Chapter 7

Row-Based Global Routing

7.1 Introduction

Global routing is the decomposition of an integrated circuit interconnection network into *net segments*, and the assignment of these net segments to regions or channels. The global routing results will be fed to a detailed router on a channel by channel basis. The detailed router will create the physical geometries necessary to manufacture the photomasks. This divide-and-conquer strategy produces global view solutions while managing the complexity of large circuit designs. It is assumed that the positions of the pins of a net have been determined in the placement phase. Global routing is known to be an NP-hard problem [105]. Therefore heuristic and approximation algorithms must be used.

7.1.1 Previous Work

One of the most common approaches in global routing is the maze route approach [130][156]. A major disadvantage of maze routing is that its space complexity is $O(n^2)$, where n represents the number of routing grids in each dimension of the plane. Another is routing order dependence. The space complexity problem can be eliminated using line probe algorithms, but the routing order dependence problem remains [89][152]. One could first route nets in a random order, ripping them up and rerouting them iteratively as Nair did. However, there is no guarantee that the rip-up order is optimal [162].

Global routing using a graph to model the network has also been proposed [4][138]. The vertices of the graph denote the cell terminals of the net and feedthrough ports. The edges correspond to minimal connections between adjacent pins within one channel. Global routing is accomplished by finding a Steiner tree for each net on the graph. The main drawbacks of this approach are the net routing order and mapping of feed-through pins. Also, as circuits continue to get larger, the increasing number of feedthrough pins will require more computer memory.

Hierarchical methods have also been proposed to handle large scale global routing problems. The problem is first partitioned into a hierarchy of global routing detail. Two schemes are possible: top-down [21][87] and bottom-up [144]. All nets are routed simultaneously at each hierarchical level eliminating the net ordering problem. The results of the current level are then refined in the next level of the hierarchy. Hierarchical methods work extremely fast and are useful in solving large-size problems. However, since the solutions at each level rely heavily on the solutions of previous levels and on the quality of the partitions, the resulting global routing often is suboptimal.

Constructive algorithms have been offered for gate array global routing. Li proposed routing from the chip's periphery and proceeding inward [138]. This method does not minimize total channel density or wire length. It only attempts to complete all of the routes. In 1987, Blair and company introduced an odd-even heuristic able to produce a solution within a factor of 1.5 of the optimal solution, but it only applies for two pin nets [14].

Linear programming techniques have been applied to the global routing problem as well [163][175][232][233]. Raghavan and others have attempted to solve the global routing integer linear program through the use of a randomized rounding technique [163][175]. An initial solution was found using a linear program solver. The final solutions were obtained by rounding any fractional numbers in the solution to 0 or 1 using a randomized technique. Although this approach finds the global routing solutions of all the nets simultaneously, it does not handle multiple pin nets correctly.

Single and multicommodity flow algorithms have also been applied to global routing. These methods use a graph model to describe the global routing problem. Each edge is assigned a capacity equal to the number of nets that may be routed through the corresponding channel. A network flow (single or multicommodity) algorithm is executed to determine the number of nets that may flow through a channel. Meixner and Lauther used a single-commodity flow formulation to improve an existing global routing solution and to guarantee integer solutions [150]. Carden and Cheng used Shahrokhi and Matula's multicommodity network flow algorithm to quickly derive a fractional solution for multiple pin nets [24]. Randomized rounding was used to obtain an integer solution. However, they can no longer guarantee an optimal solution.

In order to reduce execution time for global routing, algorithms for parallel computer architectures have been proposed [17][185]. Rose implemented a parallel global router for standard cells [185]. This global router generates a large set of routes between two pins and uses Nair's rip-up and reroute method to reduce the routing order dependency.

Global routers have been proposed to handle each of the design styles. They have been designed for gate arrays [49][138], standard cells [40][60][131][157][185], sea-of-gates [97][132][167][208], and macro cells [29][32][183][231]. Rose and colleagues have implemented global routers tailored for *island-style* field programmable gate arrays (FPGAs) [186][187]. However, this global router does not handle row-based field programmable gate arrays.

Other global routers have been augmented to handle additional constraints. Timing-driven global routers have also been proposed. Prasitjutrakul and Kubitz presented a timing-driven global router which maximized the minimum delay slack [169]. Cong and colleagues have produced a global router which bounds the length of the longest interconnection path using a bounded-radius minimum routing tree [41]. Cong also proposed a standard cell global router which minimizes area [40]. This router condenses the width of the chip by minimizing feedthroughs and the height by minimizing total channel density. However, it does not explicitly minimize area - the product of width times height. It also does not add Steiner points to further reduce the wire length. Other global routers have used linear assignment to optimally allocate feedthrough resources [150][167]. Meixner and Lauther presented a global router which uses linear assignment when placing feedthroughs to minimize wire length and vertical constraints [150]. However, it does not consider congestion and only minimizes vertical constraint loops during feedthrough assignment.

From the discussion above, we see that all of the previous global routing approaches suffer from one or more of the following shortcomings:

(1) nets are routed sequentially, and thus suffer from net ordering.

(2) multiterminal nets are not properly handled.

(3) Steiner points to reduce the total interconnection length are not added.

(4) the total channel density is not minimized.

(5) the number of feasible routing pattern shapes are limited.

(6) area is not explicitly minimized.

(7) timing constraints are ignored.

(8) feedthrough resources are not optimally allocated.

(9) congestion is ignored during feedthrough assignment.

(10) FPGAs or multiple row feedthroughs are not handled.

(11) vertical constraint loops are not minimized.

The TimberWolfSC global router [199][131] and its sea-of-gates derivative, SGGR (Sea-of-Gates Global Router) [132], do not suffer from the first five problems. In Section 7.3, we will present a global routing algorithm which addresses all of these shortcomings. For this reason, the TimberWolfSC global router will become the basis for our new global routing algorithm.

7.1.2 Previous Work in Steiner Tree Generation

The primary subproblem in global routing is the generation of a Steiner tree for a set of terminal pins. In 1966, Hanan showed that all points in an *minimum rectilinear Steiner tree* (MRST) must be a member of the Cartesian cross-product of the terminal points [82]. In 1976, Hwang showed that the *minimum spanning tree* (MST) is an approximation to the MRST with a worst-case ratio $\frac{\text{cost}(MST)}{\text{cost}(MRST)} \le \frac{3}{2}$ [95]. Kahng and Robins have recently categorized all previous Steiner tree algorithms [103]. They have shown that a large class of “minimum spanning tree-based” rectilinear Steiner tree heuristics have a worst case performance arbitrarily close to $\frac{3}{2}$ times optimal. They further subdivide the class into MST-overlap algorithms and Kruskal-Steiner algorithms. In MST-overlap algorithms, a MST is first generated, and then a shorter Steiner tree, which lies completely within the union of bounding boxes of the MST edges, is found. Algorithms given by Hwang [96], J. H. Lee [133], K. W. Lee [131], Ho [92], Hasan [84], and others fall broadly within this category. Kruskal-Steiner algorithms build a Steiner tree by connecting the closest pair of components until only one component remains. Kruskal-Steiner algorithms have been proposed by Bern [12][13], Richards [181], Servit [202] and others [39]. Kahng and Robins have proposed a new Steiner heuristic which iteratively finds optimal Steiner points to be added [104]. This iterative Steiner heuristic has a worst-case bound of $\frac{4}{3}$ times optimal and outperforms all known algorithms. Recently, Chua and Lim presented a $O(n\log n)$ algorithm which can generate the *k*-shortest Steiner trees [34].

7.2 Problems with Steiner Tree Algorithms

Table 7.1 shows a comparison between a previous TimberWolf global router and the TimberWolfSC global router modified with the iterated Steiner tree algorithm of Kahng and Robins[104]. This Steiner tree algorithm has been shown to outperform all known Steiner tree algorithms. With this addition to the global router, the best results should be achieved.

The first circuit in Table 7.1 is representative of circuits run with the TimberWolfSC global router. The wire length is reduced slightly and the number of feedthroughs are slightly increased using the iterated Steiner tree method. However, the final total density remains virtually unchanged. In the second row of Table 7.1, SGGR (Sea-of-Gates Global Router) was compared to TimberWolfSC augmented with the iterated Steiner method. In this case, the modified TimberWolfSC clearly outperformed SGGR in terms of wire length. However, the final total density is much higher. TimberWolfSC with the iterated Steiner tree method fails to take advantage of the many feedthrough cells available in this circuit. It is clear from this experiment that the shortest wire length Steiner trees for all nets does not always yield optimum area results.

Table 7.1 Comparison between Steiner tree algorithms.

Circuit	TimberWolfSC /SGGR			TimberWolfSC with iterated Steiner		
	wire length	feeds	tracks	wire length	feeds	tracks
primary1[1]	944360	717.8	163.5	942816	749	163.4
primary2[2]	4251730	4383	346.4	3811600	2982	359.6

[1] TimberWolfSC placement/TimberWolfSC global router.
[2] TimberWolfSC placement/SGGR global router.
Data is averaged over 8 runs.

It is important that the global router understands the feedthrough resources that are available for the design. If many feedthroughs exist, additional Steiner points may be added without penalty to reduce the wire length. However, if feedthrough locations are

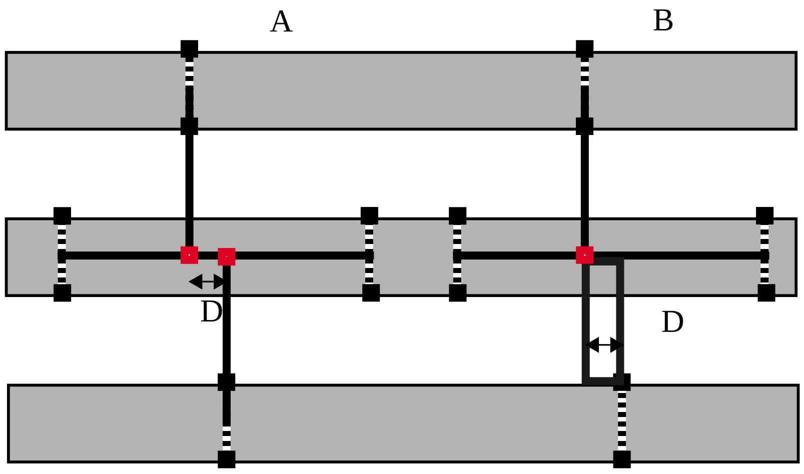

Figure 7.1 Feedthrough resources must be taken into account when building Steiner trees. A) Desired Steiner tree if Steiner point separation *D* is greater than average feed separation. B) Otherwise, only one Steiner point should be inserted.

scarce, additional Steiner points will increase the chip area. Each Steiner point will become a feedthrough cell. Since implicit feeds are limited, explicit feeds will need to be added, increasing the width of the chip. This case is shown in Figure 7.1b.

7.3 New Global Router

We will describe a new generalized row-based global router suitable for standard cell, gate-array, sea-of-gates, and field-programmable gate array (FPGA) circuits. This global router is the first row-based global router to explicitly minimize chip area. During optimization, the Steiner trees are dynamically modified to minimize chip area. Reducing the number of feedthroughs required in the longest row and minimizing the number of routing tracks enables the global router to adapt to the diverse routing resources of radically different technologies. The proposed global router uses exact port locations and density calculations throughout ensuring the correct construction of Steiner trees and calculation of chip area. By using a linear assignment algorithm to minimize wire length and maximize freeway utilization, the global router has been extended to handle FPGA *freeways* or feedthroughs which span several rows. In addition, the algorithm has been augmented to handle any number of *pin map*s, or versions of a cell instance - the program choosing the version which minimizes area. After all feedthroughs have been added and chip width has

```
Algorithm Global-Router(placement, netlist)
  1    Region-Generation()
  2    Area-Minimization()
  3    Assign-Multi-Row-Feedthrus()
  4    Assign-Single-Row-Feedthrus()
  5    Remove-Cell-Overlap()
  6    Cell-Swap-Optimization()
  7    Switchable-Segment-Optimization()
  8    Maze-Route()
  9    Vertical-Constraint-Loop-Minimization()
 10    Route-Verification()
```

Figure 7.2 Algorithm Overview

been determined, maze routing transforms the minimized Steiner trees to a minimum density solution. Finally, a unique vertical constraint loop minimization step breaks cycles in the vertical constraint graph allowing the use of left edge algorithms for detailed channel routing. Figure 7.2 show the overview of the global routing algorithm.

The input to the global router is a placement of row-based cells and a netlist which describes the signal interconnects between cells. Optionally, a cell may have more than one geometric view or *version* specified.

7.3.1 Region Generation

In the first step of the global routing algorithm, routing regions are defined. For each routing region a *density* calculation is performed. Density is the maximum number of nets crossing the width of a region. The routing regions are defined using the corner stitching method of Magic [166]. The tiles are merged using a heuristic to form the largest tiles possible in an area. Routing regions may be arbitrarily shaped rectilinear figures as shown in Figure 7.3. A routing region is defined as the set of tiles which encompasses the rectilinear area. Each region has a capacity which is specified on a per tile basis. For row regions, the capacity is fixed; it is zero for standard cell circuits and nonzero for sea-of-gates circuits. For gate arrays, the channel region's capacities are fixed.

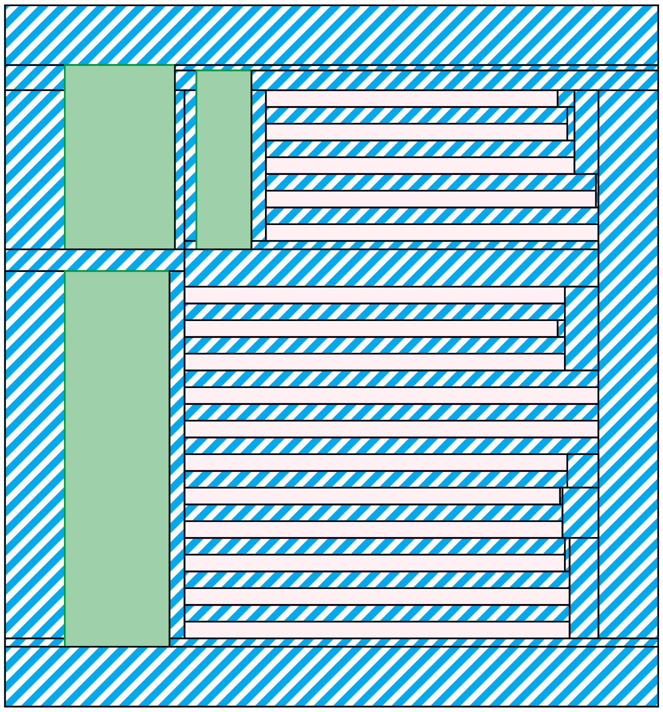
Figure 7.3 The routing tiles for a mixed macro/standard cell design.

7.3.2 Area Minimization

The area minimization step distinguishes this global router from its predecessors. This is the first global router which explicitly minimizes chip area. Figure 7.4 shows the area minimization algorithm. The loop in lines 1-2 seeks to generate a *minimum rectilinear Steiner tree* for all nets. A *minimum rectilinear Steiner tree* (MRST) is defined as follows: Given a set P of n signal pins, find an additional set S of *Steiner points* such that the minimum spanning tree over $P \cup S$ has minimum cost. The cost of the MRST is the summation of the lengths of all the edges where the length of an edge is measured using the rectilinear or Manhattan metric. Both the SGGR [132] and iterated Steiner tree algorithms [104] have been implemented. Lines 3 through 5 compute the initial height, width, and timing penalty. The initial height is the sum of the heights of the routing regions plus the row heights. The initial width is the length of the longest row counting any explicit feedthroughs that need to be added. The timing penalty is computed as discussed in Chapter 6. In line 6, the initial area is computed. The loop in lines 7-21 is a greedy algorithm

```
Algorithm Area-Minimization()
 1   for $i \leftarrow 1$ to $numnets$ do
 2       $T_i$ = Build-Steiner-Tree($i$, MINIMUM_WIRELENGTH)
 3   compute $H = \sum_{r=1}^{numregions} trackpitch \cdot tracks_r + \sum_{b=1}^{numrows} height_b$
 4   compute $W = \max_{b \in \{1, numrows\}} (length_b + feedwidth \cdot feeds_b)$
 5   compute $P_T$
 6   $oldcost \leftarrow W \cdot H + \alpha \cdot P_T$
 7   do
 8       $n \leftarrow Random(1, numnets)$
 9       pick segment $s$ of net $n$
10       if $s$ crosses maximum density then
11           $cost \leftarrow$ Flip-Segment()
12       else
13           if feed limited then
14               $T_n \leftarrow$ Build-Steiner-Tree($n$, MINIMUM_FEEDS)
15           else if density limited then
16               $T_n \leftarrow$ Build-Steiner-Tree($n$, MINIMUM_DENSITY)
17           else if Instance-Versions exists for net then
18               $\{T_i\} \leftarrow$ Swap-Instance-Versions($n$)
19           else continue
20           $cost \leftarrow$ Calculate-New-Cost()
21       if $cost \le oldcost$ then
22           Accept-Move()
23           $oldcost \leftarrow cost$
24   while $cost$ improves
```

Figure 7.4 Area minimization algorithm

which minimizes the chip area. A greedy algorithm suffices for area minimization since there are many alternative states which allow the algorithm to escape from local minima. Each iteration of the loop is as follows: First, a net is randomly selected. Next, a switchable segment of that net is picked. If that segment crosses the maximum density area of a channel, change the switchable segment to another state and calculate the new area and timing penalty. The possible switchable segment moves are shown in Figure 7.5.

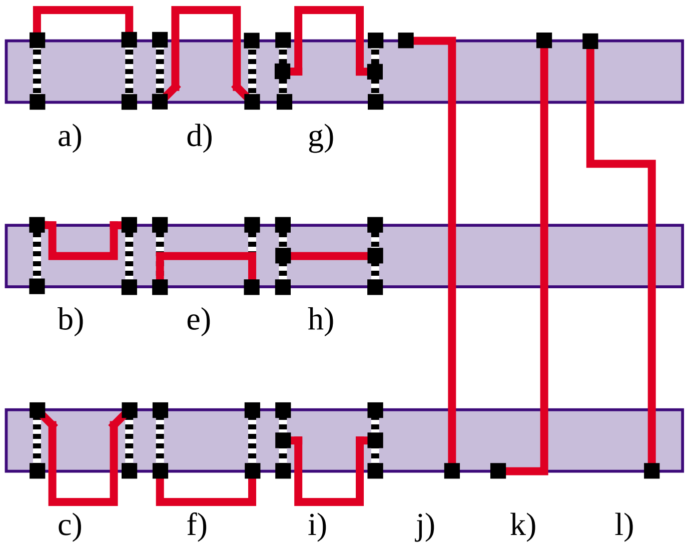

Figure 7.5 Switchable segment moves. Dashed lines indicate equivalent ports. Solid lines denote the segment connecting two ports. States a, f, j, k, l are valid for row-based circuits where the row area is a keep out area (gate-arrays and standard cells). States j, k, and l will require feedthrough insertion. Additional states b, c, d, e, g, h, and i become valid when routing over the row is permissible (sea-of-gates).

If the current segment does not cross a maximum density region, an attempt is made to rebuild the Steiner tree for that net. Line 13 determines if the net contributes to an explicit feedthrough in the longest row. If it does, any segments of this net connected to feeds in the longest row are removed, and the Steiner tree for the net is locally rebuilt. By changing the cost function for the Steiner algorithm, we can generate a Steiner tree which minimizes the number of feeds crossing this row. Figure 7.6a shows an example of a Steiner tree constructed to minimize feeds. Figure 7.7 shows an example of a Steiner reconstruction move to reduce the width of the chip.

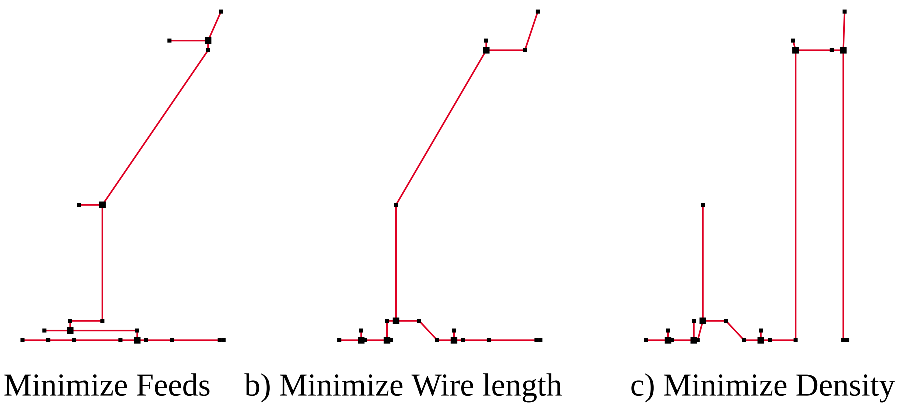

Figure 7.6 Various Steiner trees for a 16 pin net. The Steiner tree may minimize feedthroughs, wire length, or density.

If this net fails the test in line 13, it is then checked to see if it crosses any maximum density regions (i.e. portions of a routing region in which the local density equals the density). Note that all segments are tested in line 15, but only switchable segments are checked in line 10. If the net does cross a maximum density region, the Steiner tree for the net is regenerated for minimum density as shown in Figure 7.6c.

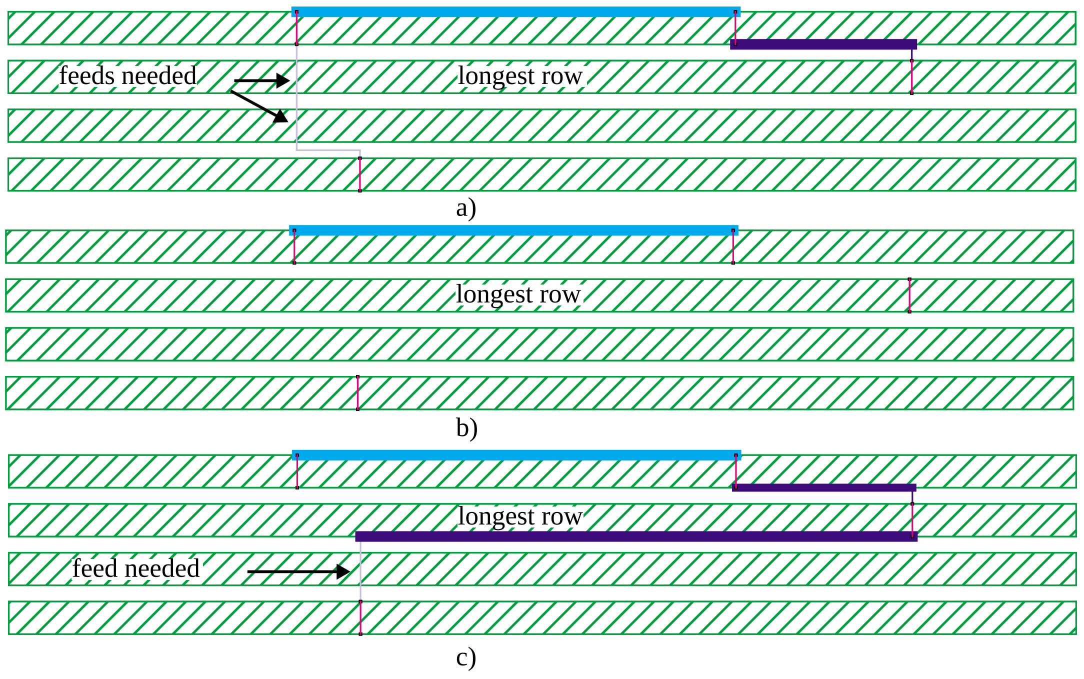

Figure 7.7 An example of a Steiner reconstruction move. a) the original minimum wire length Steiner tree requires two feedthoughs, one in the longest row. b) net segments which cross the longest row are removed. c) the reconstructed Steiner tree only needs one feedthough and none in the longest row. Hence, the longest row has been shortened.

If the test of line 15 fails, we then check to see if any cell instance connected to this net has multiple cell versions. If such a cell exists, another version of the cell is selected. Figure 7.8 shows an example of a cell instance with multiple versions.

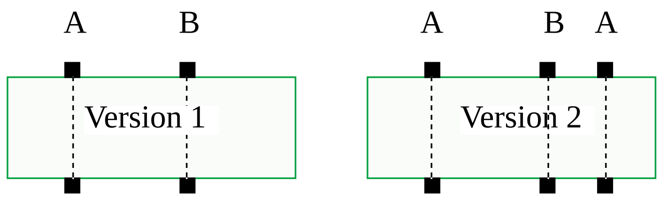

Figure 7.8 Cell instance may have multiple versions. Each instance has a unique pinmap. The number of ports may differ between pinmaps but the number of signals must remain constant.

If at this point the global routing has been modified, the new cost is calculated in line 20. This move is accepted if the cost decreases; otherwise, the move is rejected. Looping continues until the cost does not improve after *O*(*switchable segments*) moves. This same method was proposed in the previous TimberWolfSC global routers.

7.3.3 Feedthrough Assignment - Multiple Row Assignment

Feedthrough assignment is broken into two stages: multiple row and single row. To perform the assignment optimally, wire length and congestion must be considered for each phase. We define a *crossing* to be a net segment that crosses one or more rows and requires a feedthrough. A *freeway* is a feedthrough which spans multiple rows as shown in Figure 7.9. We define the cost for assigning a freeway *i* to a crossing *j* to be

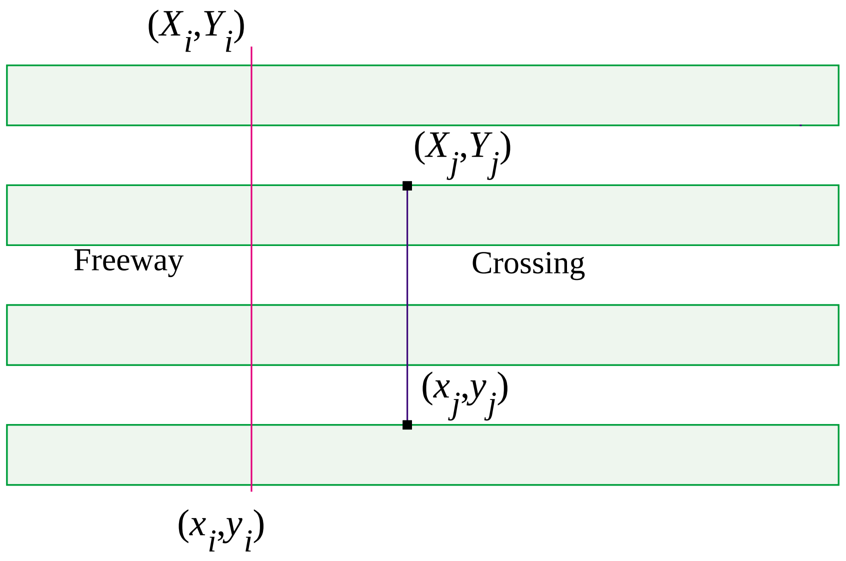

Figure 7.9 Definitions for multiple row feedthrough assignment. A multiple row feedthrough or freeway is shown at the left and a crossing is shown on the right.

$$C_{ij} = |X_i - X_j| + \{ (Y_i - y_i) - [\min(Y_i, Y_j) - \max(y_i, y_j)] \} + P_1 \tag{7.1}$$

where $f_i(x) = [x_i, X_i]$, $f_i(y) = [y_i, Y_i]$

$c_j(x) = [x_j, X_j]$, $c_j(y) = [y_j, Y_j]$,

defines the positions of the crossing and freeway segments, and

$$P_1 = K \cdot \Delta density \quad (7.2)$$

is the penalty due to congestion. The first term in the cost function minimizes the wire length, and the second term (in braces) is the freeway utilization cost. The congestion penalty is proportional to the change in density due to the addition of new horizontal segments between the crossing and freeway ports. The constant *K* is chosen such that,

$$rowlength \ll K \quad (7.3)$$

to ensure the penalty term receives the highest priority.

```
Algorithm multi_row_feed_assignment()
 1   work_to_do ← TRUE
 2   do
 3       {$Region_i$} ← Find-Available-MultiRowFeed()
 4       num_free ← Find-Number-MultiRowFeeds({$Region_i$})
 5       num_cross ← Find-Row-Crossings({$Region_i$})
 6       if num_cross = 0 then break
 7       if num_cross > 0 and num_free = 0 then error break
 8       if num_cross > num_free then
 9           Relax-Crossings({$Region_i$})
10       Linear-Assignment(C)
11       Update-Steiner-Trees()
12   while work_to_do
```

Figure 7.10 Multi row feedthrough assignment algorithm. This algorithm assigns both macro cell feedthoughs and FPGA freeways.

The multiple row feedthrough assignment algorithm is presented in Figure 7.10. The loop in lines 2-12 is repeated until there are no more multiple row feedthroughs to assign. In line 3, the core is partitioned into regions; one of which is picked to perform assignment. Next, all the multiple row feedthroughs which intersect this region are found. In line 5, the number of crossings in this region are determined. If crossings do not exist, we go on to the next region. If crossings exist, but there are no feedthroughs to assign, we issue a nonfatal error message and continue with the next region. The assignment of these crossings will be deferred to the single row feedthrough assignment stage. If the number of crossings is greater than the number of feedthroughs available in the region, we will relax the crossings in the region. Each crossing in the region will be ranked based on length and utilization of the freeway. In addition, a net will only be allowed to cross the region once; all crossings are consolidated into a single crossing which minimizes the total deviation from the original Steiner tree. In line 10, we use linear assignment to optimally assign the feedthroughs with respect to the cost function presented in Equation 7.1. Afterwards, the Steiner trees are updated to their new configuration.

7.3.4 Feedthrough Assignment - Single Row Assignment

The next stage of the global routing algorithm is the assignment of single row feedthroughs. Figure 7.11 shows an example of a single row feedthrough. In this formulation, we allow explicit feedthrough cells to contain multiple pairs of feedthrough ports.

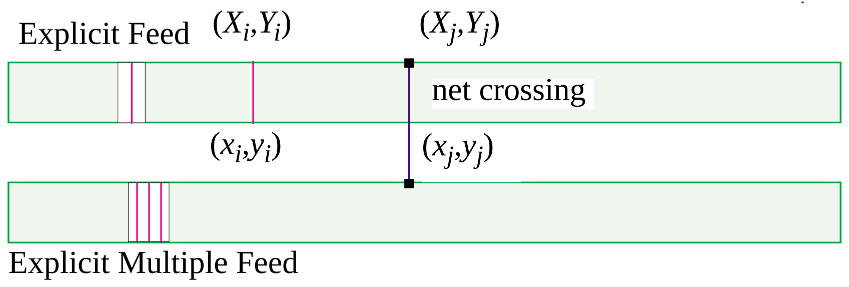

Figure 7.11 Examples of a single row feedthrough. The feedthrough cell in the bottom row has multiple feedthrough ports.

The cost of assigning a freethrough *i* to a crossing *j* is

$$C_{ij} = |X_i - X_j| + P_1 + P_2 + P_3 \quad (7.4)$$

where
$$P_1 = K \cdot \Delta density \quad (7.5)$$

$$_2 = \begin{cases} rowlength & \text{if} \quad explicit \quad feed \\ 0 & \text{otherwise} \end{cases} \quad (7.6)$$

$$_3 = \begin{cases} K & \text{if} \quad unused \quad multifeed \\ 0 & \text{otherwise} \end{cases} \quad (7.7)$$

$$rowlength \ll K \quad (7.8)$$

The first term in the cost function minimizes the wire length. In addition to the congestion penalty, the cost function also contains two penalties to control the addition of explicit feedthroughs. Penalty P_2 minimizes the number of explicit feedthroughs added. Penalty P_3 insures that all of the feedthrough ports of a multiple port feedthrough cell are used before adding an additional feedthrough.

```
Algorithm Feedthrough Assignment Phase II()
 1  for $r \leftarrow 1$ to numrows do
 2      avail ← Find-Available-Feeds(r)
 3      need ← Find-Net-Crossings(r)
 4      if avail < need then
 5          if fixed-width then
 6              Relax-Steiner-Trees(r, MINIMUM_FEEDS)
 7          Add-Extra-Feeds-Between-Cells()
 8      Eliminate-Overlapping-Crossings()
 9      Linear-Assignment(C)
10      Update-Steiner-Trees()
```

Figure 7.12 Single row feedthrough assignment algorithm.

The algorithm for single row feedthrough assignment is presented in Figure 7.12. We assign each of the rows in turn. In line 2, we find the available implicit feedthroughs in the row. Next, we find the net segments which cross this row. If the number of required crossings exceeds the number of implicit feeds available, we check to see if we are allowed to add this many explicit feedthroughs in line 5. If the number of feedthroughs that we are allowed to add in this row is less than the number we need to add (such as in a gate-array design), we relax all Steiner trees which cross the row. These Steiner trees are converted to Steiner trees requiring the minimum amount of feeds. In line 7, we add explicit feedthroughs between cells. For each crossing intersecting a cell, we add an explicit feedthrough at both ends of the cell. The cost function insures that the minimum number of explicit feeds will actually be added. That is, unassigned explicit feeds will be deleted. In the case that Relax-Steiner-Trees fails to reduce the number of crossings sufficiently, the global router will be able complete the routing, albeit infeasible. In line 8, overlapping crossings, or crossings at the same *x*-coordinate, are untangled using a modified version of Groeneveld's wiring order algorithm [78]. Feedthrough assignment is then performed optimally using linear assignment with the cost function of Equation 7.4. Afterwards, the Steiner trees are updated to reflect the addition of the feedthroughs.

7.3.5 Cell Overlap Removal

If any explicit feedthroughs have been added to the design, we must remove any overlap created. Overlap removal is accomplished by sorting cells by their left edge. Ties are broken using the Groeneveld ordering information. The sorted cells may then be placed end to end as shown in Figure 7.13.

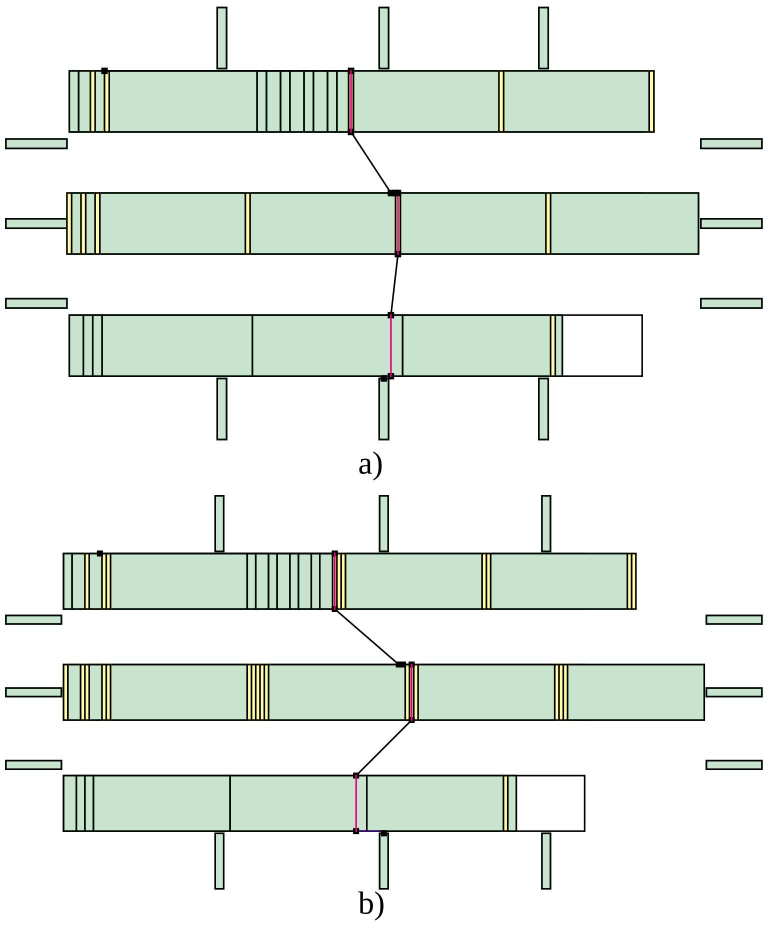

Figure 7.13 Removal of cell overlap. a) After explicit feeds have been added. b) After overlap has been removed.

7.3.6 Cell Swap Optimization

The next major step in the global routing algorithm is the cell swap optimization phase. Two types of placement modifications are attempted: a pairwise interchange of neighboring cells and an orientation change for a cell. The cost function is used to minimize wire length and the timing penalty:

$$C = \sum_{n=1}^{nets} S_x(n) + S_y(n) + P_T \qquad (7.9)$$

where

S_{x_n}, S_{y_n}	Steiner wire length
P_T	timing penalty

A placement modification is randomly selected; the new state is accepted if the cost is not increased. The algorithm stops when further improvement appears unlikely. It has a major impact on designs that require many explicit feedthroughs.

7.3.7 Switchable Segment Optimization

Next, the area minimization algorithm of Figure 7.4 is re-executed. Since the feed positions are known, the chip width is fixed. The area minimization algorithm reduces to a problem of optimizing track density under timing constraints. In addition, only states *a* through *i* inclusive of Figure 7.5 are considered for swapping. This insures that wire length, and hence the timing penalty, do not change during optimization. This step is analogous to similar steps performed in the previous TimberWolf global routers.

7.3.8 Maze Routing

The previous switchable segment optimization does not allow the Steiner tree to exceed the bounding box of the ports of the net. Maze routing is used to transform the minimized Steiner trees to a minimum density solution not necessarily constrained by the bounding box. In order to perform maze routing efficiently, we define a graph $G(V, E)$

where the nodes V denote port locations, and the edges E are formed between adjacent ports in the same region. An example of the maze graph construction step is shown in Figure 7.14.

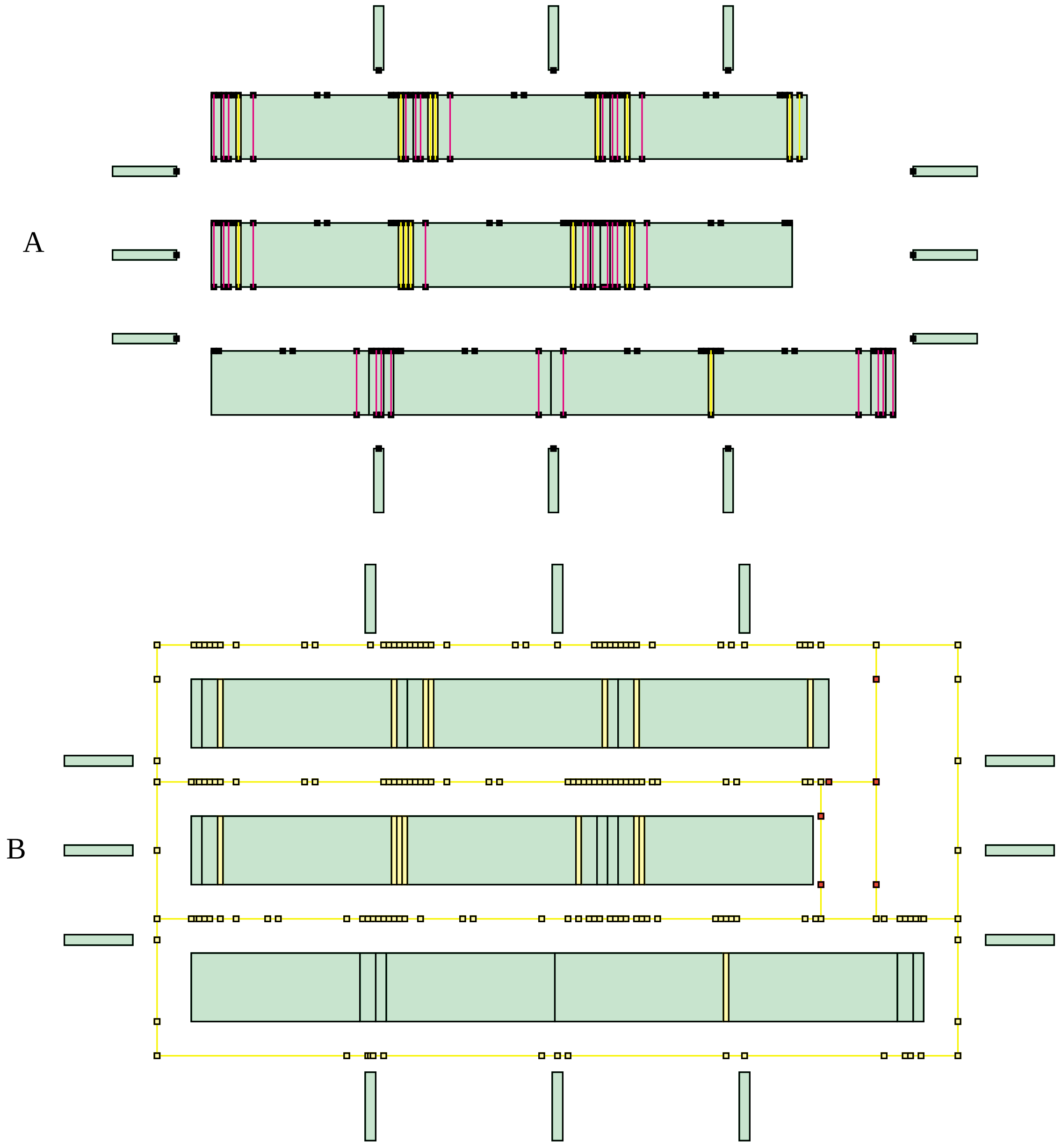

Figure 7.14 An example of the maze graph construction step for a small circuit. Figure *a*) shows the port locations and *b*) displays the resulting graph.

Furthermore, we observe that the track count can only make transitions at the nodes of the graph; hence, we store the density information at the nodes. We define a critical node as a node which is within one track of the maximum density of the region:

$$ritical \equiv \begin{cases} 0 \ \text{if} \ \ density\,(node) < maxdensity\,(region) - 1 \\ 1 \qquad\qquad\qquad\qquad \text{otherwise} \end{cases} \tag{7.10}$$

Any new routing path that contains a critical node will either maintain or increase the density. Our goal is to find an alternate path for a segment in a maximum density region which does not contain any critical nodes.

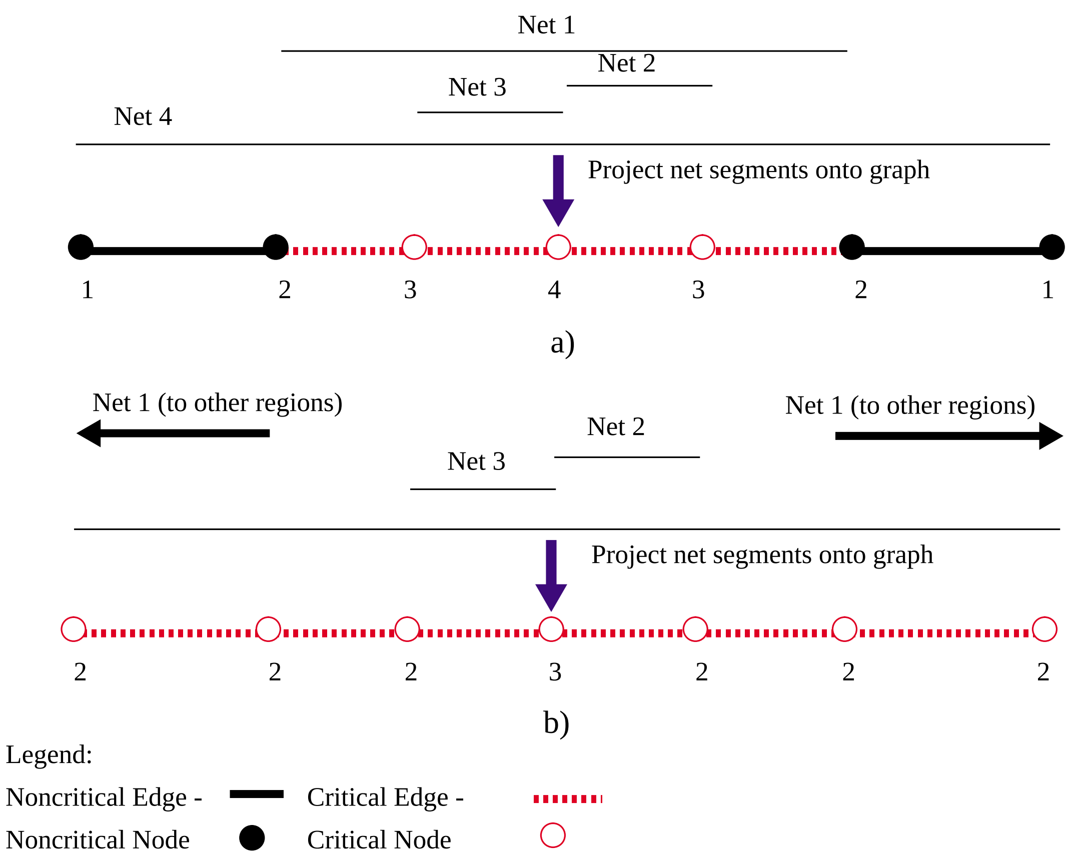

Figure 7.15 Example of incremental maze routing. a) Original routes for nets and their projection onto the graph. The maximum density is 4. b) If we reroute net 1 using other regions, we reduce the density to 3. It does not help to reroute nets 2 or 3 since their ports are at critical nodes. After reroute, all ports are critical. Notice that if the new route does not contain any critical nodes, we are guaranteed that the new route reduces the density by 1.

In order to perform maze routing on the graph, the nets are projected onto the graph, and the track count is summed at each node as shown in Figure 7.15a. Maze routing *candidates* are those net segments which span a maximum density region and have at least one port at a noncritical node. In the example of Figure 7.15a, nets 1 and 4 are valid candidates.

```
Algorithm Maze-Route()
1   Build-Maze()
2   Initialize-Maze()                    /* calculate initial maximum density regions */
3   $\{S\} \leftarrow$ Find-Candidates()     /* S is the set of candidate segments */
4   do
5       $reduction \leftarrow$ FALSE
6       while $\{S\} \neq \{\ \}$ do
7           for $S \in \{S\}$ do
8               for $e \in S$ do             /* for all edges which make up S */
9                   $w[e] \leftarrow \infty$           /* set weight of edge to infinite cost */
10              $reduced \leftarrow$ Maze-Route $(S, C)$ /* maze route segment using dynamic cost function
11              if reduced = TRUE then
12                  Update-Steiner-Trees()
13                  $reduction \leftarrow$ TRUE
14          if reduction = TRUE then
15              $\{S\} \leftarrow$ Find-Candidates()
16  while reduction = TRUE and $S \neq \{\ \}$
```

Figure 7.16 Algorithm for incremental maze routing.

Maze routing is performed using the algorithm presented in Figure 7.16. After the initialization (lines 1-3), the loop in lines 4-16 attempts to reduce the total density by rerouting a segment that crosses the maximum density region. For each candidate segment, the edge weights are initialized to an infinite cost so they cannot be part of the new route. In line 9, maze routing is performed on the graph for the net segment candidate *S*. The cost function which defines the weight of an edge *e* as

$$[e] = \begin{cases} \infty & \text{if} \quad \text{critical}(n_1) \quad \text{or} \quad \text{critical}(n_2) \\ |x_1 - x_2| + |y_1 - y_2| & \text{otherwise} \end{cases} \tag{7.11}$$

where n_1 and n_2 are the two nodes of the edge.

Observe that if the sum of the edge weights of a path is finite, a new route of shortest length has been found which reduces the total density. If a new route is found, the Steiner tree for the net is updated and the reduction flag is set. The outer loop continues until further improvement is not possible. Figure 7.17 shows an example of incremental maze routing.

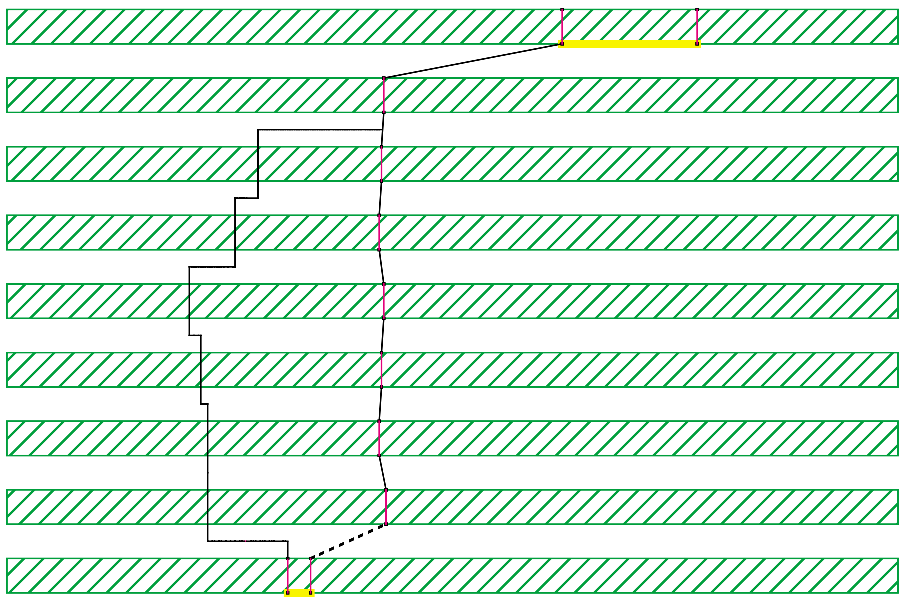

Figure 7.17 Maze route rip-up and reroute. The broken line is a segment that crosses the maximum density region. The segment is removed and the net is reconnected using the new route on the left which does not cross any maximum density regions. Hence, the total density has been reduced by one. The rerouted segment extends beyond the minimum bounding box of the ports.

7.3.9 Vertical Constraint Minimization

Another unique feature of this global router is vertical constraint loop minimization. If a cycle exists in the vertical constraint graph, the resulting global routes cannot be detailed routed using a left-edge algorithm (LEA) channel router. An LEA router must break the cycle by ignoring a constraint that is a part of the cycle. The remaining net segments may then be routed using the left edge algorithm, but the deleted constraint must be routed by a special technique (usually maze routing) [179].

The algorithm for minimizing vertical constraints is shown in Figure 7.4. The set of

Algorithm Vertical-Constraint-Minimization()

```
 1  $C \leftarrow \{\ \}$                                  /* C is the set of cycles */
 2  for $r \leftarrow 1$ to numregions do
 3      $G_v \leftarrow$ Build-Vertical-Constraint-Graph $(r)$
 4      $C \leftarrow C \cup$ FindCycles $(G_v, r)$      /* find cycles in the vertical constraint graph */
 5  $oldcost \leftarrow$ Initialize-Cost()               /* calculate initial track density */
 6  while $C \neq \{\ \}$
 7      $n \leftarrow Random(1, numnets)$
 8      pick segment $s$ of net $n$
 9      if $s \in C$ then                            /* check to see if segment s occurs in any cycle */
10          cost$\leftarrow$ Flip-Segment()
11          $broken \leftarrow$ Check-Cycle $(C, s)$     /* does the flip break the cycle */
12          if $cost \leq oldcost$ and $broken$ = TRUE then
13              Accept-Move()
14              $oldcost \leftarrow cost$
15              $C = C - \{C_s\}$
16      if cost does not improve for $k$ moves break
```

Figure 7.18 Vertical constraint loop minimization algorithm

cycles is initialized to the empty set in line 1. For each region, a vertical constraint graph is constructed using the method described in Chapter 1. In line 4, any cycles found in the region's vertical constraint graph are added to the set of cycles. The initial cost is calculated in line 5. The loop from line 6 through 15 is a greedy algorithm which seeks to break cycles in the vertical constraint graphs similar to the one used in area minimization. Moves are accepted if they do not increase the total track density, and they break a cycle without creating a new one. The loop continues until all cycles are broken or further improvement appears unlikely.

7.3.10 Route Verification

The final step in global routing is a route verification stage. The route verification stage uses a depth-first search of each Steiner tree to see if all the necessary ports are connected. In addition, the search can detect cycles if they exist.

7.4 Algorithmic Complexity

Table 7.2 shows the time complexity for each step of the routing algorithm. Notice that all steps of the algorithm have polynomial time complexity. In addition, all steps have linear space complexity. The most expensive operation is linear assignment with $O(F^3)$ time complexity. This can be reduced to $O(F)$ by using divide-and-conquer techniques if suboptimal solutions are permissible.

Table 7.2 Algorithmic Complexity

Step	Time Complexity	Space Complexity
Region-Generation	$O(R^2)$	$O(R)$
Area-Minimization	$O(NS)$	$O(P)$
Assignment I	$O(F^3)$	$O(F)$
Assignment II	$O(F^3)$	$O(F)$
Remove-Cell-Overlap	$O(C\log C)$	$O(C)$
Cell-Swap-Optimization	$O(C)$	$O(C)$
Switchable-Segment-Opt	$O(N\mathrm{S})$	$O(P)$
Maze-Route	$O(P[\log V + E])$	$O(E+V)$
Vertical-Constraint-Min.	$O(NS)$	$O(P)$
Route-Verification	$O(V+E)$	$O(P)$

Legend: C = number of cells, E = number of edges in maze graph, F = number of feedthroughs in a single row/region, N = number of nets, P = max. number of pins of a single net, R = number of regions, S = number of net segments, V = number of vertices in the maze graph.

7.5 Sea-of-Gates Extensions

This global router has been extended to handle sea-of-gates arrays. In the sea-of-gates design style, routing is performed in the same area as the rows of transistors. If there is not enough area to complete the routing in a region, an entire row of transistors may be left unconnected. Since the cell level routing is eliminated when the transistors are left unused, more resources are available to complete the routing. However, this increases the area of the chip dramatically - just one additional routing track over the capacity of the row area causes another entire row to be allocated for routing.

In order to minimize the number of partially-filled rows, two modifications must be made. First, the height of the chip (line 3 of Figure 7.4) must be redefined to sum integral multiples of the row height in each region as

$$H = \sum_{r=1}^{numregions} \left\lceil \frac{trackheight(r)}{rowheight(r)} \right\rceil \cdot rowheight(r) \qquad (7.12)$$

where

$$trackheight(r) = trackpitch \cdot tracks(r) \qquad (7.13)$$

for a region *r*.

Secondly, a penalty term is incorporated to create a gradient for the case of states which have the same area. In Equation 7.14, a single track over the capacity of the

$${}_{sog} = \begin{cases} 0 & \text{if} \quad trackheight(r) \text{ mod } rowheight(r) = 0 \\ (trackheight(r) \text{ mod } rowheight(r) - rowheight(r))^2 & \text{otherwise} \end{cases} \qquad (7.14)$$

row region has the greatest penalty. This minimizes the number of partially-filled rows.

7.6 Row-based FPGA Extensions

In order to handle row-base field programmable gate arrays, the global router has been augmented to understand FPGA freeway maps. A freeway map is shown in Figure 7.19.

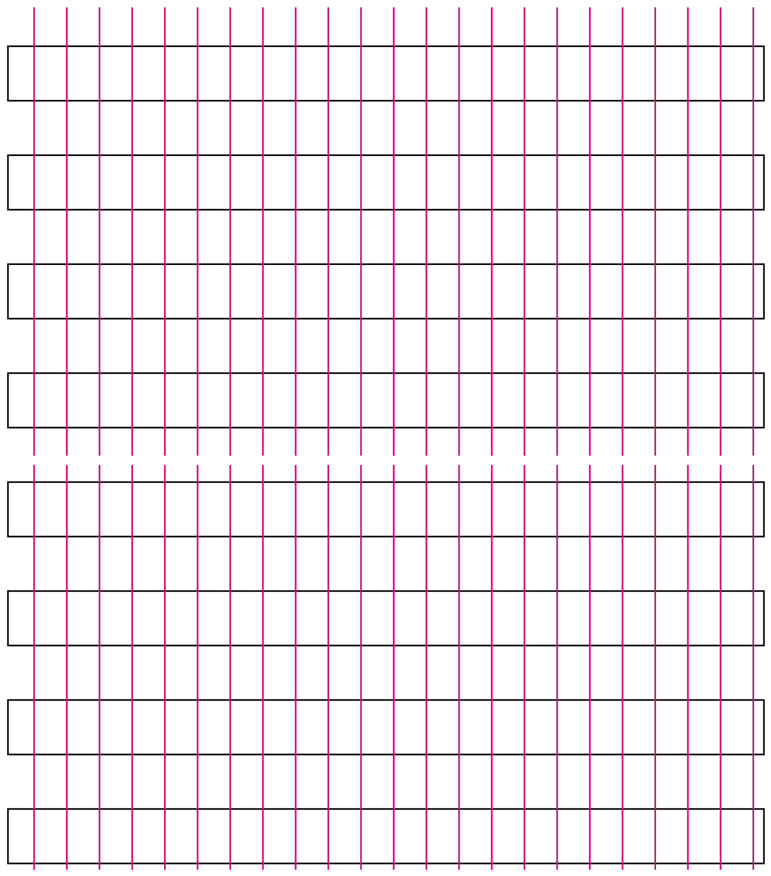

Figure 7.19 Example of freeway map.

The multiple row feedthrough assignment step comprehends feedthroughs which have more than two ports. In the case of an FPGA freeway, a port exists in every region that it intersects. The global router has also been modified so that output ports may span multiple rows.

In addition, the various FPGA pin-maps for a cell instance are optimized during area minimization through the use of instance version moves. An example of a Steiner tree for an FPGA circuit is shown in Figure 7.20.

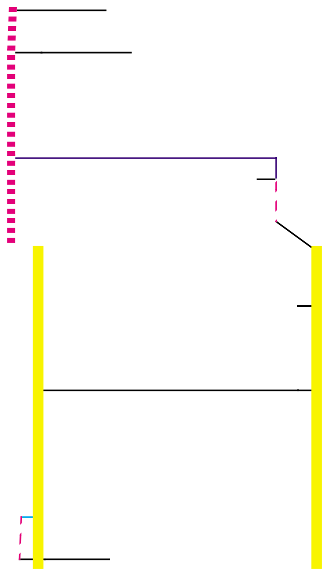

Figure 7.20 Example of a Steiner tree for a FPGA. Solid lines are freeways. Dashed line is a multiple row output pin.

7.7 Results

In order to determine the effectiveness of the global router for different technologies, we performed two experiments. In the first experiment, all implicit feeds were removed from the benchmark circuits. In the second, the implicit feeds were left in place. We compared the global routing results for the same placement in all cases.

Table 7.3 presents the results for the MCNC benchmark circuits where implicit feedthroughs were removed. The SGGR global router could not be compared because it cannot add explicit feeds. The version 6.0 global router outperformed all other global routers on the set of MCNC benchmark circuits requiring the insertion of feedthrough cells. In all cases, the new global router outperformed the previous version. Both the number of tracks and area were reduced.

Table 7.3 Results for the MCNC benchmarks - Explicit feeds case.

Circuit	TimberWolfSC version 6.0			TimberWolfSC version 7.0		
	width	tracks	area	width	tracks	area
sp1	5210	163	2.18×10^7	5200	160	2.16×10^7
sp2	11730	401	8.76×10^7	11600	374	8.34×10^7
guts	8344	1817	5.76×10^7	8300	1734	5.59×10^7

In the second case, (Table 7.4) the implicit feedthroughs in the circuits were utilized. In all cases, additional explicit feedthroughs were not necessary. Hence, the width of the design is fixed, and the track density determines the area of the design. Previously, SGGR produced the best results ever reported for the MCNC benchmark circuits. For every circuit, TimberWolfSC version 7.0 outperforms all other global routers.

Table 7.4 Results for the MCNC benchmarks. Track count for implicit feeds case.

Circuit	TimberWolfSC 6.0			SGGR			TimberWolfSC 7.0		
	avg	min	max	avg	min	max	avg	min	max
small	60	51	65	55	53	59	49	47	51
primary1	150	142	167	141	140	141	141	140	143
primary2	386	368	407	346	342	352	340	338	344

Notice from Table 7.5 that the new global router uses much less total wire length compared to SGGR. The latter program is based on maze routing and hence is not suitable for performance driven global routing.

Table 7.5 Results for the MCNC benchmarks. Wire length comparison for implicit feeds case.

Circuit	SGGR			TimberWolfSC 7.0		
	avg	min	max	avg	min	max
small	109719	109082	111474	95501	94280	96810
primary1	801217	794165	811950	736283	735290	737600
primary2	4257644	4234530	4276180	4007742	3987810	4017180

7.8 Conclusions

We have presented a new generalized row-based global router suitable for standard cell, gate-array, sea-of-gates, and FPGA integrated circuits. It is the first row based global router to explicitly minimize chip area. The global router uses adaptive Steiner trees to minimize chip area. Results were vastly improved over typical minimum wire length Steiner trees. This global router automatically adapts to technologies. In addition, optimal feedthrough placement is accomplished using linear assignment. Throughout the algorithm, timing constraints are taken into account. Furthermore, a unique vertical constraint loop minimization step eases the task for LEA channel routers. Finally, it has been shown that this global router outperforms other global routers for all the MCNC benchmark circuits which were tested.