis tool, is also helpful in the development of design synthesis tools. Many designs make use of machine generated geometry. Some examples are PLA generators[17] or Bristle Blocks[18]. The design rule checker could check the designs these programs generate. We need only check the program generated design once, just as a symbol is checked once (see Appendix I). Many designs combine the use of machine generated symbols with hand coded symbols. Even though the symbols are correct, the interactions involving machine generated geometry still need to be checked. The design rule filter supplies this capability.

5.2. Further Research

In addition to the Design Rule Checker, other analysis tools which exploit the hierarchy can be developed. These include: circuit extraction and verification, and net list extraction. Then, the extracted hierarchical net list can be verified against

a hierarchical circuit diagram system's net list. The basic approach is the same, though there are a different set of problems in each different type of analysis tool. Other analysis tools which could use a similar approach are functional verification systems, timing analyzers and electrical rules checkers.

The problem of performing the geometric manipulations on the primitive elements in order to do design rule checking needs to be investigated[11]. If we introduce the concept of primitive symbols to represent devices and other structural pieces of the design, then the design rule checker would "know" more about the designer's intent. With primitive symbols, the design rule checker could not only perform design rule checks, but also extract a hierarchical net list as it looks at the pairs of primitive elements. The net list information would eliminate many false errors.

The design rule filter is a first attempt at a hierarchical design analysis tool. VLSI designers need many tools which use or produce a hierarchy.

References

[1] H.S. Baird, "Fast algorithms for LSI Artwork Analysis", Proc of 14th D.A. Conf., 303-311, June 1977.

[2] I.E. Sutherland, C.A. Mead, "Microelectronics and Computer Science", Scientific American 237:3, September 1977, p. 210-228.

[3] R.Sproull, R. Lyon, S. Trimberger, "The Caltech Intermediate Form for LSI Layout description", Silicon Structures Project Display File #2686, California Institute of Technology. April 1979.

[4] C. Mead, L. Conway, Introduction to VLSI Systems, Addison-Wesley, Reading, Massachusetts, 1980.

[5] I.E. Sutherland, "The Polygon Package", Silicon Structures Project Display File #1438, California Institute of Technology, February 1978.

[6] G. Tarolli, J. Rowson, "A SIMULA CIF2.0 Parser", Silicon Structures Project Display File #2777, California Institute of Technology, April 1979.

[7] W. Newman and R. Sproull, Principles of Interactive Computer Graphics, McGraw-Hill, New York, 1973.

[8] G. Birtwistle, L. Enderin, M. Ohlin, J. Palme, DECSYSTEM-10 SIMULA Language Handbook, Swedish National Defense Research Institute and the Norwegian Computer Center (reports C8398, C8399 and C10045).

[9] E.E. Barton, I. Buchanan, "The Polygon Package", Computer Aided Design, 12:3, January 1980, p. 3-11.

[10] B. Locanthi, J. Rowson, "Things": Simula Program, Computer Science Department, California Institute of Technology, 1978.

[11] E.J. McGrath, T. Whitney, "Design Integrity and Immunity Checking: A New Look at Layout Verification and Design Rule Checking", Proc of 17th D.A. Conf., 263-268, June 1980.

[12] J. Rowson, Understanding Hierarchical Design, Ph.D. Thesis, California Institute of Technology, 1980.

[13] J.L. Bentley, D. Haken, R.W. Hon, "Statistics on VLSI Designs", Carnegie-Mellon Report, Carnegie-Mellon University, April 1980.

[14] T. Whitney, "Description of Telle - a CIF2.0 Processor", Silicon Structures Project Display File #4028, California Institute of Technology, October 1980.

[15] G. Tarolli, C. Peters Digital Equipment Corporation, private communication, August 1980.

[16] J.C. Mudge, Digital Equipment Corporation, private communication, September 1980.

[17] R. Ayres, "Silicon Compilation-A Hierarchical Use of PLAs", Proc of 16th D.A. Conf., 314-326, June 1979.

[18] D. Johannsen, "Bristle Blocks: A Silicon Compiler", Proc of 16th D.A. Conf., 310-313, June 1979.

[19] E.E. Barton, A Self Timed PDP8, Masters Thesis, California Institute of Technology, pending.

[20] G. Moore, "Are We Really Ready for VLSI?", Proceedings of Caltech Conference on VLSI, 3-14, January 1979.

[21] C.M. Baker, C. Terman, "Tools for Verifying Integrated Circuit Designs", Lambda, Vol. I, No. 3, 22-30, 1980.

[22] H.J. Simon, "The Architecture of Complexity", Procedure of the American Philosophical Society, vol. 106, no. 6, December 1962.

Appendix I

Hierarchical Design Rule Checker CIF User Extensions

The following section describes the CIF user extensions supported by the hierarchical design rule filter, and eventually the design rule checker. For each extension, the paper presents the syntax and then describes how special cases are handled.

In the syntax description, anything contained in curly brackets ({}) is optional, and anything outside the brackets is not. The order in which information is given is fixed.

A separator (sep) is defined to be:

```
sep   ::= blank sep | blank
blank ::= ' ' | ','
```

The DRC supports the following user extensions:

User Extension Purpose

7 - Special Purpose cell information for DRC

8 - Element names for boxes, wires, polygons, and calls

9 - Symbol Definition Names

A symbol contains user extension 7 in order to define special attributes of the symbol definition for the DRC. A designer may designate a symbol as having up to three different attributes. These attributes are not mutually exclusive, though a precedence exists which will be described later. All three have a separate indicator. If more than one attribute is present, the order in which they are declared is important.

The syntax is as follows:

7 {sep} {C {hecked} sep} {L {eaf} sep} {P {rimitive} sep type} ';'

The C indicates a symbol definition is not to be checked. The DRC assumes it has been previously checked. A prechecked symbol is one designated as design rule correct by the designer. The DRC assumes that any symbol referenced by a prechecked symbol is also correct in the given context. The implication is that if symbol A is referenced solely by the prechecked symbol B, then the definition of A will never be checked. But if symbol C also references A, then A's definition is examined. A prechecked symbol's interactions with other elements are still checked.

The L indicates a Leaf cell. This construct allows the designer to explicitly identify leaf cells[12], even if instances of other symbols are used in a leaf cell definition. The DRC replaces all instances of symbols in a leaf cell with the corresponding geometry. If the DRC is a hierarchical filter only, the instantiated geometry of the leaf cell is written out to the CIF output file. The filter does not spend time checking interactions between pieces of geometry in a leaf cell, but rather spends time more appropriately looking at the cell's interactions with other cells. The DRC notes cells as leaf cells automatically if they contain only geometry and references to primitive symbols.

A leaf cell is similar to a prechecked symbol in that calls to other symbols are not checked by the DRC. The difference is that the DRC fully instantiates and eventually checks a leaf cell whereas a prechecked symbol definition never gets checked. If both a Checked flag and a Leaf flag are present in a single definition, the Checked flag takes precedence.

The Hierarchical DRC requires that Leaf cells be design rule correct, e.g. a wire cannot be half the minimum width. The DRC must be able to find all real errors by checking a symbol definition once. This approach places a constraint on the method by which layouts are done. One of the reasons the concept of a Leaf cell was introduced is to allow the use of "construction cells". These cells are not necessarily design rule correct, but they can be used to build leaf cells which are design rule correct. A common example of this is in RAM cell designs. In the course of

designing a RAM cell, a quarter of the cell is designed, and then repeated four times to form the RAM cell. The RAM cell is the leaf cell.

The P indicates a Primitive Symbol. A primitive symbol is any cell that should not be broken down into pieces of geometry. All devices, and contact cuts are primitive symbols. Associated with each primitive symbol is a type. This type defines what the primitive symbol is supposed to be or do. These types could eventually have a one to one correspondence with a set of design rule checking procedures. An example of a primitive symbol is an enhancement mode transistor. The transistor could have a corresponding checking procedure which checks the transistor for design rule violations.

There are two implementations possible for the DRC. The first is a hierarchical filter, which extracts the minimum set of geometry that needs to be checked, in order to check the entire chip. This minimum set of geometry is written out to a CIF file. The leaf cell designation is used to indicate entire sets of geometry that need to go out to the file. The second DRC implementation is an extension of the filter, which incorporates true design rule checking. Primitive symbols are used in this second implementation to represent devices. If a leaf cell is declared, all levels of hierarchy present down to but not including primitive symbols are brought up to the current level.

A user extension 7 command may occur anywhere within the symbol definition. If a 7 is encountered outside a symbol definition, the DRC produces a CIF error message, and ignores the statement. Multiple commands may occur within a symbol, but if a field is changed, a warning is issued, and the field is updated. As an example, consider the following symbol definition:

```
DS 1;
B 40 40 0,0;
7 PRIMITIVE ET24;
7 CHECKED;
P 0,0 90,0 90,90 0,90;
7 PRIMITIVE DP44;
DF;
```

In this example, the DRC believes the symbol is a primitive and prechecked symbol. When the DRC encounters the third 7 command, a warning is issued, since the

primitive status is redefined, and the last definition is taken.

The user extension 8 command allows the designer to label pieces of geometry. When the DRC encounters an 8, the text between the 8 and the following semicolon, with the front and trailing blanks removed, is used to label the next piece of geometry encountered in the file. If the DRC encounters multiple 8's before a piece of geometry, a warning is issued, and the last label encountered is used. A 'DF;' command will cause any 8 that has not been assigned to a piece of geometry to be ignored. If this happens, a warning is issued. An 8 command can never occur outside a symbol definition, since nothing but the top level call may occur outside a symbol definition. The top level call cannot be labeled.

The user extension 9 allows designers to label a symbol definition. A 9 command may occur anywhere in the symbol definition. If the DRC encounters multiple 9's in a symbol, a warning is issued and the most recent label is used. The filter follows the same procedure described for user extension 8 while creating the label. If a 9 is encountered outside of a symbol definition, a CIF error is generated, and the statement is ignored.

CIF design files containing the above user extensions can easily be translated into files for software not supporting user extensions. Since the special cell designation indicates attributes of a symbol only, these statements may be deleted. The labeling commands may be deleted or replaced by a comment. These extensions to CIF are helpful, but do not in anyway destroy the integrity of the language as it exists today.

Appendix II

User Manual

Introduction

The hierarchical design rule filter is a program that takes a Caltech Intermediate Form (CIF2.0) file and filters out redundant geometrical information. Its purpose is to act as a preprocessor to a standard Design Rule Checker (DRC). If a design contains a great number of repetitious structures, the filter will eliminate much of the design's geometrical information, and create a filtered, fully instantiated design description which can then be design rule checked. If no design rule violations are found in this filtered version of the design, then there are no design rule violations in the original design description. This approach cuts down on DRC runtime and cost besides limiting the amount of computer memory that must be available to the DRC.

User's Manual

The hierarchical Design Rule Checker FILter (DRCFIL) is a program that accepts as input a CIF2.0 file, and extracts the minimum set of geometry that needs to be checked, according to a given set of design rules, in order to check the entire design. This set of geometry is then written out to another CIF2.0 file.

In order to run DRCFIL, the following information must be available:

1. input file - must be specified, if it is not the first file name on the command line, the filter will ask for the name. If an extension is not specified, the default extension is .CIF.

2. design rule file - must be specified, if it is not the second file name on the command line, DRCFIL will ask for the name. This file may be created using the program DESRUL. The default extension is .RUL. The specification file for the Mead and Conway rules is NMOS.RUL.

3. output file - may be specified as the third file on the command line, but the default is {input file}.CNV.

An example of using the filter on the CIF design file PLA.CIF would be:

@drcfil pla nmos

DRCFIL also has several modes available in which it prints information about what it is currently doing. The modes may be set by specifying a switch on the command line. These may be in any order, and may be placed among the above file names. By default, a given mode is off unless the switch is present. The switches are:

1. /d - Debug mode. DRCFIL prints all kinds of miscellaneous information about what it is doing. This mode is probably not useful to most users.

2. /f - Non filter mode. The default setting for DRCFIL is in filter mode. This means the program produces a file of geometry that needs to be checked. If this switch is set, the program will attempt to do some real design rule checks. Since the polygon package which it uses to do this has a great number of bugs, I would not suggest using this switch. The program will bomb.

3. /s - Statistics mode. If this switch is set, the program will produce a file with relevant (or irrelevant) statistics about the input file. This file will be {input file}.STA. Try it once, see what it comes up with. You might be interested.

4. /i - This is also a different kind of debugging switch. With this switch set, the program will print out all kinds of information about the interaction list, as the list is accessed.

5. /a - Disk accesses. This switch causes the program to issue a message each time a symbol definition is sent out to disk, or read in from disk. If you want to see how much thrashing is going on, this is a good switch to set.

DRCFIL performs many pairwise comparisons between pieces of geometry. Since a piece of geometry may need to be checked in several different contexts, the

filter does not always know whether a piece of geometry has already been written out to the file. Therefore a great deal of redundancy occurs in the output file. If the DRC package being used performs a merge of all pieces of geometry on the same layer, this does not present a problem. However if a plot is needed, you definitely want to get rid of the redundancy. There is a program to rid the file of it's geometric duplication, called CNVCIF. The following information is needed by the program:

1. input file - Once again, this must be available as the file name on the command line. If the name is not present, the program will ask for it's name. The default extension is .CNV.

2. output file - may be specified as the second file name on the command line. The default name is {input file}.DRC.

An example of the commands needed to Filter the CIF file PLA.CIF, and then create a file to plot is the following:

```
@drcfil pla nmos
@cnvcif pla
```

When the above commands have completed, the file to plot will be PLA.DRC.

DRCFIL has encountered certain limitations in the size of symbol definitions it is capable of accepting. Any design which is broken into symbols, where the number of elements contained in each symbol definition does not exceed 500, should not run into this limitation. In order to overcome this obstacle, rather than introducing new management into DRCFIL, I decided to write a program capable of modifying input files in order to meet the demands of the filter. This program is called TELLE[14].

Program Error and Warning Messages

Errors - Design Rule Filtering Aborted

a. Tried to place a polygon/wire/box on the ZZZ layer.

A piece of geometry was placed on an undefined layer. The piece of geometry needs to be placed inside a symbol definition in order to maintain consistency.

b. A DD statement is unsupported.

The filter is unable to handle DD statements. Since the designer's intention is not known, design rule checking is aborted.

c. Polygon/Wire/Box defined outside of a symbol definition.

The filter is unable to handle geometry defined outside of a symbol.

d. A Flash command is unsupported.

The filter is unable to handle CIF flashes.

e. Top Level Symbol is undefined.

There is a call to a symbol at the end of the file, but the symbol referenced is undefined.

f. No final call in CIF file, so cannot check.

The Filter needs to know which symbol is the entire design in order to check it. The file contained no final call.

Warnings - May affect the output of the filter

a. Polygon/Wire/Box defined with angles which are not a multiple of 45 degrees

A piece of geometry contained angles which were not 45 degrees. Since many tools do not support any angles non-45, the designer may want to change the piece of geometry.

b. Symbol n is called but not defined.

A call exists in the file to an undefined symbol or a symbol containing no geometry. The call is ignored.

c. Redefining symbol n "symbol name (if any)"

There are two definitions of the same symbol, The second definition is taken.

d. Cannot eject symbol from memory.

There are more data structures in memory than the filter thinks there should be, and the program is unable to get rid of any information. The possibility exists of the program blowing up with not enough memory. There isn't much to be done about this. Normally the filter will finish if the symbol definitions are small enough, even if this message appears. The symbol limit set by the is somewhat arbitrary and it leaves quite a bit of room to play with.

e. Different Scale factors in file. The first set is used on output.

CIF allows each symbol to have associated with it scale factors. The filter uses the same scale factors in it's output file, that the input file uses. If the input file contains multiple sets of scale factors, only the first set is used for all output symbols. This could conceivably lead to round off errors. Usually this is not a problem.

f. Layer command outside symbol definition - ignored.

A layer command existed outside a symbol definition. The command is ignored.

g. Transformation not supported on top level symbol, so ignored.

The top level symbol is always assumed to have no transformation associated with it. If a transformation is associated, it is ignored. The lack of a transformation should have no affect on whether or not the design is correct.

h. User Extension n is not supported.

The filter did not recognize the given user extension, so the extension is ignored.

i. User Extension command outside symbol definition - ignored

A user extension was encountered outside any symbol definition. The statement was ignored.

j. Redefining a geometry name in symbol n.

The user extension 8 which allows a user to name a piece of geometry was encountered twice before a piece of geometry was encountered.

k. Redefining the name for symbol n.

Duplicate user extensions 9 for naming symbols were present in the symbol definition.

l. Multiple use of prechecked status in symbol n.

A symbol was marked prechecked more than once.

m. Multiple use of leaf status in symbol n.

A symbol was marked a leaf symbol more than once.

n. Primitive symbols are currently not supported.

Primitive symbols are defined to be symbol definitions which are the basic building blocks of a design, such as transistors and contacts. The intention is to

eventually incorporate this idea into the filter. The filter recognizes the user extension which designates a primitive symbol, but it does not do anything with this knowledge. The statement is ignored.

o. Currently, arrays are not supported.

The filter recognizes the CIF user extension which defines an array, but does not yet understand arrays as a construct. The statement is ignored.

p. A geometry label was still present at the end of symbol n.

A geometry label was present and no geometry existed to use it. The statement is ignored.

Informational Messages

a. Throwing away a call to symbol n.

A call to an undefined symbol was encountered. This call is thrown out.

b. No valid CIF in symbol n, so not included.

If a symbol contains no elements, the definition is discarded.

c. Throwing out symbol n

Only a limited number of symbol definitions may be in memory at a given time. Symbol n is being put on disk.

d. Bringing in symbol n

A symbol definition is needed by the filter which is currently on disk. The definition is brought in.

Error Messages - Problem with the program - abort

a. Index into symbol n's array is too large.

The program attempted to access more geometry then was available in the definition.

b. Geometry asked for in symbol n where geometry doesn't exist.

The program attempted to access a piece of geometry that didn't exist.

c. The geometry being thrown to disk is of unknown type.

The program attempted to write to disk an unknown quantity.

d. A spacing rule was requested where none exists.

The program requested a spacing rule between two layers that were defined as having no rule.

e. A width rule was requested where none exists.

The program requested a width rule for an unknown layer.

f. Problem with interaction matrix.

Interactions between symbols are defined by a transformation matrix. The inverse transformation matrix is being computed, and the divisor is 0. Some kind of problem with the matrix computation.

g. Transformation stack ran out.

At all times the program keeps a transformation which tells it where in the design it currently is. The program thought there was one more transformation than there really was. A definite problem.

h. The information has been destroyed in the disk file.

The program creates a temporary file to take some of the information out of memory. This file has somehow been garbaged up.

i. Attempted to write out a null interaction.

The interaction list is kept out on disk. The program wants to add a null interaction to the list.