An O (N log N) Algorithm For Boolean Mask Operations

Ulrich Lauther

SIEMENS AG, Munich, FRG

Abstract

A new algorithm is presented which calculates Boolean combinations (AND, OR, EXOR, AND NOT) between two layers of an integrated circuit layout. Input and output of the algorithm is an edgebased description of the set of polygons which represent the artwork. The algorithm has O (N log N) time and O ($\sqrt{N}$) space complexity, i.e. it is faster than previously published methods. Moreover, we believe that it is easier to understand and to implement than the previously leading method in the field.

1. Introduction

Calculation of Boolean mask combinations (AND, OR, EXOR, AND NOT) between different layers of an integrated circuit is a basic procedure in designrule checking, connectivity checking and device recognition from the layout (see [1,2,5,8] for applications). This task seriously stresses the computational resources (cpu-time and storage) of today's computers: runtimes in the range of tens of hours are often reported. It is sometimes argued that increasing speed of computers and dropping cost of main memory will solve these problems, but this argument does not hold because the size of layouts to be analysed increases at least as fast as our computational power: We have to use today's (if not yesterday's) hardware to develop tomorrow's computers. The exploitation of hierarchical design methodology will not solve this problem either unless a large fraction of the layout is of a repetitive nature. Therefore, instead of calling for the sledgehammer of superfast computers, we should look for fast algorithms with modest memory needs, i.e. runtime should grow linearly or near linearly with size of input and only a small (sublinearly growing) fraction of the layout should be held in main memory. An algorithm with these characteristics will be presented in this paper.

2. Model of computation

To evaluate previous and our own work we make some assumptions about the computational environment and the data to be handled:
We assume a general purpose computer with a fairly limited main memory allowing fast random access and a practically unlimited peripheral storage, for which fast access is possible only in a serial manner. Typically, it is not possible to keep the whole layout to be processed in main memory. As part of the programming environment we assume an external sort package which works with time comlexity O (N log N) if N denotes input size, as it is state of the art [15].
The layout to be processed is stored as a set of polygons per layer which specify the opaque regions. These regions are bounded by straight lines (the polygon edges) which may have any slope. (We do not restrict ourself to orthogonal or $n \cdot 45^o$ artwork.) Polygons may contain "doughnut holes" either by selfoverlapping or by explicit description of these windows. Polygons (and if appropriate) windows are described by a sequence of points defining the polygon edges. Polygon edges are oriented in a way that the opaque part of the layer lies always on the right side of the edge. The total number of edges in the completely intersected edge set (to be defined below) is denoted by N.
We use the usual O - notation to describe time and space complexity of algorithms: a function g (n) is said to be O (f (n)) if exist constants c, n_0 such

0146-7123/81/0000-0555$00.75

that $g(n) < c \cdot f(n)$ for all $n > n_o$.

To simplify complexity considerations we assume that for a given technology the number S of different x-coordinates is $O(\sqrt{N})$ and that the average number H of polygon edges crossing a vertical cut through the layout is also $O(\sqrt{N})$. To justify these assumptions let us make the following experiment:
Let a given chip contain N edges in the completely intersected edge set. If we now double the chip in x-direction then the numbers N and S will also double but the number H will not change. If we double again - now in y-direction - then N and H will double, but S will (asymtotically) stay constant due to the gridded structure of layouts. (For a random districution of points over the length of the chip we would get $S = S_o \ (1 - e^{-2N/S_o})$, S_o beeing the maximal possible value of S for a given chip length. In real layouts we may expect an even faster saturation of S.)

To summarize, we increased N by a factor of 4 and both S and H were doubled; so we may assume S and H to be $O(\sqrt{N})$.

3. Previous work

Basically two different approaches are generally used for the calculation of boolean mask combinations: The bit map approach and edgebased methods.[2,6]
In the bit map approach (see [2,6] for instance) the layout of each mask is mapped into a twodimensional matrix of bits which represent transparent or opaque grid points of the layout. On most computers basic instructions such as OR, AND, EXOR can be used to calculate boolean combinations between these matrices. The method is conceptually simple and has time and space complexity $O(N)$ on conventional computers, but with a large factor hidden in the O - notation. Compression techniques [7] can be applied to reduce the storage needs. A special bit map processor has recently been presented [14] which would reduce time complexity to $O(1)$ but this methods seem not to be practical in near future.

A major drawback of the bit map approach is the difficulty of dealing with nonorthogonal artwork.
These problems are avoided in edgebased methods: Here the polygons of which the layout is composed are represented by a list of edges. Basic steps of applied algorithms are:

step 1 calculation of all intersections between polygon edges and splitting of edges at intersection points; as result of this step no two edges intersect other than at endpoints of edges (the set of edges is "completely intersected")

step 2 decision, which subset of the completely intersected set of edges is visible (i.e. represents a boundary between opaque and transparent regions) on the output mask (true edges)

step 3 reconstruction of polygones from the list of true edges; this poses no special problems, but will not be discussed here

Figure 1 shows for a simple example (boolean OR between two rectangles) the set of input edges, the completely intersected set of edges and the subset forming the boolean OR.

The two steps above can be applied to a pair of polygons at a time or to the whole set of edges in the two layers considered. The crucial point in step one is to organize the search for edge intersections in a way that excessive runtime is avoided (the naive approach of testing each edge against each other for intersection yields a prohibitive $O(N^2)$ time complexity).

Various methods have been applied for step two. The basic idea is to trace along the edges of one of the polygons [9,10] or along a set of scanlines [13] accumulating enough topological information to classify all edges in the completely intersected set.

An edge based algorithm was first described by Yamin [9], and similarily by Szanto [10]. In both papers a pair of polygons was considered and the issue of time complexity was not discussed.
Baird [3,4] described a set of procedures which work on the set of all edges of the layers considered. He used the concept of sorting (which was already folklore to his time) to cut down the expected time complexity of intersection calculation to $(N^{1.5})$ and devised a sophisticated but not easily to understand algorithm to determine the role which each edge plays on the output mask. This topological analysis worked on the set of edges incident to the currently processed point, which was stepped through the layout from left to right and from bottom to top in lexicographical sorted of x and y.

Some topological decisions had to be postponed until the total layout was processed. The expected space complexity of his algorithm is O ($\sqrt{N}$).

Recently, an algorithm for reporting all intersections between straight lines in the plane has been described by Bentley and Ottmann [12] basing on previous work by Shamos and Hoes [11] which has O (N log N) worstcase time complexity. We will use a simplified version of this method, enhanced by an additional scanning procedure for classification of output edges to solve our problem. Let us first - in the next section - shortly recall the Bentley - Ottmann algorithm.

4. The Bentley - Ottmann algorithm

The main idea in this algorithm is, to sweep a vertical scanline from left to right through the plane. The scanline defines a vertical order on the line segments crossing it. Only such segments which at some time are adjacent in this order can intersect each other and have therefore to be checked for intersection. The algorithm uses two datastructures Q and R. Q contains initially all segment endpoints and later on also the crosspoints between segments. The entries in Q are sorted according to x. R contains the segments which currently cross the scanline and is ordered according to the y - values of these segments at the current position of the vertical scanline. Sweeping the scanline through the plane is implemented by processing the points of Q in x - order: If the point being processed is the startpoint of a segment, the segment is inserted into R; if it is an endpoint, the pertinent segment is deleted from R; if it is a crosspoint, the two pertinent segments are adjacent in R and have to be interchanged. Whenever two segments become adjacent in R (during insertion, deletion or interchange) they are checked for intersection and the intersection point is added to Q (if not already in Q). For each point of Q being processed at most two such checks occur.
Operations on Q are INSERT, DELETEMIN and MEMBER, those on R are INSERT and DELETE (see [16] for nomenclature).

If appropriate datastructures * are used, the total runtime is O (N log N). Vertical line segments and the case of more than two lines crossing at one point pose problems which have not been handled in detail in [12]. We will see below that for our application these are nonproblems.

5. Complete intersection, a new technique

As pointed out earlier, we have to solve two problems: To find all intersections between input edges and to classify the edges in the completely intersected set for output. To solve the first problem, we modify the Bentley - Ottmann algorithm: We are interested only in such edge intersections which truely split at least one of the two edges. (Other edge intersections are the polygon nodes already known). To circumvent problems with vertical segments, these are omitted from the input; we will see later, that the significant vertical edges can easily be reconstructed.
If a true intersection between two edges is detected, we will immediatly split the pertinent edge(s) thus avoiding the operation of swapping entries in the vertical order.
The algorithm uses four datastructures EDGEFILES, QUEUE, OLDSCANLINE and NEWSCANLINE. EDGEFILES is a set of peripheral sequential files (one per layer) containing all nonvertical input edges sorted according to lexicographic order of the x- and y-values of their leftmost endpoint and their slope. QUEUE is a mainmemory datastructure containing edges to be processed and allows for MIN, INSERT and DELETEMIN operations, maintaining the same order as on EDGEFILES. QUEUE is used to buffer inputedges coming from EDGEFILES and new edges resulting from splitting. A procedure NEXTEDGE delivers and deletes the next edge from QUEUE and - if this edge had come originally from EDGEFILES - transfers the next edge (if any) from the respective file to QUEUE. OLDSCANLINE and NEWSCANLINE are linear lists containing all segments crossing associated scanlines in vertical order.

We are now ready to describe our algorithm in pseudo code:

* Bentley and Ottmann suggest a balanced binary searchtree for R and a heap for Q, but the latter seems not to be sufficient, due to the MEMBER operations needed on Q.

```
begin
   initialize EDGEFILES; (the input poly-
   gons are decomposed into edges which
     are stored - one edge a record - on
     peripheral files and sorted. No
     vertical edges are generated.)
   initialize QUEUE; (the first  edge from
   each of the two layers to be
     processed is inserted into the empty
     QUEUE.)
   x_o:=x_left(MIN (QUEUE)); (position of
   first scanline) OLDSCANLINE:= empty;
   repeat
     (set up new scanline :)
     NEWSCANLINE:=empty;
     while x_left(MIN (QUEUE)) =x_o do begin
       NEXTEDGE (e); INSERT (NEWSCANLINE,e)
   end;
   (update OLDSCANLINE:)
   for all edges in OLDSCANLINE calculate
   the y-value at x = x_o;
     (the vertical order in OLDSCANLINE
     is not affected by this step.
     We now have two lists of edges
     crossing the current position x_o:
     OLDSCANLINE containing "inherited"
     edges and NEWSCANLINE containing all
     edges starting at x = x_o)
   merge lists OLDSCANLINE and NEWSCAN-
   LINE into a common list OLDSCANLINE'
   preserving vertical order. Whenever
   during the merge two edges become
   adjacent and at least one of them is a
   new edge, then call the procedure
   INTERSECT;
   OUTPUTTRUEEDGES; (this procedure is
   described later)
   delete all edges from OLDSCANLINE
   which end at the current position x_o;
   Whenever during deletion two edges be-
   come adjacent, then call the procedure
   INTERSECT;
   (determine position of next scanline:)
   x_o:=∞;
   for all edges in OLDSCANLINE do x_o:=
   min (x_o,x_right(edge) );
   x_o:=min (x_o,x_left(MIN (QUEUE)));
   until x_o=∞;
end;
```

The procedure INTERSECT checks two edges for true intersection; if intersection occurs, the respective edge(s) is (are) split. Keeping the left part(s) in OLDSCANLINE, the right part(s) is (are) inserted into QUEUE.
The process of scanline maintenance is illustrated in Fig. 2. OLDSCANLINE contains the edges 1,2 and 3. Edges a, b and c are inserted into NEWSCANLINE. After merging the sequence is a, b, c, 2, 3.

The pairs (a,b), (b,1), (1,c) are checked and the intersection (b,1) is detected. Due to deletion of edge 2 the pair (c,3) becomes adjacent and leads to detection of another true intersection.

The data flow of the algorithm is shown in Fig.3.

As far as the algorithm has been described it solves the first problem, to bring the set of edges into completely intersected form.

6. Classification of edges for output

The second problem - classification of edges for output - can also be solved using the scanline concept: For this purpose, with each edge we keep information about the layer from which it came and about its direction (forward, backward).
If we scan a scanline from top to bottom (using the list OLDSCANLINE after completion of the merge) we can easily maintain two counters COUNT representing the "opaqueness" in the two layers and a logical variable BLACK indicating the state of the layer combination. The two counters are initialized with zero at the top of the scanline and increased (decreased) when a forward (backward) edge is crossed in the pertinent layer. At any time the color of the mask combination is defined as

$$\text{BLACK} = (\text{COUNT}[\text{layer1}] > 0) \text{ op } (\text{COUNT}[\text{layer2}] > 0)$$

with op $= \vee, \wedge, \wedge\neg, \neq$ for the OR, AND, AND NOT, EXOR operation.
Whenever the value of BLACK changes, we have crossed a true edge. True edges which end at the current position are passed to output.
The scan described is executed simultaneously "along the left side" of the scan line (taking into account crossing and ending edges) and "along the right side" (taking into account crossing and starting edges). If the two scans deliver different BLACK values, then a vertical edge is existent on the output mask. Vertical edges also are passed to output when they end.

7. Comlexity of the algorithms

We will discuss complexity in terms of N, the number of edges in the completely intersected edge set. (It makes no sense to discuss complexity in terms of the number of input edges because one can easily construct examples where M input edges generate O (M^2) intersections. Every sequential algorithm then would take at least O (M^2) steps.)

We have to look at the implementation of datastructures to analyse the complexity of the algorithms:

EDGEFILES is kept on peripheral storage and the associated sort takes O (N log N) time.

Operations on QUEUE are INSERT, DELETEMIN and INSPECTMIN. This can conveniently be done with a leftist tree [15,17]. Each edge from the completely intersected edge set is exactly once inserted and deleted. This also takes O (N log N) time (worst case).

On OLDSCANLINE and NEWSCANLINE we have insertions only at the front of the list, merging of two lists and deletions during a sequential scan thru the list. Therefore, a simple linear linked list structure is sufficient.

The time spent for intersection checks is linear in N because each edge is checked at most two times during insertion and may cause one additional check at deletion. This also is clearly a worst case bound.

The time for maintenance of the scanlines is linear in H•S where H is the average number of entries per scanline and S is the number of different scanlines. From the assumptions discussed in section 2 follows that H•S is O (N). Therefore the time for scanline maintenance is O (N) exept for the QUEUE-operations.

The same holds for the vertical scan operations in the procedure OUTPUTTRUE-EDGES. Both these estimations are e x p e c t e d time complexities. Thus the overall expected time complexity is O (N log N).

The expected space complexity is O ($\sqrt{N}$), since we keep only the SCANLINES and QUEUE in main memory.

8. Implementation and results

The algorithms described here are beeing implemented on a SIEMENS mainframe computer (paged **memory** , timesharing operating system, about 1 Mops) in Pascal. The problem calls for very careful coding to avoid trouble caused by rounding errors.

The table below shows some results obtained for the two examples in Fig. 4.

Sample	4 a	4 b
Operation	AND	AND
Number of		
input edges	100	100
intersections	0	240
nonvertical sections (N)	50	336
intersection checks	55	880
checks per section	1.1	2.6
scanlines	11	335
sections per scanline (average)	32.5	40
sections in QUEUE (max/aver.)	3/2.84	7/[illegible].52
output edges	240	240
Time[sec] cpu for		
macro expansion and decomposition into edges	0.30	0.31
sorting	1.01	1.13
Boolean operation	0.23	4.90
Memory needs[kbyte]	4	4

The number of intersection checks is in both samples well below the theoretical bound of 3 N.

9. Conclusions

A new algorithm for Boolean mask operations has been presented which is faster and (hopefully) easier to understand than previously published edge based algorithms.

We will give more experimental results in the oral presentation.

10. Acknowlegements

I would like to thank an unknown referee who did a very careful reading and triggered a rewriting of the sections on complexity. He also provided the example of Fig. 4 b.

Wolfgang Peine did most of the programming and testing work.

11. References

1. B.W. Lindsay, B.T. Preas: "Design Rule Checking and Analysis of IC Mask Designs", Proc. 13th DA Conf., San Francisco, June 1976, pp. 301-308

2. I. Dobes, R. Byrd: "The Automatic Recognition of Silicon Gate Transistor Geometries: An LSI Design Aid Program", Proc. 13th DA Conf., San Francisco, June 1976, pp. 327-335

3. H.S. Baird: "Fast Algorithms for LSI Artwork Analysis", Proc. 14th DA Conf., New Orleans, June 1977, pp. 303-311

4. H.S. Baird: "Design of a Family of Algorithms for Large Scale Integrated Circuit Mask Artwork Analysis", M.S. Thesis, Dept. of Computer Sience, Rutgers University, New Brunswick, New Jersey, May 1976

5. C.S. Chang: "LSI Layout Checking Using Bipolar Device Recognition Technique", Proc. 16th DA Conf., June 1979, San Diego, pp. 95-101

6. P. Losleben, K. Thompson: "Topological Analysis for VLSI Circuits", Proc. 16th DA Conf., June 1979, San Diego, pp. 461-473

7. J.A. Wilmore: "A Hierarchical Bit-Map Format For The Representation of IC Mask Data", Proc. 17th DA Conf., June 1980, Minneapolis, pp. 585-590

8. U. Lauther: "Simple But Fast Algorthms For Connectivity Extraction and Comparison in Cell Based VLSI Designs", Proc. ECCTD 80, September 1980, Warsaw, pp. 508-514

9. M. Yamin: "Derivation of All Figures Formed by the Intersection of Generalized Polygons", The Bell System Technical Journal, Vol. 51, No. 7, Sept. 1972, pp. 1595-1610

10. L. Szanto: "Network Recognition of a MOS Integrated Circuit from its Masks", Tesla electronics, 9(1976), pp. 67-75

11. I. Shamos, D. Hoey: "Geometric Intersection Problems", Proc. 17th Ann. Conf. Foundations of Computer Science, Oct. 1976, pp. 208-215

12. J.L. Bentley, T.A. Ottmann: "Algorithms for Reporting and Counting Geometric Intersections", IEEE Trans. Comp; Vol. 6-28, No. 9, Sept. 1979, pp. 643-647

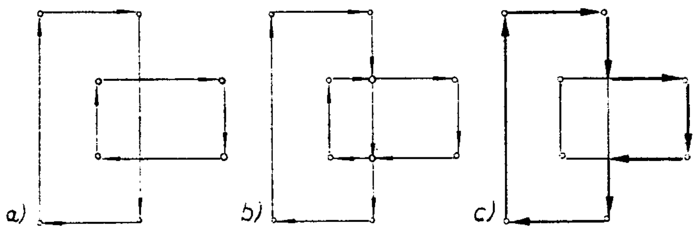
13. D. Alexander: "A Technology Independent Design Rule Checker", NCA Corporation, 383 Oakmead Parkway, Sunnyvale, Cal.

14. T. Blank: "A Bit Map Processing Maschine Architecture", IEEE DA Workshop, Lansing, Oct.1980

15. D.E. Knuth: "The Art of Computer Programming Vol 3: Sorting and Searching" Reading, MA: Addison-Wesley, 1973

16. A.V. Aho, J.E. Hopcroft, S.D. Ullmann: "The Design and Analysis of Computer Algorithms", Reading, MA: Addison-Wesley, 1975

17. C.A. Crane: "Linear lists and priority queues as balanced binary trees", Ph. D. Thesis, Stanford University, 1972



Fig, 1

Boolean OR between two rectangles.
a) Input
b) Completely intersected edge set
c) True edges (bold)

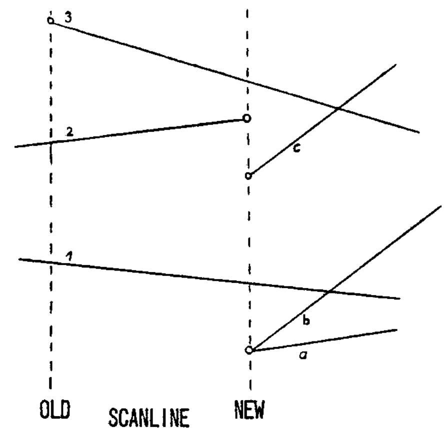


Fig. 2

A set of edges and two scanlines

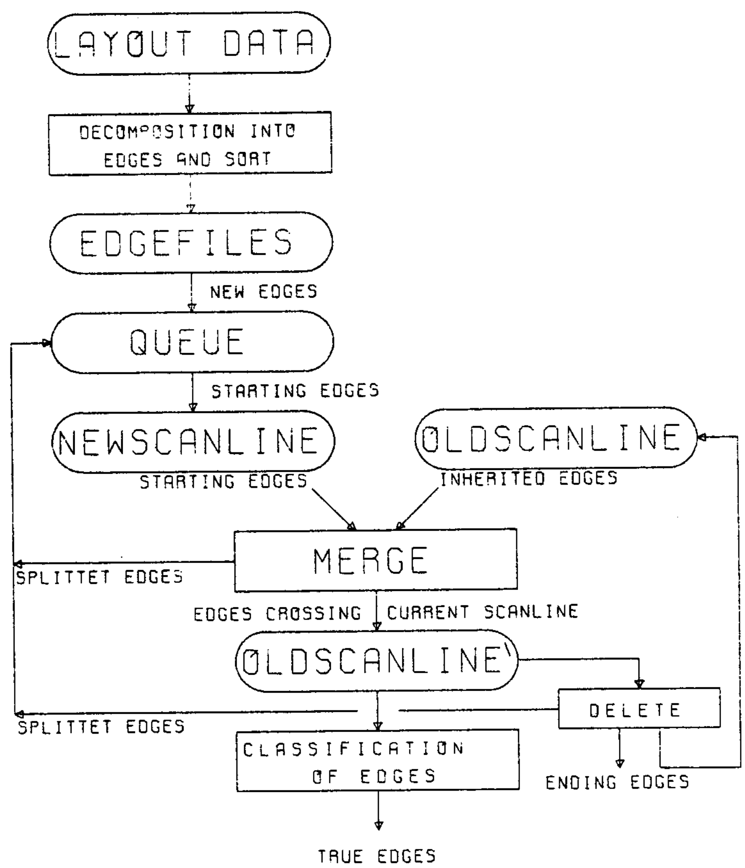
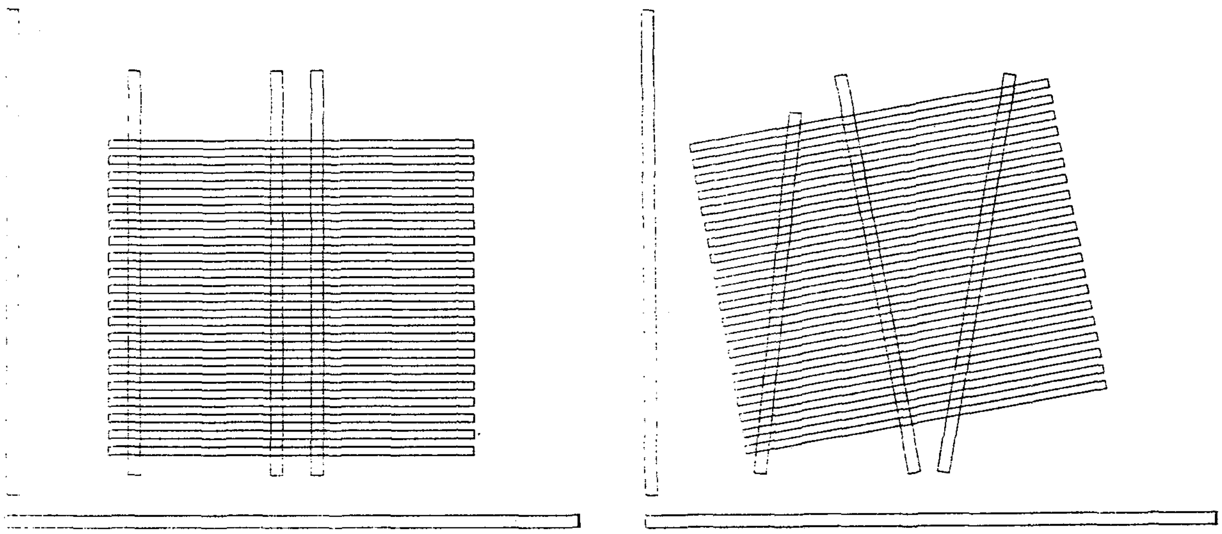


Fig. 3
Data flow of the algorithm for Boolean mask operations



a

b

Fig. 4
Two sets of test data