A New Approach to Perform Circuit Verification Using an O(n) Algorithm

Nunes, R.B, Anido, M.L and Oliveira, C.E.T.
Núcleo de Computação Eletrônica - Univ. Federal do Rio de Janeiro
e-mail: mlois@vidigal.nce.ufrj.br

Abstract

This paper presents a new approach to perform circuit verification, particularly Design Rule Checking (DRC) and Circuit Extraction, using a very fast data structure which supports an O(n) algorithm. A hierarchical DRC using such data structure is described by using an object oriented programming methodology. The paper discusses the main problems involved in geometric verification and presents a suitable language to describe the design rules. The paper also discusses the main algorithms employed to implement the DRC and analyses their applicability to circuit extraction. Finally, the hierarchical operation of the DRC is addressed.

Keywords: Circuit Verification, DRC, CAD, VLSI Tools.

1. Introduction

Teaching VLSI design is mostly a practical task that demands a considerable number of workstations, when large groups of students are to be taught. Undoubtedly, IBM-PC like workstations are (and will be in the near future) the cheapest and most easily available workstations. Consequently, providing VLSI CAD tools that can run on such machines, is one of the best choices to improve the dissemination of VLSI design. Moreover, the performance of high-end IBM-PC workstations approaches (or, in some cases, surpasses) the performance of well-known professional workstations. This CAD tool is part of the second generation of tools of the TEDMOS system [1][2].

Among the most efficient data structures to represent rectangular two-dimensional objects are Corner Stitching [3] and multidimensional binary trees (K-D trees)[4]. However, these data structures presuppose linear memory addressing of large amounts of memory.
Unfortunately, this is not the reality of IBM-PC-like machines. Besides the hardware limitations (poor segmentation system and small physical memory), the most used operational system (MS-DOS) cannot address more than 640Kb. Fortunately, this can be tackled with the latest versions of Borland's Pascal compiler (Borland Pascal 7.0®), which allows dealing with memory through DPMI (up to 16Mb) or by recompiling the system for protected mode operation. However, data segments are still limited to 64 Kbytes and the physical limit still exists. Due to these limitations, another data structure, more adequate to the computer hardware, had to be used. Windows compilers also have similar restrictions.

2. Circuit Representation

In order to achieve a compact and efficient data structure for the representation of circuit layout, several points have to be considered, such as efficiency, storage overhead, difficulty of implementation and hardware adequacy. Additionally, the data structure must support the operation of several, and distinct, CAD tools, such as a graphics editor, a DRC and a circuit extractor. The graphics editor of the TEDMOS system (CIRCO) uses a rectangle list for each layer. To accelerate editing operations (which are mostly region based), the rectangle list is organized as a Bucket structure, which presupposes a great deal of locality to quickly find rectangles to be inserted or deleted. In other words, the design area is subdivided into small regions that can be quickly dealt with.

Data structures like Corner Stitching[3], require many additional pointers associated to each rectangle and generate additional rectangles for free spaces and interlayer intersections. Our main objective is to be efficient with small memory overhead.

To achieve our main goal we use two sorted lists (sorted collections) of spans (Y-spans and X-spans). To store the layout and to support design rule checking, each data structure is a class called Layer. An object belonging to the Layer class corresponds to one layer of the layout. The concept of spans is the most important in this approach. The Layer class supports messages to add and delete rectangles, to make operations with other layers and all the necessary primitives to define design rule checking. A rectangle is defined as a set of coordinates X1, Y1, X2, Y2 with the following restriction: Y1<Y2 and X1<X2.

The Layer class represents the layout by using spans. Firstly, the layout is divided into Y-spans. A Y-span is an horizontal stripe of the layout represented by a pair of rectangular coordinates Y1-Y2. This provides a collection (sorted on Y2) of horizontal stripes. Each Y-span points to a collection of maximally horizontal X-spans, where each X-span in a Y-span represents a rectangle. This is represented by figure 1 below.
A maximally horizontal X-span means that there will be a minimum number of X-spans to represent one stripe of the layout and that two X-spans in the same Y-span will never touch each other, otherwise, they will be merged. There is a Y-span for each coordinate referred in each rectangle inserted in the layer (if the rectangle is not enclosed by another one). This is depicted by figure 1.

The layer class constitutes a sorted collection of Y-spans (another class) and each Y-span has a sorted collection of X-spans. This data representation is very compact and very efficient, providing O(n) algorithms for most DRC and circuit extraction operations. It is very fast to find a neighbour, because geometric neighbours are also neighbours in the Y-span and X-span collections, meaning that there is no need to search all X-span list for each Y-span element to find a neighbour. The number of Y-spans is limited to half the number of Y coordinates of the layout and one rectangle is represented as a single pair of X coordinates. If non-Manhattan geometry has to be used, trapezoids can be employed as the basic geometric elements. In this case, each X-span will contain two pairs or X co-ordinates (upper and lower sides of the trapezoid). Other more complicated geometric forms, such as circles or ellipses, can be dealt with by slicing them into rectangles or trapezoids. The main algorithms that deal with this data structure are described in section 4.

3. Design Rules Description Language

The language employed to describe the design rules, has to be simple enough to permit the easy coding of the design rules and, at the same time, has to be powerful enough to allow coding difficult design restrictions. The language used by the DRC described in this paper accepts logic expressions among layers (e.g.: C.(D+P), D.P-C.M, where D, P, C, M represent layout layers), as an operand of any of its design rules. These expressions may contain “and”, “or” or “minus” operators. The result of these expressions is a pseudo layer used within the design rule. To specify the design rules and facilitate the specification of design restrictions the DRC
uses the ES2 nomenclature. Basically, there are four types of design rules: Width, Spacing, Intersection and Cover. The grammar rules are shown below.

```
Rules → ε
    | Rule Rules
Rule → widthRule
    | spacingRule
    | intersectRule
    | coverRule
    | IDENTIFIER "=" exp
widthRule → WIDTH exp relop INTEGER
spacingRule
    → SPACING exp relop INTEGER
    | SPACING exp, exp relop INTEGER
intersectRule → INTERSEC exp, exp intersectTests
coverRule → COVER exp, exp relop INTEGER
intersectTests
    → intersectTest
    | intersectTest intersectTests
intersectTest → testType relop INTEGER
testType → MARGIN
    | COIN
    | OVERLAP
exp → termo
    | termo "+" exp
    | termo "-" exp
termo → fator
    | fator "*" termo
fator → IDENTIFIER
    | "(" exp ")"
relop → "<" | "<=" | "="
```

4. Design Rule Checker Operation

The Layer class must support a generic set of operations on a layer or among layers (or pseudo layers). Each design rule is translated into a sequence of messages to one or more Layer objects (instances). The Layer class supports operations on rectangles, among layers and other DRC basic operations, as explained below.

4.1 Operations on Rectangles

- Inserting a rectangle into a layer. This operation inserts the rectangle into the data structure by creating additional Y-spans or by splitting existing ones. This operation is performed in four steps and is illustrated by figure 2.

  Firstly, find the first Y-span that intersects the rectangle (Y-spans are ordered according to Y2). Secondly, insert the upper part of the rectangle (bottom of Y-span) by creating a new Y-span or by splitting the present Y-span. Thirdly, insert the middle of the rectangle in to the next Y-spans. Finally, insert the lower part of the rectangle by creating a new Y-span or by splitting the last Y-span.

  All steps are performed by shrinking the rectangle and if the rectangle is empty the operation is finished. All rectangle insertion into Y-spans is accomplished by considering rectangles maximally horizontal.
- **Deleting** a rectangle from a layer. This operation deletes the rectangular region defined in the data structure by creating additional Y-spans or by splitting existing ones. Similarly to insertion, this operation can be subdivided in four steps and is illustrated by figure 3. Firstly, find the first Y-span that intersects the rectangle. Secondly, delete the upper part of the rectangle (Y-span bottom), splitting the present Y-span and then deleting the space of the bottom part. Thirdly, delete the middle part of the rectangle in the next Y-spans. Finally, delete the lower part of the rectangle by splitting the last Y-span and deleting the space defined in the rectangle.
Qualifying the presence (intersection) of a rectangular region in a layer. The result of this operation is an identification of Null presence, Partial presence or Total presence, or in other words it means if the rectangular region is totally outside the layer, partially inside the layer or totally inside the layer. The corresponding algorithm is presented below. The Insertion and Deletion algorithms are similar to this one and for this reason they will not be presented.

Function TLayer.qualifyPresence ( R : TRectangle ) :TPresenceStatus

If Y-SpansCollection search( R.Y1, Index ) Then
    Inc(Index); { If R.Y1 exist an Y-Spans Y2 get the next span }

If does not exist Y-span on this position or
the actual Y-span.Y1 < R.Y2 Then
    return( Null );

    { Check Upper Part }
If R.Y1 < Y-span.Y1 Then
    Presence is Not Total
    R.Y1 = Y-span.Y1 { Shrink R };

    { Check middle Part}
Case Y-span.X-spans.qualifyXPresence (R.X1, R.X2) of
    Null : Presence is Not Total
    Complete : Presence is Not Null
    Partial : return( Partial ) { Presence is Partial };

    R.Y1 = Y-span.Y2 { Shrink R };

    { Check Lower Part }
While R.Y1 < R.Y2 do { rectangle still exists ? }
    Get Next Y-span
        if does not exists next Y-span or
        Y-span.Y1 > R.Y2 Then
        Presence is Not Total
        Break { Go out of while }

    Case Y-span.X-spans.qualifyXPresence (R.X1, R.X2) of
        Null : Presence is Not Total
        Complete : Presence is Not Null
        Partial : return( Partial ) { Presence is Partial };

    R.Y1 = Y-span.Y2 { Shrink R };

    If Presence is Not Total and Not Null Then
    return( Partial ) { Presence is Partial };

    ; { End while }

    If Presence if Not Null Then
        return (Null)
    Else
        If Presence is Not Complete Then
        return ( Total )
        Else
        return ( Partial );
End of qualifyPresence;
• For each rectangle in a layer apply the following procedure. Create a temporary rectangle R. For each Y-span make R.Y1=Y-span.Y1 and R.Y2=Y-span.Y2. For each X-span in Y-span make R.X1=X-span.X1 and R.X2=X-span.X2, call such procedure passing R as parameter.

This procedure can be used to plot the layer or to perform a very simple operation on rectangles.

• Compacting a layer. Compact the representation of a layer by joining equalsY-spans.

For each Y-span check if Actual.X-spans are equal to Next.X-spans. If equal make Actual = union (Actual and Next) and repeat the verification for Actual.

4.2 Operations Among Layers

This section addresses the main operations among layers and describes the logical AND (.) operation in detail.

• Logical Or (+) - Add (include) the rectangles of a certain layer (passed as a parameter) to another layer. For each rectangle in the first layer insert it into the second layer.

• Minus (-) - Delete the rectangles of a certain layer (passed as a parameter) from another layer. For each rectangle in the first layer delete that rectangular region from the second layer.

• Logical And (.) - A certain layer is obtained from its intersection with another layer, which is passed as a parameter. Below we describe the procedure to perform a logical AND of two layers.

Procedure TLAYER.AndLayer(aLayer) Create a New Layer intersecLayer;

    aIndex := 0;
mIndex := 0;

    While mIndex < Count and (aIndex < aLayer.Count) do
        mY = Y-Spans.At(mIndex);
        aY = aLayer.Y-Spans.At(aIndex);

        Y1 := Max(mY.Y1, aY.Y1);
        Y2 := Min(mY.Y2, aY.Y2);

        If Y2 > Y1 Then { They Intersect }
            intersecLayer.Add Y-Span(Y1, Y2) with
X-Spans = IntersecX-Spans (mY.X-Spans, aY.X-Spans);

{ Get Next Y-Span }
If mY.Y2 > aY.Y2 Then
   Inc(aIndex) { Drop Y-Span with lower Y2 }
Else
   Inc(mIndex);
Dispose Self; { Dispose layer }
Self = itersecLayer; { New layer }
End AndLayer;

4.3 DRC Operations

In this section we describe the main DRC operations, getting into more detail of the Width Rule operation.

- **WidthRule (minimum, anErrorFunction)** - This procedure checks the width rule of a layer and calls an error function for each part of the layer that has width less than minimum.

Procedure TLayer.widthLessThan (minimum, anErrorFunction)

Procedure X-SpanTest( X-Span : PSpanX );
R.X1 = X-Span.X1
R.X2 = X-Span.X2

If (R.X1-R.X2) < minimum Then { maximally horizontal}
   anErrorFunction( R )
Else
   { In Vertical direction we must check other rects }
   Size = R.Y1 - R.Y2;

   While Size < minimum do
      Get upper rectangle that covers present completely
      if none exists break
      else inc (Size, Upper.Y2-Upper.Y1);

   While Size < minimum do
      Get bottom rectangle that covers present completely
      if none exists break
      else inc (Size, bottom.Y2-bottom.Y1);

   If Size < minimum Then
      anErrorFunction( R );
return;

{ Correct rectangles may fail the } { rule in contact with others }
If exists upper Y-Span
   Test both X-spans in the intersection regions to
   check regions of contact
If exists bottom Y-Span
   Test both X-span in the intersection regions to
   check regions of contact
End of X-SpanTest

Begin of width

For Index := 0 To Count-1 do
  Y-Span = Y-Spans.At(Index);
  R.Y1 = Y-Span.Y1;
  R.Y2 = Y-Span.Y2;
  Y-Span.X-Spans.ForEach(@X-SpanTest);

End of width

- **SpacingRule (minimum, anErrorFunction)** - This operation checks the spacing rule among the rectangles of a layer. It calls an error function for each rectangle of the layer that exceeds the minimum distance to another.

  For each rectangle in a layer get all neighbour rectangles. If neighbours do not fail the test, no other rectangle will. If neighbours fail the spacing rule, check if they are connected in some way. If not, indicate the error.

  Note: the connection test has to be done with some restrictions. Connections are only searched in the neighborhood of the rectangle. An extensive test is not necessary and can be very inefficient. A better approach is to make the DRC obtain the connection information from the circuit extractor.

- **SpacingRule(layer, minimum, anErrorFunction)** - This operation checks the spacing rule among rectangles of layer1 and layer2. It calls an error function for each layer rectangle that exceeds the minimum distance to another. This test is similar to the previous one, although there are some differences. Neighbourhood checking in a single layer is accomplished only in two directions (left to right and top to bottom). When testing spacing among different layers all four directions have to be checked.

- **IntersectionRules (layer, Margin, Coincidence, Overlap, anErrorFunction)** - This operation checks the intersection rules which consist of Margin, Coincidence and Overlap tests.

5. Hierarchical and Incremental Operation

Physical layout typically involves creating hierarchical designs by building larger cell blocks from smaller ones. There are several advantages in using a hierarchical DRC such as saving time that would otherwise be spent verifying several copies of the same cell. The DRC described in this paper has three modes of operation: Flat, Hierarchical and Incremental. **Flat operation** means that all hierarchies are removed, that is, the layout is exploded into a single and probably large cell. **Hierarchical operation** is performed by traversing the hierarchy of
cells, when necessary. When a cell instance is too close or intersects another cell, or a wire, it is necessary to traverse the hierarchy of cells to check for any design rule violations. In this mode of operation all master cells are tested. Any errors found are stored into the cells to be seen at any cell instance. The most important thing is that the DRC is executed only once for each master cell. In the Incremental Mode of Operation only the cells that had not been verified previously are verified. To support a consistency check, each cell includes a timestamp information. Thus, master cells have to be older than cells containing instances to them. Additionally, each cell has a flag indicating if it had been checked previously and if it is free of errors.

6. Conclusions

This paper presented a new approach to perform circuit verification, particularly Design Rule Checking, using a very fast data structure which supports an O(n) algorithm. The paper focused on the description of the main operations of a DRC based on such data structure, using an object oriented programming approach.

The data structure employed is based upon two sorted lists of spans (Y-spans and X-spans) and is highly efficient for graphics editing, DRC checking and circuit extraction operations because most tasks can be performed using O(n) algorithms. Additionally, this data structure can also support non-Manhattan geometry, because a trapezoid (instead of a rectangle) can be used as the basic geometric unit.

The main advantage of this data structure over Corner Stitching is that it does not presuppose linear memory addressing and also requires less memory.

7. References


[02] Borges, Jos{\textdegree} A. e Schmitz, Eber A.; "Projetos de circuitos integrados" - Livros T{\textacute}cnicos e Cient{	extacute}ficos Editora Ltda. 1990

