Systolic algorithms for rectilinear polygons

Rajiv Kane* and Sartaj Sahni

We develop systolic algorithms for the OR, AND, Oversizing, and Undersizing of rectilinear polygons. These algorithms work on an edge representation of the polygons rather than on a bit map representation. The algorithms are to be run on a systolic chain of processors. The edges are input at the left end of this chain. From here, they 'float' as far to the right as necessary. As edges float to the right, they compare themselves with edges that are resident in the processors they are floating through. During this comparison the output polygons are generated. Output polygons float to the left. These polygons are output from the left end of the chain. The throughput of the sytolic system can be improved by increasing the length of the processor chain.

rectilinear polygons, systolic algorithms, oversize, undersize, circuit design

Rapid advances in manufacturing technology have made it possible to fabricate chips of ever-increasing complexity. This has posed a severe challenge to existing design automation tools. Existing algorithms take more computer time than is desirable and in some cases require more time and memory than is practical.

One way to meet this challenge is to design new computer architecture and corresponding algorithms for design automation tasks. This approach has been the subject of many recent research efforts. Special architectures for design rule checks are described in Blank *et al*[1], Seiler[2] and Kane and Sahni[3], wire routing is considered in Blank *et al*[1], Mudge *et al*[4], Nair *et al*[5], and Ueda *et al*[6], Iosupovici *et al*[7] and Dah-Juh and Breuer[8] consider module placement. New architectures for simulation are proposed in Abramovici *et al*[9], Denneau[10], Kronstadt and Pfister[11], and Pfister[12]. It is anticipated that through the use of these specialized architectures, one can increase the circuit size that can be handled by a few orders of magnitude.

Kane and Sahni[3] have proposed a systolic design rule checker. This is essentially a hardware algorithm that checks for width and spacing errors. (The reader unfamiliar with systolic designs is referred to Kung[13] for an excellent introduction.) This paper is a continuation of the work reported in Kane[3]. Specifically, systolic algorithms are developed for some functions that are commonly performed on rectilinear polygons. These functions are:

- OR
  - ○ find the logical OR of a set of rectilinear polygons belonging to k layers
- AND
  - ○ find logical AND of a set of rectilinear polygons belonging to k layers
- UNDERSIZE
  - ○ reduce the height (width) of each rectilinear polygon by a specified amount $2d$
- OVERSIZE
  - ○ increase the height (width) of each polygon by a specified amount $2d$

The details of these functions are described in the next section. The systolic algorithms described in this paper are edge based. This coupled with the fact that the algorithms are to be implemented in hardware should result in a substantial reduction in the computing time required.

RECTILINEAR POLYGONS AND FUNCTIONS

In this paper only rectilinear polygons are dealt with explicitly. A rectilinear polygon is composed solely of horizontal and vertical edges. Further, it is assumed that all polygons are well formed. This means that open polygons and polygons with self overlaps (Figures 1(a) and (b)) are not permitted. However, polygons are permitted to contain holes which are themselves rectilinear polygons (Figure 1(c)).

The set of polygons to be handled is assumed to belong to different layers. It is assumed that a polygon belonging to a given layer satisfies the spacing and width requirements as described in Kane[3]. These requirements can actually be relaxed for the AND, and OR operations. In particular, polygons on a layer can share an edge, and a hole may share an edge with the enclosing polygon (see Figure 2). However, these are not allowed for oversize and undersize operations.

The restriction to rectilinear polygons allows a compact

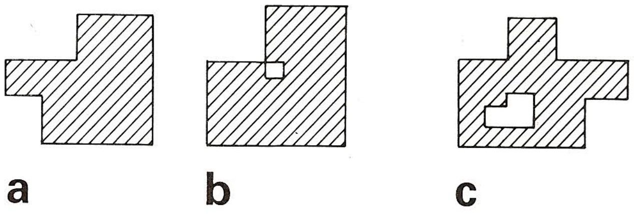


Figure 1. Polygons (a) open, (b) with self overlap and (c) with hole that is a rectilinear polygon. Polygon interior is shaded

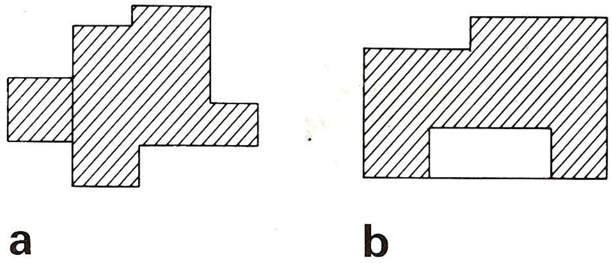


Figure 2. Polygons (a) on a layer sharing an edge and (b) enclosing and sharing an edge with a hole. Polygon interior is shaded

Computer Science Dept., 136 Lind Hall, University of Minnesota, Minnesota, MN 55455, USA
*Present address: Daisy Systems, San Jose, CA, USA

0010—4485/87/010015—10 $03.00 © 1987 Butterworth & Co (Publishers) Ltd

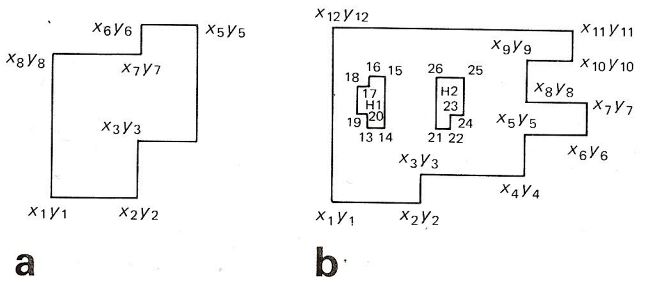


Figure 3. Polygons (a) without holes and (b) with two holes H1 and H2

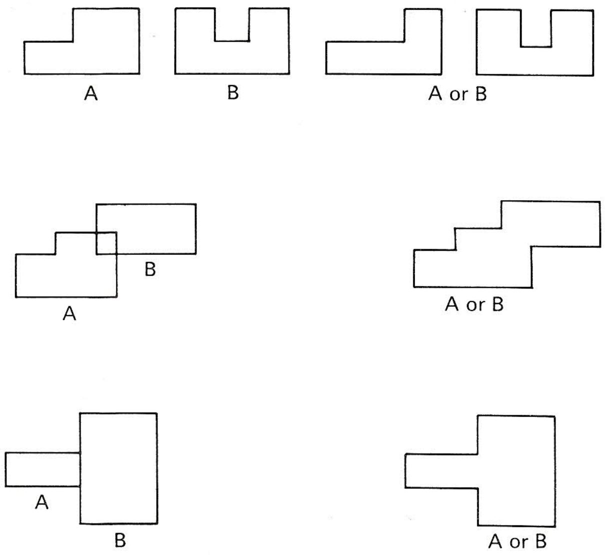


Figure 4. The OR of two rectilinear polygons A and B

representation for each polygon. This representation consists of the following:

- Polygon number
  - ○ Each polygon is assigned a unique number. Holes within a polygon are assigned the same number as the enclosing polygon.
- Layer number
  - ○ The layer number to which the polygon belongs.
- Sequence of polygon vertices

This sequence begins at the lowermost left hand vertex of the polygon and is obtained by traversing the polygon so that its interior lies to the left of the edge being traversed. Since all edges are either horizontal or vertical, the polygon vertices (except the first) may be described by providing a single coordinate. Thus, the polygon of Figure 3(a) is represented as

$$p, n, l, x_1, y_1, x_2, y_3, x_4, y_5, x_6, y_7, x_8, y_1$$

The first symbol *p* identifies this as an enclosing polygon; *n* is the polygon number. *l* is the layer number. In case of a hole, an *h* is used in place of the *p*. Holes are traversed such that the interior is to the left of each edge traversed. The representation for the polygon and holes of Figure 3(b) is

$$p, n, l, x_1, y_1, x_2, y_3, x_4, y_5, x_6, y_7, x_8, y_9, x_{10}, y_{11}, x_{12}, y_1$$

$$h, n, l, x_{13}, y_{13}, x_{14}, y_{15}, x_{16}, y_{17}, x_{18}, y_{19}, x_{20}, y_{13}$$

$$h, n, l, x_{21}, y_{21}, x_{22}, y_{23}, x_{24}, y_{25}, x_{26}, y_{21}$$

OR

The OR of two rectilinear polygons *A* and *B* is a set of rectilinear polygons that includes the area occupied by *A* as well as that occupied by *B*. Figure 4 gives some examples.

AND

The AND of two rectilinear polygons *A* and *B* is a third rectilinear polygon *C* which includes the area that is common to both *A* and to *B*. Figure 5 shows some examples.

Oversize

Oversizing in the *y* direction is described here. Oversizing in the *x* direction is similar. The objective is to enlarge the polygon by an amount 2*d* in the *y* direction. For each horizontal edge, with *ud* = 0, the *y* coordinate is decreased by *d* and for each horizontal edge with *ud* = 1, the *y* coordinate is increased by *d*. The OR of these elongated polygons with the original set (assumed to be a set of non-overlapping polygons) yields a set of polygons oversized by 2*d* in the *y* direction. Note that the OR is needed to properly account for holes that shrink to zero during the oversize and also to account for the fact that oversizing might cause polygons that did not overlap earlier, to do so now. Figure 6 gives some examples.

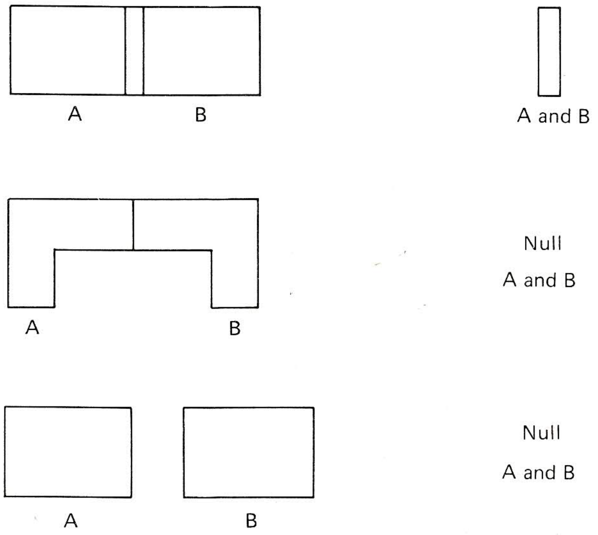
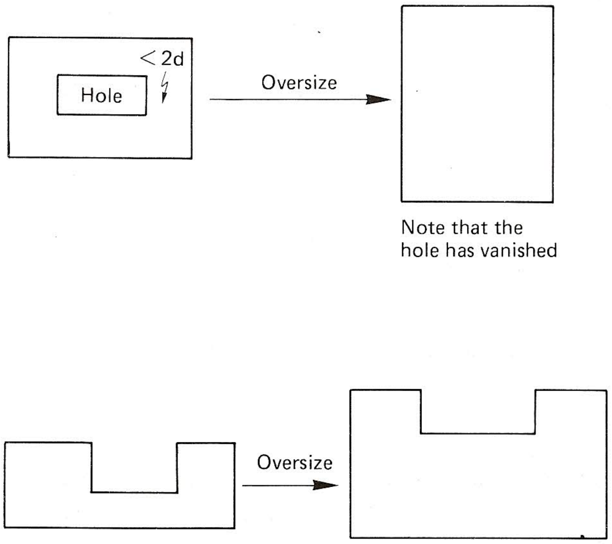


Figure 5. The AND of two rectilinear polygons A and B



Note that the hole has vanished

Figure 6. Examples of oversizing

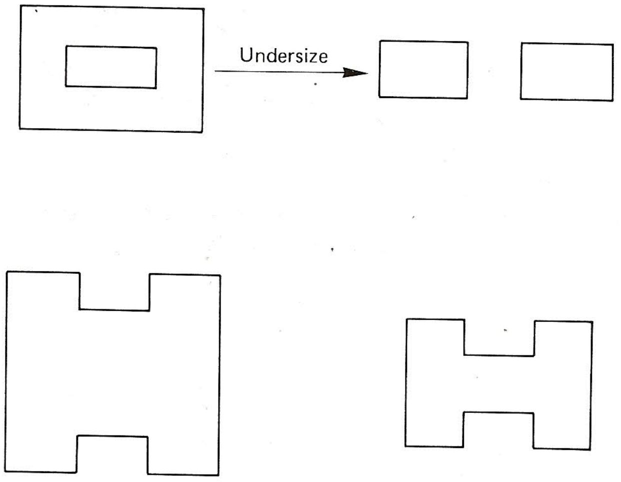


Figure 7. Examples of undersizing

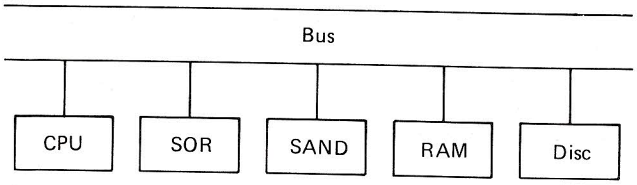


Figure 8. SOR and SAND attached to a computer system as a peripheral

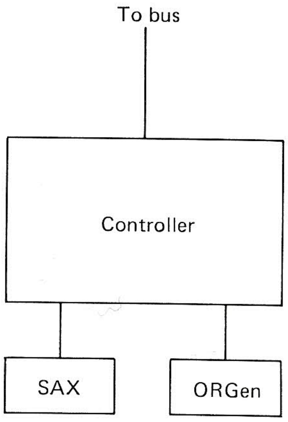


Figure 9. SOR architecture

Undersize

As in the case of oversize, only the operation for the y direction is described. Undersizing in the x direction is similar. The y coordinate of each horizontal edge is decreased by d if ud = 1 and increased by d if ud = 0. The AND of these shortened polygons with the original polygon set (assumed to be a set of nonoverlapping polygons) yields the set of undersized polygons. Figure 7 gives some examples. Note that if polygons are oversized by 2d we may not get back the original polygon. This happens, for example, when holes vanish due to oversize.

SOR AND SAND ARCHITECTURE

The SOR (systolic OR generator) and SAND (systolic AND generator) are hardware devices that may be attached to a computer system as a peripheral (see Figure 8) or directly to a CPU as in the case of a floating point processor. A high level description of the SOR architecture is explicitly provided. The architecture of the SAND is similar.

A block diagram of the SOR architecture appears in Figure 9. As can be seen, the systolic OR generator is comprised of three major units: a controller, SAX, and ORGen. The controller essentially communicates with the outside world and also coordinates the activities of the SAX and ORGen. SAX is a hardware sorter and ORGen is a hardware device that generates the OR.

The controller receives the compact polygon descriptions and generates horizontal edges of each polygon. Each horizontal edge has a four field descriptor as shown in Figure 10; y is the y coordinate of the edge; x_l the left coordinate; x_r is the right x coordinate; and ud (up/down) is 0 if the interior of the polygon is above the edge and 1 otherwise. The polygon number and layer number are not recorded in the edge descriptors.

As an example, consider the polygon in Figure 11. The horizontal edge descriptors are

$y_1, x_1, x_2, 1, 0$

$y_7, x_7, x_8, 1, 1$

$y_{16}, x_{16}, x_{15}, 1, 0$

$y_{10}, x_{10}, x_9, 1, 0$

$y_{11}, x_{11}, x_{12}, 1, 0$

$y_6, x_6, x_5, 1, 1$

$y_{14}, x_{14}, x_{13}, 1, 0$

$y_4, x_4, x_3, 1, 1$

The horizontal edges generated by the controller are transmitted to SAX, which is a sorter. This device sorts the horizontal edges into non-decreasing order of the multikey (y, x_l, ud) (i.e. y is the primary key, x_l is the first secondary key, and ud the second secondary key). For SAX, we propose using the systolic priority queue developed by Leiserson[14]. This device will periodically output the next horizontal edge in the desired sorted order. In this design[14], the first edge comes out two cycles after the last edge has been input. Thereafter, each succeeding edge may be extracted every two cycles.

The controller accepts the edges from the SAX in the sorted order and transmits them one by one to ORGen. The ORGen generates the horizontal edges of the polygons that form the OR of all the input polygons. These horizontal edges may be passed back to the controller or directly to the CPU.

In some applications it is adequate to describe the result of the OR by a sequence of horizontal edges. In others, it is necessary to provide the vertical edges explicitly too. This may be accomplished by carrying out a vertical edge OR generation in parallel with the generation of horizontal

y	x_l	x_r	ud

Figure 10. Four field descriptor of horizontal edge

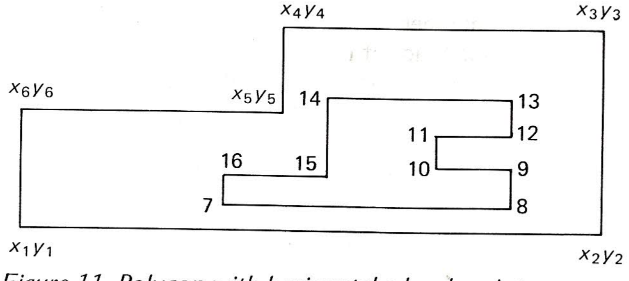


Figure 11. Polygon with horizontal edge descriptors

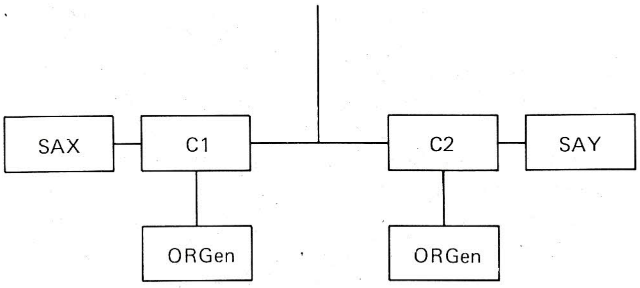


Figure 12. Block diagram of the SOR after a vertical edge OR generation in parallel with the generation of horizontal edges

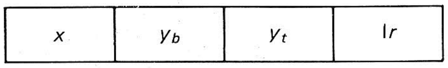


Figure 13. Four field descriptor of vertical edge

edges. The block diagram of the SOR now takes the form given in Figure 12. C1 generates horizontal edges while C2 generates vertical edges. Each vertical edge has a four field descriptor (see Figure 13). x is the x coordinate; y_b the bottom y value; y_t the top y value; and lr (left/right) is 0 if the polygon interior is to the right of the edge and 1 otherwise. The descriptors for vertical edges of Figure 11 are

$x_2, y_2, y_3, 1, 0$

$x_8, y_8, y_9, 1, 1$

$x_{12}, y_{12}, y_{13}, 1, 1$

$x_{10}, y_{10}, y_{11}, 1, 1$

$x_{15}, y_{15}, y_{14}, 1, 0$

$x_5, y_5, y_4, 1, 1$

$x_7, y_7, y_{16}, 1, 1$

$x_1, y_1, y_6, 1, 1$

SAY is identical to SAX except that it sorts vertical edges rather than horizontal. Another solution to the vertical edge problem is to generate the vertical edges in the OR from the horizontal edges as in Figure 14. This is slower than the solution of Figure 12 which generates the horizontal and vertical edges in parallel.

ORGEN : THE OR GENERATOR

Preliminaries and overview

The ORGen is a systolic array processor that accepts as input the horizontal (vertical) edges of a set of polygons and generates as output the horizontal (vertical) edges of the polygon obtained by computing the OR of the input polygons. In describing the ORGen, it is assumed that the horizontal edges are input. (Consequently, the terminology used in this section is with respect to horizontal edges.) As mentioned earlier, the horizontal edges are input to the ORGen in non-decreasing order of the multikey (y, x_l, ud).

The processors of the ORGen (also called processing elements) are connected to form a chain (see Figure 15). The horizontal edges of the polygons are input at the leftmost processor in this chain. These edges float towards the right. As an edge moves towards the right, it determines if it overlaps with edges already in the processor it is currently at. (Two horizontal edges overlap if they have a common x coordinate, the y coordinate of the edges need not be the same.) If so, the polygons described by the overlapping edges are to be combined in the OR. The edges are suitably processed to reflect this combining of polygons. Since edges are input in order of increasing y coordinate, it is possible to output polygons in the OR before processing all the horizontal edges. Specifically, once the y coordinate of an input edge exceeds the maximum y coordinate of a polygon in the ORGen array, this polygon may be output as further input edges cannot interact with this polygon. Polygons (or more correctly, their horizontal edges) are output from the leftmost processor. This is accomplished by making these 'float' from their resident processors leftwards along the processor chain.

The remainder of this paper provides the details of how the horizontal edges of the OR are generated and how these float towards the left and are output. Each processing element (PE) of the ORGen contains four sets of registers A, $B1$, $B2$, C as shown in Figure 15. The PEs are connected together to form a logical chain. Adjacent PEs need to be able to exchange the data from A registers. Also a PE needs to be able to send $B1$ and $B2$ registers to the right.

Edges enter the ORGen through the $B2$ register of PE 0. Output edges, ie, edges that form the polygon in the OR, exit the ORGen through the A register of PE 0. The A registers contain two kinds of edges:

- edges to be output
- edges not ready for output

This second category of A register edges have the following properties:

- No two edges overlap. If L_i and L_j are the left x coordinates of the edges in PEs i and j respectively, and R_i and R_j respectively are their right x coordinates, then either Rsubj $\leqslant L_i$ or $R_i \leqslant L_j$.

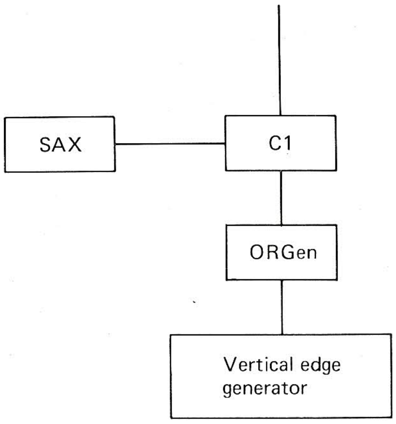


Figure 14. Vertical edges in the OR generated from the horizontal edges

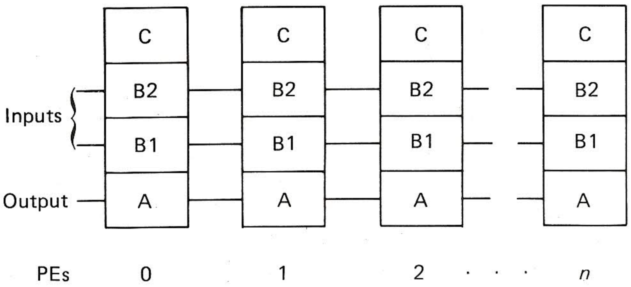


Figure 15. Processors of the ORGen connected to form a chain

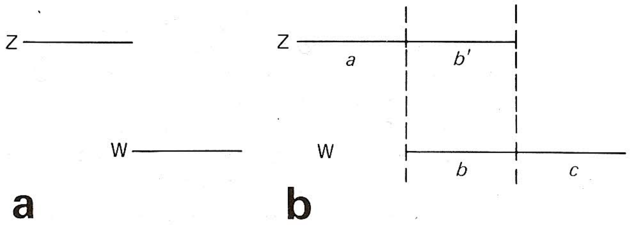


Figure 16. Z reaches a PE (a) whose A register edge W lies entirely to its right and (b) with which it overlaps

- Edges are ordered by their x coordinates. So, if $i < j$ then $R_i \leqslant L_j$.
- Edges are all lower boundary edges.

The B registers contain edges that are ready to be output or are looking for a *PE* to settle into (this can only be done by moving into an A register such that all the above properties are preserved). Let Z be an edge looking for a PE to settle into. Z moves to the right (through B registers and possibly going through C registers too) until it reaches a PE whose A register edge W lies entirely to its right (Figure 16(a)) or with which it overlaps (Figure 16(b)) is one such case).

If case Z is entirely to the left of W (Figure 16(a)) then Z settles in this PE and W moves towards the right through B registers. If there is an overlap, the edges are split. In Figure 16(b) we will assume that the level of W is 1. The Z and W get split into segments a, b and c. a settles in this PE while segments b and c move to the right through B registers. If the interior of the polygon (whose edge Z is) lies above Z, the number of open polygons above edge b becomes 2. In this case b' is discarded since it does not form an edge of the output polygons. If the interior of the polygon (of edge Z) lies below, then level number of edge b becomes 0. Thus no open polygons are above edge b. Thus an output edge is generated. This consists of edge b together with y coordinate of Z, which denotes an area bounded by b and b'.

Details of ORGen are presented below.

Each register will hold edge descriptor fields x_l, x_r, y and *ud*. Each set of register has the following additional fields:

- PR: a two bit priority field used to control the flow of the edges, the four possible values of PR have the following interpretation:
  - ○ PR = 11: this signifies an empty register
  - ○ PR = 10: the register contains an edge that has yet to settle
  - ○ PR = 01: this denotes an edge that has settled (this value is possible for A registers only)
  - ○ PR = 00: this denotes an edge in the OR output
- y_2: this field is used only when PR = 00, it allows the storage of two edges having the same x_l and x_r values (but different y values) in the same register
- level: the number of active (or open) polygons so far that are above this edge

Before the edges are input into the ORGen, every register in the ORGen is initialized so that PR = 11 and *ud* = 0. Following this, ORGen repeatedly executes its basic cycle. At the start of each iteration of this cycle, the ORGen checks for an overflow condition. Overflow occurs if the rightmost PE has a nonempty $B1$ or $B2$ register. If this is the case, then the ORGen cannot faithfully compute the OR of the input polygons and the problem needs to be partitioned into smaller subproblems.

If no overflow is detected, the edges in $B1$ and $B2$ registers of every PE are moved one PE right. In case there are new edges to enter into the ORGen, one of these (in the order stated earlier) is entered into the $B2$ register of the leftmost PE. The $B1$ register of this PE is set to be empty (ie, PR = 11). Each PE now processes the edges in its registers. Only the edges in A, $B1$, and $B2$ are involved in this processing. Following the processing, the edges in $B2$ and C registers of each PE are interchanged. This is done so as to cause a delay in the rightward movement of the $B2$ edges. Edges that appear in the OR of the input polygons are called output edges. These edges move to the left through A registers at the end of each cycle.

The processing that takes place in each PE during each cycle of the ORGen has the following objectives:

- control the rightward movement of edges
- ensure that all A register edges that are not output edges are ordered left to right and have no overlaps
- at the start of each cycle, the $B2$ edge (if any) is to the right of $B1$ edge (if any); there is no overlap between $B1$ and A edges

The details of the ORGen are spelled out in the next subsection.

```
procedure OR_Cycle
{basic cycle of ORGen}
begin
   if B1[n].PR <> 11 or B2[n].PR <> 11
   then
      overflow; stop
   endif

   {shift B edges right}
   B1[i], B2[i] ← B1[i-1], B2[i-1] i > 0
   {set B edges in the first PE (i.e. PE 0) }
   B1[0].PR ← 11; B1[0].ud ← 1

   if there is a new edge
   then
      [B2[0] ← new edge; B2[0].PR ← 10;
      if B2[0].ud = 0
      then
         B2[0].level ← 1
      else
         B2[0].level ← -1
      endif]
   else
      [B2[0].PR ← 11]
   endif

   OR_process_in_each_PE

   {exchange C and B2}
   C[i] ←→ B2[i] , i > 0
   {move output A edges left}

   if A[i].PR = 00 and A[i-1].PR <> 00
   then
      A[i] ←→ A[i-1], i > 1
   if A[0].PR = 00
   then
      [output edge A[0]; A[0].PR ← 11; A[0].ud ← 1]
   endif
end OR_Cycle;
```

Figure 17. Procedure OR_Cycle

```
procedure OR_managelevel (X)
begin
    {update the level of edge in register X}
    if B2.ud = 0 {polygon bounded by B2 is above B2}
    then
        X.level ← X.level - 1
    else {B2 is closing edge}
        [X.level ← X.level - 1
        if X.level = 0 {B2 and X are two output edges}
        then
            [X.PR ← 00; X.y2 ← B2.y; B2.PR ← 11]
        endif]
    endif
end OR_managelevel
```

Figure 18. Procedure OR_managelevel

Formal description of ORGen

The ORGen basic cycle

We shall use the notation $X[i].Y$ to refer to field Y of the register X of PE i. Thus, $B1[2]$.PR refers to the priority field of register $B1$ of PE 2. Every PE in the ORGen is initialized so that X.PR = 11 and $X.ud = 0$ for $X \in \{A, B1, B2, C\}$. Following the initialization, procedure OR_Cycle (Figure 17) is repeatedly executed until all edges have been output from the ORGen.

OR_managelevel

Before describing the procedure OR_process_in_each_PE that is used by OR_cycle, another procedure OR_managelevel (Figure 18) that is used extensively by this procedure is described. OR_managelevel life OR_process_in_each_PE is local to every PE. It has one parameter X which is a register name (the only possible values are A and $B1$). At the time it is invoked, it is known that the edges in $B2$ and X have the same x_l and x_r values. So, the presence of the edge in $B2$ will either increase or decrease the level number of the edge in X. If the level number of X drops to zero then X and $B2$ are respectively lower and upper bounding edges in the OR of the input polygons.

OR_process_in_each_PE

Each PE uses this procedure to process the edges in its A, $B1$ and $B2$ registers. In order to understand this procedure, one should keep the following in mind:

- 1 Edges may settle only in A registers. Thus only A registers may have a priority value 01. Furthermore, all settled edges have $ud = 0$.
- 2 Settled edges are ordered by their x_l values left to right in the A registers. The sequence of settled edges (ie PR = 01) may be interspersed with output edges (ie PR = 00) and empty A registers (ie PR = 11).
- 3 Edges that have yet to settle must do so by moving right through the $B1$ and $B2$ registers.
- 4 Edges left in $B2$ registers at the end of processing first move to the corresponding C register (see procedure OR_Cycle). Thus $B2$ edges move right once every two cycles.
- 5 A polygon edge may get split or discarded during processing. In general, edge splits and discards are carried out so as to ensure that the set of active edges (ie PR = 01 or 10) have no overlap of their x coordinates.
- 6 At the start of every cycle of PE processing the following is true in every PE:
    - (a) If a PE has both a B1 and B2 edge, then B2 edge is completely to the right of the B1 edge.
    - (b) Every B1 edge is completely to the left of all settled A edges (ie PR = 01) to its right.
    - (c) Every C register edge lexically precedes the $B2$ register edge (if any) in its PE. If the A register has PR = 01 then C register edge (if any) is completely to its right.

The details of the processing carried out in each PE during each cycle are provided using algorithmic notation in Appendix 1.

The correctness of conditions 6(a), (b) and (c) is readily established by induction on the cycles. The necessity of a C register in each PE is seen by examining case 3 of Figure 20(c). If the $B1'$ and $B2'$ segments proceed to the right as a packet then when $B1'$ finds a PE to settle into (the first PE with A.PR = 01 and $B1'.x_r <= A.x_l$) it is quite possible that $B2'$ now overlaps with the A edge. Handling this condition is now quite difficult. By using C registers as described here, $B2$ edges travel slower than $B1$ edges and it can be ensured that whenever a $B1$ edge is present, the $B2$ edge is completely to its right.

A final note about ORGen is that its output is really edge segments rather than complete edges. These segments are readily combined by creating two sorted lists; one sorted on x_l and the other on x_r using a systolic sorter. Next, these two lists are merged combining edges that abut.

Example

The working of ORGen is illustrated by means of an example. The polygons considered in the example are shown in Figure 21. Inside the PEs, edges are represented by their coordinates, but for sake of clarity their names will be used (as shown in Figure 22). During the processing, some of the edges get split into segments. The names of the segments and their coordinates appear in Figure 22. The output edges are stored in the registers as edges with PR field set to 00. The y_2 field in such cases will be shown along with the edge field.

Figure 23 shows the registers in the PEs as they will be depicted in the illustration. In Figure 23, the contents of the registers during the respective processing cycles are shown. The contents represent the status after the $B1$ and $B2$ register transfers have taken place. The processing that takes place in each of the cycles is now described.

- 0 Initialization. In all the PEs the PR field is set to 00 and ud to 0 (not shown in Figure 23).

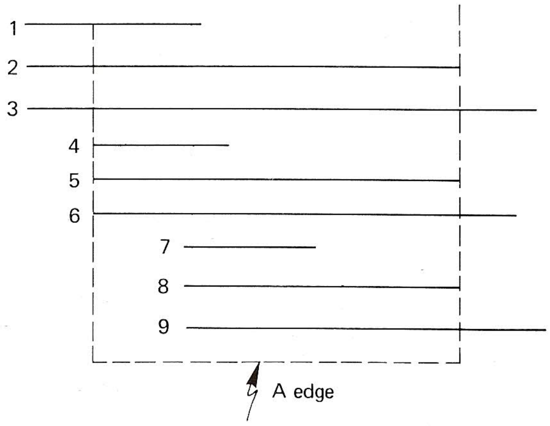


Figure 19. Nine possibilities for B2 relative to A

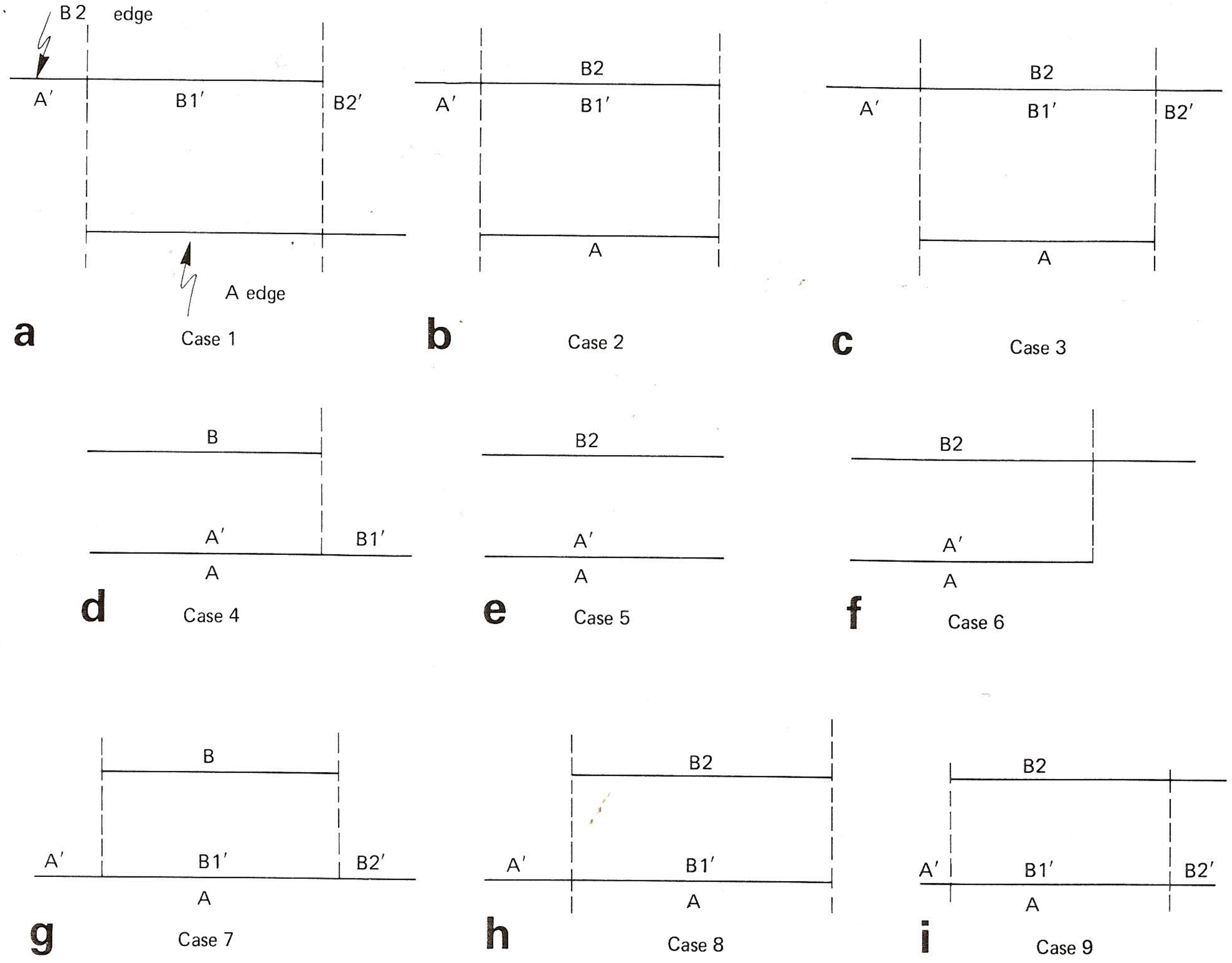


Figure 20. Nine cases; primed quantities are after processing and unprimed quantities are before processing

- 1 Edge a is input. This will settle in the A register of PE 0.
- 2 Edge b overlaps partially over a, in PE 0. The edges a and b are split giving rise to three segments h, f, and k; k goes to $B2$, f goes to $B1$ and h goes to A. $B2$ and C are exchanged and $B1$, $B2$ are transferred to PE 1.
- 3 In PE 0, c is to the right of h. In PE, f settles.
- 4 In PE 0, d and h have some overlap, thus splitting takes place. Segment h is ready to be output. Its PE field is set to 00 and a segment l is formed in $B2$. In PE 1, k lies entirely to the right of f. The result (h, 3) is taken out of the PE 0. In the next cycle $A[0]$.PR is set to 11 and $A[0]$.ud to 1.
- 5 In PE 0, A.PR is 11 with ud 1. Thus e does not settle in PE 0. In PE 1, c covers f completely. c is split and the level of f is reduced by 1. Segment m is created.
- 6 In PE 1, l covers f partially. After the splitting, segment g with PR 00 results. This is to be output. Another segment n is created which goes to $B1$. In PE 2, k settles. After OR_process_in_each_PE, $A[1]$ and $A[0]$ are exchanged. $A[0]$ contains the output which is taken out.
- 7 In PE 2, n is in $B1$ register, thus $B1$ and A are exchanged; so are $B2$ and C.
- 8 Segment k settles in PE 3.
- 9 In PE 2 and 3, $B2$ segments overlap A segments exactly. Thus results are generated. A register exchanges take place between PEs 1 and 2.
- 10 Results are pulled to the left through A registers in this and the remaining cycles (11, 12 and 13). Results come out from $A[0]$.

ANDGEN: THE AND GENERATOR

The operation of the ANDGen is very similar to that of the ORGen. The basic cycle of the ANDGen is the same as that of the ORGen except that OR_process_in_each_PE is replaced by AND_process_in_each_PE. The processing in each PE differs only in the specification of the managelevel procedure and handling of cases 7 and 9 (Figure 20(g) and (i)). First, cases 7 and 9 are modified, so that edge splitting

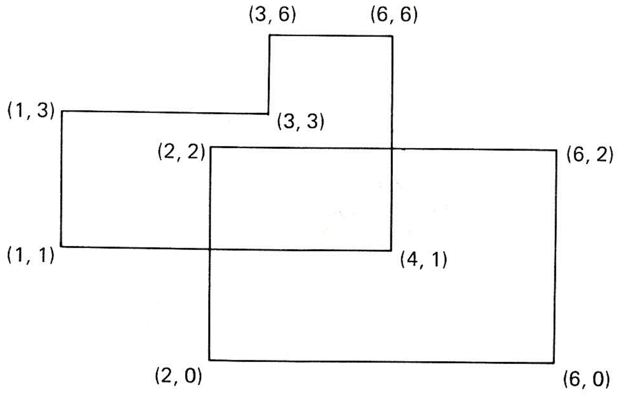


Figure 21. Polygons considered in the example of the working of ORGen

edge	x_l	x_r	y	ud
a	2	6	0	0
b	1	4	1	0
c	2	6	2	1
d	1	3	3	1
e	3	4	4	1
Segments				
f	2	4	0	1
g	2	3	0	1
h	1	2	1	0
k	4	6	0	0
l	2	3	3	1
m	4	6	2	1
n	3	4	0	0

Figure 22. Edge names of polygons

takes place as in Figure 24. This essentially delays some of the splitting that took place as in Figure 20(g) and (i). Note that these two cases no longer result in a call to managelevel. All uses of OR_managelevel are changed to AND_managelevel. Note that AND_managelevel is invoked only in cases 4, 5, 6, and 8 of Figure 20.

AND_managelevel has two parameters *X* and *Y*. *X* is the same register as before and *Y* register is the name of a register that will be empty following the processing (in case 4, $Y = B2$; in case 5, $Y = B1$ or $B2$; in case 6, $Y = B1$; and in case 8, $Y = B2$). This empty register *Y* is used to store the bounding edge of a result polygon. Procedure AND_managelevel is given in Figure 25.

Another alternative for the ANDGen is to handle cases 7 and 9 as in Figure 20 when $B2.ud = 0$ and as in Figure 24 when $B2.ud = 1$.

PARTITIONING

The total number of edges to be handled will often exceed the capacity of the ORGen and ANDGen. Hence there is a need to be able to decompose a large problem into smaller ones. This decomposition is quite straightforward for the AND and OR operations. Each layer may be partitioned as in Figure 26 and the AND/OR of each partition computed separately.

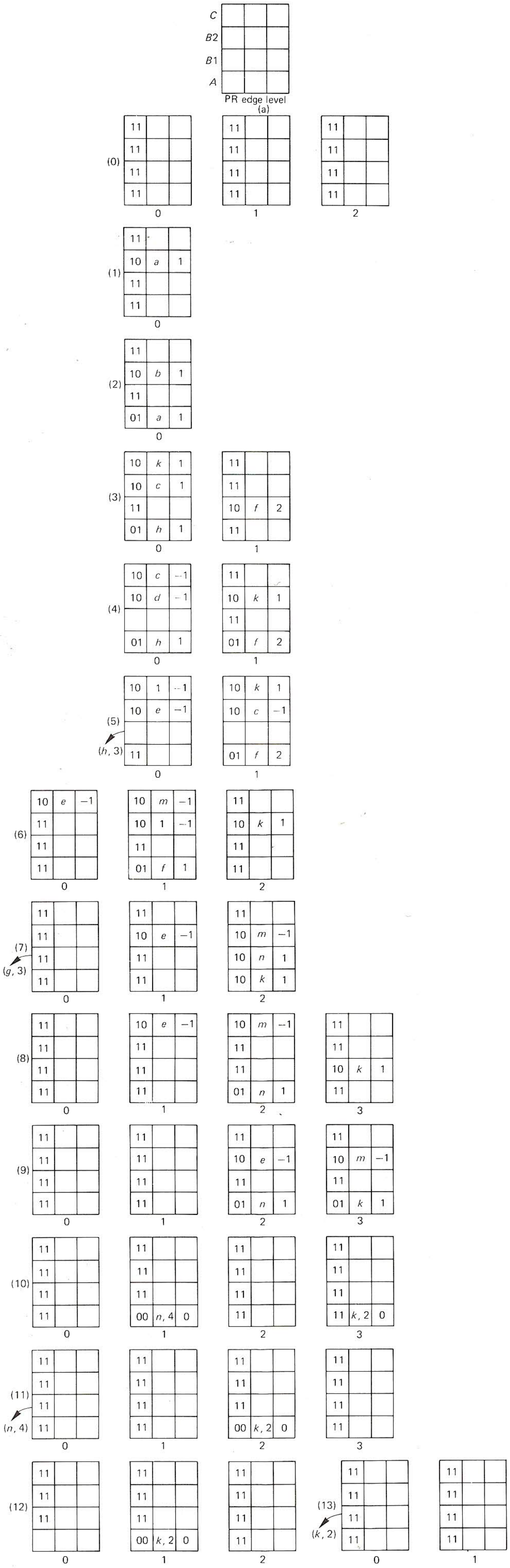
CONCLUSIONS

It has been demonstrated how the OR and AND of several polygons may be computed efficiently using a systolic array. This array has a reasonably simple architecture. Furthermore, since the oversize and undersize operations can be carried out by some simple processing (change of *y* coordinates) followed by OR or AND, these two operations can also be computed efficiently using systolic arrays.

ACKNOWLEDGEMENTS

This research was supported in part by the Office of Naval Research under contract N00014-80-C-0650.

Figure 23. (right) Registers in the PE showing contents during the processing cycles



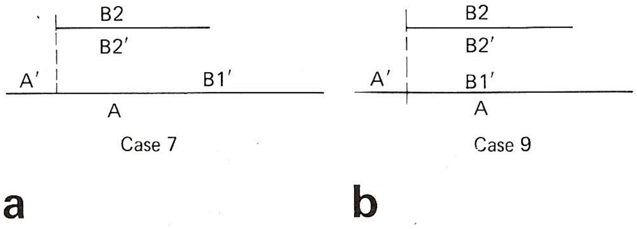


Figure 24. Modification of cases 7 and 9 so that edge splitting takes place

```
procedure AND_managelevel(X,Y) begin {k is the number of layers}
   if B2.ud = 0
   then
      [X.level ← X.level + 1
       X.y ← B2.y]
   else
      [X.level ← x.level - 1
       if X.level = k - 1
       then
          [Y ← X; Y.PR ← u0 Y.y₂ ← B2.y]
       if X.level = 0
       then
          [X.PR ← 11; X.ud ← 1]
      ]
   endif end AND_managelevel
```

Figure 25. Procedure AND_managelevel

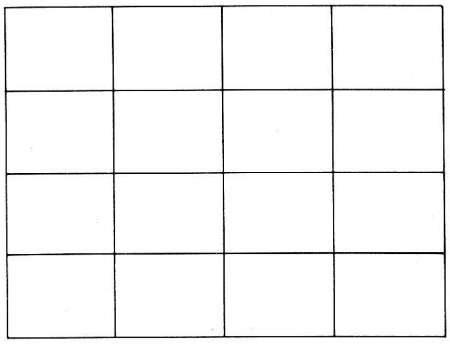


Figure 26. Partitioning into 16 partitions

REFERENCES

1 Blank, T, Stefik, M and vanCleemput, W 'A parallel bit map processor architecture for DA algorithms' *ACM IEEE 18th Des. Automation Conf. Proc.* pp 837–845

2 Seiler, L 'A hardware assisted design rule check architecture' *ACM IEEE 19th Des. Automation Conf. Proc.* pp 232–239

3 Kane, R and Sahni, S 'A systolic design rule checker' *TR 83-13* Department of Computer Science, University of Minnesota, MN, USA

4 Mudge, T N, Ratenbar, R A, Lougheed, R M and Atkins, D E 'Cellular image processing techniques for VLSI circuit layout validation and routing' *ACM IEEE 19th Des. Automation Conf. Proc.* pp 537–543

5 Nair, R, Jung, S, Liles, S and Villani, R 'Global wiring on a wire routing machine' *ACM IEEE 19th Des. Automation Conf. Proc.* pp 224–231

6 Ueda, K, Komatsubara, T and Hosaka, T 'A parallel processing approach for logic module placement' *ACM IEEE Trans. Comput. Aided Des.* Vol CAD-2 No 1 (Jan 1983) pp 39–47)

7 Iosupovici, A, King, C and Breuer, M 'A module interchange machine' *ACM IEEE 20th Des. Automation Conf. Proc.* pp 171–174

8 Chyan, D-J and Breuer, M A 'A placement algorithm for array processors' *ACM IEEE 20th Des. Automation Conf. Proc.* pp 182–188

9 Abramovici, M, Levendel, Y H and Menon, P R 'A logic simulation machine' *ACM IEEE 19th Des. Automation Conf. Proc.* pp 65–73

10 Denneau, M M 'The Yorktown simulation engine' *ACM IEEE 19th Des. Automation Conf. Proc.* pp 55–59

11 Kronstadt, E and Pfister, G 'Software support for the Yorktown simulation engine' *ACM IEEE 19th Des. Automation Conf. Proc.* pp 60–64

12 Pfister, G F 'The Yorktown simulation engine, introduction' *ACM IEEE 19th Des. Automation Conf. Proc.* pp 51–54

13 Kung, H T 'Let's design algorithms for VLSI systems' *CMU-CS-79-151* Department of Computer Science, Carnegie Mellon University, USA

14 Leiserson, C E 'Systolic priority queues' *Proc. Conf. on VLSI: Architecture, Des. Fab.* California Institute of Technology, CA, USA (Jan 1979) pp 199–214

APPENDIX 1: DETAILS OF PROCESSING IN EACH PE DURING EACH CYCLE

```
procedure OR_Process_in_each_PE begin
   case A.PR of
      00: {output edge in A; do nothing}

      10: {unsettled edge in A; not possible}

      11: {A register is empty}

         case B1.PR of

            00:A ←→ B1 {move output edge to A register}

            01: {not possible}

            10:if A.ud = 0 {no edges to the right}
               then
                  [A ←→ B1; A.PR ← 01]
               endif

            11:case B2.PR of

                  00:A ←→ B2 {move output edge to A register}

                  01: {not possible}

                  10:if A.ud = 0
                     then
                        [A ←→ B2; A.PR ← 01]

                  11: {do nothing}

            end case {B2.PR}

         end case {B1.PR}

   01: {register A contains a settled edge}

      case B1.PR of

         00: A ←→ B1; B1.PR ← 10 {move output edge to A}
```

```
10:if B1.$x_r$ $<=$ A.$x_l$
   then [A $\leftarrow\rightarrow$ B1; A.PR $\leftarrow$ 01; B1.PR $\leftarrow$ 10]

01: {not possible}

11: {B1 is empty but B2 may have an edge}

   case B2.PR of

      00: {transfer output edge to A; and A to B1}
         B1 $\leftarrow$ A; B1.PR $\leftarrow$ 10
         A $\leftarrow$ B2; A.PR $\leftarrow$ 01; B2.PR $\leftarrow$ 11

      01: {not possible}

      11: {do nothing}

      10: {A contains a settled edge and the edge in
         B has yet to settle }

         case

            :A.$x_l$ $>=$ B2.$x_r$: {B2 has to
                                   settle here}
             B1 $\leftarrow$ A; B1.PR $\leftarrow$ 10
             A $\leftarrow$ B2; A.PR $\leftarrow$ 01; B2.PR $\leftarrow$ 11

            :A.$x_r$ $\leqslant$ B2.$x_l$: {B2 is to the right
                                      of A; do nothing}

            :else : {B2 and A overlap; there are
                    nine cases (see Figure 19) }

               :case 1: {Figure 20(a). B2.ud = 0 as otherwise
                         the edge would have been split by now}
                     B1 $\leftarrow$ A; B1.PR $\leftarrow$ 10;
                     B1.$x_r$ $\leftarrow$ B2.$x_r$;
                     B1.level $\leftarrow$ B1.level + 1;
                     A $\leftarrow\rightarrow$ B2; A.$x_r$ $\leftarrow$ B2.$x_l$; A.PR $\leftarrow$ 01
                     B2.PR $\leftarrow$ 10

               :case 2: {Figure 20(b).B2.ud = 0 as otherwise the
                        edge would have been split by now}
                     B1 $\leftarrow$ A; B1.PR $\leftarrow$ 10;
                     B1.level $\leftarrow$ B1.level + 1
                     A $\leftarrow\rightarrow$ B2; A.$x_r$ $\leftarrow$ B1.$x_l$; A.PR $\leftarrow$ 01
                     B2.PR $\leftarrow$ 11

               :case 3: {Figure 20(c): B2.ud = 0 as otherwise the
                        edge B2 would have been split by now}
                     B1 $\leftarrow$ A; B1.PR $\leftarrow$ 01
                     B1.level $\leftarrow$ B1.level + 1
                     A $\leftarrow$ B2; A.PR $\leftarrow$ 01; A.$x_r$ $\leftarrow$ B1.$x_l$
                     B2.$x_l$ $\leftarrow$ B1.$x_r$

               :case 4: {Figure 20(d) }
                     B1 $\leftarrow$ A; B1.$x_l$ $\leftarrow$ B2.$x_r$
                     B1.PR $\leftarrow$ 10;
                     A.$x_r$ $\leftarrow$ B2.$x_r$
                     B2.PR $\leftarrow$ 11;
                     OR_managelevel(A);

               :case 5: {Figure 20(e) }
                     B2.PR $\leftarrow$ 11;
                     OR_managelevel(A);

               :case 6: {Figure 20(f) }
                     B2.$x_l$ $\leftarrow$ A.$x_r$;
                     OR_managelevel(A);

               :case 7: {Figure 20(g) }
                     B1 $\leftarrow$ B2; B1.y $\leftarrow$ A.y;
                     OR_managelevel(B1);
                     B2 $\leftarrow$ A; B2.$x_l$ $\leftarrow$ B1.$x_r$; B2.PR $\leftarrow$ 10
                     A.$x_r$ $\leftarrow$ B1.$x_l$;

               :case 8: {Figure 20(h) }
                     B1 $\leftarrow$ B2; B1 $\leftarrow$ A.y;
                     OR_managelevel(B1);
                     A.$x_r$ $\leftarrow$ B1.$x_l$;
                     B2.PR $\leftarrow$ 11

               :case 9: {Figure 20(i) }
                     B1 $\leftarrow$ B2; B1.y $\leftarrow$ A.y; B1.$x_r$ $\leftarrow$ A.$x_r$
                     OR_managelevel(B1);
                     A.$x_r$ $\leftarrow\rightarrow$ B2.$x_l$
            end else
         end case
      end {B2.PR = 10}
   end {B1.PR = 11}
  end case {B1.PR}
 end {A.PR = 01}
end case {A.PR}
end OR_Process_in_each_PE
```