ALGORITHMIC RESULTS ON PHYSICAL DESIGN PROBLEMS IN VLSI AND FPGA

BY

YACHYANG SUN

B.S., National Cheng Kung University, 1985
M.S., Northwestern University, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

Abstract

Various physical design problems in Very Large Scale Integrated (VLSI) circuits and Field-Programmable Gate Arrays (FPGA) are studied in this thesis. Specifically, we study (1) the area minimization problem in floorplans with L-shaped modules; (2) the channel routing problem in the single layer customization technology; (3) the area routing problem in the channelless single layer customization technology; (4) the design of new routing architectures and algorithms for FPGAs; (5) the congestion-balanced approach for the FPGA placement problem.

To my parents, Ming-Teh Sun and Yen-Bin Wang, and my wife, Christine Hewijin Jiau

Acknowledgements

I am deeply indebted to my advisor, Prof. C. L. Liu, for his exceptional guidance during the course of my research. I also thank Prof. R. Gupta and Prof. S. Kang for serving on my committee.

I also thank the members of my research group, Cheng Chang, Ki-seok Chung, Jason Cong, Sai-Keung Dong, Tong Gao, Ran Libeskind-Hadas, Nany Hasan, Taewhan Kim, Anmol Mathur, Unni Krishnan Narayanan, Chaeryong Park, Piechen Pan, Too-Seng Tia, and Xian-Jin Yao, for their suggestions, discussions and all the good times we had together. I also want to thank Chinese student basketball and volleyball clubs of this town for the happiness we shared with.

I am grateful to Mr. Shinji Sato at Fujitsu, Dr. Ting-Chi Wang at Lsi Logic, Dr. C. K. Wong at T. J. Watson research center in IBM for their helpful suggestions and comments on my work.

Especially thanks to my parents and my wife for their encouragement, support, and love throughout these many years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Integrated circuit design is usually divided into three design steps, namely, *functional design*, *logic design* and *physical design*. Functional design defines the behavioral specification of the circuit in terms of a functional representation. Logic design transforms the functional representation into a minimum logic representation. Physical design transforms the technology-independent logic representation into a geometric representation. The top-down hierarchical physical design approach which is adopted and studied extensively in the context of Very Large Scale Integrated (VLSI) circuits and Field-Programmable Gate Arrays (FPGA) can be divided into the following four stages:

1. *logic partitioning and technology mapping* determines how a logic representation is to be partitioned into *logic cells* and *interconnections*. In VLSI, these logic cells are circuit modules which are well-defined logical entities, and can be implemented using a layout style which is suitable for a specific integrated circuit technology. In FPGA, these logic cells can be as simple as 2-input NAND gates or more complex, such as multiplexers or lookup tables. The functions of these FPGA logic cells can be defined by programming

the provided programmable switches. The interconnections are a net-list connecting these logic cells.

2. *floorplanning and placement* generates the relative positions of the logic cells based on their interconnections and dimensions.

3. *global wiring* (approximately) interconnects the logic cells by deciding the rough path for each interconnection and *pin assignment* assigns pin positions to each logic cell based on the global wiring result.

4. *detailed wiring* completes the interconnections and *cell generation* transforms each logic cell into transistors and wires in the case of VLSI and programs the function of each logic cell in the case of FPGA.

In the thesis, we shall examine some of the problems arising in the four stages stated above. Specifically, we shall examine

- the area minimization problem in floorplans with L-shaped modules in Chapter 2. Given a floorplan with L-shaped modules, we want to choose an implementation from a set of possible implementations for each module such that the resultant floorplan will have minimum area, subject to the constraint that the topological relations between the modules of the floorplan must be preserved. This problem is known to be NP-complete. Consequently, it is unlikely that a polynomial time algorithm for solving the problem optimally can be found. However, for a special case of floorplans with rectangular regions, there is an efficient algorithm for an optimal solution of the corresponding area minimization problem [48]. In this chapter, we present an extension of some of the concepts embodied in this algorithm.

- the channel routing problem for the single layer customization technology in Chapter 3. It is known that using one metal layer for customized wiring reduces the fabrication turn-around time significantly, without appreciable sacrifice in performance and integration [35, 61]. However, since there is only one customized layer, the limited routing flexibility in this technology makes the routing problem more challenging than that in the conventional two-layer Manhattan routing. In this chapter, we prove that this problem is NP-complete. Then, we propose an effective algorithm that guarantees the generation of a routing solution, if one exists. The routing solution generated also guarantees a minimum number of vias used.
- the area routing problem for the channelless single layer customization technology in Chapter 4. Gate arrays with fixed routing channels in single layer customization technology suffer from inefficient usage of silicon and low gate utilization rate as in conventional gate arrays. To overcome such drawbacks, a channelless gate array architecture in single layer customization technology was introduced [13]. In this architecture, there is no reserved area for channel routing. This chapter formulates the routing problem for such an architecture. We prove that the problem is NP-complete, suggesting that the existence of a polynomial time algorithm for solving the problem optimally is unlikely. We also propose a heuristic area routing algorithm which captures special features of this particular architecture.
- the routing architectures and algorithms for FPGA in Chapter 5 and Chapter 6. In Chapter 5, we propose a new FPGA routing architecture with fixed orthogonal wire segments and field programmable switches at the intersections of the wire segments. In comparison with the conventional two dimensional FPGA routing structure, this routing architecture

has the advantage of using a smaller number of programmable switches. An existing Field-Programmable Interconnect Chip (FPIC) routing structure can also be included as a special case in our class of routing structures. Using a probabilistic model, we show the effectiveness of this routing architecture. We present a sequential routing algorithm based on the solution of the single net routing problem. We take into account the delay introduced by the programmable switches on a routing path and formulate the single net routing problem as a Node-Weighted Steiner Minimum Tree (NWSMT) problem in a bipartite graph G. Since our single net routing problem is NP-complete, a polynomial time approximate algorithm is proposed. We show that our single net routing algorithm produces an optimal solution for some special classes of bipartite graphs. In general, the solution obtained by our algorithm has a performance bound. In addition, we also prove that it is NP-complete to determine a solution that approximates the optimal solution within any constant bound. In Chapter 6, we propose another new FPGA and FPIC routing architecture which improves upon the one proposed Chapter 5 by providing continuing switches along the horizontal and vertical wire segments. These continuing switches can be activated to connect two horizontal/vertical segments to form a longer horizontal/vertical segment. The addition of continuing switches leads to higher routability and better timing performance than that for the routing architecture in Chapter 5. A two-phase routing algorithm for this new routing architecture is developed. Both the initial routing phase and the rip-up and reroute phase employ a dynamic programming technique. The rip-up and reroute phase can also be applied to the segmented channel routing problem for row-based FPGA routing structures.

- the congestion-balanced approach for the FPGA placement problem in Chapter 7. In this chapter, we propose to use routing congestion as a criterion in solving the FPGA placement problem. Due to limited routing resources, complete routing for a given FPGA placement might not be feasible. The effect of channel congestion has been overlooked in existing placement algorithms as most of them were developed for standard cell and macro cell technologies where a given placement is always routable. Based on the notion of *congestion balance*, we propose a placement algorithm that spreads out evenly routing congestion while minimizing the total wire length. This algorithm is based on the min-cut strategy which balances the numbers of interconnections in the two portions during each application of the min-cut partitioning procedure. The algorithm also minimizes the *actual* number of nets crossing horizontal and vertical cut-lines instead of using an estimated number of interconnects as was commonly done. We also propose an alternative way of employing the cut-line that allows the examination of a larger number of solutions to enhance the quality of the final solution. It also obviates the need for conventional terminal propagation techniques that are inherently less accurate in estimating net congestion. Based on our experiments, we propose a novel ordering according to which the cut-lines are applied. The ordering is related to the distance of a cut-line from the center of the chip. The cut-line closer to the center is applied earlier. This avoids the occurrence of a cut-line that has a large cut size and is close to the center of the layout area (as it often happens in a traditional top-down min-cut based placement).

Chapter 2

Area Minimization Problem for Floorplans with L-shaped Regions

2.1 Introduction

Floorplanning is an early and, thus, important step in the physical design process of VLSI circuits. A floorplan is a division of the chip area into regions such that each region is large enough to accommodate the circuit module (or module for short) assigned to the region. A module is said to be *rigid* if it has fixed dimensions and is said to be *flexible* if it has two or more possible implementations in terms of dimensions. An optimal floorplan is one with minimum total area. (Other objective functions have been introduced to measure the quality of floorplans, we shall restrict our discussion to the area measure.) To minimize the total area, a floorplan design algorithm needs to select an implementation among all alternatives for each flexible module. There is an extensive literature on floorplan design algorithms [9, 33, 36, 37, 42, 48, 55, 56, 57, 58]. However, most of the existing algorithms only deal with floorplans with rectangular regions:

- For slicing floorplans, Otten [36, 37] and Stockmeyer [48] proposed an optimal algorithm to solve the area minimization problem in polynomial time.
- For non-slicing floorplans, the area minimization problem has been proved to be NP-complete by Stockmeyer [48]. A branch-and-bound algorithm was proposed by Wimer et al. [57]. Recently, Pan et al. [39] proposed a more efficient algorithm.
- Wang and Wong [55] generalized the technique used in [48] to obtain an optimal algorithm for a special class of non-slicing floorplans, called wheels.

As was pointed out by Wong and Liu [58], the inclusion of L-shaped regions in a floorplan provides additional flexibility in achieving the goal of minimizing the total area of a floorplan. This chapter considers a new area minimization problem of floorplans with L-shaped regions. We formulate this area minimization problem in Section 2.2, and propose an efficient algorithm in Section 2.3. Section 2.4 reports and discusses some of our experimental results.

2.2 Preliminaries and Problem Definition

For the rest of this chapter, by an L-shaped module or region we shall include a rectangular module or region as a special case. A *floorplan* is a dissection of a rectangle into non-overlapping L-shaped regions. Since we regard a rectangular region as a special case of an L-shaped region, we call a floorplan in which all non-overlapping regions are rectangular a *rectangular floorplan* for the purpose of distinction. The non-overlapping regions are referred to as *basic blocks*. Figure 2.1 shows a floorplan with 8 basic blocks. A region consisting of more than one basic

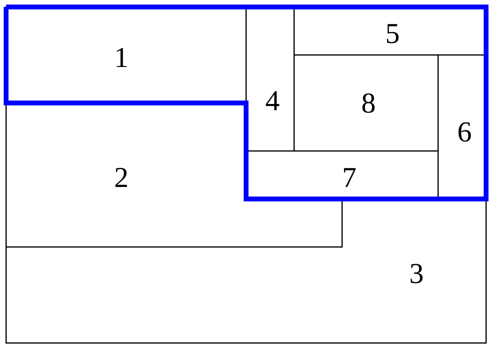


Figure 2.1: An example of a floorplan with L-shaped regions

block is called a *super-block.* The L-shaped area bounded by bold lines in Figure 2.1 is a super block consisting of 6 basic blocks. We shall use *block* to refer to either a super-block or a basic block, if the distinction is not needed. As was pointed out in [48] and [49], we follow the convention that two dissecting lines can meet, but not cross, since each crossing can be replaced by two $\top$'s as shown in Figure 2.2.

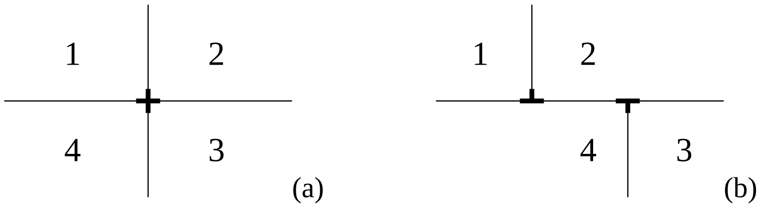


Figure 2.2: Illustration of the replacement of a cross in (a) by two $\top$'s in (b)

A floorplan can be regarded as a plane graph[1] with the edges oriented either vertically or horizontally and with the vertices being the intersection of horizontal and vertical dissecting line segments. A *vertical* (*horizontal*) *cut line* is a vertical (horizontal) path whose two extreme vertices (vertices of degree 1) are external vertices (vertices on the exterior face of the embedding). A *slicing rectangular floorplan* is a rectangular floorplan that can be recursively

[1]A plane graph is a planar graph with an embedding on a plane.

decomposed into basic blocks by vertical and horizontal cut lines. The hierarchy of such a decomposition can be represented by a binary tree called a *rectangular slicing tree.* Figure 2.3 shows a rectangular floorplan and its corresponding rectangular slicing tree, where a "+" ("*") in the slicing tree represents a horizontal (vertical) cut line.

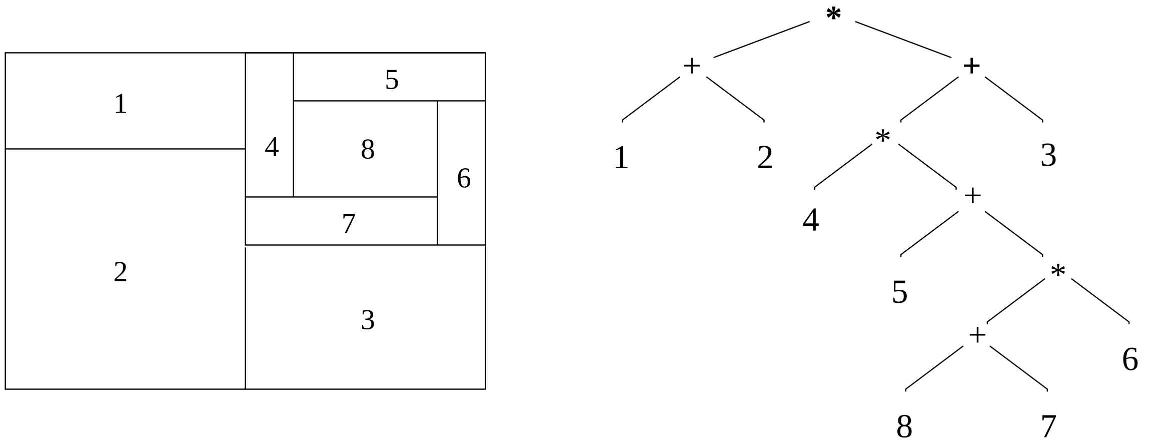


Figure 2.3: An example of a rectangular floorplan and its corresponding rectangular slicing tree

In the literature [9, 33, 48, 56], a pair of planar acyclic directed graphs is used to capture the topological relation between the regions of a floorplan, which must be preserved during the area minimization process. These two graphs are called *vertical* and *horizontal constraint graphs* in [9], *block neighboring graphs* in [33] and *dual polar graphs* in [48, 56]. [42] combines these two graphs into one and calls it a *relative position graph.* For a given floorplan F, we construct a graph V_F which specifies the "above-relation" between the regions as follows. There is a vertex in V_F corresponding to each horizontal line segment, including the top and the bottom sides of the enclosing rectangle of F. Two vertices h_i and h_j in V_F are connected by a directed edge from h_i to h_j if (part of) the segment corresponding to h_i and (part of) the segment corresponding to h_j are horizontal sides of a basic block with the segment corresponding to h_i being above that corresponding to h_j. A graph, H_F, which specifies the "to-the-left-relation", is defined analogously for the vertical segments. Two floorplans are said to have *the same topological relation* if they have the same dual polar graphs. Figure 2.4 shows the graphs V_F and H_F of the floorplan in Figure 2.1.

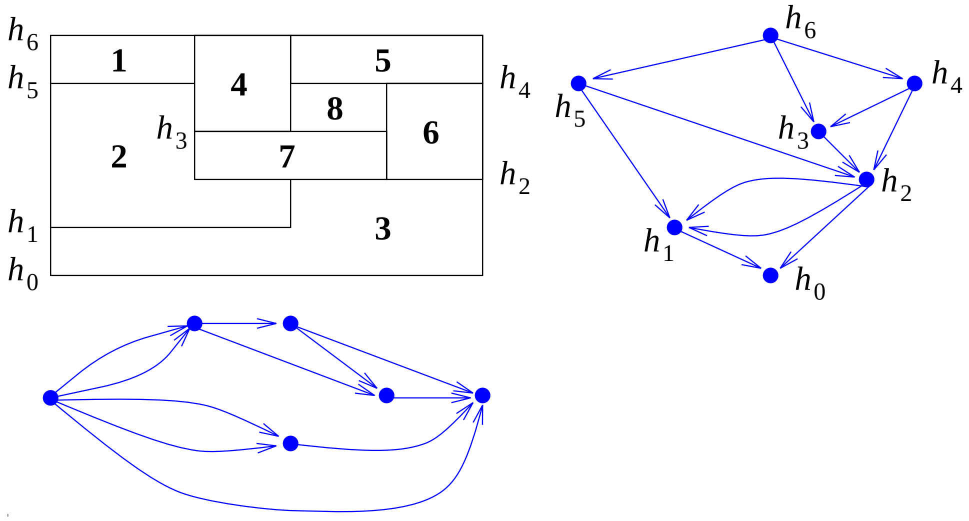


Figure 2.4: The floorplan in Figure 2.1 and its dual polar graphs

Let F be a floorplan and V_F and H_F its dual polar graphs. Suppose we have already chosen an implementation among all possible implementations for each module. Then, the minimum area of F for this particular implementation choice can be calculated using the dual polar graphs as follows. Suppose edge e in V_F is directed from h_i to h_j because there is a basic block in F of which h_i and h_j are horizontal sides. We set the weight of edge e to be the corresponding dimension of the chosen implementation of the module which is to be placed in the basic block. Let the coordinates of the bottom left corner of the floorplan be (0, 0). Then, the y-coordinate (x-coordinate) of each horizontal (vertical) line segment is equal to the longest distance from the corresponding vertex in V_F (H_F) to the vertex corresponding to the bottom (left) boundary. The coordinates of the line segments define completely a corresponding floorplan which is denoted by F'. The following properties of F' can be proved easily: (See [56] for details.)

1. F' has the same dual polar graphs as F;

2. The *height* (*width*) of F' is the longest path in V_F (H_F). The *area* of F' is, therefore, the product of the lengths of the longest paths in V_F and H_F.

In this chapter, we consider the following area minimization problem :

Given a floorplan F with L-shaped basic blocks and a set of possible implementations for each module, determine the implementation of each module so that the area of F' is minimized.

There is a very limited literature on area minimization algorithms for floorplans with L-shaped regions. Sun and Sarrafzadeh [49] proposed a polynomial time algorithm to test if a given adjacency graph that specifies the required adjacency relation between regions admits a floorplan with L-shaped regions. A polynomial time algorithm was also proposed to construct a floorplan, if one exists. Nevertheless, the area minimization problem was never addressed.

In [58], a simulated annealing algorithm was proposed which enumerates a large number of floorplans with L-shaped regions. Postfix expressions were used to represent floorplans. An operand in a postfix expression corresponds to a basic block and an operator defines the way in which two blocks are to be assembled. Their algorithm determines the shape and dimensions of the composite of two blocks according to the operator and the dimensions of these two blocks. However, how to find a postfix expression for a given floorplan is not easy and never addressed in the paper. Moreover, for a given postfix expression, basic blocks of the same shape but different dimensions may yield floorplans with different dual polar graphs. Figure 2.5 shows two floorplans with different dual polar graphs due to the different heights of block B, where the

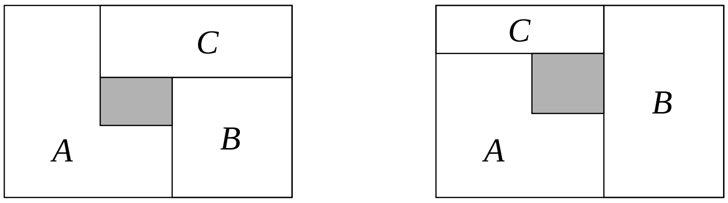


Figure 2.5: Different topological relations among regions

shaded regions are added dead space. Therefore, their algorithm can not handle the situation in which the topological relation among modules in terms of dual polar graphs is pre-specified.

Since the problem we consider is NP-complete even when all the regions in the given floorplan are rectangular [48], it is unlikely that a polynomial time algorithm for the solution of our problem exists. In this chapter, we provide an effective algorithm to solve this problem optimally. Our approach can be summarized as follows.

1. A polynomial time algorithm is developed to decompose a given floorplan F and represent this decomposition as a cut tree which is a generalization of a rectangular slicing tree define earlier.

2. By traversing the cut tree in a bottom-up fashion, we compute the areas of the irredundant composites of blocks while maintaining the dual polar graphs as invariants.

3. Among all irredundant candidates for the floorplan with minimum area, we choose the one with the smallest area as the solution.

The detailed description of each step is provided in the next section.

2.3 An Area Minimization Algorithm

In this chapter, an optimal algorithm is proposed to solve the area minimization problem defined in Section 2.2. As we mentioned earlier, a floorplan can be regarded as a plane graph with the edges being vertical and horizontal line segments. We define a *cut line* of a floorplan to be a connected component of the plane graph whose extreme vertices are external vertices. Note that our definition of a cut line extends the definition of a vertical (horizontal) cut line that is a vertical (horizontal) "path" in the plane graph with its extreme vertices being external vertices. We shall define seven additional types of cut lines that can be used to decompose floorplans with L-shaped regions, besides the vertical and the horizontal cut lines. These cut lines are

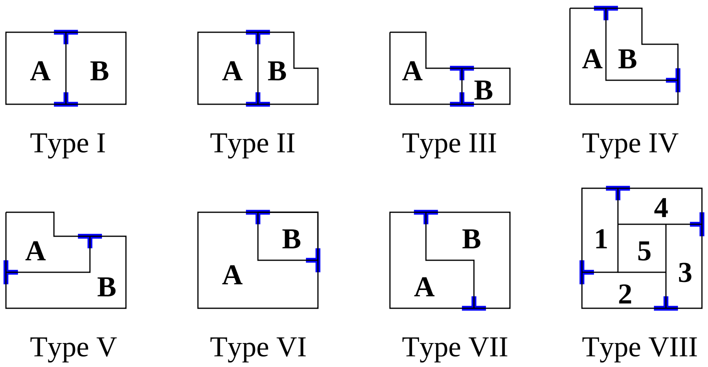


Figure 2.6: Eight types of cut_lines with symmetric cases omitted

shown in Figure 2.6 with symmetric cases omitted. Type-I cut lines are horizontal cut lines and vertical cut lines defined earlier. A Type-I, II, or III cut line is called a *straight* cut line. Note that Type-II through Type-VIII cut lines preserve the following property of vertical and horizontal cut lines:

For each extreme vertex of a cut line, the orientation of the $\top$ *is such that the* $180°$ *angle is toward the external region.*

A Type-VIII cut line is called a *wheel* in [55]. A cut line of each type, except a Type-VIII cut line, decomposes a rectangular super-block into two smaller L-shaped blocks. A Type-VIII cut line decomposes a rectangular super block into five smaller L-shaped blocks. We call these smaller L-shaped blocks *constituent blocks*. A *decomposable floorplan* is a floorplan that can be recursively decomposed into basic blocks using these eight types of cut lines. The decomposition can be described by a rooted ordered tree[2], called a *cut tree*, which has the following properties:

1. Each leaf corresponds to a basic block.

2. Each sub-tree rooted at a nonleaf node corresponds to a super-block.

3. Each nonleaf node is labeled with the type of the cut line that is used to decompose the corresponding super-block. It has either two or five children depending on the type of the cut line, each of which corresponds to a constituent block.

A cut tree for the floorplan in Figure 2.1 is shown in Figure 2.7.

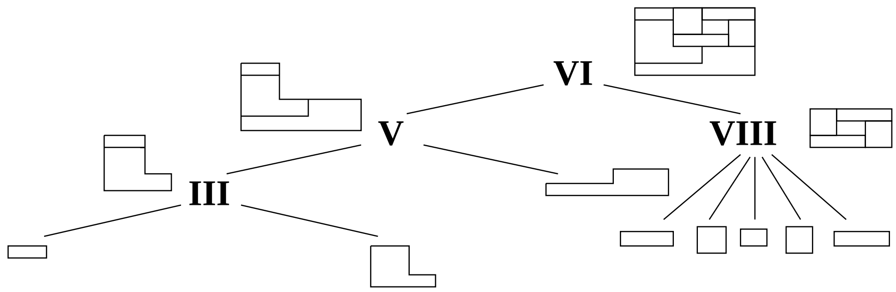


Figure 2.7: The cut tree for the floorplan in Figure 2.1

For a given floorplan F, an $O(n^3)$ time algorithm to test whether F is decomposable using the eight types of cut lines is described in Section 2.3.1. There is no prior work on the decomposition problem for floorplans with L-shaped regions, although decomposability of a rectangular

[2]An ordered tree is a tree such that interchanging the sub-trees of a node yields a different tree

floorplan using straight cut lines can be easily tested. If the algorithm answers YES, a corresponding cut tree is constructed. For a decomposable floorplan, Section 2.3.2 presents an algorithm for selecting an implementation for each module. For floorplans that are not decomposable, we extend our algorithm by incorporating a branch-and-bound search. Our algorithm guarantees minimum total area and experimental results indicate that its running time is very short, although the problem is NP-complete.

2.3.1 A Decomposition Algorithm

In this section we present an $O(n^3)$ time algorithm to test whether a given floorplan is decomposable using the cut lines defined earlier. In general, a floorplan can be decomposed in more than one way. An example is shown in Figure 2.8. Our algorithm guarantees to find one

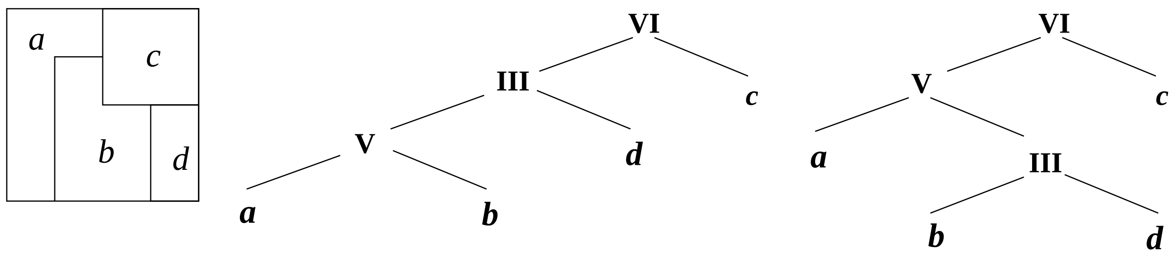


Figure 2.8: A floorplan with two different cut trees

successful decomposition and constructs a cut tree accordingly. Consider the floorplan shown in Figure 2.9. If cut line $\bar{l}$ is used first as shown in Figure 2.10(a), then it is impossible to separate basic blocks a and b. On the other hand, if cut line l is used first, then we can decompose the floorplan as shown in Figure 2.10(b). Figure 2.11 shows another example. Therefore, choosing a wrong cut line may render the resultant super-blocks indecomposable. Nevertheless, we have the following observations:

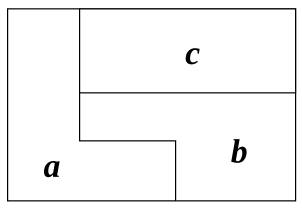


Figure 2.9: A simple floorplan with only 3 regions

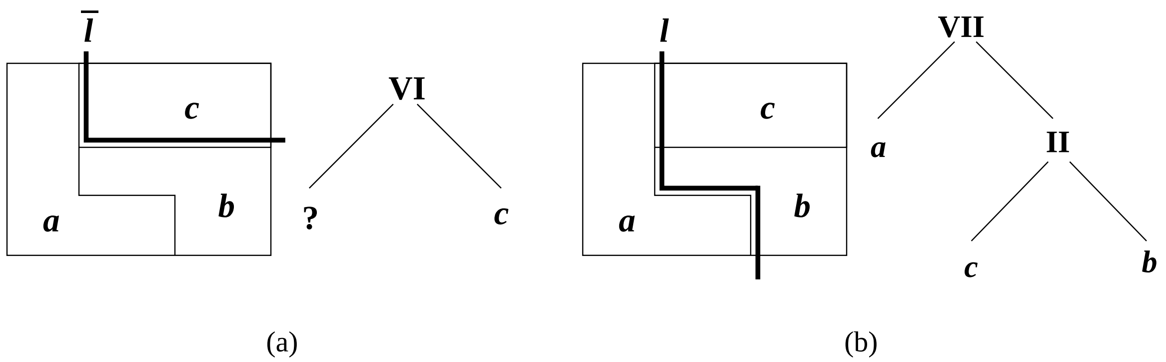


Figure 2.10: An unsuccessful and a successful attempts to carry out a decomposition

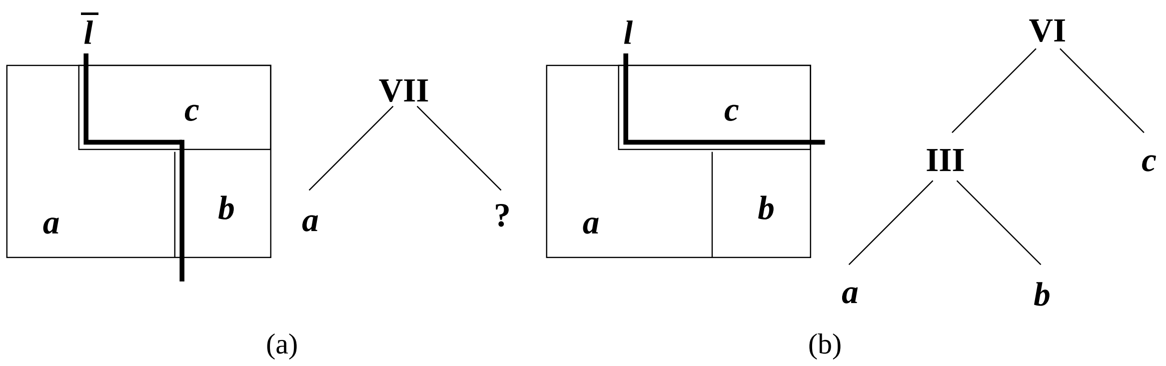


Figure 2.11: A situation similar to that in Figure 2.10

Observation (1). It is easy to verify that if a straight cut line (*i.e.* Type-I, II, and III) l can be applied to a super block f, then using cut line l prior to cut lines of other types will not change the possibility to decompose f. For example, the floorplan in Figure 2.12 (a) is decomposable according to the cut tree in (c). However, choosing the straight cut line first as in (b) will also lead to a decomposition. Consequently, our algorithm always uses a cut line of Type-I, II, or III first to decompose f, whenever it is possible.

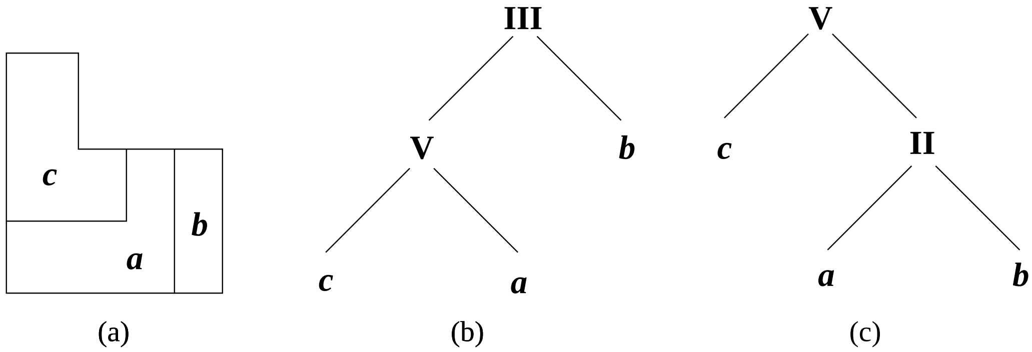


Figure 2.12: Illustration of the effect of choosing a straight cut-line

Observation (2). The property mentioned in Observation (1) holds for a Type-VIII cut line.

Observation (3). Consider a super-block f that has more than one cut lines of Type-x, where $x \in \{\text{IV, V, VI}\}$. Each figure in Figure 2.13 has, for example, two cut lines of the same type. Now, if we draw a line from the lower left corner of f to the upper right in Figure 2.13, the cut line always encountered first is called the *longest* cut line, which is shown as a dotted line in the figure. The longest cut line can be defined analogously for other symmetric cases where f is flipped and/or rotated. Examining each such f shows that choosing the longest cut line will not affect the possibility to decompose f, whereas choosing any other might. Based on this observation, our algorithm always chooses the longest one, whenever there are more than one cut lines of Type-IV, V, or VI.

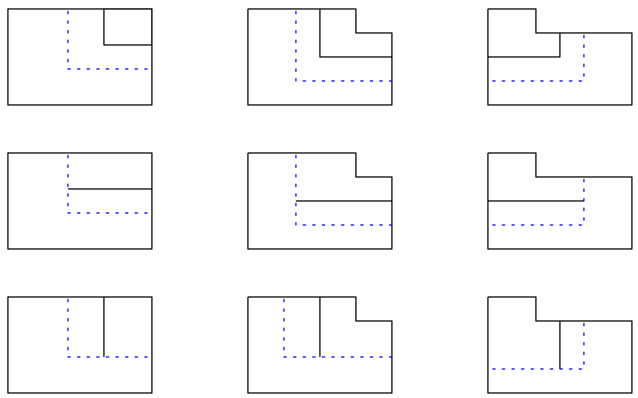


Figure 2.13: Longest cut-lines

Observation (4). For an L-shaped super-block f that is not rectangular, Type-IV and V are the only two types of cut lines that can decompose f into two L-shaped blocks as shown in Figure 2.14. Note that the cut lines in Figure 2.14(a) and (c) are of the same type but

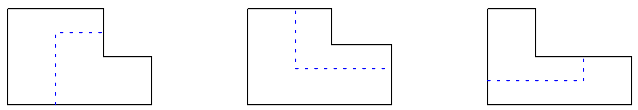


Figure 2.14: Three possible cut lines for an L-shaped region

have different orientations. However, it is easy to verify that only one of these three can exist at any time. A crossing will appear in f, otherwise. Therefore, no decision has to be make on which one of the above three to be used to decompose an L-shaped super-block.

Our algorithm written in pseudo code is shown in Table 2.1 and 2.2. It consists of two recursive procedures, *rectangle* and *L_shape*.

Procedure *rectangle* (*L-shape*) accepts a rectangular (L-shaped) block as input, and returns the *successful* signal. if the input block is decomposable. If the input block is not decomposable, then *rectangle* prints the message and stops the execution; whereas *L-shape* returns the

Procedure *rectangle*(f)

/* f is a rectangular block. */

/* This procedure returns *successful*, if f is decomposable; stops, otherwise. */

0. *if* f is a basic block,

then return(*successful*);

1. *if* there exists a Type-I or Type-VIII cut line in f,

then

1.1 apply arbitrarily one Type-I or Type-VIII cut line;

1.2 *for* each constituent block X *do*

rectangle(X);

1.3 *return*(*successful*);

2. *if* there exist a Type-VI cut line in f,

then

/* There are four possible orientations for a Type-VI cut line. */

for each possible orientation *do*

2.1 apply the Type-VI cut line, if there is exactly one;

2.2 apply the longest Type-VI cut line, if there are more than one;

/* Let A and B be the L-shaped and rectangular constituent block, respectively. */

2.3 *if* both *L-shape*(A) and *rectangle*(B) are *successful*,

then return(*successful*);

3. *if* there exists a Type-VII cut line in f,

then

for each Type-VII cut line *do*

/* Let A and B be the two constituent blocks. */

if both *L-shape*(A) and *L-shape*(B) are *successful*,

then return(*successful*);

4. *print* ("Not decomposable!") and *stop*;

Table 2.1: Procedure *rectangle*

Procedure *L-shape*(f)

/* f is a L-shaped block. */

/* This procedure returns *successful*, if f is decomposable; *unsuccessful*, otherwise. */

0. *if* f is a basic block,

then return(*successful*);

1. *if* there exists a Type-II or Type-III cut line in f,

then

/* Let l_A and l_B be the L-shaped and rectangular constituent block, respectively, for each Type-II and Type-III cut line l. */

1.1 apply one Type-II or Type-III cut line l whose corresponding l_A does not have any cut line of the same type as l;

1.2 *if L-shape*(l_A) is *successful*,

then return(rectangle(l_B)*)*,

else return(*unsuccessful*);

2. *if* there exists a Type-IV (Type-V) cut line in f,

then

2.1 apply the Type-IV (Type-V) cut line, if there is exactly one;

2.2 apply the longest Type-IV (Type-V) cut line, if there are more than one;

2.3 *for* each constituent block X *do*

L-shape(X);

2.4 *return*(*successful*);

3. *return*(*unsuccessful*);

Table 2.2: Procedure *L-shape*

unsuccessful signal. We shall describe briefly the steps in these two procedures below. Both procedures return the *successful* signal, if the input block is a basic block. Based on Observation (1), when procedure *rectangle* detects either a Type-I or a Type-VIII cut line, it will use an arbitrary one to decompose f in Step 1. Then, *rectangle* is called recursively for each constituent block to check if it is decomposable. If no Type-I or Type-VIII cut line was found in Step 1, *rectangle* checks the existence of Type-VI cut lines and use one (the longest one, if there are more than one such cut lines). If no Type-VI cut line leads to a successful decomposition, Step 3 is executed, instead of returning the *unsuccessful* signal. In Step 3, each Type-VII cut line will be used and test if there is any such that both the L-shaped constituent blocks are decomposable. If there is one, then return the *successful* signal. Otherwise, "Not decomposable!" message will be printed and the algorithm is stopped in Step 4.

As Step 1 of procedure *rectangle*, procedure *L-shape* tries to find Type-II and Type-III straight cut lines in the first step. If there are more than one cut line of Type-x, $x \in \{$II, III$\}$, procedure *L-shape* chooses the one cut line such that there is no Type-x cut line in the resultant L-shaped constituent block. Then, the L-shaped constituent block is examined by calling *L-shape* recursively. The rectangular constituent block is examined only if the L-shaped constituent block is decomposable. The *unsuccessful* signal will be returned, otherwise. Step 2 of *L-shape* looks for Type-IV and Type-V cut lines. As stated in Observation (3), a cut line of one type of these two can not co-exist with a cut line of the other type. We use the longest one, if there are at least two cut lines of a type according to Observation(2). For each constituent L-shaped block, check the decomposability and return the *successful* signal if each one is decomposable. Return the *unsuccessful* signal, otherwise.

Theorem 1 *Let* F *be a floorplan with L-shaped regions.* rectangle(F) *determines if* F *is decomposable correctly.*

Proof: Suppose *rectangle*(F) returns the *successful* signal. Then, we can construct a cut tree by tracing the recursive calls of *rectangle* and *L-shape* involved. Therefore, F is decomposable. To prove the other direction, suppose *rectangle*(F) returns the *unsuccessful* signal. Note that Step 4 of *rectangle* is the only step that the indecomposability of F is reported. We assume f be the input rectangular block of the procedure call, *rectangle*, in which Step 4 is executed. According to Procedure *rectangle*, we have

1. f is a rectangular super-block;

2. there is no Type-I nor Type-VIII cut line that can be used to decompose f into smaller blocks;

3. there is no Type-VI cut line or there are some, but none of them can be used to decompose f into a decomposable L-shaped constituent block and a decomposable rectangular constituent block in Step 2.3;

4. there is no Type-VII cut line or there are some, but none of them can be used to decompose f into two decomposable L-shaped constituent blocks.

Suppose to the contrary that F is decomposable and T_F is a cut tree. In that case, f must not correspond to any subtree in T_F. Assume that f is cut into two parts by a cut line l at some internal node v of T_F. The rest of the proof proceeds according to the type of the cut line l:

Type-I, Type-II, and *Type-III:*

In this case, (part of) cut line l is also a Type-I cut line in f, contradicting to the second property of f listed above.

Type-IV:

Since there is no crossing in f, Figure 2.15 shows the only two possible configurations, where

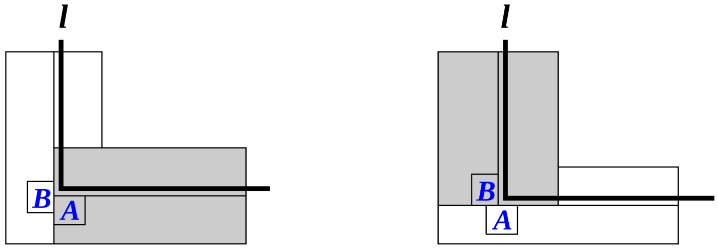


Figure 2.15: Case Type-IV

the shaded regions are maximal rectangular super blocks. Note that no cut line can be used to separate blocks A and B which contradicts the assumption that T_F is a cut tree of F.

Type-V:

Since there is no crossing in f, there are three possible configurations as shown in Figure 2.16,

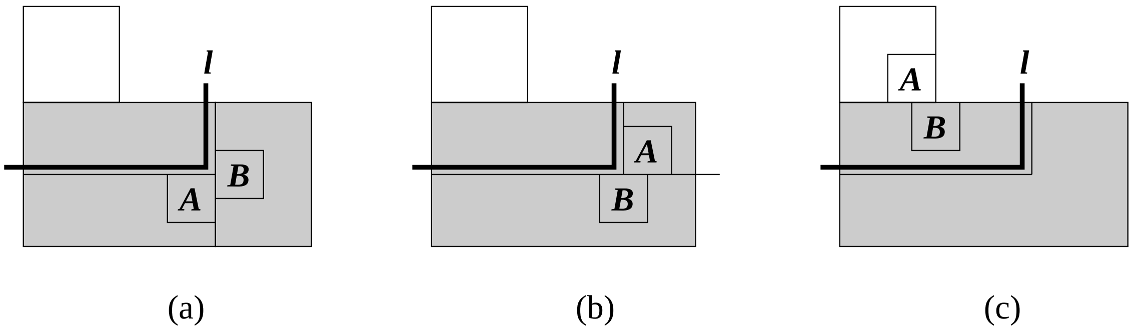


Figure 2.16: Case Type-V

where the shaded regions are maximal rectangular super blocks. As in the case of a Type-IV cut line, no cut line can be used to separate blocks A and B which contradicts the assumption that T_F is a cut tree of F.

Type-VI:

In this case, the subtree rooted at v corresponds to a super block consisting of three decomposable blocks, two being L-shaped and the other one being rectangular, as shown in Figure 2.17 where the shaded maximal rectangular super block appears in (b), but not (a). However, this

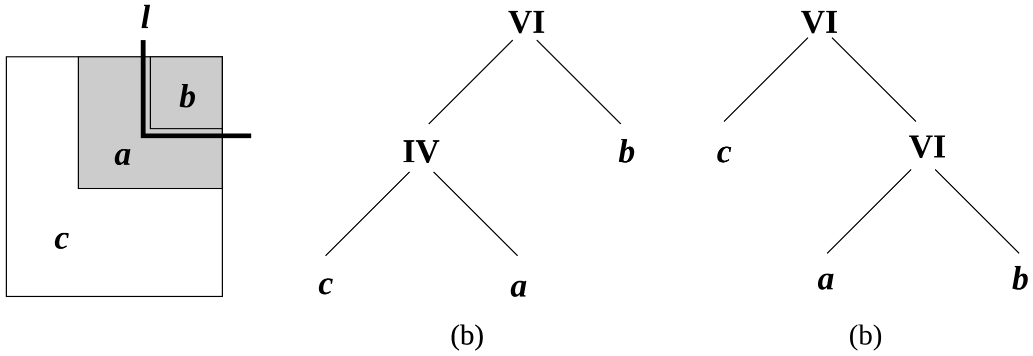


Figure 2.17: Case Type-VI

contradicts to the third property of f listed above.

Type-VII:

To avoid crossings, the only possible configurations are shown in Figure 2.18. As in Type-IV and Type-V cases, no cut line can be used to separate blocks A and B in Figure 2.18(a), 2.18(b), and 2.18(c) which contradicts the assumption that T_F is a cut tree of F. Figure 2.18(d) contradicts the fourth property of f.

Type-VIII:

It is impossible for a Type-VIII cut line to cut f into two parts without introducing a crossing. Therefore, it is impossible to have a cut line l that cut f into two parts. The theorem is proved by contradiction. $\square$

To analyze the time complexity of our algorithm, we need the following definition. For a super-block f, let $rec(f)$ denote the number of cut lines which were attempted in the decomposition of f and let $|f|$ denote the number of basic blocks in f. We have the following theorem:

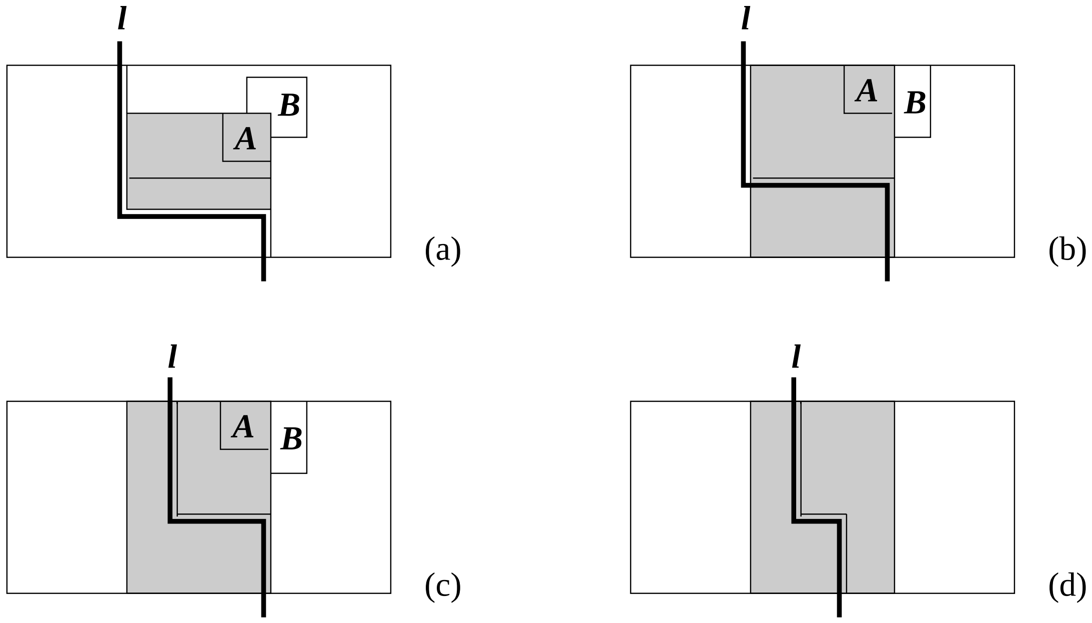


Figure 2.18: Case Type-VII

Theorem 2 $rec(F) = O(|F|^3)$.

Proof: We count $rec(F)$ by tracing the cut tree, T_F, constructed during the decomposition process. Let v be an internal node of T_F such that

(1). the subtree rooted at v corresponds to a rectangular super-block;

(2). all the descendants of v in T_F correspond to L-shaped super-blocks.

We use f to denote the rectangular super-block corresponding to the subtree rooted at v. If the first cut line chosen in T_F to cut f is a Type-I or Type-VIII cut line, then $rec(f) = 1$ by the definition of v. If the first cut line chosen in T_F to cut f is not a Type-I nor Type-VIII cut line, in the worst case, procedure *rectangle* has to enumerate all possible cut lines of other types for the first cut line to decompose f. The number of possible candidates attempted for the first cut line is at most $4 + O(|f|^2) = O(|f|^2)$, since there are at most 4 (longest)

Type-VI cut lines and there are at most $O(|f|^2)$ Type-VII cut lines, based on the fact that the number of vertices where each of the two extreme vertices of a Type-VII cut line can be located is bounded by $\mathrm{O}(f)$. Subsequent cut lines will be determined uniquely according to Observation (3). Therefore, $rec(f) = \mathrm{O}(|f|^3)$. Note that since *rectangle* either decomposes the input rectangular block into basic blocks successfully or reports the indecomposability of the given floorplan and stops the whole process, each rectangular super-block in F can be the input of *rectangle* at most once. Therefore, the counting above for f will contribute at most once in the $rec(F)$. We can replace f by a rectangular basic block and repeat the argument to the ancestor until the root of T_F is reached. The summation of all the cubic terms accounting for the intermediate rectangular super-blocks is bounded by $\mathrm{O}(|F|^3)$. $\square$

2.3.2 An Algorithm for Selecting the Implementations

The size of an L-shaped module is specified by six dimensions, d_1, d_2, d_3, d_4, d_5, and d_6, as shown in Figure 2.19. Although we can compute d_5 and d_6 from d_1, d_2, d_3, and d_4 ($d_5 = d_1 - d_3, d_6 = d_2 - d_4$), it is more convenient in our algorithm to use all six dimensions. The size of a rectangular module are specified by its height and its width. Suppose that we obtain a cut tree from the decomposition algorithm and each module has a number of possible implementations. Since the implementations differ in dimensions, our problem is that of choosing an implementation for each module such that the area of the resultant floorplan is minimum.

The following definitions are needed to state our algorithm. Let d_i^X, where $i = 1, 2, 3, 4, 5, 6$, be the six dimensions of an implementation, X, of a block f, if f is L-shaped. Also, let d_i^X, where $i = h, w$, be the height and the width of an implementation, X, of block f, if f is rectangular. Let A and B represent two possible implementations of a super-block, f. If f is L-shaped, but

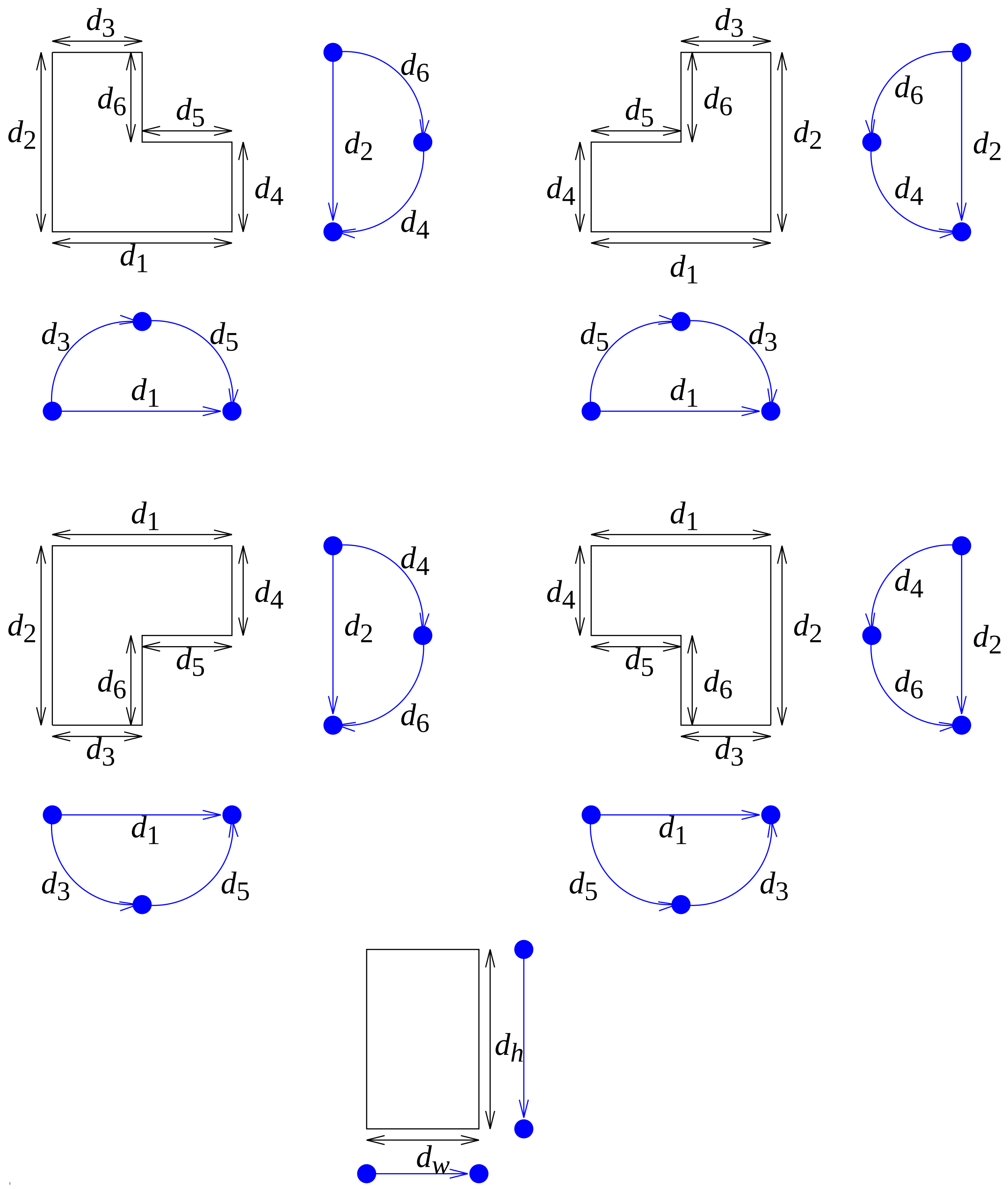


Figure 2.19: Notations for the dimensions of an L-shaped region and a rectangle

not rectangular, then A is said to *dominate* B if and only if $d_i^A \geq d_i^B$, for $i = 1, 2, 3, 4, 5, 6$. If f is rectangular, then A is said to *dominate* B if and only if $d_i^A \geq d_i^B$, for $i = h, w$. If A dominates B, then we say A is *redundant with respect to* B. Moreover, a floorplan which uses A can not have a smaller area than another one which uses B. Therefore, we only need to keep those irredundant implementations for each processed super-block. Given an area minimization problem, our algorithm recursively determines all possible implementations of the super-blocks corresponding to the sub-trees of the cut tree in a bottom-up fashion. Depending on the type of the cut line and the possible implementations of the left and right child, the implementations of the resultant super-block are determined such that none of them is redundant. The rules determining the dimensions of the super-block for each type of cut line are stated in Table 2.3, where A and B being an implementation of the two super-blocks as shown in Figure 2.6, respectively, and C being the implementation of the resultant super-block. Lemma 1 follows immediately from the rules listed in Table 2.3.

Lemma 1 *The value of each horizontal dimension of C (d_1, d_3, or d_5, if f is L-shaped, but not rectangular; d_h, if f is rectangular) is equal to the length of the longest path between the two vertices in the dual polar graphs H_F which represent the two corresponding sides of C. The same property holds for each vertical dimension of C.*

Therefore, C is large enough to accommodate both A and B.

At the beginning, we are given a set of irredundant implementations for each basic block. The algorithm traverses the cut tree in a bottom-up fashion. Consider an internal node, C, whose corresponding super-block is f. Suppose the irredundant implementations of the blocks corresponding to the two children of C, A and B, have already been computed. Then, the rules listed in Table 2.3 are used to assemble each pair of the implementations(one from A, the

Type I

$d_h^C = max\{d_h^A, d_h^B\}$

$d_w^C = d_w^A + d_w^B$

Type II

$d_1^C = d_w^A + d_1^B$

$d_2^C = max\{d_h^A, d_2^B\}$

$d_3^C = d_w^A + d_3^B$

$d_4^C = d_4^B$

$d_5^C = d_5^B$

$d_6^C = d_6^B$

Type III

$d_1^C = d_1^A + d_w^B$

$d_2^C = max\{d_2^A, d_6^A + d_h^B\}$

$d_3^C = d_3^A$

$d_4^C = max\{d_4^A, d_h^B\}$

$d_5^C = d_5^A + d_w^B$

$d_6^C = d_6^A$

Type IV

$d_1^C = max\{d_1^A, d_3^A + d_1^B\}$

$d_2^C = max\{d_2^A, d_4^A + d_2^B\}$

$d_3^C = d_3^A + d_3^B$

$d_4^C = d_4^A + d_4^B$

$d_5^C = d_5^B$

$d_6^C = d_6^B$

Type V

$d_1^C = max\{d_1^A, d_3^A + d_1^B\}$

$d_2^C = max\{d_2^A + d_6^B, d_4^A + d_2^B\}$

$d_3^C = d_3^B$

$d_4^C = max\{d_2^A, d_4^A + d_4^B\}$

$d_5^C = d_3^A + d_5^B$

$d_6^C = d_6^B$

Type VI

$d_h^C = max\{d_2^A, d_4^A + d_h^B\}$

$d_w^C = max\{d_1^A, d_3^A + d_w^B\}$

Type VII

$d_h^C = max\{d_2^A, d_2^B, d_6^A + d_6^B, d_4^A + d_4^B\}$

$d_w^C = max\{d_1^A + d_3^B, d_3^A + d_1^B\}$

Type VIII

$d_h^C = max\{d_h^1 + d_h^2, d_h^4 + d_h^5 + d_h^2, d_h^4 + d_h^3\}$

$d_w^C = max\{d_w^1 + d_w^4, d_w^1 + d_w^5 + d_w^3, d_w^2 + d_w^3\}$

Table 2.3: Merging rules.

other from B) to obtain an implementation of f. Therefore, super-block f has $n_1 \times n_2$ possible implementations. If we maintain all of these implementations, then $O(m^n)$ of time is needed to obtain an optimal solution, where each of the n modules is assumed to have m different implementations. However, it is possible that some of them are dominated by others, especially when f is rectangular. It turns out that we only need to retain the irredundant implementations of f according to the following lemma.

Lemma 2 *Let A and B be two implementations of f and A be redundant with respect to B. Then, the minimum floorplan with B being the implementation of f will not be larger than that with A being the implementation of f.*

The proof follows immediately from Lemma 1 and the fact that replacing the weights of the edges by smaller values will not increase the lengths of the longest paths in the dual polar graphs.

A naive approach to determine the irredundant implementations of f will have $O(n_1^2 \times n_2^2)$ time complexity, where $n_1(n_2)$ is the number of irredundant implementations of the block corresponding to $A(B)$. However, since the problem of removing redundant implementations can be transformed into the problem of finding the maxima of a point set [41] in computational geometry. A more efficient approach is to apply a maxima finding algorithm to remove the redundant implementations after all possible implementations of f have been generated. The fastest known algorithm for this problem has $O(Nlog^{d-2}N)$ time complexity, where N is the number of the points and d is the dimension of the space the point set lies. If f is rectangular, then the complexity of the algorithm is dominated by the sorting procedure. If f is L-shaped, then the algorithm has time complexity $O(Nlog^4N)$, where N is $n_1 \times n_2$.

After obtaining all irredundant implementations of the floorplan corresponding to the whole cut tree, we scan them once and choose the one with the minimum product as our solution. It should be noted that our algorithm guarantees an optimal solution as stated below:

Theorem 3 *Given a decomposable floorplan F and a set of implementations for each module, our algorithm determines the implementation for each module such that the area of the resultant floorplan is minimum.*

Using Lemma 1 and 2, we can prove this theorem by induction on the level of the cut tree.

2.3.3 An Algorithm for the Indecomposable Floorplans

If the given floorplan F is not decomposable, the execution of our algorithm will terminate in a procedure call *rectangle*(f), where f is a rectangular super-block in F. Our algorithm can be extended to handle this situation as follows. Our algorithm first finds all maximal rectangular super-blocks[3] in f using the algorithm developed in [8]. These maximal rectangular super-blocks together-ed with the remaining L-shaped basic blocks form a partition of f. Then, our extended algorithm greedily applies the following operations until none of these two operations can be applied:

1. merge an L-shaped block g with a maximal rectangular super-block h, if they form an L-shaped super-block j in f and j can be dissected into g and h using one of the eight types of cut lines;

2. merge two L-shaped blocks, g and h, if they form an L-shaped super-block j in f and j can be dissected into g and h using one of the eight types of cut lines;

[3] A rectangular super-block that is not contained in any other rectangular super-block in f.

These maximal rectangular super-blocks are then used as inputs of Procedure *rectangle* for further decomposition. Figure 2.20 shows an example, where bold lines show a partition of an undecomposable super block. Since the super-block in Figure 2.20 can not be decomposed using

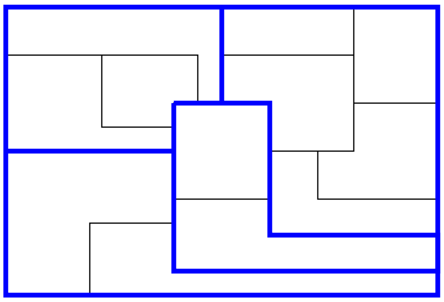


Figure 2.20: A partition of an undecomposable floorplan

the eight types of cut lines, it is partitioned into four super-blocks using the two operations described above. Now, suppose inductively that the irredundant implementations of the blocks in such a decomposition of f have been obtained. Then, we use a branch-and-bound algorithm to try all possible combinations of the implementations of these blocks as in [8, 57] to obtain irredundant implementations of f. Therefore, incorporation of the branch-and-bound algorithm in [8, 57] into our algorithm will enable us to handle indecomposable floorplans.

2.4 Experimental Results and Discussions

Our algorithm was implemented in C and executed on a Convex computer. Two examples were used to test the algorithm, one consists of fourteen, and the other forty modules. They are denoted Ex14 and EX40, and are shown in Figure 2.21 and 2.22. EX14 is an example taken from [49], and EX40 is a problem in [58].

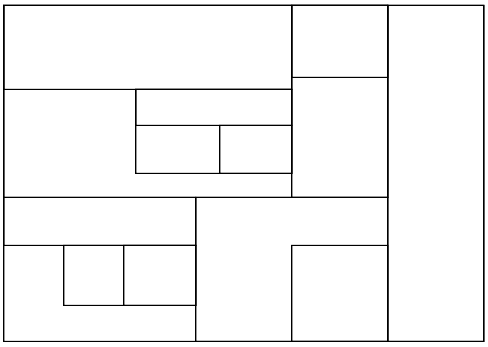


Figure 2.21: EX14

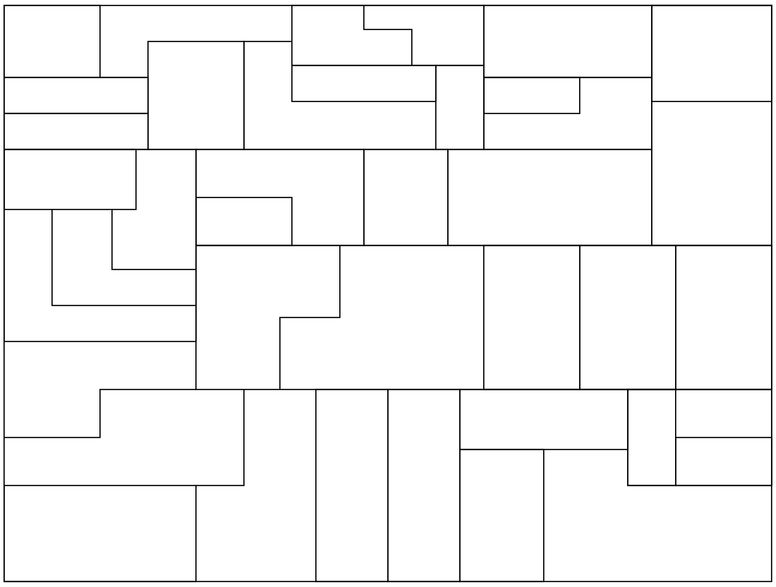


Figure 2.22: EX40

Note that our problem can also be formulated as a quadratic 0-1 integer programming problem as follows. Suppose module i has n different implementations. We use n 0-1 integer variables, $x_i^1, x_i^2, \ldots, x_i^n$, to represent the n possible implementations. If $x_i^j = 1$, where $1 \leq j \leq n$, then the jth implementation will be chosen for module i. Since exactly one of these 0-1 integer variables can assume the value 1 in the solution, we need the constraint:

$$\sum_{k=1}^{n} x_i^k = 1.$$

If module i is rectangular, we need the constraints:

$$h_u - h_b \geq \textstyle\sum_{k=1}^{n} x_i^k \times a_k \qquad\qquad w_l - w_r \geq \textstyle\sum_{k=1}^{n} x_i^k \times b_k,$$

where a_k and b_k are the height and width of the kth implementation and h_u, h_b, w_l, and w_r are the variables corresponding to the coordinates of segments containing the four sides of the basic block in which module i will be placed. If i is an L-shaped module, we need the constraints:

$$\begin{array}{ll}
h_u - h_b \geq \sum_{k=1}^{n} x_i^k \times a_k & \qquad h_m - h_b \geq \sum_{k=1}^{n} x_i^k \times c_k \\
h_u - h_m \geq \sum_{k=1}^{n} x_i^k \times e_k & \qquad w_l - w_r \geq \sum_{k=1}^{n} x_i^k \times b_k \\
w_l - w_m \geq \sum_{k=1}^{n} x_i^k \times d_k & \qquad w_m - w_r \geq \sum_{k=1}^{n} x_i^k \times f_k,
\end{array}$$

where a_k, b_k, c_k, d_k, e_k, and f_k are the six dimensions of the kth implementation and h_u, h_m, h_b, w_l, w_m, and w_r are the variables corresponding to the segments containing the six sides of the basic block in which module i will be placed as shown in the left figure of Figure 2.23(b). The objective function to be minimized is the distance between the segments containing the top and bottom sides of the floorplan times the distance between the segments containing the left and right sides of the floorplan.

To obtain an optimal solution to the quadratic 0-1 integer programming problem, we use a package, MPOS (Multiple-Purpose Optimization System) from Northwestern University. First of all, we assume each module has only two implementations, namely, rotation by 90° for rectangular modules and flipping along the diagonal for L-shaped modules as illustrated in Figure 2.23. The dimensions of the modules are randomly generated. Table 2.4 shows the

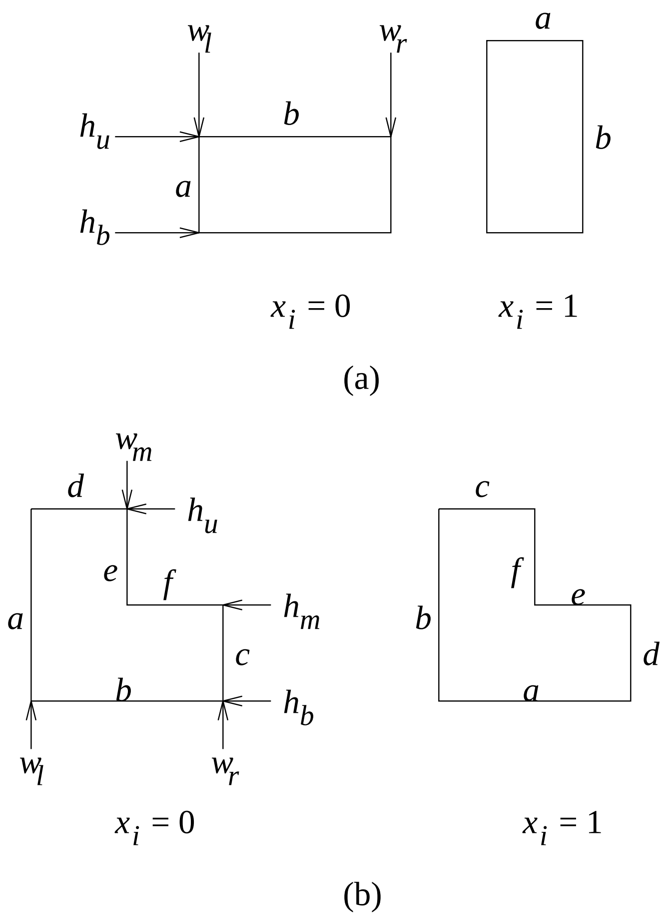


Figure 2.23: Two possible implementations for a rectangular and an L-shaped module

computation time and the area of the solution produced by our algorithm for both examples. Each row in Table 2.4 corresponds to a set of randomly generated data for the dimensions of the modules. Table 2.4 also shows the results produced by MPOS. We ran each test problem

example	MPOS result	our result	time (seconds)	# of implementations
EX14	3432	3432	0.07	$2^{14} = 16384$
EX40	4756	4756	0.27	$2^{40} = 1.1\times10^{12}$
EX40	19557	19557	0.5	$2^{40} = 1.1\times10^{12}$
EX40	8900	8900	1.15	$2^{40} = 1.1\times10^{12}$

Table 2.4: Experimental results in which each module has two implementations

example	MPOS result	our result	time (seconds)	# of implementations
EX14	2028	2028	0.2	$6^{14} = 7.8\times10^{10}$
EX40	12257	10640	172.5	$6^{40} = 1.34\times10^{31}$
EX40	*	11858	82	$6^{40} = 1.34\times10^{31}$
EX40	11990	11990	50.2	$6^{40} = 1.34\times10^{31}$

Table 2.5: Experimental results in which each module has six implementations

on MPOS for an hour and extracted the best feasible solution produced. In this table, MPOS also produced an optimal solution. But, our algorithm runs in less than 2 seconds for all cases.

For the test examples in Table 2.5, we randomly generated the values of the dimensions of 6 implementations for each module. Again, we used MPOS to obtain a feasible solution by running it for at least an hour. MPOS can produce an optimal solution in the first and the fourth rows. Its result is by 15.2% worse than that produced by our algorithm in the second example. MPOS could not produce any feasible solution within hours in the third example, whereas our algorithm obtained an optimal one in 82 seconds.

2.5 Summary

In this chapter we consider the area minimization problem for floorplans with L-shaped regions which has been proved to be NP-complete. The problem is more difficult than that studied in [58] in the sense that there is an additional constraint on the topological relation among the modules which is usually specified before the area minimization stage. First, we develop a new decomposition scheme for floorplans with L-shaped regions. The two conventional types of cut lines, that is, the horizontal and vertical cut lines, are generalized to eight types of cut lines in order to allow L-shaped regions in the floorplan. A $O(n^3)$ time algorithm to test whether a given floorplan is decomposable by using these eight types of cut lines is presented. A cut tree is constructed if the floorplan is decomposable. Based on the cut tree so constructed, we design an algorithm to solve the area minimization problem optimally and obtain results in less than 3 minutes in the most complex test problem while a mathematical programming package runs for an hour without any solution guarantee. The branch-and-bound algorithms in [8, 57] can be incorporated easily into our algorithm to handle floorplans which are not decomposable by the eight types of cut lines. It should be noted that for different cut trees the amount of computation time is different. It is an interesting problem to find the cut tree which gives our algorithm the fastest performance.

Chapter 3

Channel Routing in Quickly Customized Logic

3.1 Introduction

In conventional VLSI gate array technology, the entire wafer consisting of an array of identical gates or cells is prefabricated. Wirings that customize the wafer according to circuit specifications are routed using two or more routing (metal) layers. The multi-layer wiring capability provides sufficient flexibility in the design of highly integrated circuits. However, the turn-around time (TAT) from logic to parts is usually long and a high design cost is usually incurred. In contrast, field-programmable technology [15, 21] has the advantage of short turn-around time, but suffers from the inevitable drawback of low integration due to limited layout resource and the adverse effect of programmable elements on circuit speed. The recently proposed Quickly Customized Logic (QCL) architecture [35, 61] combines the advantages of mask-programmable and field-programmable technologies. In QCL architecture, two routing metal-layers are utilized. The mask of the first metal is fixed, and wire segments on this layer have pre-determined positions and lengths. Only the second metal is for customized routing, and the mask for this layer will be custom-designed. In comparison with conventional two-layer customization, QCL reduces the TAT for the fabrication of engineering samples to a few days (typically 2 days), since all the fabrication processes before the second-metal layer can proceed in advance. (Masks for the vias and contact holes are also pre-designed.) Moreover, according to [61], using 1.5μm CMOS standard process, QCL achieves 0.9ns gate delay for a 2-input NAND gate with chip complexity up to the equivalence to 12K gates, which are comparable with gate arrays designed by two-layer customization. A similar architecture, called Metal-Programmable Gate Array (MPGA), was recently developed and used for the design conversion of EPLD and FPGA. Such design conversion is another important application of the concept of pre-programmed architecture [25].

QCL gate arrays have rows of transistors and well-defined channel areas in the inner region of the chip, as shown in Figure 3.1. Within a row of transistors, there are transistor islands separated by pre-placed segments. A transistor island consists of a number of transistors that are chained together by diffusion. Cell circuits are formed by interconnecting transistor islands. Pre-placed segments between transistor islands are used as bypass wires to cross the transistor rows. In the channel, there are aligned first-metal wire segments used for interconnections between cell circuits. At each end of such a wire segment, there is a via hole connecting the first-metal layer and the second-metal layer. This chapter considers the channel routing problem in QCL structure. Section 3.2 presents a formulation of this new channel routing problem and reviews related work. In Section 3.3, we prove that this problem is NP-complete. Section 3.4 gives several necessary properties for any feasible routing solution. Recognition of these

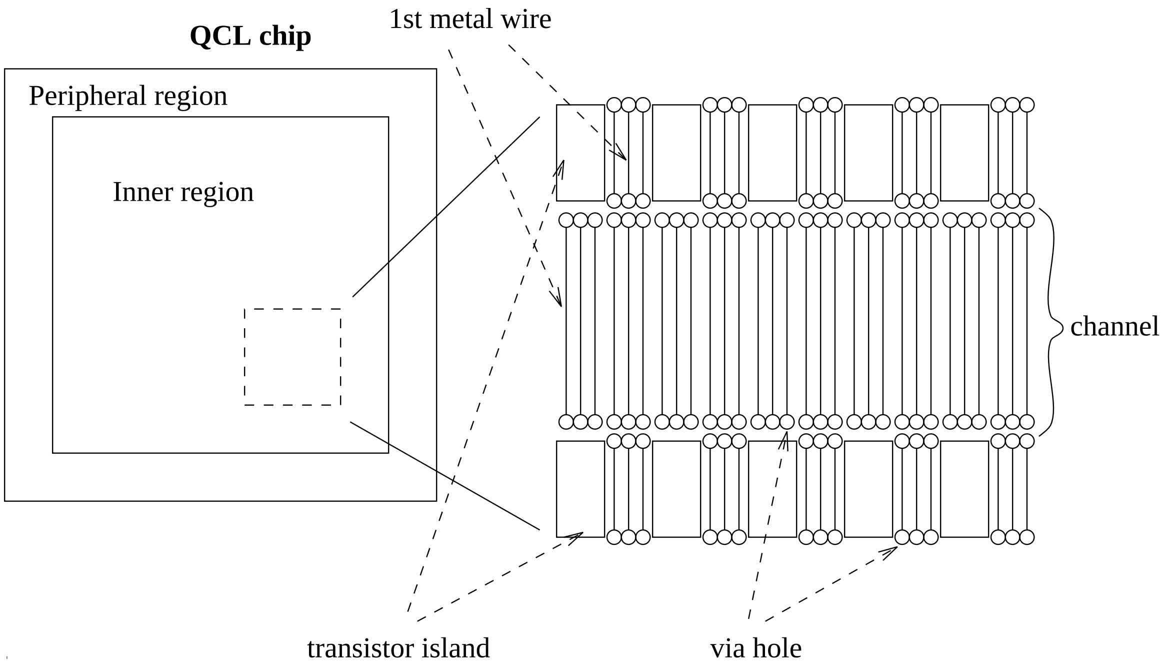


Figure 3.1: QCL architecture

necessary conditions leads to an effective exhaustive search algorithm. Experimental results are provided in Section 3.5 to show that our algorithm can produce a solution, if one exists, within a reasonable amount of running time.

3.2 Problem Formulation

As in a conventional routing channel, we assume that there is an underlying grid for the channel. Vertical grid lines, called *columns*, are labeled 1, 2, ..., w from left to right. Horizontal grid lines, called *tracks*, are labeled 1, 2, ..., k, as shown in Figure 3.2. A channel in QCL structure has *terminal positions* along its top boundary (line TOP) and bottom boundary (line BOTTOM). Terminals placed at terminal positions on lines TOP and BOTTOM will be referred to as *top terminals* and *bottom terminals*, respectively. Two metal layers are used for routing in the channel. They are referred to as *Layer I* and *Layer II.* Customized wires are allowed only on Layer II. There are *pre-placed vertical wire segments* (or *pre-placed segments* for short) on Layer I in the channel. Moreover, each end of a pre-placed segment has a via hole between Layer I and II. One via, called *top via*, is located right beneath a terminal position on line TOP and another, called *bottom via*, right above a terminal position on line BOTTOM. Such a (top or bottom) via is said to be *associated with* the terminal position it is adjacent to. A terminal is connected to a routing wire through its associated via, since there is no routing track between a terminal position and its associated via, as depicted in Figure 3.2. Consequently, for a top (bottom) terminal on TOP, no terminal of another net can be located at the bottom (top) terminal position of the same column, since otherwise there will be a short circuit between the two terminals. Figure 3.2 shows an example of the channel, where a cross on TOP and BOTTOM represents a terminal position.

In our routing problem, n nets, numbered 1, 2, …, n, with p top terminals and q bottom terminals, are given. We denote the top terminals and the bottom terminals by $\alpha_1, \alpha_2, \ldots, \alpha_p$ and $\beta_1, \beta_2, \ldots, \beta_q$, respectively, where $1 \leq \alpha_i, \beta_j \leq n$ for $i = 1, 2, \ldots, p$ and $j = 1, 2, \ldots q$. Thus, the terminals in $\{\alpha_1, \alpha_2, \ldots, \alpha_p, \beta_1, \beta_2, \ldots, \beta_q\}$ are partitioned into n nonempty subsets corresponding to the n nets. Let $x(\alpha_i)$ $(x(\beta_j))$ denote the x-coordinate of the terminal α_i (β_j). As indicated above, two terminals belonging to two different nets cannot be located in the same column. Thus, we have $x(\alpha_i) \neq x(\beta_j)$ unless $\alpha_i = \beta_j$. Consequently, if there is a terminal located in column j, whether it is a top terminal or a bottom terminal, the pre-placed segment in column j cannot be used by another net. Therefore, as far as routing is concerned, there is no need to distinguish a top terminal from a bottom terminal in a column. Scanning the columns from left to right, we use t_j to label the jth terminal encountered, and each two terminals of

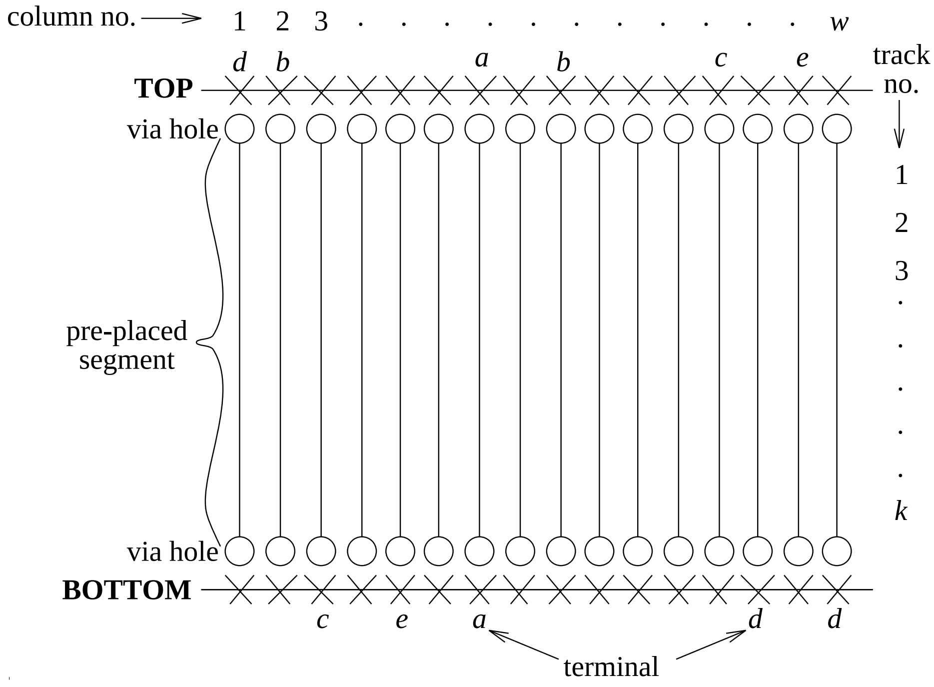


Figure 3.2: A channel in QCL structure

the same nets located in the same column have the same label. In the contrary case, if there is no terminal located in a particular column, the pre-placed segment in the column can be used by any net. In Figure 3.3, there is no terminal in column l. The pre-placed segment can be used by any net in either of the two ways shown in (a) and (b).

Consider two terminals, t_i and t_j, that belong to the same net. If $i = j$, then these two terminals are located in same column, with one being a top terminal and the other being a bottom terminal. They are connected by the pre-place segment in the column. Net a in Figure 3.4 is an example. If $i \neq j$, these two terminals are to be connected using wires in Layer II. Since Layer II is the only layer that can be used for customized interconnections, only non-intersecting wires can be used. Consider a particular case, where we allow only wires of shapes $\sqcup$ and $\sqcap$, as illustrated by net b and c, respectively, in Figure 3.4. To avoid crossing between

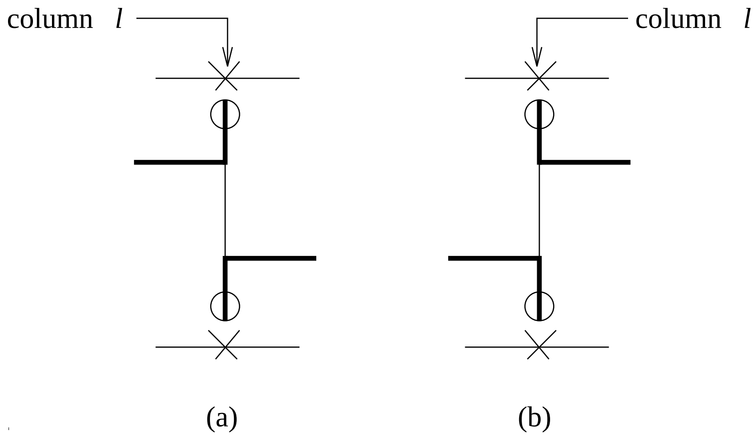


Figure 3.3: The usage of a 'free' pre-placed segment

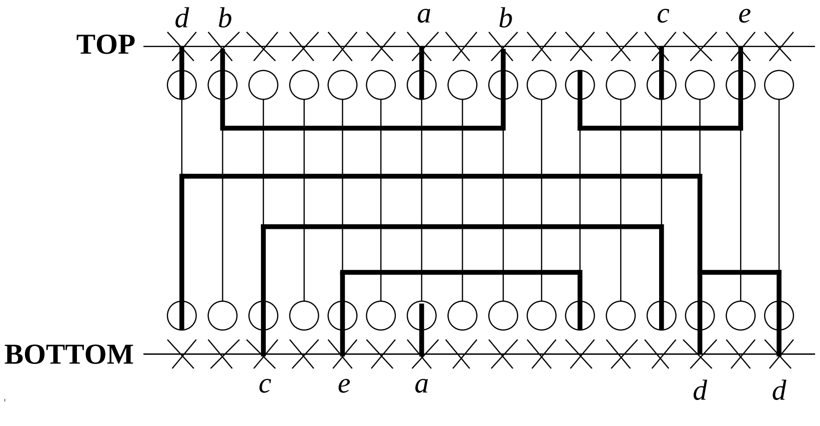


Figure 3.4: A routing solution for the channel in Figure 3.2

routings of different nets, wires of these two types can be used repeatedly in the routing of a single net, as in net e and net d in Figure 3.4. Zig-zag wires, shown in Figure 3.5 (a), are not allowed due to the adverse effects on the yield caused by the bends of wires. Moreover, in order to avoid long paths, any vertical line between two adjacent pre-placed segments can intersect the wire for each net at most once, as depicted in Figure 3.5 (b). Figure 3.4 shows an example of the routing solutions to the channel routing problem in Figure 3.2.

The formal definition of our QCL channel routing problem is stated as follows:

CHANNEL ROUTING PROBLEM IN QCL(CRQCL)

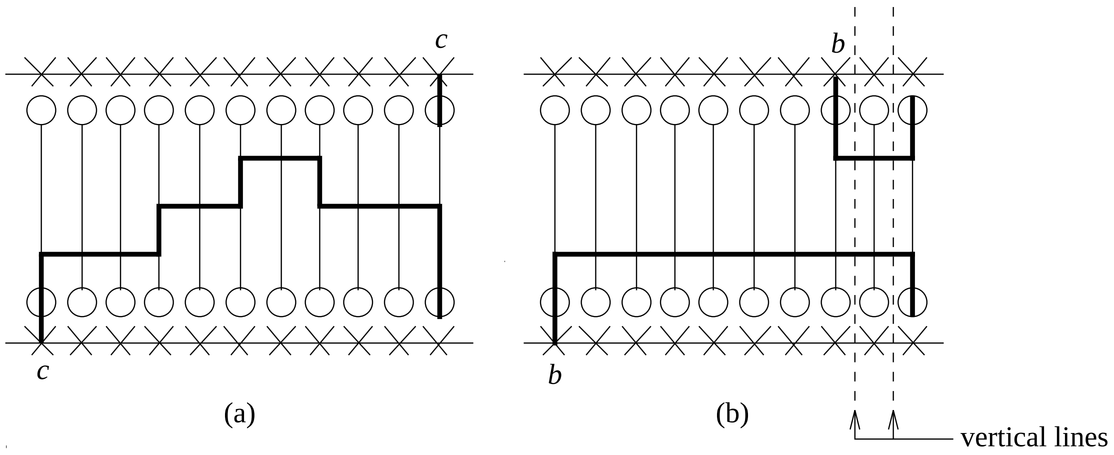


Figure 3.5: (a)Zig-zag wires (b)Parallel wires

Instance: A collection of n nets with p terminals t_1, t_2, …, t_p in a channel with w columns, and a non-negative integer k.

Question: Is there a routing for the n nets where the number of tracks used in the channel is less than or equal to k?

3.3 An NP-Completeness Proof

We shall prove that CRQCL is NP-complete by reducing the bandwidth problem to a restricted version of CRQCL where $k = 2n$. We use RCRQCL to denote this restricted version of CRQCL. The bandwidth problem is known to be NP-complete [17], and is stated as follows:

BANDWIDTH PROBLEM(BWP)

Instance: Graph $G = (V, E)$ and positive integer $B \leq |V|$.

Question: Is there a linear ordering of V with bandwidth B, i.e., a one-to-one function f: $V \rightarrow \{1, 2, \ldots, |V|\}$ such that, for all $(u, v) \in E$, $|f(u) - f(v)| \leq B$?

Before we present the proof, let us consider RCRQCL. Since the routing of a net can intersect a vertical line between two adjacent columns at most once, each net can occupy at most two tracks in each column. The two cases in which the routing of a net occupies two tracks are shown in Figure 3.3. Moreover, since the number of tracks is equal to twice the number of nets, we can conclude that if there is a routing solution for these n nets, there must be a routing solution which uses at most k tracks. Therefore, letting $k = 2n$ imposes no restriction on the number of tracks used in the channel. The similarity between the QCL routing problem and the single-row routing problem leads to another important observation. Suppose that we have a routing solution for a QCL channel routing problem. If we merge line TOP with line BOTTOM (the resultant line is called *node line*), then we have an instance of the classical single-row routing problem [27, 54]. Moreover, a routing solution for the QCL channel routing problem yields a corresponding solution for the single-row routing problem. For the QCL routing problem in Figure 3.2, Figure 3.6(a) shows the corresponding single-row routing problem, and Figure 3.6(b) shows the single-row routing solution corresponding to the QCL routing solution in Figure 3.4. Note that in Figure 3.6 net a becomes a degenerated single-terminal net, which requires no wire in its routing. On the basis of the observation above, RCRQCL can be transformed to the single-row routing problem with unlimited street capacities and limited space between adjacent nodes.[1] We will use RSRR to denote such a version of the single-row routing problem. Thus, we need only to prove that RSRR is NP-complete.

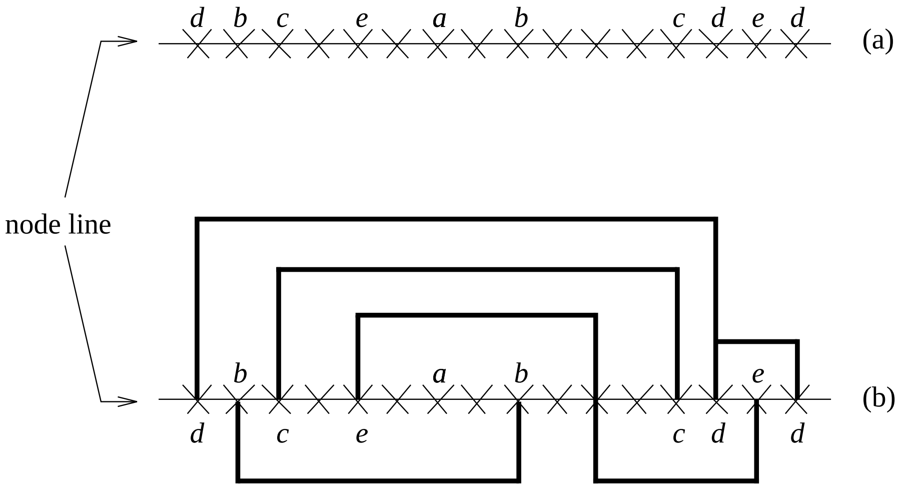


Figure 3.6: (a) Single-row routing problem corresponding to Figure 3.2 (b) Single-row routing solution corresponding to Figure 3.2

Kuh et al. [27] introduced an *interval graphical representation* for a routing solution in the single-row routing problem. In this representation, each net is represented by a line interval parallel to the node line. There are as many nodes on the line interval as the number of terminals in the corresponding net. These nodes are located according to the positions of the terminals. The dotted line that connects the nodes of all the nets is called the *reference line*. For example, Figure 3.7 shows the interval graphical representation of the single-row routing shown in Figure 3.6(b). If we stretch the reference line, place it on top of the node line, and map the line intervals into vertical and horizontal paths, we obtain the routing solution shown in Figure 3.6(b) for the corresponding single-row routing problem. Note that, if we order the

[1] Refer to [27] for the terminology used in the context of the single-row routing.

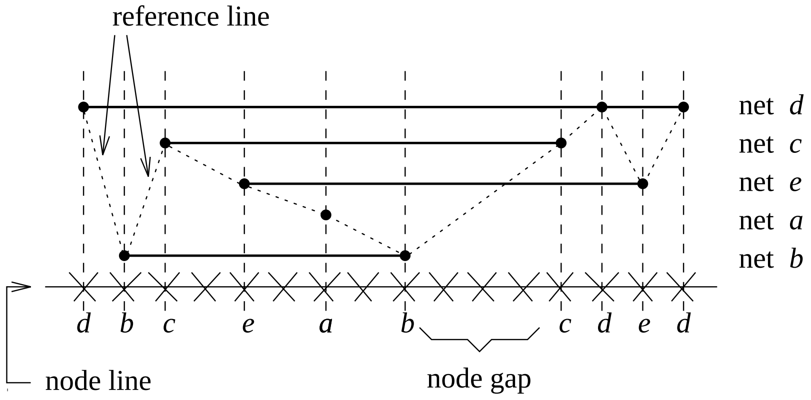


Figure 3.7: The interval graphical representation of the single-row routing in Figure 3.6 (b)

intervals according to their vertical relative positions in the interval graphical representation, for each ordering, the process above uniquely defines a routing solution for the corresponding single-row routing problem. Moreover, the intersections of the reference line with any line interval determine the node gaps where the routing of the corresponding net crosses from one side of the node line to the other side. For example, the line interval corresponding to net e intersects with the reference once between terminals of net b and c. Consequently, the routing of net e in Figure 3.6(b) crosses the node line once between the corresponding nodes.

Lemma 3 *RSRR is NP-complete.*

Proof: Obviously, RSRR is in NP. To show that RSRR is NP-complete, it suffices to reduce BWP to RSRR. Given $G = (V, E)$ in BWP, for each vertex $v_i \in V$, construct a net i with nodes x_i and y_i. These x and y nodes are located along the node line in the order of (x_1, x_2, $\ldots$, $x_{|V|}$, $y_{|V|}$, $y_{|V|-1}$, $\ldots$, y_1). For each edge (v_i, v_j), we introduce two adjacent nodes, $r_{i,j}$ of net i and $s_{i,j}$ of net j, between the x nodes and the y nodes. All the r nodes and s nodes are arranged in an interleaving way, as shown in Figure 3.8. The gap between adjacent nodes that are of the form $r_{i,j}$ and $s_{i,j}$ is such that at most B vertical wires can cross the node line

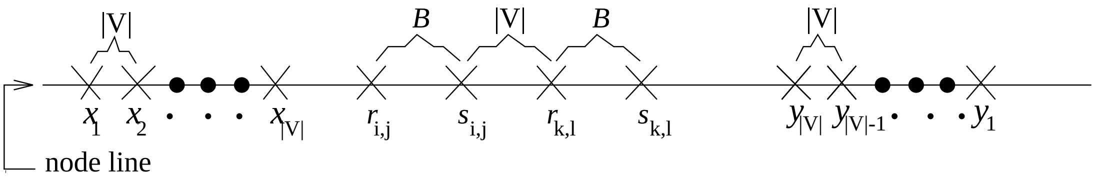


Figure 3.8: The reduction

between these two nodes. The gap between any other adjacent nodes on the node line is equal to $|V|$. By identifying the ordering of the line intervals in the interval graphical representation of RSRR with the linear ordering of the vertices in the BWP, we can easily prove that the given BWP problem has a feasible linear ordering if and only if there is a routing solution in the corresponding RSRR problem. $\square$

Based on the equivalence of the RCRQCL problem and the RSRR problem, we have the following result:

Theorem 4 *The channel routing problem in QCL is NP-complete.*

3.4 A Routing Algorithm

Since the QCL channel routing problem is NP-complete, it is unlikely that there exists a polynomial-time algorithm that decides whether there exists a routing solution for a given QCL channel routing problem. To guarantee that a solution will be found if such a solution exists, we employ an exhaustive search approach. In this section, we present a search algorithm which employs an effective scheme to prune many of the branches in the search tree. As will be seen later, the running time of our algorithm is quite reasonable for problems of practical size.

Before stating our algorithm, we need to introduce some terminology. Given a routing R of a channel and a column l in the channel, we can list the nets whose routing appears on the k

tracks between columns l and $l+1$, with 0 representing an empty track. We call the resultant sequence of length k a *frontier for column l in routing R.* For example, the frontier for the 3rd column in the routing shown in Figure 3.4 is $(b, d, c, 0)$. If a frontier is a sequence of k zeros, we call such a frontier a *trivial frontier.* If all the non-zero entries appear consecutively in f, then we say the frontier f is *compact.* Otherwise, f is not compact. Therefore, the frontier $(b, d, c, 0)$ mentioned above is compact, whereas the frontier for the 3rd rightmost column in the routing shown in Figure 3.4, $(e, 0, 0, d)$, is not compact. It is easy to see that in any frontier, nets whose wires are $\sqcup$-shaped must appear before the nets whose wires are $\sqcap$-shaped. Therefore, for any compact frontier, we insert a special mark, α, to separate the nets whose wires are of different types. Two frontiers are *symmetric* if flipping one gives the other. Therefore, the frontiers (0, 0, 2, 5, α, 11, 9, 1, 0) and (0, 1, 9, 11, α,5, 2, 0, 0) are symmetric. Moreover, if (r_1, r_2, ..., r_k) is a compact frontier and r_p and r_q ($p \leq q$) are the first and the last non-zero entries, then we call the nets r_p and r_q the *outermost* nets, the nets r_{p+1} and r_{q-1} the *second outermost* nets, etc. However, this ordinal number assignment does not apply beyond the special symbol α. That is, net 3 is the fourth outermost net in frontier (0, 4, 1, 5, 3, α, 2), instead of the second outermost net. If the frontier for each column in a routing R of a channel is compact, then we say R is a *compact routing.*

We are now ready to describe our channel routing algorithm. The algorithm processes the channel in a column-by-column fashion from left to right. Before processing the first column, we have exactly one frontier for column 0, namely, a trivial frontier. Then, the algorithm processes one column at a time. In each column, for example column l, all possible frontiers for column $l-1$ were generated. Based on these frontiers for column $l-1$ and the terminal located on column l, we generate all possible frontiers for column l. If the frontier generating process continues and eventually generates a trivial frontier for the rightmost column, then by tracing backward column by column, we can construct a routing for the channel. It can be shown that our algorithm always produces a routing solution, if there is one. If, however, no frontier is generated for a column, say column l, then our algorithm proceeds as follows. We insert a new column between column $l-1$ and l, and/or a track as the topmost track, with the intention of generating more frontiers before considering column l. By inserting columns and rows whenever necessary, we are assured of obtaining a routing solution for any given channel.

Despite the exhaustive nature of our algorithm, the following lemma says that we actually need not consider all possible frontiers in each column. In fact, only compact frontiers for each column need to be retained.

Lemma 4 *Given a channel routing problem, there is a routing solution if and only if there is a compact routing solution.*

Proof: We need only prove the "only if" part, since the other direction is obvious. Given a routing of a channel, we scan the routing column-by-column from left to right, during which we move down wires of type $\sqcup$ and move up wires of type $\sqcap$ as long as no short occurs. The resultant routing must be compact. An example of this construction is shown in Figure 3.9. $\square$

Therefore, our algorithm only considers compact frontiers during the generating process. Hereafter, unless otherwise specified, by a frontier (routing) we mean a compact frontier (routing). After a compact routing solution is found, a linear scan is performed from left to right through the channel in order to reduce the length of second-metal wires for each net. During the linear scan, the wires of type $\sqcup$ are moved up toward the top boundary of the channel, and the wires of type $\sqcap$ are moved down toward the bottom boundary. This post-process is the reverse operation illustrated in Figure 3.9.

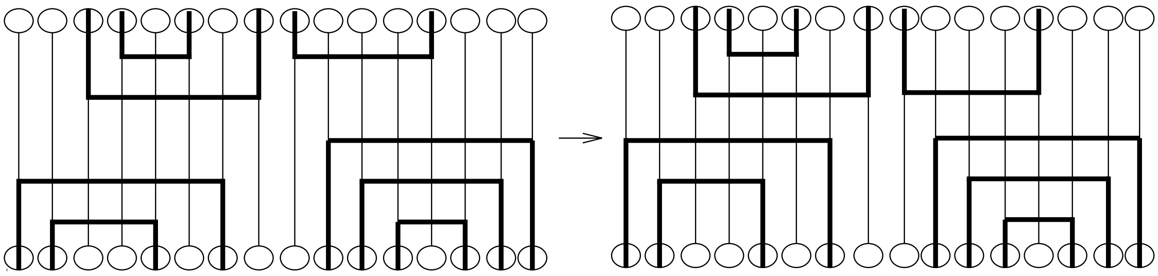


Figure 3.9: The conversion of a routing into a compact routing

Suppose $f = (r_1,\, r_2,\, \ldots,\, r_p,\, \ldots,\, r_m,\, \alpha,\, r_{m+1},\, r_{m+2},\, \ldots,\, r_q,\, \ldots,\, r_k)$ is a frontier of column l. Let r_p and r_q be the two outermost nets, where $p \leq q$ and $1 \leq p,\, q \leq k$, that is, $r_i = 0$, for $i = 1, 2, \ldots, p-1$ and $q+1, q+2, \ldots, k$. We define the following operations on a frontier f.

$insert\_head(f)(u) = (r_1,\, r_2,\, \ldots,\, r_{p-2},\, u,\, r_p,\, \ldots,\, r_m,\, \alpha,\, r_{m+1},\, r_{m+2},\, \ldots,\, r_q,\, \ldots,\, r_k)$,

where u is a net and $u \notin f$.

$insert\_tail(f)(u) = (r_1,\, r_2,\, \ldots,\, r_p,\, \ldots,\, r_m,\, \alpha,\, r_{m+1},\, r_{m+2},\, \ldots,\, r_q,\, u,\, r_{q+2},\, \ldots,\, r_k)$,

where u is a net and $u \notin f$.

$remove\_head(f) = (r_1,\, r_2,\, \ldots,\, r_{p-1},\, 0,\, r_{p+1},\, \ldots,\, r_m,\, \alpha,\, r_{m+1},\, r_{m+2},\, \ldots,\, r_q,\, \ldots,\, r_k)$.

$remove\_tail(f) = (r_1,\, r_2,\, \ldots,\, r_p,\, \ldots,\, r_m,\, \alpha,\, r_{m+1},\, r_{m+2},\, \ldots,\, r_{q-1},\, 0,\, r_{q+1},\, \ldots,\, r_k)$.

$copy(f) = f$.

$head\_to\_tail(f) = (r_1,\, r_2,\, \ldots,\, r_{p-1},\, 0,\, r_{p+1},\, \ldots,\, r_m,\, \alpha,\, r_{m+1},\, r_{m+2},\, \ldots,\, r_q,\, r_p,\, r_{q+2},\, \ldots,\, r_k)$.

$tail\_to\_head(f) = (r_1,\, r_2,\, \ldots,\, r_{p-2},\, r_q,\, r_p,\, \ldots,\, r_m,\, \alpha,\, r_{m+1},\, r_{m+2},\, \ldots,\, r_{q-1},\, 0,\, r_{q+1},\, \ldots,\, r_k)$.

Let t denote the net that has a terminal located on column l. If there is no terminal on column l, then t is 0. For each frontier, f, for column l, the pseudo-code to generate the frontiers for column $l+1$ is shown in Table 3.1. We say that each frontier for column $l+1$ generated from f according to the procedure in Table 3.1 is a *child* of f. It is possible that two different frontiers for column l will lead to the generation of the same frontier for column $l+1$. Clearly, we need

```
If t ≠ 0, then
      if t ∉ f, then                                                  /* start a net */
            if p ≥ 2, then
                  generate insert_head(f)(t); (See Figure 3.10.)
            if q ≤ k − 1, then
                  generate insert_tail(f)(t); (See Figure 3.11.)
      else
            if r_p = t, then
                  if no terminal of net t exists right to column l,
                  then                                                /* end a net */
                        generate remove_head(f); (See Figure 3.12.)
                  else                                                /* continue a net */
                        generate copy(f); (See Figure 3.14(a).)
                        generate head_to_tail(f); (See Figure 3.15(a).)
            if r_q = t, then
                  if no terminal of net t exists right to column l,
                  then                                                /* end a net */
                        generate remove_tail(f); (See Figure 3.13.)
                  else                                                /* continue a net */
                        generate copy(f); (See Figure 3.14(b).)
                        generate tail_to_head(f); (See Figure 3.16(a).)
else                                                                  /* continue a net */
      generate copy(f); (See Figure 3.14(c).)
      if p ≥ 2, then
            generate tail_to_head(f); (See Figure 3.16(b).)
      if q ≤ k − 1, then
            generate head_to_tail(f); (See Figure 3.15(b).)
```

Table 3.1: Pseudo-code to generate frontiers for column l + 1

to keep only one copy of such a frontier. Moreover, Lemma 5 shows that it is not necessary to keep all compact frontiers in each column during the frontier generation process. We need the following definition to state Lemma 5. A frontier f for a column l is called a *feasible* frontier if f leads to a routing solution for the channel during the generating process.

Lemma 5 *Suppose f and f' are two symmetric frontiers generated when column l is processed. Then, f is a feasible frontier for column l if and only if f' is also a feasible frontier for column l.*

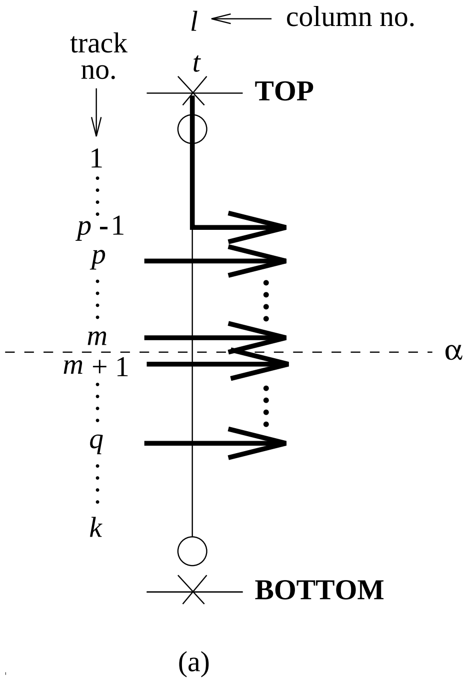


Figure 3.10: *Insert_head*

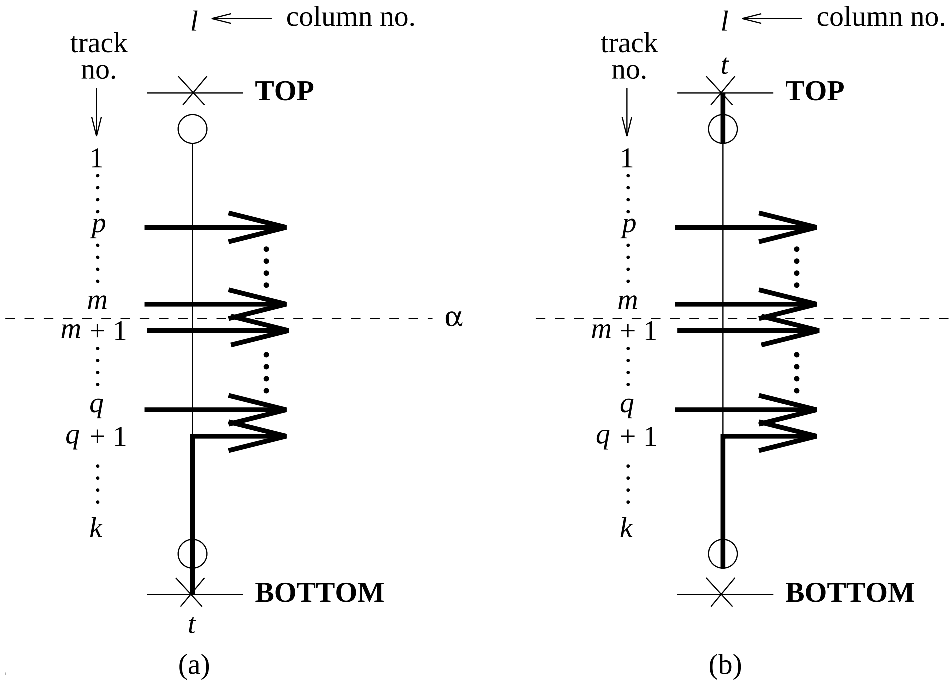


Figure 3.11: *Insert_tail*

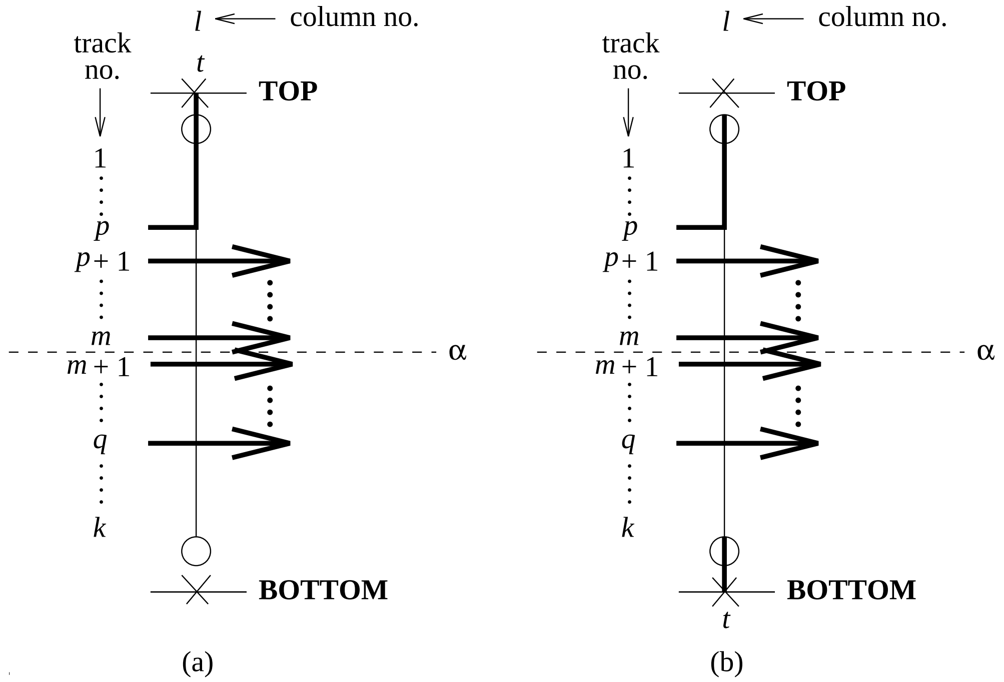


Figure 3.12: *Remove_head*

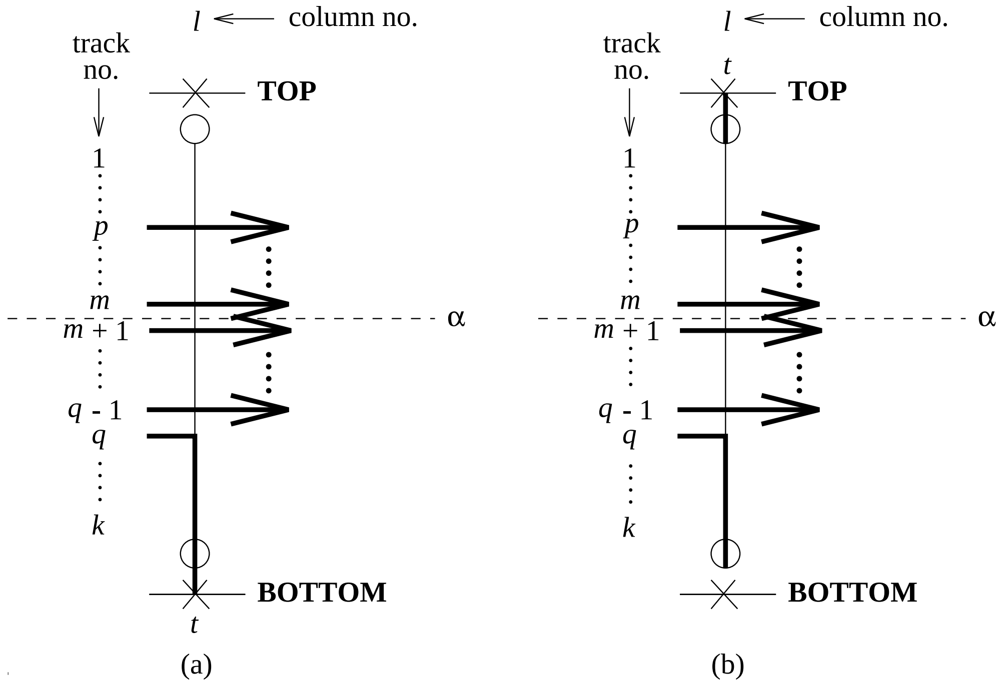


Figure 3.13: *Remove_tail*

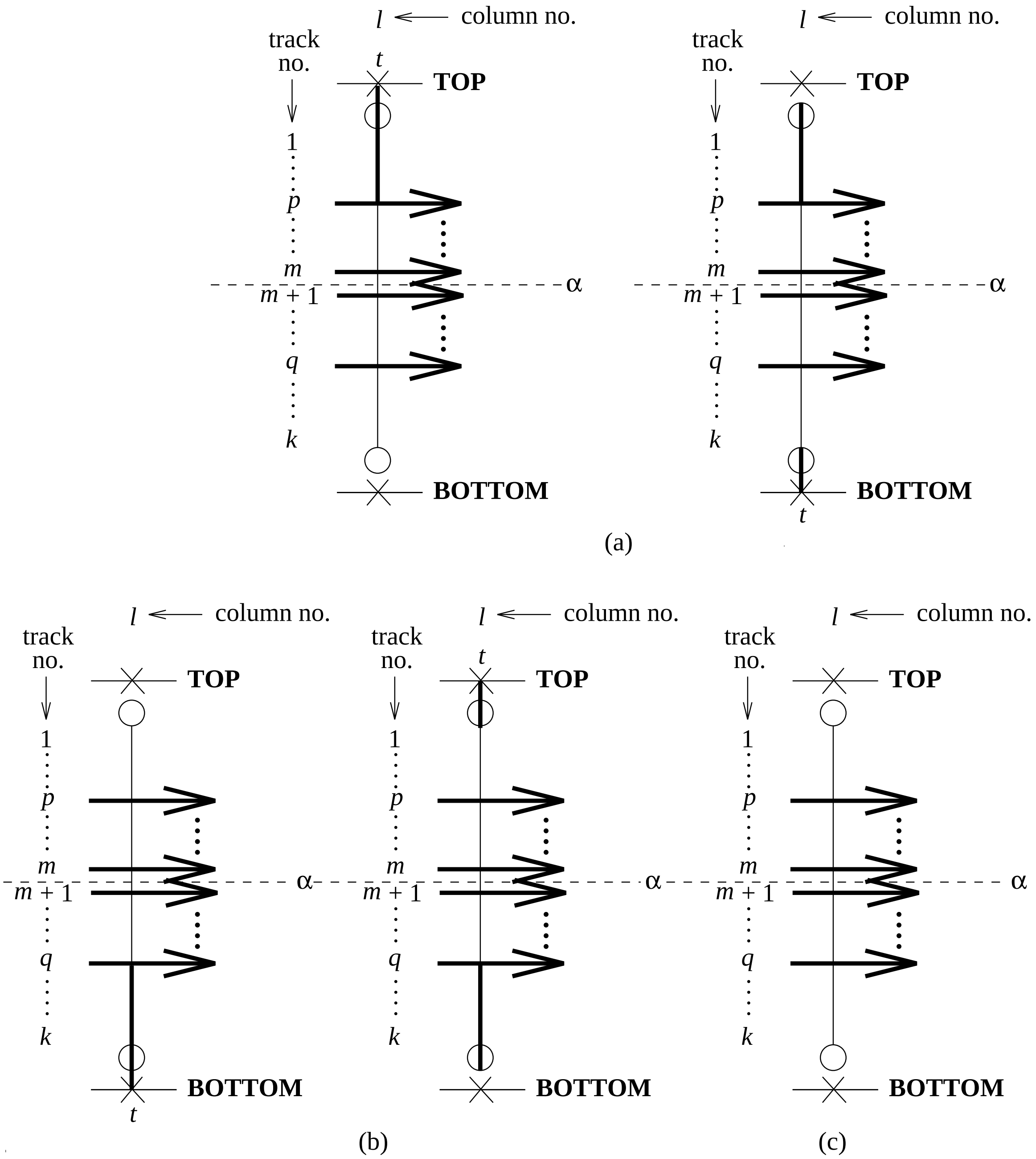


Figure 3.14: *Copy*

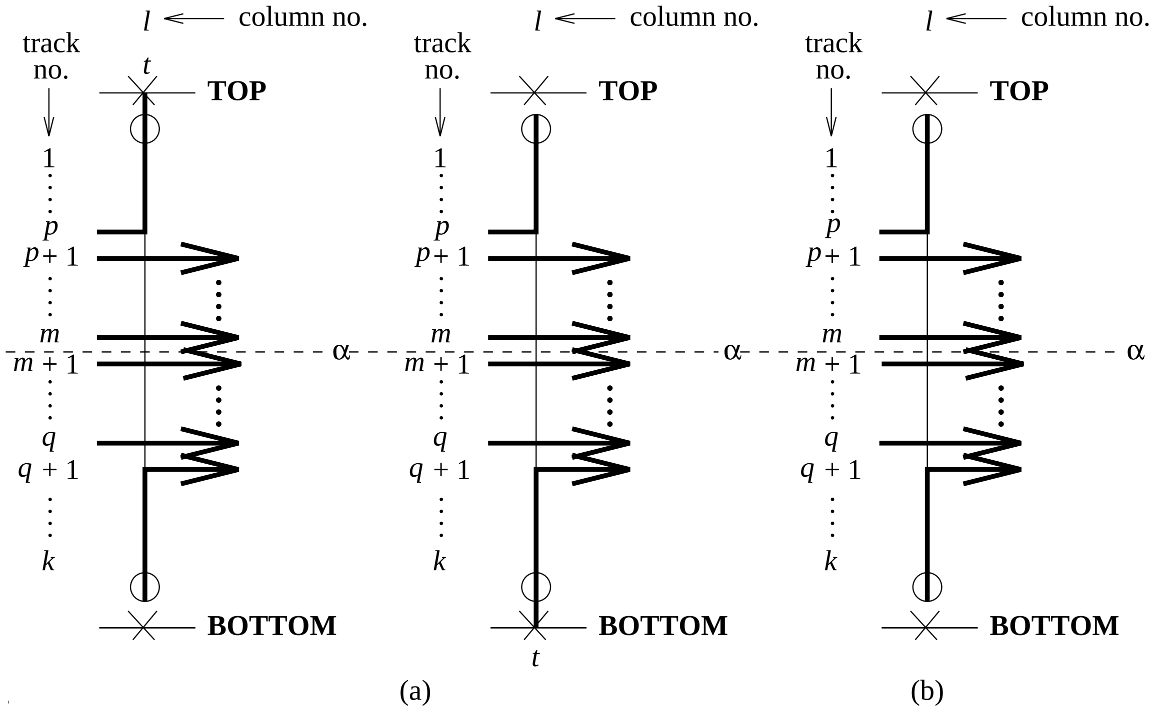


Figure 3.15: *Head_to_tail*

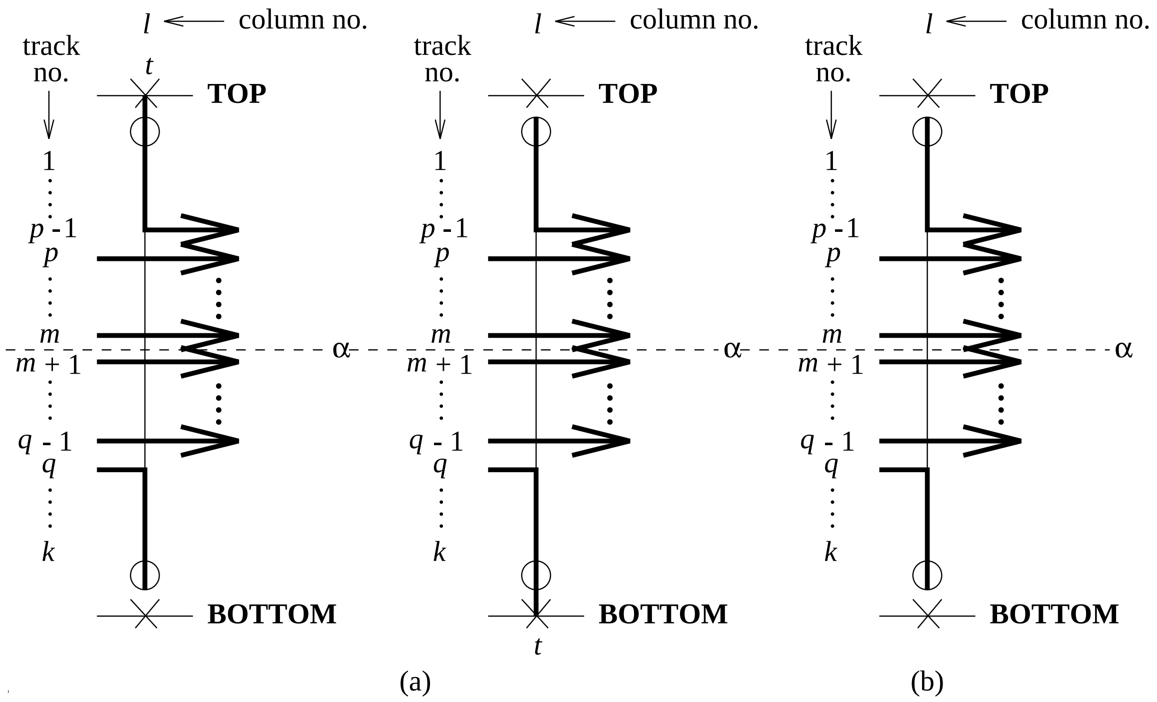


Figure 3.16: *Tail_to_head*

Proof: Suppose we are given a routing R in which f is a frontier for column l. We can construct another routing R' in which f' is a frontier for column l as follows: We take the part of the routing R involving column l and flip it vertically. The resultant routing can be incorporated with f' to provide the part of the routing R' after column l. Figure 3.17 demonstrates such a

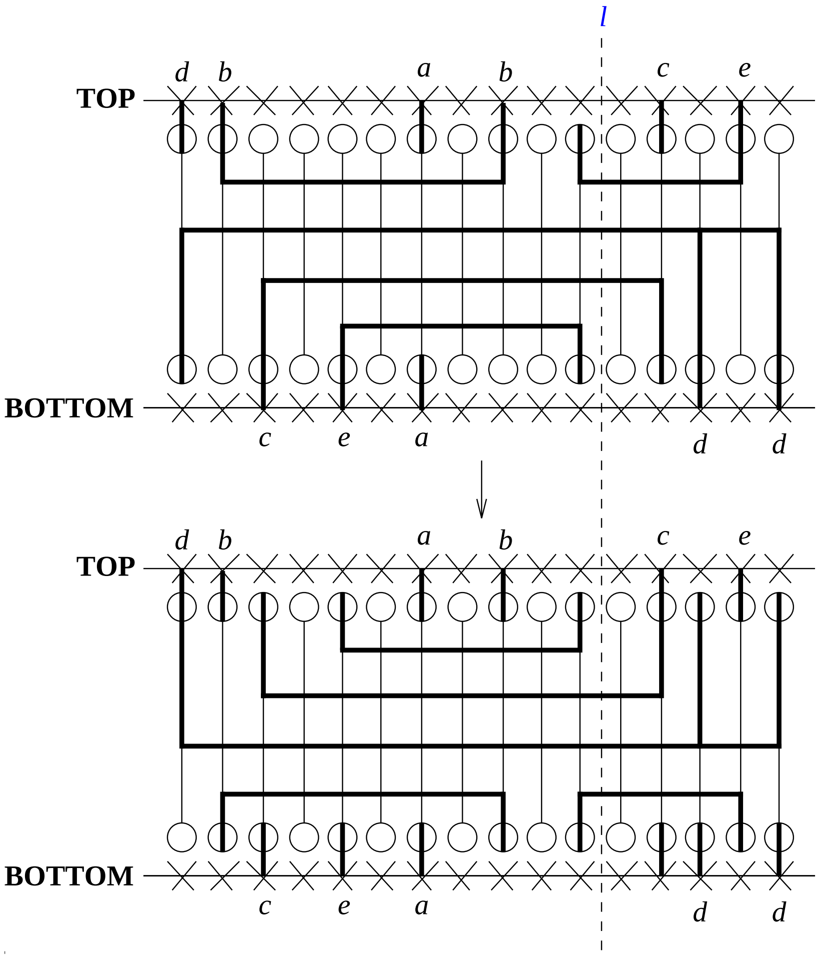


Figure 3.17: The construction used in the proof of Lemma 3

construction. □

Based on this lemma, we need to keep only one of every two symmetric frontiers for each column. In addition to saving memory space, we also save computation time, as many frontiers generated for a column will be discarded and the number of frontiers generated for the remaining columns dramatically reduces.

In the rest of this section, two necessary conditions are given to improve the running time even further. Suppose column l has a terminal which belongs to net t. Then, it is obvious that net t must be an outermost net in any feasible frontier for column $l-1$. If net t in a frontier f for column $l-1$ is not one of the outermost nets in f, then it is impossible to connect the wire carrying net t in column $l-1$ to the terminal in column l. Thus, f cannot be a feasible frontier. This necessary condition can be generalized for any column between $l-k$ and l, as the following lemma states.

Lemma 6 *Suppose there is a terminal of net t in column l and f is a frontier, containing net t, for column l. If f is a feasible frontier, then net t must be one of the mth outermost nets in f for column $l-j$, where $1 \leq m \leq j$ and $1 \leq j \leq k$.*

Proof : To prove this lemma, consider the pseudo-code of our algorithm listed in Table 3.1. Since $t \neq 0$, $t \in f$, and f is a feasible frontier, then one of r_p and r_q must be equal to t. We have proved the case $j = 1$. For $j > 1$, the proof is inductive, using the fact that each of mth outermost nets in f can become $m-1$st outermost net in its child frontier for the next column (by one of *remove_head, remove_tail, head_to_tail,* and *tail_to_head*). □

Before stating our second necessary condition, we need the following definition. If we remove all zeros in a frontier f, then we call the resultant sequence a *simplified frontier.* The second necessary condition can be explained using the example in Figure 3.18. Since nets 1, 4, and 9 have terminals in the next three consecutive columns, all simplified frontiers of feasible frontiers for column l containing nets 1, 4, and 9 must be in one of the forms shown below, where X represents any net other than 0, 1, 4, and 9.

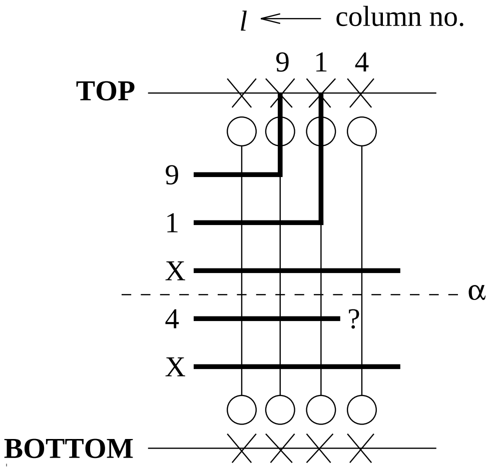


Figure 3.18: The second necessary condition

$$(9,\ 1,\ 4,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X})$$

$$(9,\ 1,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ 4)$$

$$(9,\ 4,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ 1)$$

$$(9,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ 4,\ 1)$$

$$(1,\ 4,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ 9)$$

$$(1,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ 4,\ 9)$$

$$(4,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ 1,\ 9)$$

$$(\mathrm{X},\ \ldots,\ \mathrm{X},\ \alpha,\ \mathrm{X},\ \ldots,\ \mathrm{X},\ 4,\ 1,\ 9)$$

It is easy to verify that any frontier whose simplified frontier is not in the forms shown above would not be a feasible frontier. An example of such frontiers is shown in Figure 3.18. Formally, given a sequence of nets, A, of which a terminal is located consecutively starting from column l, for instance A = (9, 1, 4) in Figure 3.18, we define a function, *extract*, which takes as an argument a simplified frontier f' for column l. The image of this function is a sequence in the form (s_1, α, s_2). s_1 is a sequence of numbers in f' that appears left to the rightmost member of A in f' and left to α. s_2 is a sequence of numbers in f' that appears right to the leftmost member of A in f' and right to α. If we let A be $\{1, 4, 9\}$ and f' be the simplified frontier of the frontier for column l in Figure 3.18, then *extract* gives (9, 1, α, 4, X). If we let f' be those eight simplified frontiers listed above, then *extract* gives the following eight resultant sequences, respectively :

$$(9,\ 1,\ 4,\ \alpha)$$

$$(9,\ 1,\ \alpha,\ 4)$$

$$(9,\ 4,\ \alpha,\ 1)$$

$$(9,\ \alpha,\ 4,\ 1)$$

$$(1,\ 4,\ \alpha,\ 9)$$

$$(1,\ \alpha,\ 4,\ 9)$$

$$(4,\ \alpha,\ 1,\ 9)$$

$$(\alpha,\ 4,\ 1,\ 9)$$

It is easy to check that each such a sequence is a subsequence of (9, 1, 4, α, 4, 1, 9). Formally, we use S to denote $(A,\ \alpha,\ reverse(A))$, where $reverse(p)$ is the resultant sequence obtained by flipping a sequence p. According to the above reasoning, for the simplified frontier of each feasible frontier for column l, *extract* must give a subsequence of S. Therefore, for each simplified frontier of frontiers for column l containing nets 1, 4, and 9 in Figure 3.18, if *extract* gives a sequence that is not a subsequence of S, then we can discard the simplified frontier. This necessary condition for the frontiers for column l can be used to derive a necessary condition for the frontiers for column $l-1$ as follows. Any frontier f for column $l-1$ whose simplified frontier f' does not satisfy one of the following two conditions can be discarded:

1. *extract*(f') must be a subsequence of $(X,\ S,\ X)$.

2. applying *extract* to the simplified frontier of either *head_to_tail*(f) or *tail_to_head*(f) gives a subsequence of S.

The first condition includes those frontiers for column $l-1$ that generate feasible frontiers for column l by *copy*, *insert_head*, *insert_tail*, *remove_head*, or *remove_tail*. The second condition includes frontiers for column $l-1$ which generate feasible frontiers for column l by *head_to_tail* or *tail_to_head*. These two necessary conditions can be recursively applied to derive the necessary conditions for the frontiers for column $l-2$, $l-3$, etc. We call each individual necessary condition a *subsequence condition*. Then, we have

Lemma 7 *All feasible frontiers for column $l-i$ must satisfy the subsequence condition for the frontiers for column $l-i$.*

Since *head_to_tail* and *tail_to_head* uses a pre-placed segment, and thus introduces extra vias in the routing which degrade the performance, we want to find a routing with a minimum number of total vias. This can be achieved as follows. We construct a layered directed acyclic graph, called a *frontier graph*, $G = (V, E)$ where

1. V is the set of feasible frontiers generated during the process;

2. there is a layer corresponding to to a column from column 0 to column w;

3. vertex u in layer l has a directed edge to vertex v in layer $l+1$ if and only if the frontier f_v corresponding to v is a child of the frontier f_u corresponding to u during the process; Moreover, if *head_to_tail*$(f_u) = f_v$ or *tail_to_head*$(f_u) = f_v$, then the weight of edge (u, v) is 1, and 0 otherwise.

Note that there is only one vertex in the first layer and the last layer, respectively. The vertices are denoted by s and t, respectively. s has a directed edge to each vertex in the layer corresponding to column 1 with weight 0, and t has a directed edge from each vertex in the layer corresponding to column $w-1$ with weight 0. Both s and t correspond to trivial frontiers for column 0 and w, respectively. Figure 3.19 shows the frontier graph for the channel with k being 4 in Figure 3.2. In Figure 3.19, the bold circles indicated correspond to the frontiers for the columns in Figure 3.4 and the cost of each bold edge is 1, while that of the rest is 0. Now, the problem of finding a routing with minimum number of total vias becomes a shortest-path problem in a directed acyclic graph from vertex s to t, which can be solved optimally in $\Theta(|V| + |E|)$[2]. Note that the number of child frontiers of any frontier is bounded by a constant, 2, according to our algorithm in Table 3.1. Therefore, $|E|$ is $\Theta(|V|)$ in the frontier graph, and the shortest-path problem mentioned above can be solved in $\Theta(|V|)$.

3.5 Experimental Results

Our algorithm was implemented in C and executed on a SPARC-2 station. Six channel routing problems were used to test our algorithm. The first three channel routing problems were for the one and half layer routing structure described in [47]. The one and half layer routing structure is similar to QCL with several exceptions. In one and half layer routing, pre-placed segments are prefabricated poly-silicon, only one metal layer is used, and masks for both the contact holes and the metal layer can be customized. Recall that there must be no vertical constraints in QCL channels, whereas one and half channel routing channels allow vertical constraints. In order to eliminate vertical constraints in the test channels for one and half layer technology, we stretch each test channel and alternate the appearance of top terminals and bottom terminals. Therefore, it is not surprising that QCL routing structure uses more columns than the one and half layer routing structure. The three channels and the comparison of our results with

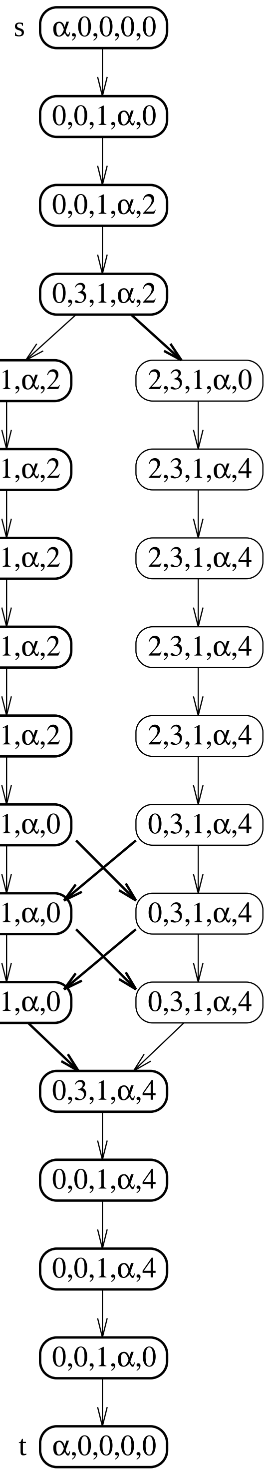


Figure 3.19: A frontier graph

	Channel1	Channel2	Channel3	Channel4	Channel5	Channel6
# Terminals	82	78	74	56	73	61
# Column(Ours)	159	146	132	142	128	129
# Column([47])	101	101	101	*	*	*
# Tracks(Ours)	11	9	10	14	18	16
# Tracks([47])	10	9	8	*	*	*
Density	7	5	7	9	12	12
Time(sec.)	7.0	2.7	21.4	37.2	80.1	78.9

Table 3.2: Experimental Results for QCL channel routing

the results in [47] are summarized in the first three columns of Table 3.2. Note that our experiments show that QCL routing structure uses only one or two more tracks than one and half layer routing structure. The routing solutions obtained for these three channels are shown in Figure 3.20. The arrow in the third channel points to a column inserted during the execution of our program. The algorithm fails to generate any frontier for the next column and, thus, inserts a column to continue the execution. Our algorithm runs very quickly in these three examples.

The remaining three channels are extracted from a bus controller circuit implemented in QCL structure. The results are summarized in the last three columns of Table 3.2. The routing solutions for these three channels are shown in Figure 3.21. Note that all the routing solutions we obtained use a minimum number of vias. Although the largest running time in these six test channels is less than one and half minutes, the running time of our algorithm depends on the number of frontiers generated in each column in general. Channels that generate a large number of frontiers in columns usually have many routing solutions. In such circumstances, generating all frontiers would incur too much computation time. Therefore, we set a control factor, K, whose value was typically between $10^2 \sim 10^3$, to bound the number of frontiers generated in each column. If K generated frontiers fail to lead to a routing of the whole channel, then we backtrack and generate a new set of K frontiers for this column and continue the process.

3.6 Summary

In this chapter, we formulate channel routing problem in QCL structure. We prove that this problem is strongly NP-complete by showing that a special case of it is NP-complete. Therefore, it is unlikely to have a polynomial-time algorithm to guarantee a routing, if one exists. We propose an exhaustive algorithm that processes one column at a time from left to right. We show that, by examining only small portion of all possible configurations in each column, we can still guarantee a routing, if one exists. Moreover, two necessary conditions are proposed to reduce the number of configurations generated in each column. The routing with a minimum number of total vias can be generated easily by using a linear-time algorithm for the single-source single-destination shortest-path problem on a directed acyclic graph. Experimental results show that this approach is very promising.

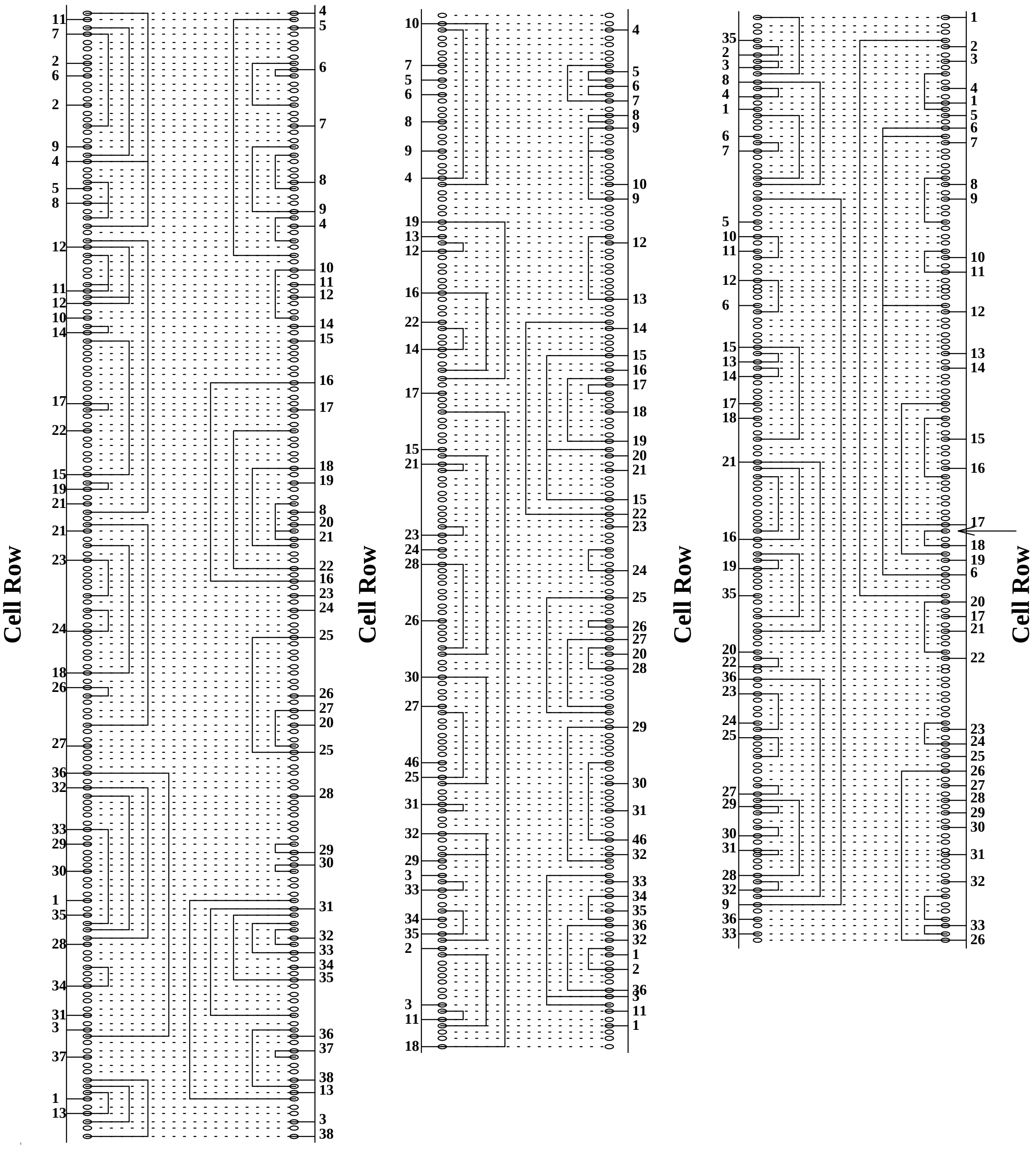


Figure 3.20: Three routing examples used in [47]

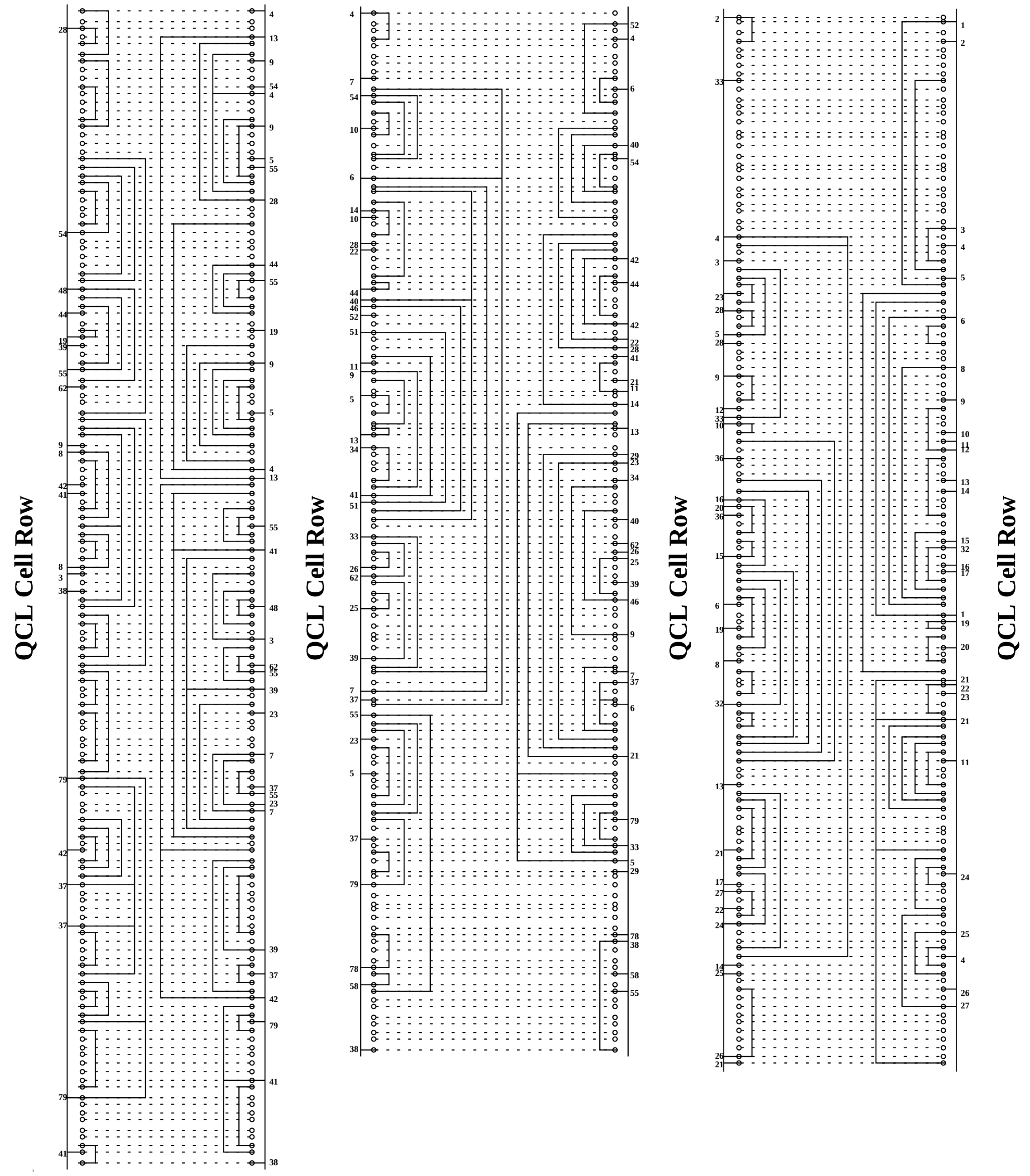


Figure 3.21: Three channels extracted from a real circuit implemented in QCL structure

Chapter 4

Area Routing for Channelless QCL Gate Arrays

4.1 Introduction

As mentioned in Chapter 3, two metal layers are utilized in QCL technology. Wiring on the first metal layer and consequently the corresponding mask are fixed. The second metal layer is for customized wiring and the mask for this layer will be custom-designed. In comparison with the conventional two-layer technology, QCL reduces turn-around time for fabrication of engineering samples to a few days (typically 2-5 days) by reducing multiple customized masks to a single mask. (As will be seen later, there are only fixed-position vias and contact holes in QCL technology. Consequently, the two masks for the vias and contact holes are also pre-designed.) Moreover, the performance of QCL chips is comparable to those designed by two-layer customization. Despite of such advantages, original QCL chips using an architecture similar to the conventional gate array architecture with fixed routing channels suffer from inefficient usage of the silicon area and low gate utilization rate. In [13], a channelless QCL gate array architecture was introduced. In this architecture no routing channel is reserved for wiring. It turns out that this architecture improves both routability and utilization of the silicon area substantially.

In this chapter, we consider the routing problem in channelless QCL gate array chips. Section 4.2 describes QCL technology in details. In Section 4.3 the routing problem is formulated. The problem is shown to be NP-complete and a heuristic algorithm based on features of this particular architecture is presented in Section 4.4. Results of an industrial example are provided in Section 4.5.

4.2 QCL Channelless Gate Array Architecture

In QCL technology there is a set of pre-placed wire segments (or pre-placed segments for short) on the first metal layer. They are equally spaced vertical wire segments of the same length and are arranged in rows. At each end of a wire segment there is a via connecting the first metal layer to the second. In convention QCL gate arrays there are rows of transistors and explicit channel areas in the inner region of the chip as shown in Figure 4.1. Within a row of transistors, there are transistor islands separated by pre-placed segments. A transistor island consists of a number of transistors which are chained together by diffusion. Cell circuits are formed by transistor islands. The pre-placed segments between transistor islands are used as bypassing wires to cross the transistor rows. In the channel, there are plenty of pre-placed segments for interconnections between cell circuits. Since the area reserved exclusively for the channel cannot be changed and the number of tracks in the channel is fixed, this architecture usually results in poor silicon usage and low gate utilization rate.

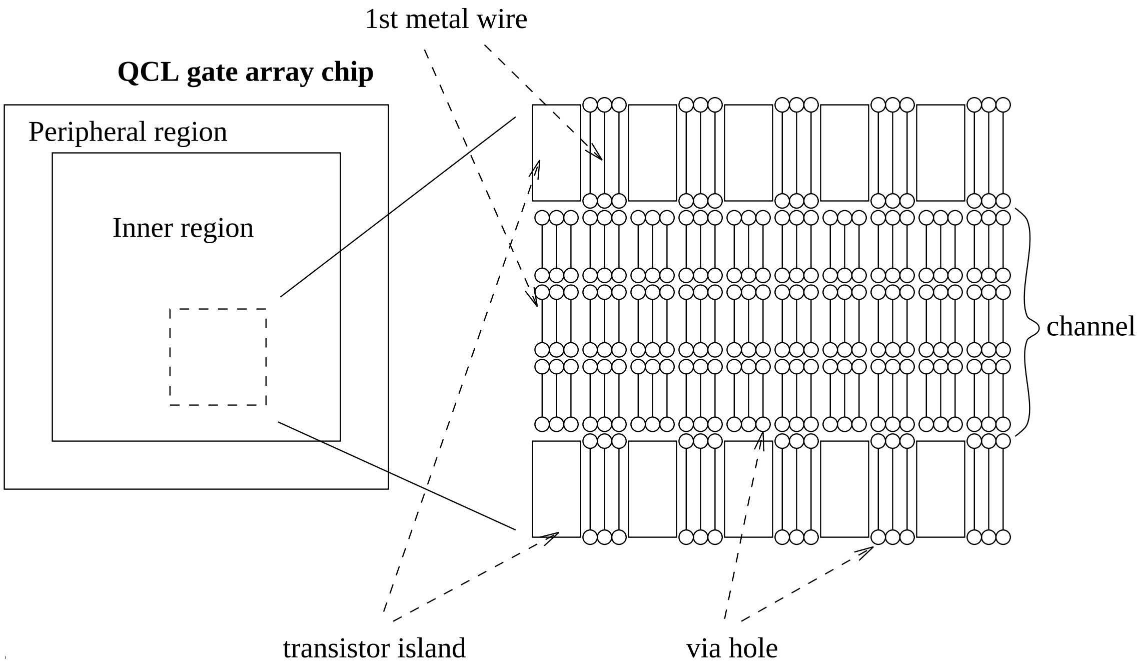


Figure 4.1: Conventional QCL gate array

A variation of the conventional QCL gate array architecture is the channelless QCL gate array architecture [13]. In this architecture, both p-type and n-type transistors are embedded in the inner region of the chip while there is no explicit channel area as demonstrated in Figure 4.2. Rows of p-type transistor chains and n-type transistor chains appear alternately in the vertical direction. There are two types of pre-placed segments on the first metal layer. Those that lie above the gate of a transistor are called *type-1 pre-placed segments*, and those that lie above the source or the drain of a transistor are called *type-2 pre-placed segments.* These two types of pre-placed segments in the p-transistor chains are shown in Figure 4.3. Note that 180 degree rotation gives pre-placed segments in the n-transistor chains. A pre-placed segment of either type has one contact hole for connecting the segment to the corresponding part of the transistor and three via holes for connecting the segment to customized wires on the second metal layer. The area between the top two vias in each pre-placed segment is reserved for the

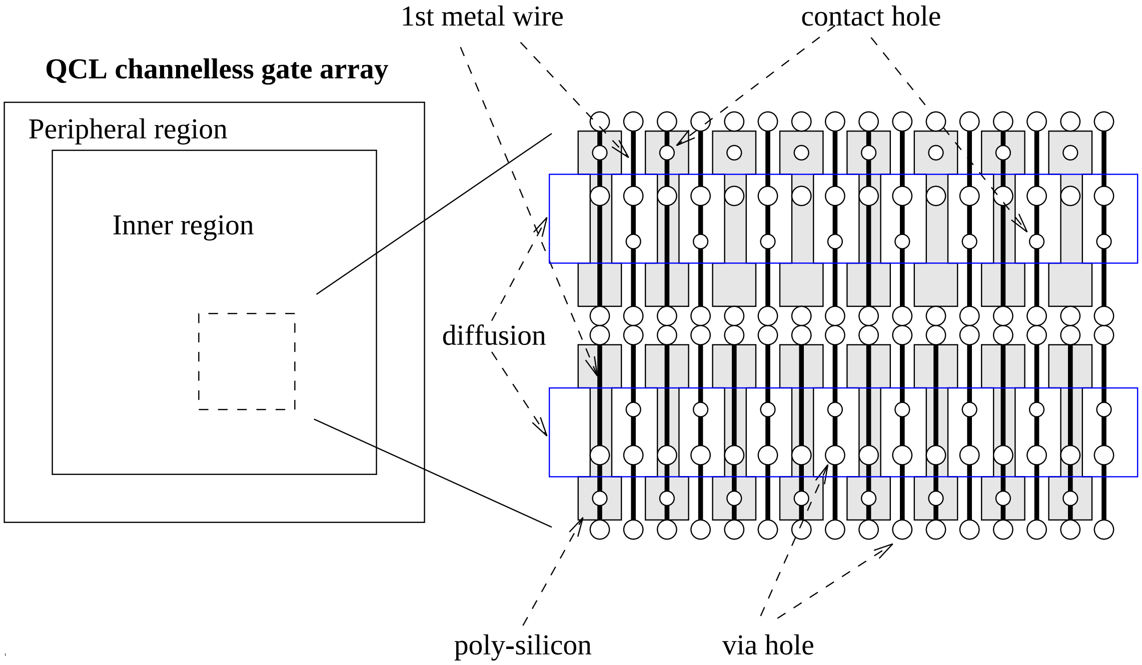


Figure 4.2: Channelless QCL gate array

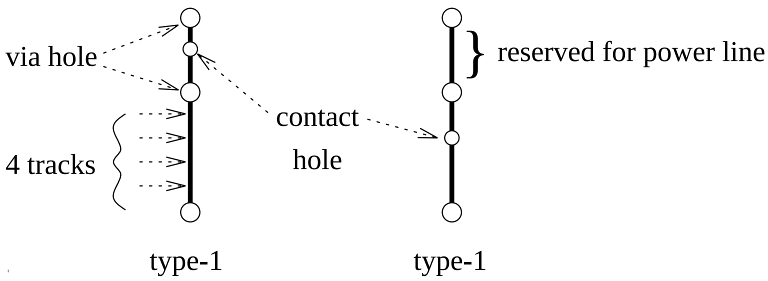


Figure 4.3: Pre-placed segments in the p-transistor chains

power line. Signal interconnections use the remaining part of the segment in which the number of tracks cannot be too large nor too small due to the trade-off considerations between area and performance efficiency. For example, 4 is a typical value in practice which is what we shall assume throughout the chapter. Fabrication of the wafer proceeds in the order of transistors, contact holes, first metal layer, vias, and second metal layer. All layers but the second metal layer in this sequence can be processed in advance.

Figure 4.4 shows how interconnections are made on the second metal layer to build a 2-input NAND gate. For simplicity, the poly-silicon layer is omitted from the figures, hereafter, since it can be identified by the type of the pre-placed segment. Because transistors are chained

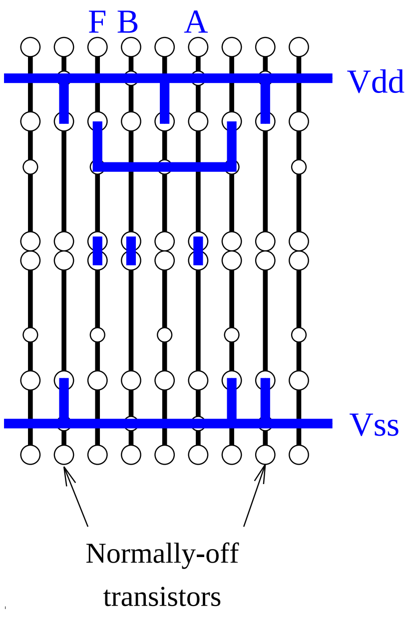


Figure 4.4: F = NAND(A, B)

by sharing the diffusion, we need to use *normally-off* transistors to isolate the 2-input NAND gate from the rest of the circuit. A normally-off transistor is built by connecting the type-1 pre-placed segments to the corresponding voltage line. Each pre-placed segment can be used for inter-cell wiring as long as it is not used for building cell circuits. However, we shall be careful when pre-placed segments are used for this purpose. When a type-1 pre-placed segment carries a '1' signal, the source and the drain of the transistor are at the same potential. Therefore, if the two adjacent type-2 pre-placed segments carry two different signals, a 'short' circuit will result. The easiest way to avoid this is to use only the type-2 pre-placed segments for inter-cell interconnection. Consequently, only every other pre-placed segment will be used for inter-cell wiring. Figure 4.5 shows an example. Since for the inter-cell routing problem considered in

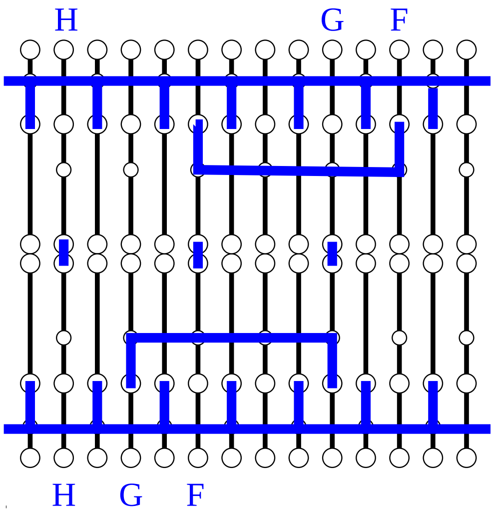


Figure 4.5: Inter-cell routing

this chapter, only type-2 pre-placed segments will be used, we shall refer to them simply as pre-placed segments hereafter, unless otherwise stated. Moreover, we shall omit the power lines in all figures for the sake of clarity.

4.3 Problem Formulation and Related Works

The basic architecture of channelless QCL gate arrays offers a variety of chip layout styles. For example, the row-based design style similar to conventional QCL gate arrays, the building block design style, etc. In the building block design style, building blocks can be built using the active devices in different rows or columns. Furthermore, they can occupy any position on the chip. The algorithm proposed in this chapter is applicable for all design styles.

We introduce first the following notations: Suppose we have a rectangular region, R, in which there are m rows and n columns of pre-placed segments. We call the pre-placed segment in the xth row and the yth column the $(x,\ y)$-*segment.* The lower left pre-placed segment is defined to be the $(0, 0)$-segment. A *net* is a set of pre-placed segments that are to be connected together. Each pre-placed segment in a net can be either a type-1 or a type-2 segment and is referred to as a *terminal.* For a given net-list, A, let $t(i)$ denote the number of terminals in net i. Moreover, We order the terminals of a net according to their x-coordinates. Thus, net i is represented as $\{(x^i_j, y^i_j) | x^i_j \leq x^i_{j+1}, 1 \leq j \leq t(i)\}$. Since inter-cell routing uses only type-2 pre-placed segments, in the figures hereafter, only type-2 pre-placed segments and type-1 pre-placed segments used by intra-cell routing will be shown.

The QCL area routing problem can be stated as follows: Suppose that the positions and layouts of all cell circuits are given in a region R. For a given net-list A which specifies the interconnection among the cell circuits, the problem is to determine the routing for each net i in A such that all $(x^i_j,\ y^i_j)$-segments $(1 \leq j \leq t(i))$ are connected within R. Figure 4.6 shows an example of a routing problem.

Conceptually, wires and pre-placed segments used in the layout of cell circuits can be viewed as obstructions when we search for a routing solution. For performance consideration, it is desirable not to use zig-zag wires (as shown in Figure 4.7(a)) in a row of pre-placed segments. That is, only those shown in Figure 4.7(b) are allowed. For simplicity, we shall restrict our discussion to utilizing only the ⊔-shaped and ⊓-shaped wires in the rest of the chapter. Extending

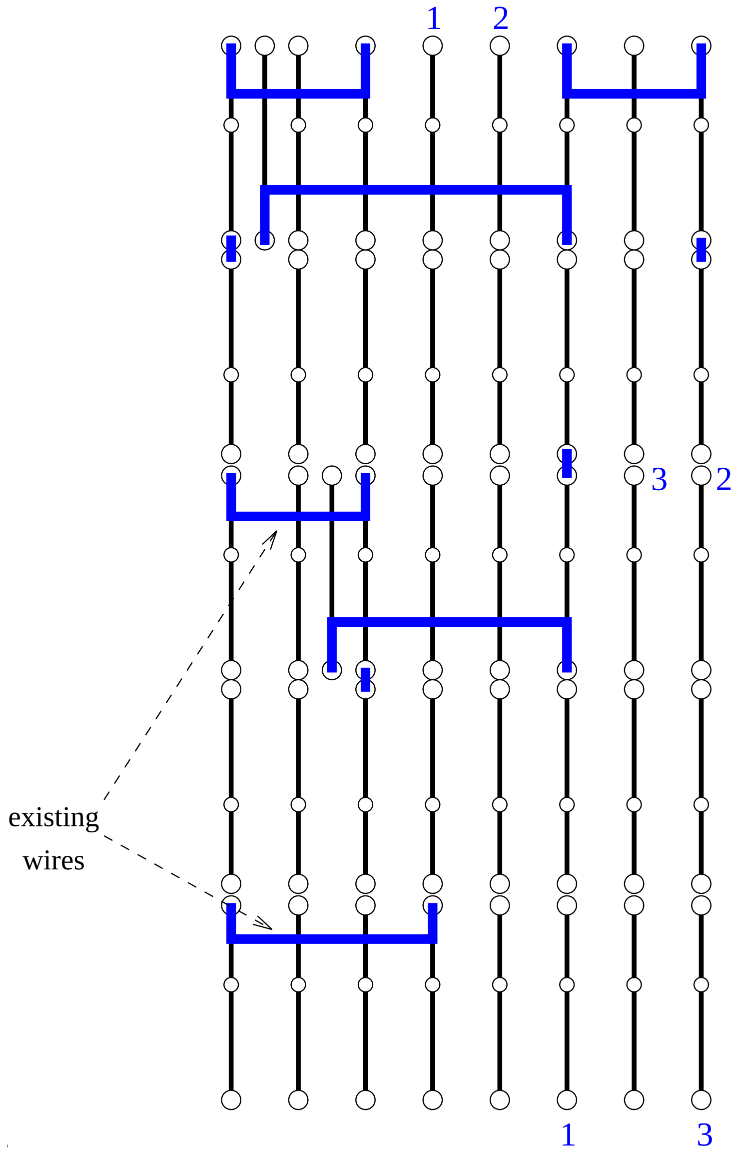


Figure 4.6: A routing problem

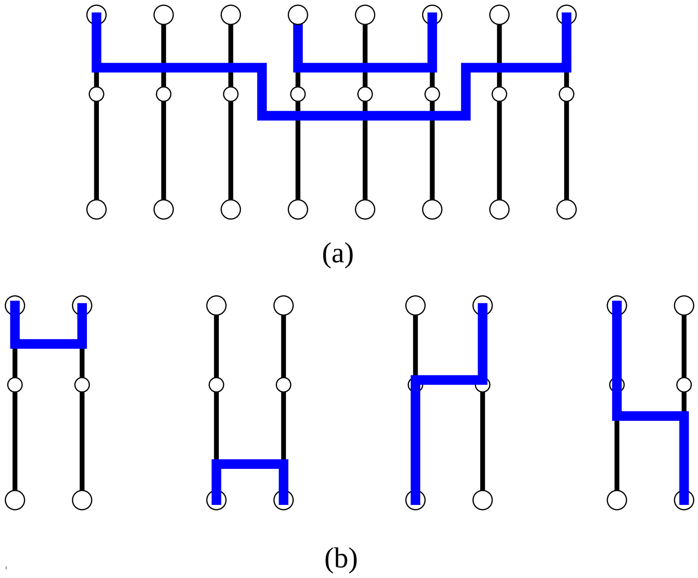


Figure 4.7: (a)Zig-zag wires (b) Four types of wires

our results to include the other crank wires is straightforward. Figure 4.8 shows a solution to the routing problem in Figure 4.6.

No existing paper addresses the routing problem in the QCL channelless gate array architecture. For the conventional channelless gate arrays, Hsu *et el.* [22] proposed a layout system for the row-based design style. A system for building-block design style was presented in [46]. ORCA [24] is another place-and-route system for both row-based and building-block design styles. However, because of the limited layout resources in QCL technology, direct application of existing approaches for conventional channelless gate arrays to our routing problem usually results in a low routing completion rate. Therefore, development of new routing algorithms capturing the special features of QCL architecture is crucial.

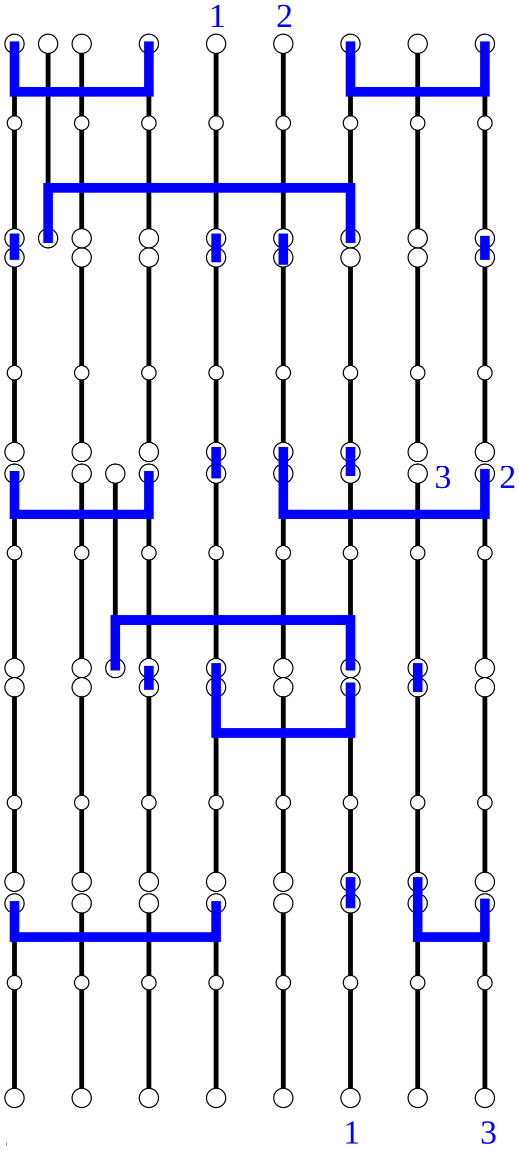


Figure 4.8: A routing solution

4.4 A Routing Algorithm

We prove that the routing problem defined in Section 4.3 is NP-complete in Section 4.4.1. A routing algorithm is presented in Section 4.4.2.

4.4.1 NP-completeness of the Channelless QCL Routing Problem

The channel-width minimization problem in Manhattan channel routing is reduced to our routing problem to show that our problem is NP-complete [17].

Theorem 5 *The routing problem in channelless QCL gate arrays is NP-complete.*

Proof: It is obvious that our routing problem is in NP. Consider the following channel-width minimization problem[1]:

CHANNEL-WIDTH MINIMIZATION PROBLEM (CWMP)

Instance. A conventional two layer channel with terminals on the top and the bottom boundaries and a non-negative integer t.

Question. Is there a routing solution using at most t tracks in the Manhattan model?

This problem was proved to be NP-complete by Szymanski [51]. Given an instance of the CWMP problem, we construct a corresponding QCL routing problem instance as follows. Let l be the number of columns in the channel in CWMP. Consider a routing region with $t \times (l+6)$ pre-placed segments. For each row of pre-placed segments, there are existing wires connecting the leftmost and the rightmost three pre-placed segments as shown in Figure 4.9. Then, we copy the terminals on the top (bottom) channel boundary in CWMP to the first (last) row of pre-placed segments such that if net i has a top (bottom) terminal at the jth terminal position counting from the left, then the $(1, j+3)$-segment ($(t, j+3)$-segment) is a terminal of net i in

[1] Refer to [32] or [62] for the terminology used in the context of channel routing.

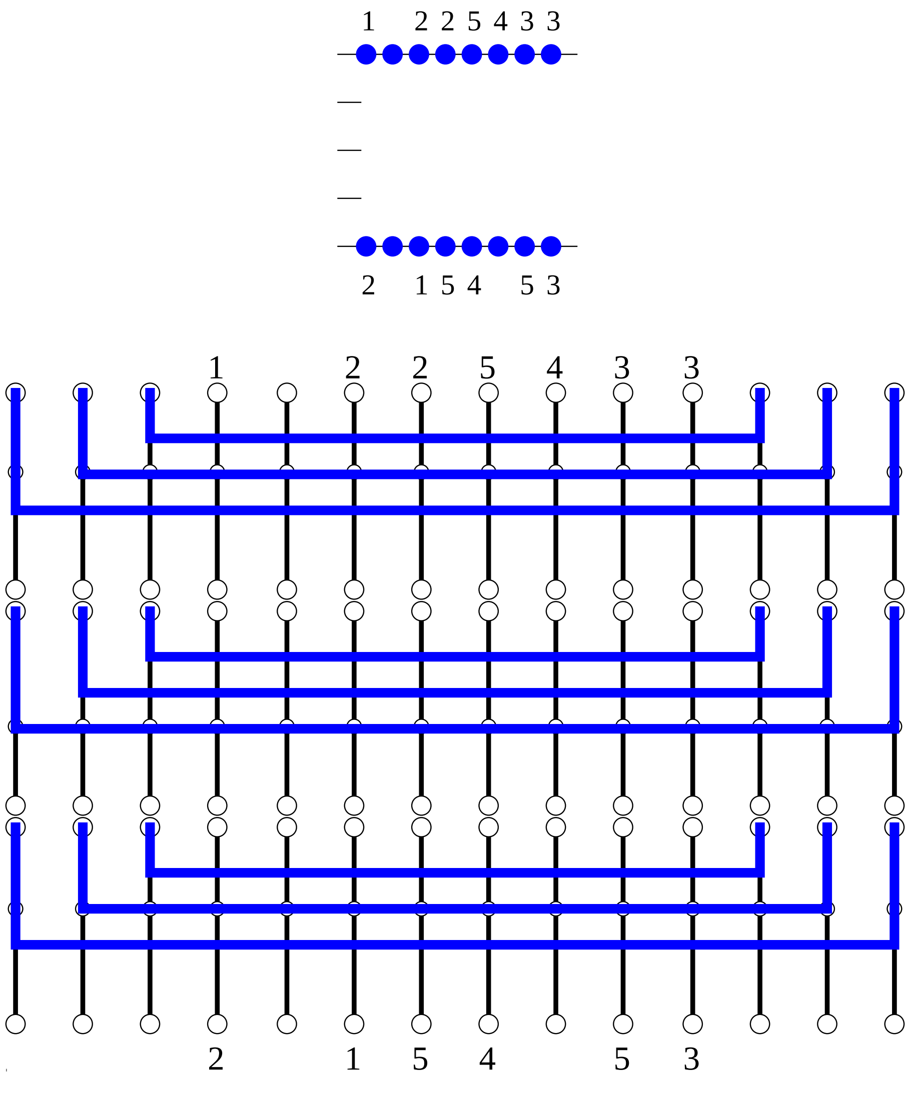


Figure 4.9: Construction of the QCL routing problem

our problem. Obviously, such a construction can be done in polynomial time. Moreover, it is not hard to see that if one of these two problems has a solution, then the other will also have a solution. The correspondence is illustrated in Figure 4.10. □

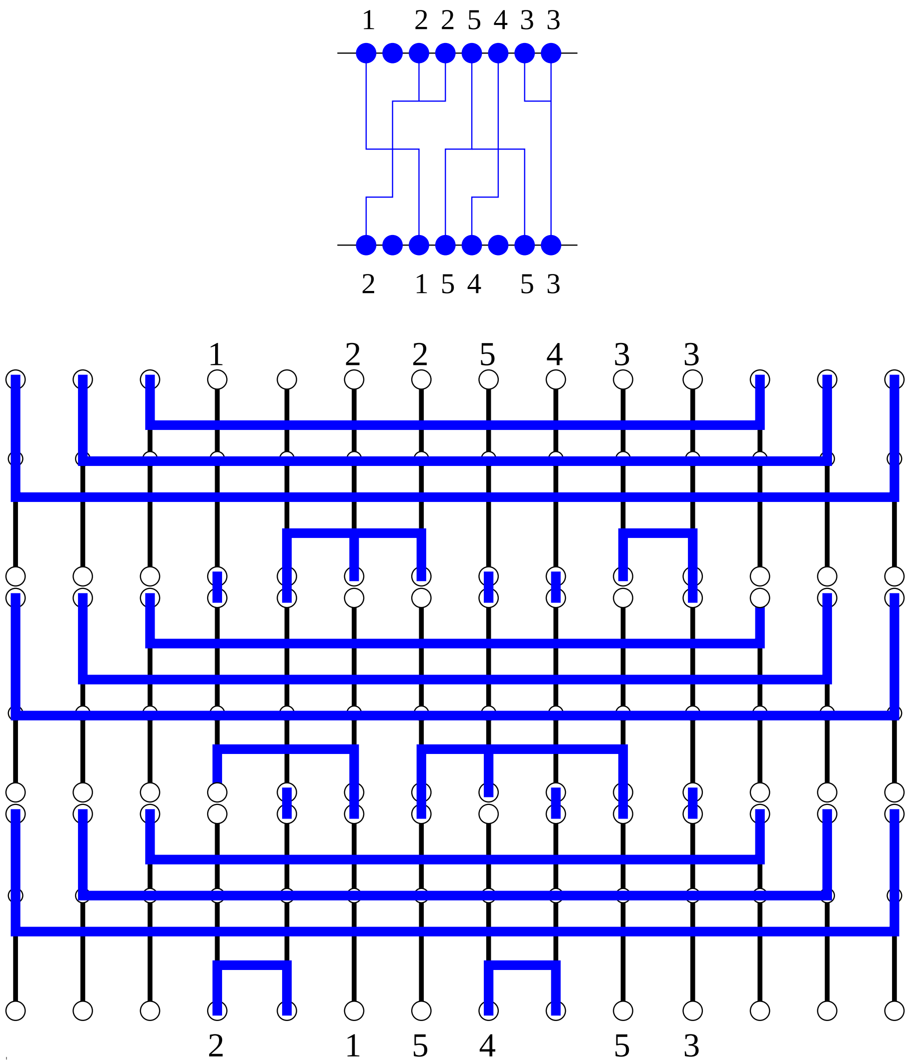


Figure 4.10: Proof of Theorem 1

4.4.2 A Routing Algorithm

To solve the channelless QCL routing problem, we use a two-phase approach, a global routing phase followed by a detailed routing phase. Consider net 1 in Figure 4.8. Since the terminals of net 1 are located in different rows of pre-placed segments, routing for net 1 must pass through the rows of pre-placed segments between the two terminals. A pre-placed segment that is used for a net to cross a row of pre-placed segments is called a *feedthrough*. In the global routing phase, we decide for each net the rows of pre-placed segments in which its ⊔-shaped or ⊓-shaped wires will reside and the set of pre-placed segments that can be used as feedthroughs. Then, a detailed router is used to decide the actual shapes, lengths, and positions of wires used by each net and the exact pre-placed segments to be used as feedthroughs. For global routing, a coarse grid is first superimposed on the chip. An area enclosed by the coarse grid lines is called a *global cell*. Each global cell consists of four consecutive pre-placed segments in the horizontal direction as shown in Figure 4.11. Each net is identified with a set of global cells in which its

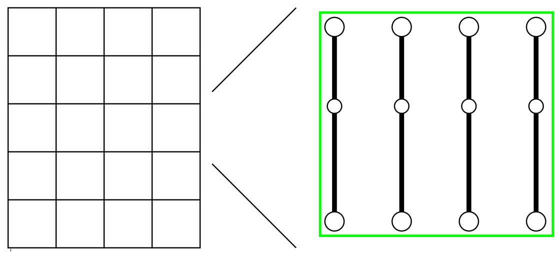


Figure 4.11: Division of the chip area into global cells

terminals are located. A global route of a net is a wire route on these global cells. Note that wires of at most four different nets can pass through each side of a global cell, depending on whether there are existing wires which cross the global cell. We define the *capacity* of a side of a global cell to be four minus the number of existing wires passing through the side. To obtain a global routing, we use the hybrid routing algorithm in [31]. The reasons for choosing this algorithm are:

1. It combines the maze-running approach and the hierarchical routing approach;
2. It can handle equivalent terminals[2];
3. It produces good global routing results and performs efficiently in the sea-of-gates layout structure.

Figure 4.12 shows an example of the result of global routing.

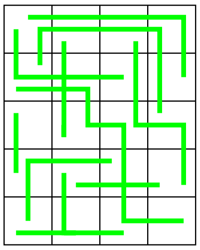


Figure 4.12: A global routing

After global routing has been obtained, our detailed router generates the detailed routes corresponding to the global routes by processing one row of pre-placed segments at a time. Consider a row of pre-placed segments. The actual positions of the two ends of a global route are already specified by the terminals of the corresponding net. However, for a global route that passes through the common vertical side of two adjacent cells we need to determine the exact horizontal track used by the corresponding detailed route. Also, if a global route passes through the bottom side of a global cell, then all four pre-placed segments in this global cell are *candidates* for connecting the corresponding detailed route to the row below. To exhaust

[2] Two terminals of a net are *equivalent* if it suffices for the routing of the net to connect one of them.

all such possibilities, we proceed column by column from left to right, generating all possible configurations of the track usage by the nets passing through a column from the configurations for the previous column. If this configuration generation process can proceed to the rightmost column, then we can backtrack the generation process to find a sequence of configurations which forms a routing solution for this row. However, it is possible that not all global routes in a row of global cells can be completed. Therefore, we want to complete as many global routes as possible during the exhaustive search process. To be more specific, suppose we are processing column l and the set of global routes in the row is A. A configuration is represented by a sequence of four ordered pairs followed by a number α. Each ordered pair is of the form $(netid, shape)$, where $netid \in A \cup \{0\}$, and $shape \in \{\sqcap, \sqcup\}$. The ith ordered pair $(netid, shape)$ specifies that net *netid* occupies the ith track in column l and the shape of the wire is *shape*. If *netid* is 0, it means that the corresponding track is unoccupied. (The corresponding *shape* can be either $\sqcap$ or $\sqcup$, since in this case the information is meaningless.) The first (last) ordered pair whose *netid* is non-zero is called the *first* (*last*) *non-zero ordered pair.* α records the cardinality of the largest subset of A, A^*, which satisfies the conditions:

(1) there is a routing solution for the nets in A^* in column 1 through l - 1;

(2) in column l, the track usage of the routing solution in (1) is specified by the four ordered pairs.

An example of a configuration for column l ((1, $\sqcup$), (3, $\sqcup$), (0, $\sqcup$), (6, $\sqcap$), α) is shown in Figure 4.13. Based on the configurations at column l, the configurations at column $l+1$ are generated according to the following rules: Suppose c is a configuration at column l.

- If column l is a column at which the left terminal of a net, say net i, in A can be located and at most three ordered pairs in c have non-zero *netid*, then we generate two configurations

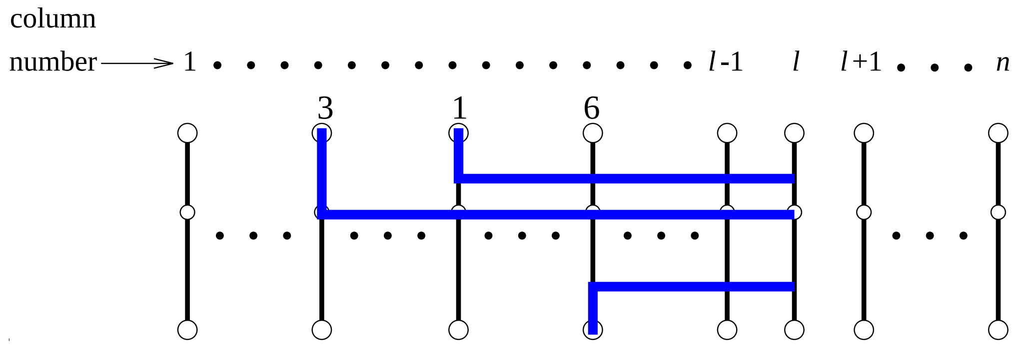


Figure 4.13: An example of a configuration

for column $l+1$, one including net i in the first non-zero ordered pair and the other in the last non-zero ordered pair. Figure 4.14 shows the configurations at column $l+1$ generated from the configuration at column l in Figure 4.13. Figure 4.14 (a) shows a configuration for column l+1: (1, $\sqcup$), (3, $\sqcup$), (6, $\sqcap$), (4, $\sqcap$), α, where net 4 uses the lowest track. Figure 4.14 (b) shows a configuration for column l+1: (4, $\sqcup$), (1, $\sqcup$), (3, $\sqcup$), (6, $\sqcap$), α, where net 4 uses the topmost track. In addition, c is included as a configuration at column $l+1$.

- If *shape* of the first non-zero ordered pair is $\sqcup$ and if its *netid* can terminate at column l, we generate a configuration for column $l+1$ in which *netid* of that ordered pair is changed to 0. The same for the case in which *shape* of the last non-zero ordered pair is $\sqcap$ and its *netid* can terminate at column l. The value α of the generated configuration is that of c plus 1. Suppose net 6 in Figure 4.13 ends at column l. Then the corresponding configuration for $l+1$ is, shown in Figure 4.15, (1, $\sqcup$), (3, $\sqcup$), (0, $\sqcup$), (0, $\sqcup$), $\alpha + 1$.

- If l is not the last candidate at which the right end of each net in c can be located, we include c as a configuration at column $l+1$ and conditionally generate the following two configurations:

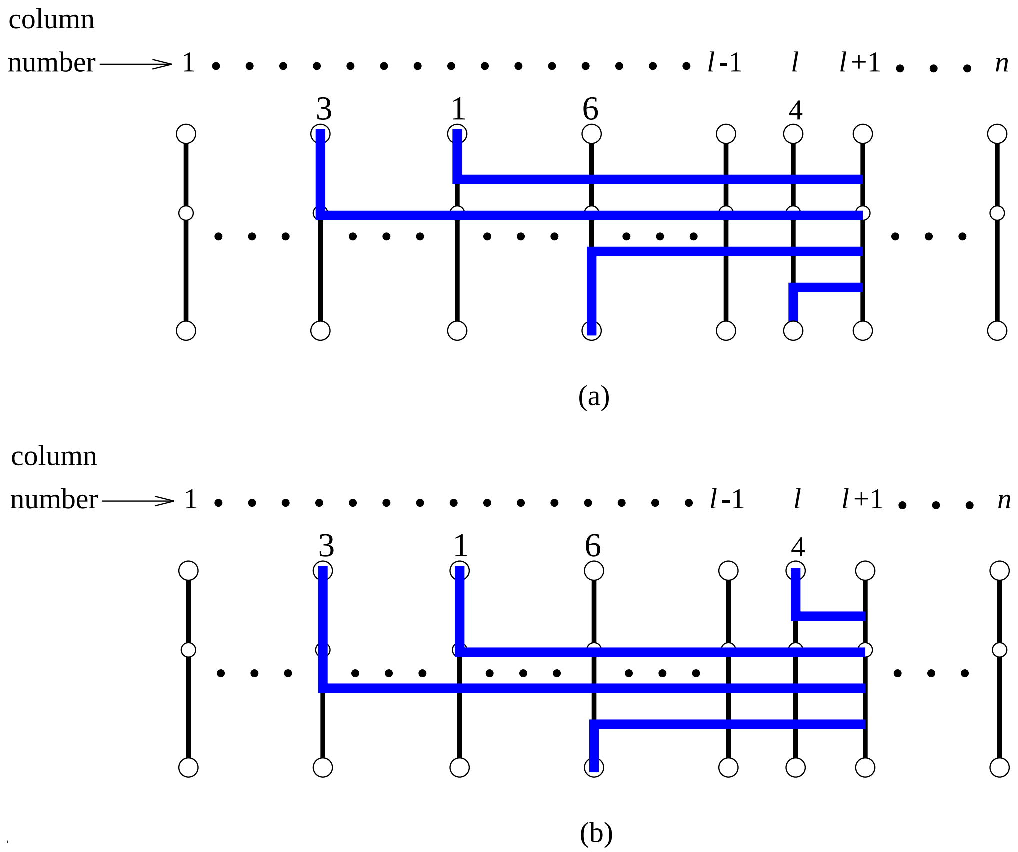


Figure 4.14: Starting a new net

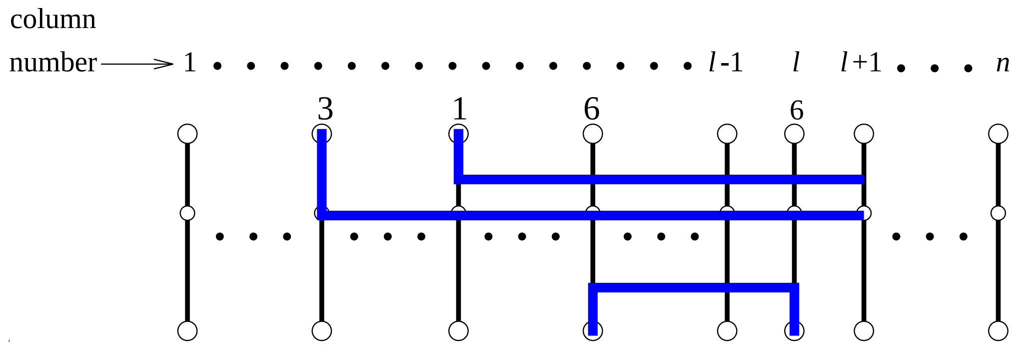


Figure 4.15: Ending a net

1. If the ith ordered pair is the first non-zero ordered pair whose *shape* is $\sqcup$ and the jth ordered pair is the last non-zero ordered pair, where $j < 4$, then we generate a configuration which is the same as c except that *netid* in the ith ordered pair becomes the *netid* in the $(j+1)$st ordered pair with shape being $\sqcap$ and *netid* in the ith ordered pair is set to 0. Figure 4.16(a) shows an example where the configuration

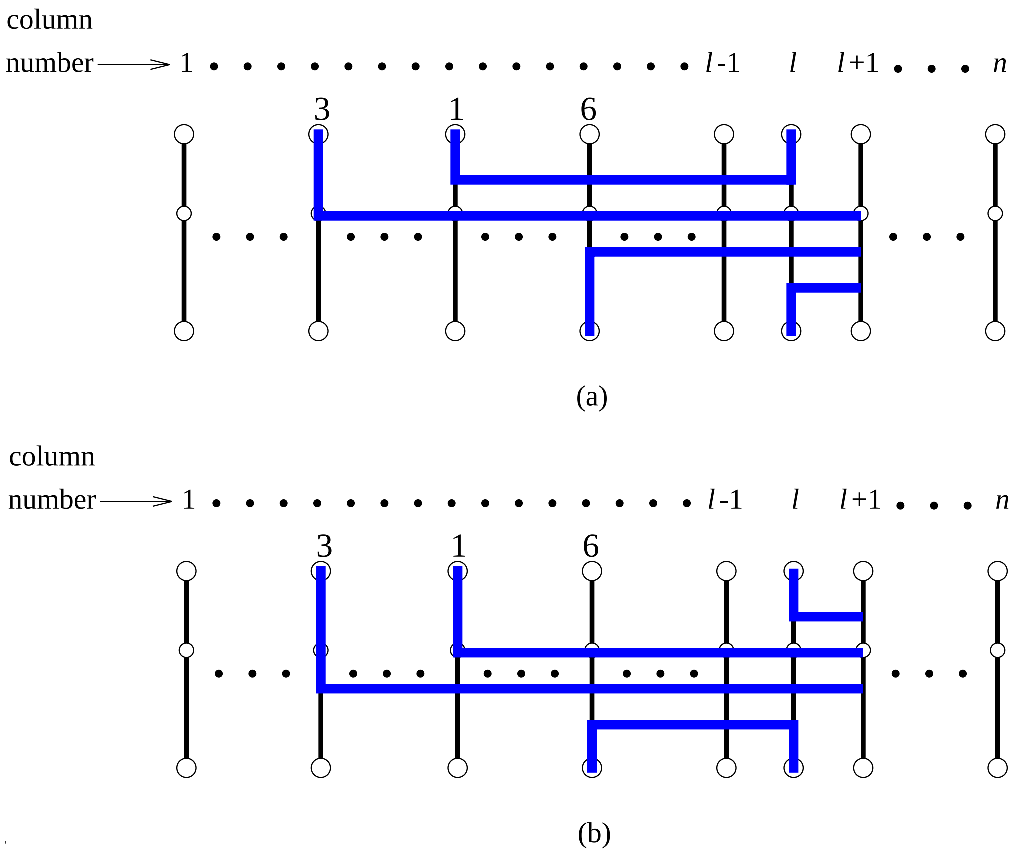


Figure 4.16: Changing tracks

for column l+1 is $(0, \sqcup)$, $(3, \sqcup)$, $(6, \sqcap)$, $(1, \sqcap)$, α.

2. If the ith ordered pair is the last non-zero ordered pair whose shape is $\sqcap$ and the jth ordered pair is the first non-zero ordered pair, where $j > 1$, then we generate a configuration which is the same as c except that *netid* in the ith ordered pair

becomes the *netid* in the $(j-1)$st ordered pair with shape being $\sqcup$ and *netid* in the i ordered pair is 0. Figure 4.16(b) shows an example where the configuration for column l+1 is $(6, \sqcup)$, $(1, \sqcup)$, $(3, \sqcup)$, $(0, \sqcup)$, α.

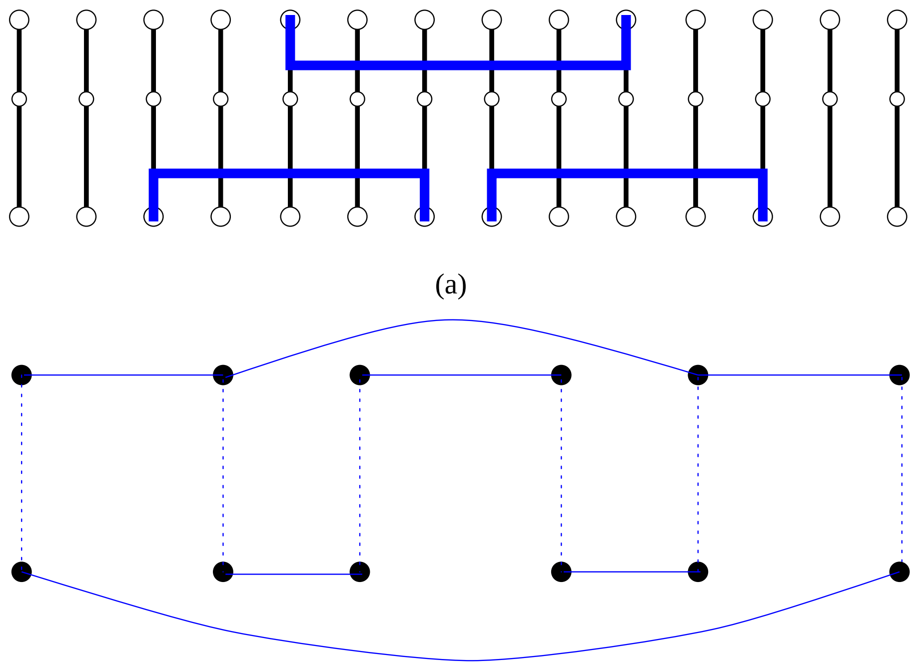
- If two configurations generated for column $l+1$ have the same four ordered pairs but different α values, we always keep the configuration with the larger α. If two configurations at column $l+1$ are identical, only one of them will be kept.

Since our algorithm exhausts all possible configurations at each column, if we pick the configuration at the last column which has the largest α value and backtrack the configuration generation steps, then the sequence of configurations at columns 1 through n will form a solution to the problem of maximizing the number of global routes completed in a row. Moreover, since the number of configurations for each column is bounded by $2^4 \times 4! \times \binom{|A|}{4} = O(|A|^4)$, the running time of our algorithm is bounded by $O(n \times |A|^4)$.

For nets that cannot be completed in the above process, they are handled as follows. Consider a row of pre-placed segments as shown in Figure 4.17(a) where there are existing wires. The unused pre-placed segments in this row are divided into groups of consecutive unused pre-placed segments. We use a pair of vertices to represent each group of consecutive vias in the unused pre-placed segments, one representing the upper vias of these pre-placed segments and the other the lower vias. The vertices corresponding to the upper (lower) vias are called the *upper* (*lower*) *vertices*. An edge is introduced between the upper vertex and the lower vertex of each pair. We call such an edge a *pair edge*. Thus, each pair edge corresponds to a group of consecutive unused pre-placed segments. Let u and v be two upper vertices. If there are two vias, one from each group of consecutive vias corresponding to u and v such that these two vias can be connected by a $\sqcup$ wire, then we say v is *u-accessible*. Furthermore, if there is no other

u-accessible vertex whose corresponding unused pre-placed segments are located between those for u and v, then we say v is a *nearest* u-accessible vertex. (In general, there are two nearest u-accessible vertices, one to the left and one to the right of u.) Note that u is v-accessible if and only if v is u-accessible by the symmetry of the definition. Similarly, v is a nearest u-accessible vertex if and only if u is a nearest v-accessible vertex. An edge is introduced between vertices u and v, if v is a nearest u-accessible vertex and the introduction of a $\sqcup$ wire between two vias, one from each group of consecutive vias corresponding to u and v, will not make the number of tracks needed exceed 4. Similarly, edges are introduced among the lower vertices. We call the resultant graph the *access graph*[3] for the row. Given a row of pre-placed segments with existing wires, the access graph for the row can be constructed in $O(n)$ time as follows. Consecutive unused pre-placed segments can be identified in a single scan during which upper vertices, lower vertices, and pair edges are constructed. To construct the edges among the upper vertices, we know that the existing wires connected to the upper vias must all be $\sqcup$-shaped. We can use a stack to identify the two nearest u-accessible vertices for each upper vertex u in a single scan. For each pair of (u, a nearest u-accessible vertex), the feasibility of inserting a wire ($\sqcup$ or $\sqcap$) can be tested in $O(n)$ time. Since there are at most two (u, a nearest u-accessible vertex) pairs for each upper vertex u, the edges among the upper vertices can be constructed in $O(n^2)$ time. Similarly, the edges among the lower vertices can also be constructed in $O(n^2)$ time. Therefore, the access graph for a row can be constructed in $O(n^2)$ time. Figure 4.17(b) shows the access graph for the row in Figure 4.17(a), where dotted edges are pair edges. Consider two adjacent rows, row r and $r+1$, of pre-placed segments and their access graphs, G_r and G_{r+1}, respectively, as shown in Figure 4.18(a). Given a lower vertex u of G_r and an upper vertex v of G_{r+1}, if

[3] A similar definition can be found in [7].



(a)

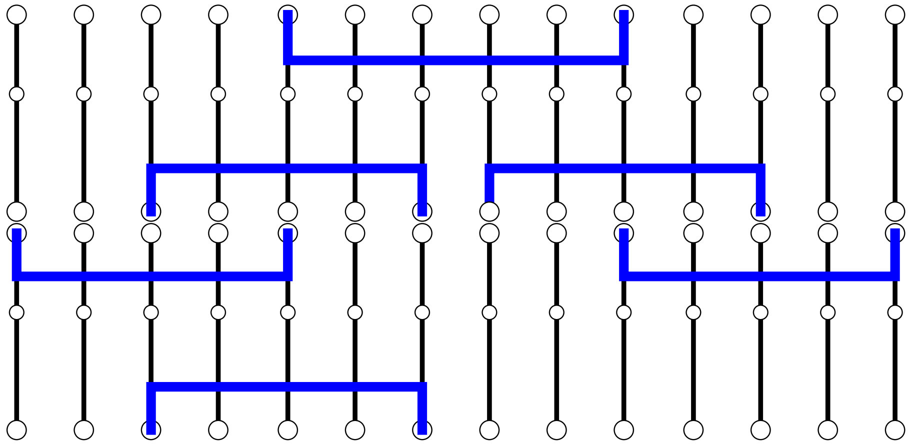
(b)

Figure 4.17: A row of pre-placed segments and its access graph

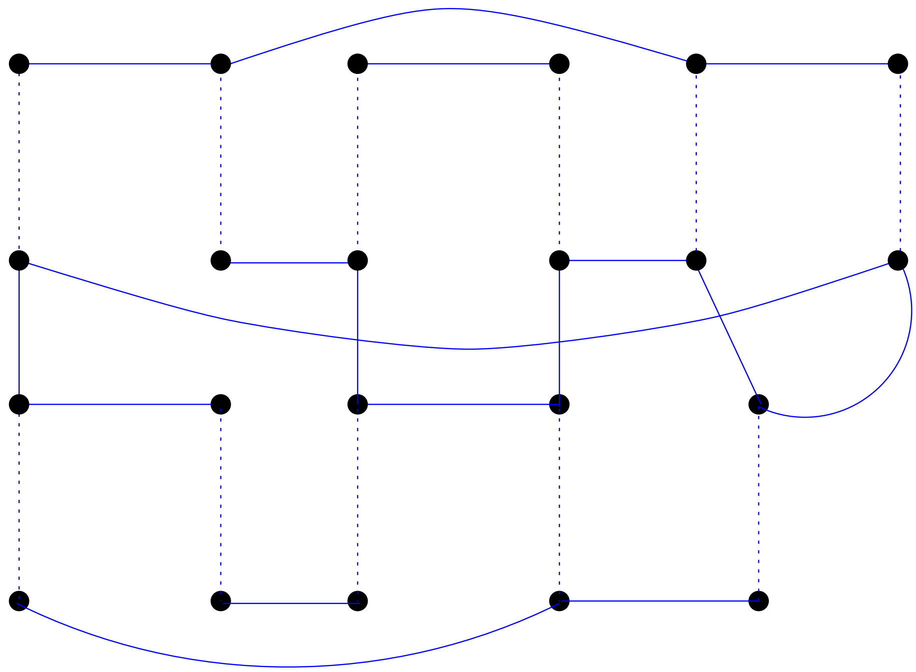
there are two vertically aligned vias, one from each group of consecutive vias, then an edge is added between u and v. The resultant graph is called the *combined access graph* for rows r and $r+1$. The construction of a combined access graph for all rows can be done in $O(mn^2)$ time. Figure 4.18(b) shows the combined access graph for the rows in Figure 4.18(a).

Conceptually, the (combined) access graph describes the possibility of finding a routing for a net that has not been completed in the exhaustive process described above. Suppose net i has k terminals. Since the k pre-placed segments corresponding to these k terminals are unused before net i is routed, there are $2k$ vertices in the combined access graph, each corresponding to a group of vias to which one of the vias of these k pre-placed segments belongs. Figure 4.19 shows an example of such vertices in the combined access graph. If we can find a connected component that spans all such vertices, then a corresponding routing can be constructed. Figure 4.19 shows an example of such correspondence.

Suppose U is the set of nets that cannot be completed by our row-by-row detailed router. First of all, we construct the combined access graph for all rows. Then, we order these nets according to the half perimeter of the rectangle enclosing the terminals in the net and process them one at a time. Nets with larger half perimeters are processed before nets with smaller half perimeters. Suppose net i is being processed. If $t(i) > 2$, then we split net i into $t(i)-1$ two-terminal nets, $\{(x_1^i, y_1^i), (x_2^i, y_2^i)\}$, $\{(x_2^i, y_2^i), (x_3^i, y_3^i)\}$, ..., $\{(x_{t(i)-1}^i, y_{t(i)-1}^i), (x_{t(i)}^i, y_{t(i)}^i)\}$. For each terminal of such a two-terminal net, in the combined access graph there is a pair edge whose corresponding group of consecutive pre-placed segments contains the terminal. For each pair edge, we choose one end vertex arbitrarily. Let u and v denote the two vertices, one from each pair edge. Then, we assign to each edge in the combined access graph a weight as follows. If the edge coincides with the global routing, then we let w, $w > 1$, be its weight.

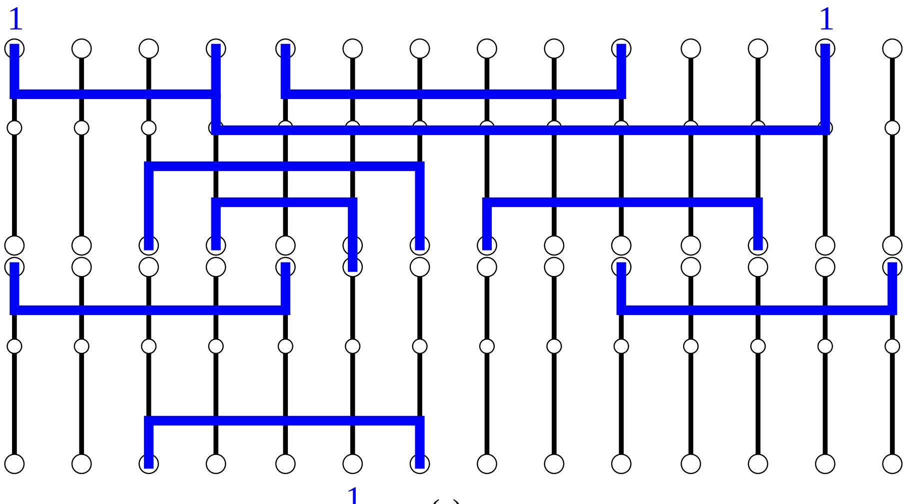


(a)



(b)

Figure 4.18: (b) is the combined access graph for the two rows in (a)



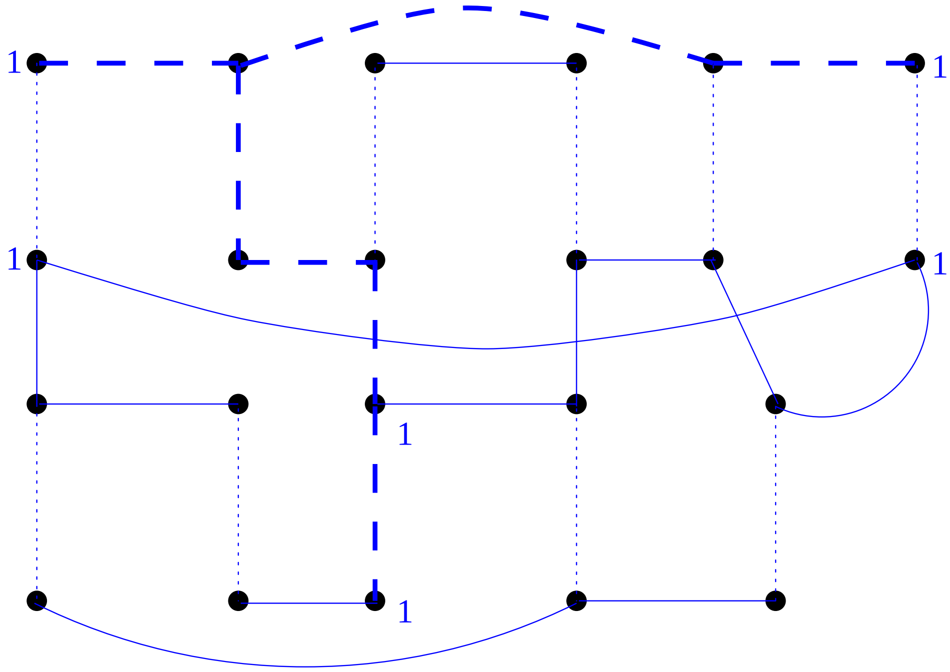


Figure 4.19: The correspondence between a connected component in the combined access graph and a routing

For the remaining edges, the weight is equal to 1. After the weight assignment, we shall find a shortest path between u and v in the weighted combined access graph. Choosing the weight $w > 1$ guides the shortest-path algorithm to favor those wires that follow the global routing obtained earlier. Figure 4.20 shows an example of such edge weight assignment. This will

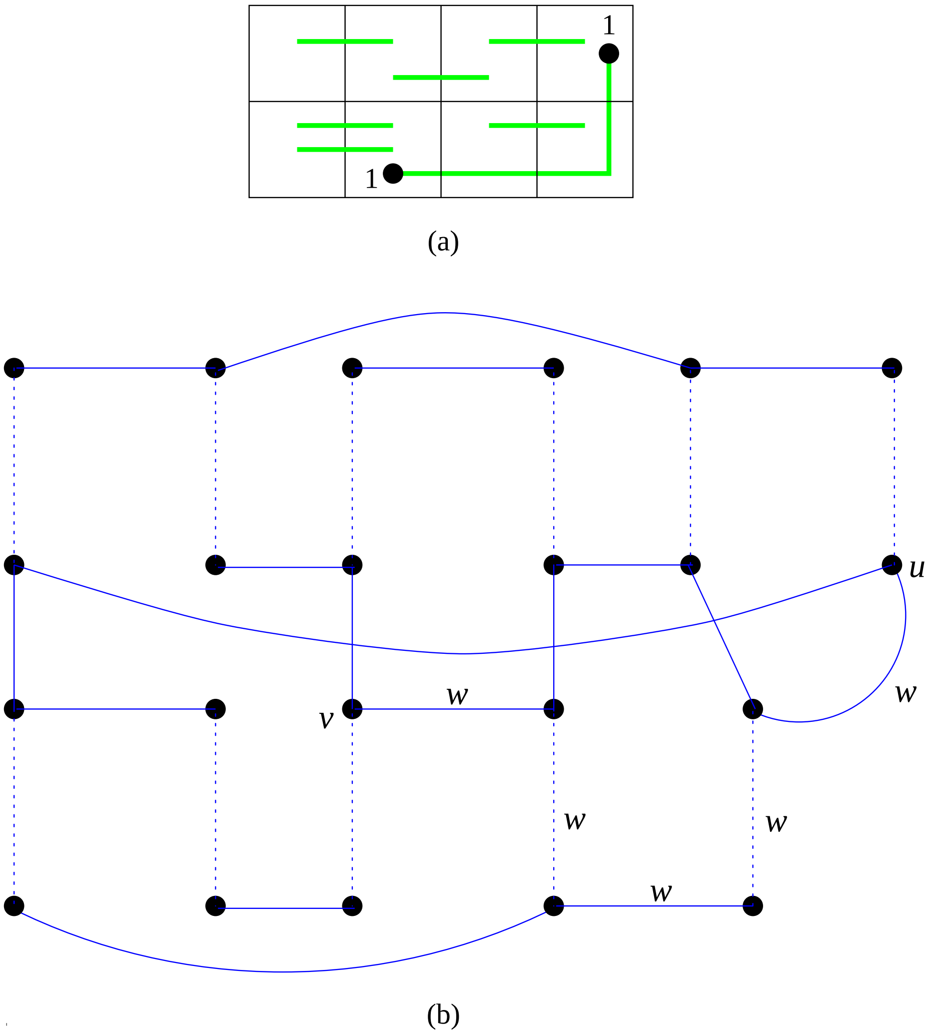


Figure 4.20: (a) A partial global routing of net 1 (b) The weighted access graph where the unmarked edges have weight 1

avoid congestion formed by nets considered earlier and might block the nets to be considered later. Therefore, a uniform distribution of the global routes will contribute to a more uniform distribution of wires in this step. After the routing for a net is found, we update the combined access graph and repeat the process for the next net.

4.5 Experimental Results

Our Algorithm was used to route a 16 bits $\times$ 16 bits multiplier circuit. The circuit consists of seven blocks, a Booth's decoder, buffers, a Booth's selector, a Wallace tree, a carry look-ahead adder, and two blocks of inverters. The total gate count is about 3000 and the number of interconnections is about 1100. Seven cell circuits in the cell library were used to form each block. The placement was done manually as shown in Figure 4.21. Half of the routing result is

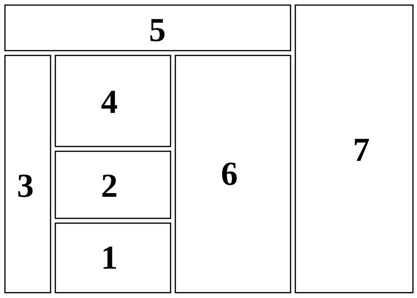


Figure 4.21: The placement of the blocks

shown in Figure 4.22. Note that the figure is simplified for the sake of clarity. Each cell layout is represented by a rectangle of the same size. We also omit all type-1 pre-placed segments except terminals in some cell layouts.

4.6 Summary

In this chapter, a routing problem in the channelless QCL gate array technology is studied. Because of the substantial routing resource limitation in this particular architecture, conven-

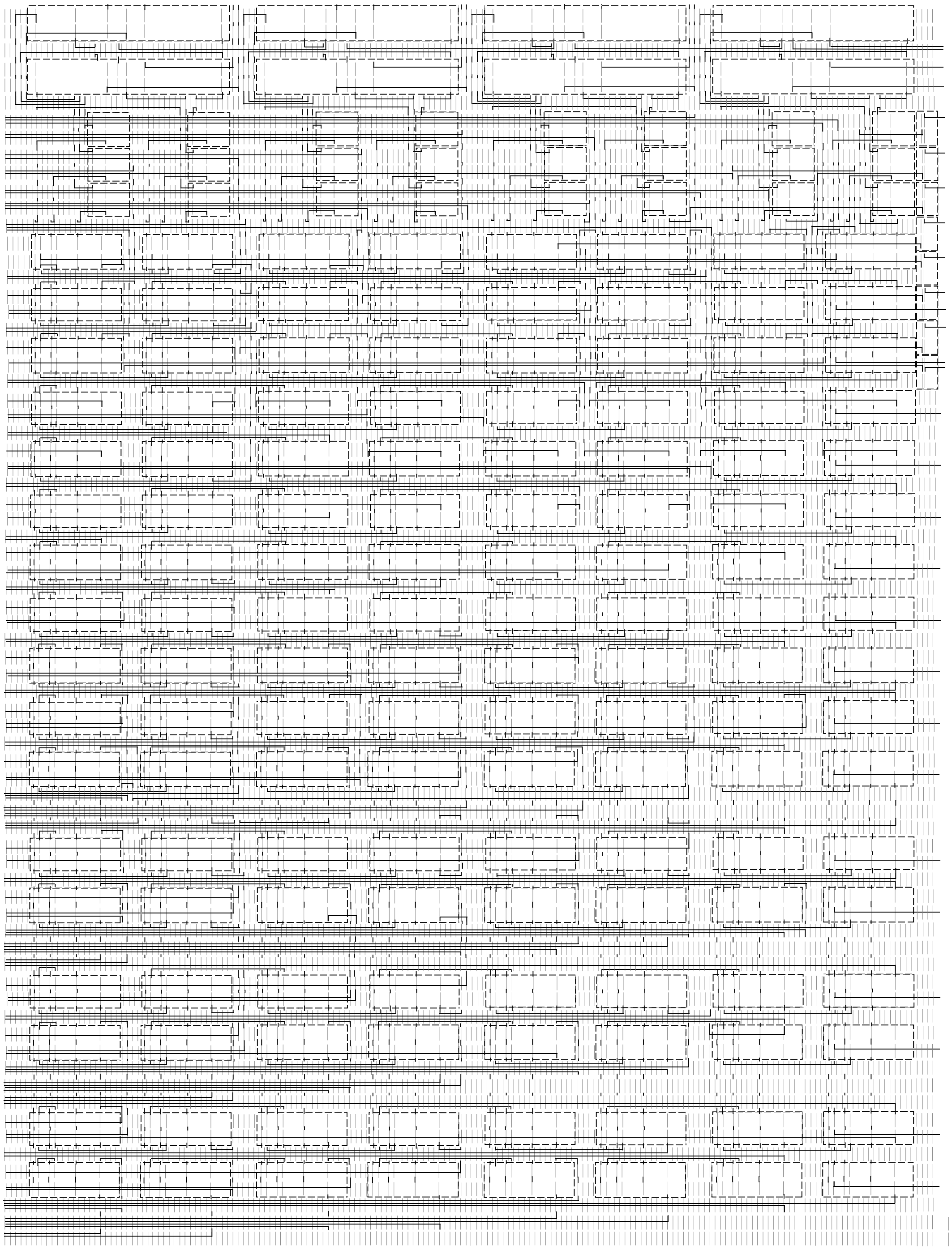


Figure 4.22: Routing result

tional routing approaches are ineffective. We formulate the routing problem and prove its NP-completeness. However, a systematic routing method which consists of two phases is proposed. After global routing, a row-by-row detailed router is employed to complete the routing. For nets that cannot be completed in the row-by-row routing step, a novel algorithm based on the access graph is used to find a routing that is as close to the corresponding global route result as possible. Experimental results are presented.

Chapter 5

Routing for Symmetric FPGAs and FPICs

5.1 Introduction

In the market for application-specific integrated circuits, field programmable chips offer the advantages of short production time and low prototyping cost. Unlike conventional mask-programmable integrated circuits, field-programmable integrated circuits use programmable switches in the connecting paths. The ON/OFF status of the programmable switches is programmed by the user without going through the foundry facility. Among different kinds of field-programmable integrated circuits, the Field-Programmable Gate Array (FPGA) is a good choice for circuits of moderate size. It has higher integration than the Programmable Logic Device (PLD) and is extensively used in the implementation of digital logic. To implement circuits that cannot be fitted in a single FPGA, the Field-Programmable Interconnect Chip (FPIC) was introduced [2, 19]. In such cases, large circuits are divided into several parts, and each part is implemented in an FPGA. An FPIC is then used to interconnect these FPGAs on a printed circuit board. Each external pin of the FPGAs is connected to an FPIC pin through a trace on the board. An FPIC does not implement any logic function. Rather, it provides interconnecting paths between its pins (and, thus, between FPGAs) through orthogonal wiring segments. Since it is re-programmable on the board, the FPIC enables a designer to change the functionality of a circuit by altering the pin connections on the FPIC. In this manner, the FPIC offers a quick and re-programmable means for inter-FPGA connections and permits fast modification of large circuits.

In this chapter, we consider the routing problem for routing structures using orthogonal segments, *i.e.*, segments that are either horizontal or vertical, of variable lengths. In a conventional two-dimensional FPGA routing structure, programmable switches are located unit lengths apart. Thus, the number of programmable switches on a connecting path is equal to the length of the path. These switches will introduce a significant delay even for a path of moderate length. In our class of routing structures, a signal passes a programmable switch only when making a turn. Signal delay is thereby reduced dramatically. This advantageous reduction is achieved with negligible sacrifice in routability. As a specific demonstration of the effectiveness of this class of routing structures, we use a probabilistic model to show that complete routing can be achieved with a high probability in a routing structure of this class. In such a complete routing, the number of tracks in each channel approaches the lower bound asymptotically. Using our class of routing structures, we can also model the routing structure in FPIC chips proposed in [2, 19].

In a manner similar to maze-routing based algorithms proposed for the conventional two-dimensional FPGA routing structure in [12, 38], we present a sequential routing algorithm based on the solution of the single net routing problem. We formulate the single net routing problem as a Node-Weighted Steiner Minimum Tree (NWSMT) problem in a bipartite graph G. An approximation algorithm with a guaranteed performance bound is provided for the single net routing problem. We also prove that approximating the optimal single net routing solution within any constant bound is NP-complete. Our approximation algorithm solves the single net routing problem optimally if the bipartite graph G is a (6,2)-graph. If the weights of all vertices and edges in G are the same, then our algorithm also produces an optimal solution for a (6,1)(8,3)-graph. Experimental results on several industrial circuits demonstrate the effectiveness of both the algorithm and the routing structure.

5.2 Routing Structure

The conventional two-dimensional FPGA routing structure proposed in [6, 38, 59] contains only wire segments of unit length as shown in Figure 5.1. Each square represents a logic block implementing logic functions. The terminals of a block are located on the boundaries of the block and are connected to wire segments called *terminal segments*. Terminal segments are connected to wire segments in the routing channel through switches at their intersections, which are shown as black circles in Figure 5.1. *Interconnect points*, shown as white circles in the diagonal of the junction of a horizontal channel and a vertical channel, make connections between wire segments in the horizontal and the vertical channels. Each interconnect point consists of six programmable switches, any subset of which can be programmed to be active and create a desired connection. A routing example of a 4-terminal net is shown in bold lines. There are eight active switches in the connecting path, four of which are interconnect points, and each of the remaining four switches connects a terminal of the net to a wire segment in a routing channel. In earlier Xilinx chips, interconnect resources could be grouped into direct

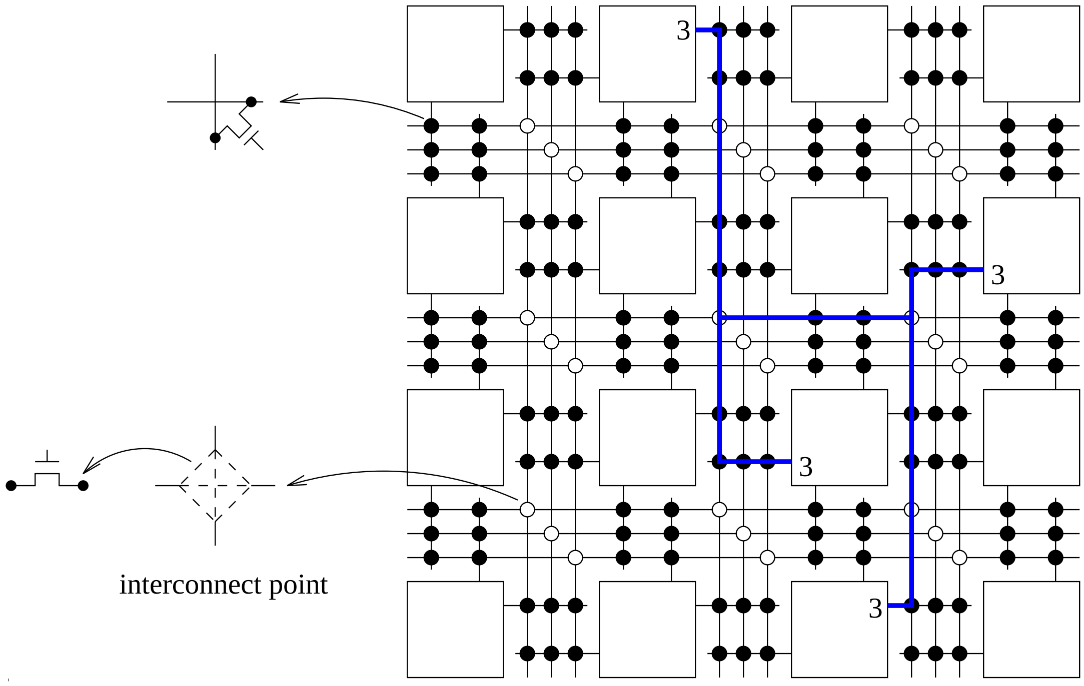


Figure 5.1: Conventional two-dimensional FPGA structure and a routing example

connects, general interconnects, and long lines [60]. A direct connect provides a direct path from a Configurable Logic Block (CLB) to one of its vertical or horizontal neighbors. A general interconnect, running horizontally and vertically for unit length between two switch blocks, is normally used to route signals within a localized area. Long lines, spanning the entire length or width of the array of CLBs, are intended for high fan-out and time-critical nets. For moderately critical nets or nets of intermediate length, usage of the shorter general interconnects induces significant delays as a result of the large number of programmable switches on the connecting paths. To improve upon the situation, the recently-developed Xilinx *XC4000* series provides double-length lines which are twice as long as single-length lines. However, the variety of wire segments of different lengths is still very limited. It is believed that there is a technical trend toward having larger variety of wire segments of different lengths, and we propose a class of routing structures that includes such a feature.

An example of the proposed routing structure is shown in Figure 5.2. The routing structure

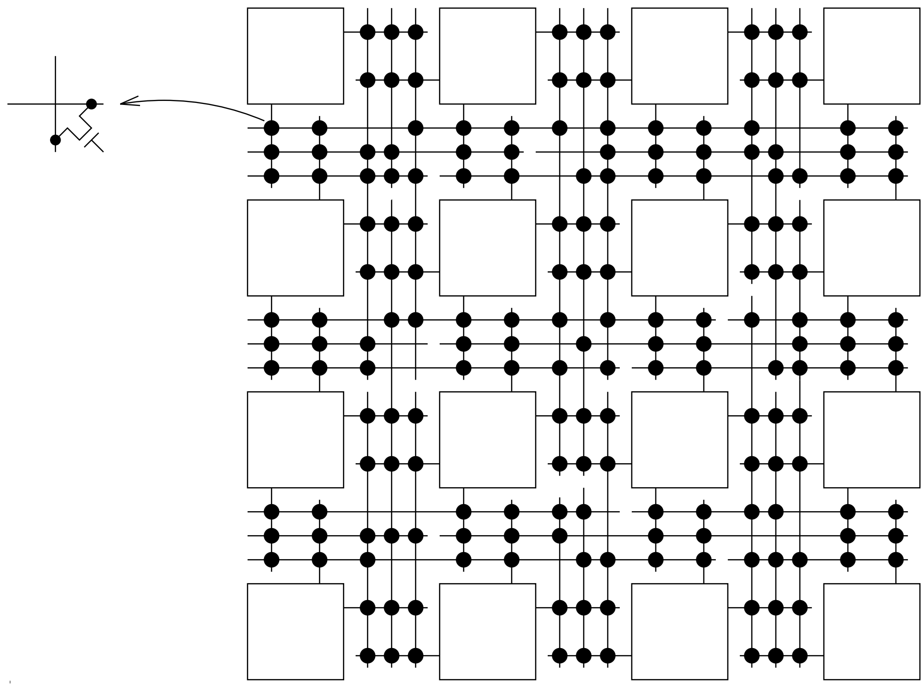


Figure 5.2: Proposed two-dimensional FPGA routing structure

consists of a two-dimensional symmetric array of *blocks*. In our model of the FPGA routing structure, each block is a logic block that may be as simple as a gate or as complex as a multiplexor or a lookup table with latches implementing sequential circuits. Terminals are located at the boundaries of the blocks. There is a horizontal interconnect *channel* between each two adjacent rows of blocks. Likewise, there is a vertical interconnect channel between each two adjacent columns of blocks. Each channel contains a set of tracks, and there are fixed wire segments of variable lengths on each track. Switches are located at some of the intersections between horizontal and vertical wire segments. As shown in Figure 5.2, each switch can be programmed to interconnect the two orthogonal, intersecting segments. Each terminal in a block is connected to a terminal segment. There is a switch at each intersection involving a terminal segment. Figure 5.3 shows a routing example for the net shown in Figure 5.1, using the

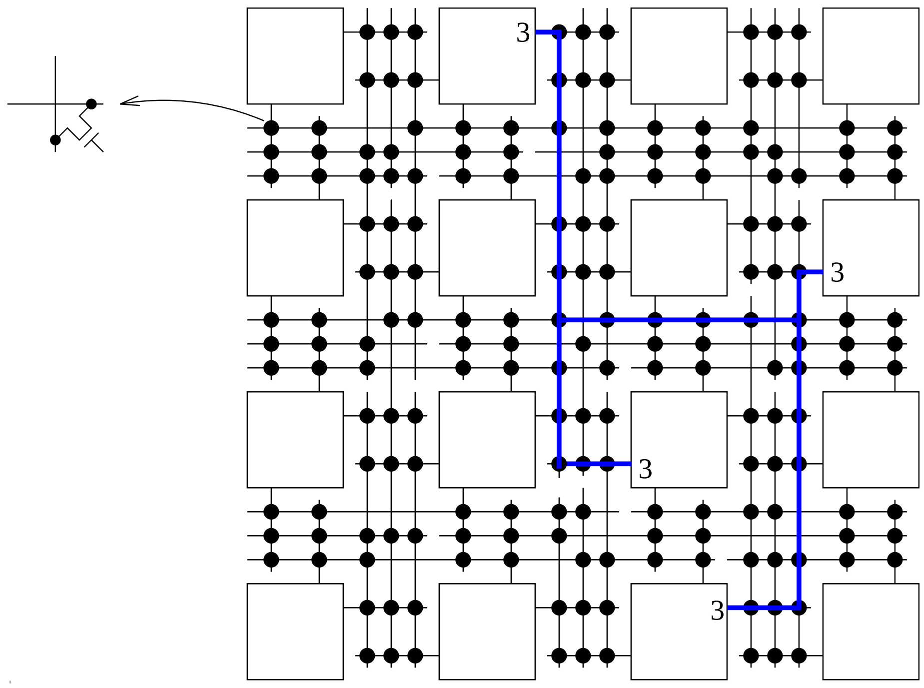


Figure 5.3: A routing example of the net shown in Figure 5.2

routing structure shown in Figure 5.2. Note that only six programmable switches are activated in the route shown in Figure 5.3, as compared with eight programmable switches in Figure 5.1. In our class of routing structures, a signal uses a programmable switch only when making a turn, whereas in the conventional FPGA routing structure one switch is used in each unit length. Accordingly, signal delay due to the programmable switches is reduced dramatically in our routing structures.

Another advantage is that our class of routing structures contains the routing structure of FPIC chips as a special case. In modeling the FPIC routing structure, each block in our routing structure is reduced to become a point and represents an I/O pin of the FPIC chip. These I/O pins carry the signals of inter-chip nets and are connected on the FPIC. Thus, our class of routing structures precisely models the routing structure of the FPIC described in [2, 19]. To model Actel's row-based routing structure, we regard each horizontal channel in our class of routing structures as a channel between two cell rows in an Actel chip. Vertical segments in our class of routing structures correspond to the vertical segments used as feed-throughs to bypass the cell rows and to connect nets spanning several cell rows in an Actel chip. According to [15], the number of anti-fuses between the driver and each input is generally limited to two for each net. Therefore, if we stipulate that each net that connects a (driver, input)-pair may use at most one horizontal segment (or, equivalently, two programmable switches) in each horizontal channel, then our class of routing structures precisely models Actel's row-based FPGA routing structure.

5.3 Probabilistic Analysis

In this section, we present a probabilistic analysis of the routability of a two-dimensional routing model in which only fixed orthogonal wire segments are used. Using a probabilistic model for nets, we show that the number of tracks needed to achieve a high probability of routing completion is asymptotically equal to a lower bound on the number of tracks in each channel. In the following, we show the analytic result for the symmetric FPGA, followed by that for the FPIC.

Our model of a FPGA/FPIC chip consists of a two-dimensional $N \times N$ lattice with fixed wire segments placed along columns and rows of the lattice points. Terminals are assumed to be located at the lattice points. Following earlier analysis carried out for gate arrays [14, 45], we assume that the number of nets emanating from a logic block is a random variable X according to the Poisson distribution, with mean λ. That is,

$$P\{X = k\} = \frac{\lambda^k e^{-\lambda}}{k!}, \qquad k = 0, 1, \ldots$$

In our analysis, we assume that each net has two terminals. To approximate the case where there are multiple-terminal nets, we can increase the value of the parameter λ. To simplify our discussion, we assume that the chip is embedded on the surface of a torus, *i.e.*, column 1 and column N are adjacent and so are row 1 and row N. Note that removing this assumption permits the extension of our results. We further assume that the rectilinear distance[1] between the two terminals in a net is chosen independently according to a geometric distribution with mean $\gamma = N^{\rho}, 0 < \rho < 1$, as [14]. Consider a net of length 1 emanating from the block (i, j). There are four configurations for this net as shown in Figure 5.4, where $+_N(-_N)$ denotes addition (subtraction) modulo N and solid lattice points represent (i, j). For a net of length 2,

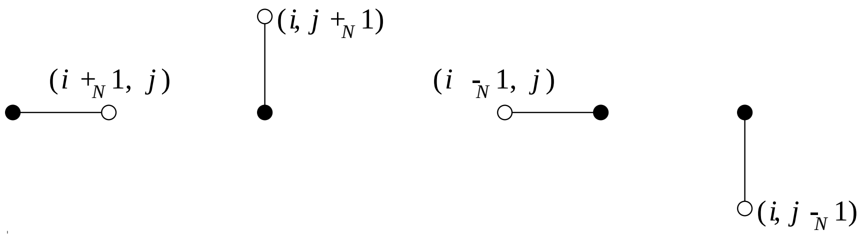


Figure 5.4: Four configurations for a net of unit length

there are 12 possible configurations, as shown in Figure 5.5. We introduce the restriction that

[1] The distance between two adjacent lattice points that are vertically or horizontally aligned is regarded as unit length.

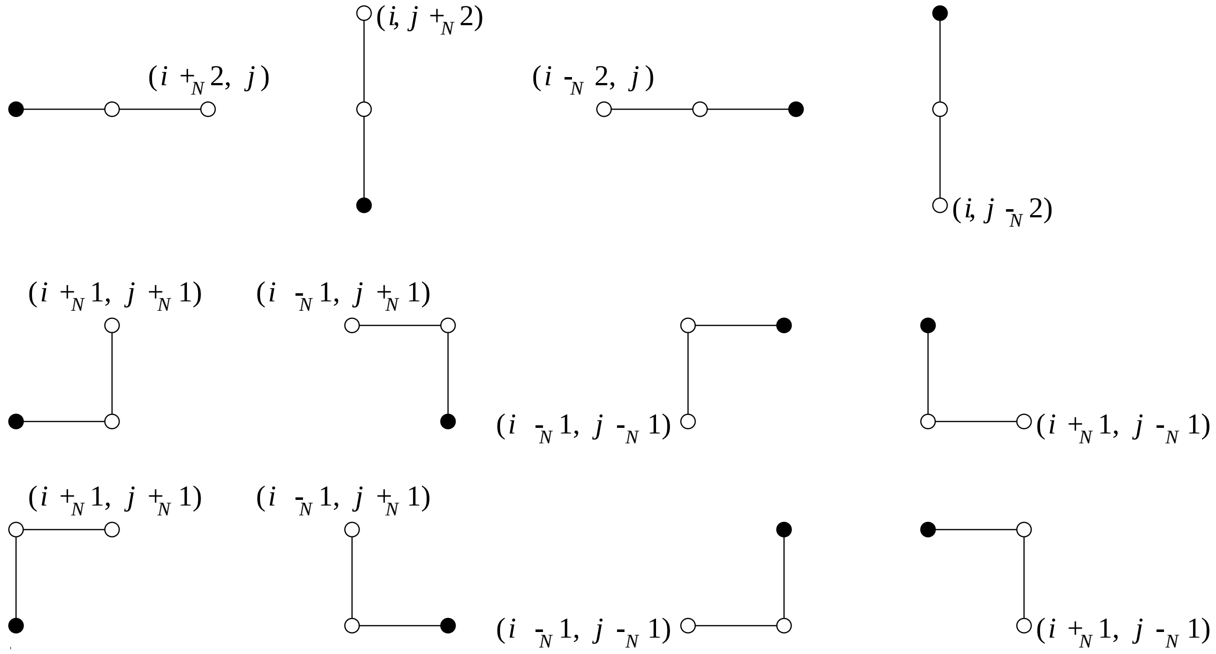


Figure 5.5: Twelve configurations for a net of double length

each net can only be routed using *2-segment routing*[2]. In general, for a net of length l, there are $8l - 4$ possible configurations. We assume each of these $8l - 4$ configurations is equally likely.

Since the number of nets emanating from each lattice point has mean λ and the average length of each net is N^ρ, the expected value of the total length of the nets emanating from a lattice point is λN^ρ. Then, the total wire length for all $N \times N$ blocks is $\lambda N^{(2+\rho)}$. Thus, for successful routing, even in the case of a two-dimensional array without the restriction of using fixed wire segments, the number of tracks in each channel is at least $\frac{\lambda N^{(2+\rho)}}{N^2} = \lambda N^\rho$. The following theorem states that $O(\lambda N^\rho)$ tracks in each channel will, in fact, suffice even when fixed wire segments are used.

[2]A routing is called a *2-segment routing* if and only if it uses at most two wire segments and one programmable switch, that is, the routing can make at most one turn.

Theorem 6 *For any* N*,* $0 < \rho < 1$*,* $\lambda > 0$*, there is an arrangement of fixed wire segments along each column and row of lattice points such that the probability of failure in attaining a complete routing solution in which each net uses only 2-segment routing is* $\mathrm{O}(\mathrm{N}^{-\rho})$*. Moreover, the number of tracks along each column and row of lattice points is* $\mathrm{O}(\lambda \mathrm{N}^{\rho})$*.*

Proof : The strategy we use for arranging fixed wire segments in each column/row is similar to that in [16, 63]. However, two major differences between our two-dimensional array model and the single routing channel in [16, 63] prevent a direct application of their results. In [16, 63], M nets are chosen uniformly in the channel, whereas we assume that the number of nets emanating from a block has a Poisson distribution with mean λ. Also, we need to take into consideration the fact that a net of length l can have $8l-4$ possible configurations in a two-dimensional array, whereas in [16, 63], there is only one configuration for each net.

Since our two-dimensional array is symmetric, we consider, without loss of generality, only one row (horizontal channel) in the array of lattice points. Given the two points corresponding to the terminals of a net in a two-dimensional array of lattice points, the horizontal distance between these two points is referred to as the *horizontal net length.* The strategy for arranging the fixed wire segments in the channel is to divide all possible horizontal net lengths l into a number of ranges, $L_j < l \leq L_{j+1}$, where $0 \leq j \leq j_{max}$. Define $L_0 = 0$, $L_1 = N^{\rho}$, and $L_j = eL_{j-1}$ for $2 \leq j \leq j_{max}$. Since the horizontal net length can be at most N, we have $L_{j_{max}} \leq N$, thus, $j_{max} \leq (1-\rho) log N + 1$. We shall have a set of tracks dedicated to nets with lengths in each range. Figure 5.6 shows the tracks dedicated to nets whose horizontal net length is in the range of $(L_j, L_{j+1}]$. This is done by the following procedure [16]. A group of m_j identical tracks are placed first, where each track is divided into wire segments of length $(1+\alpha)L_{j+1}$ for some constant $0 < \alpha \leq 1$. Then a second group of m_j tracks is placed to the right of the previous

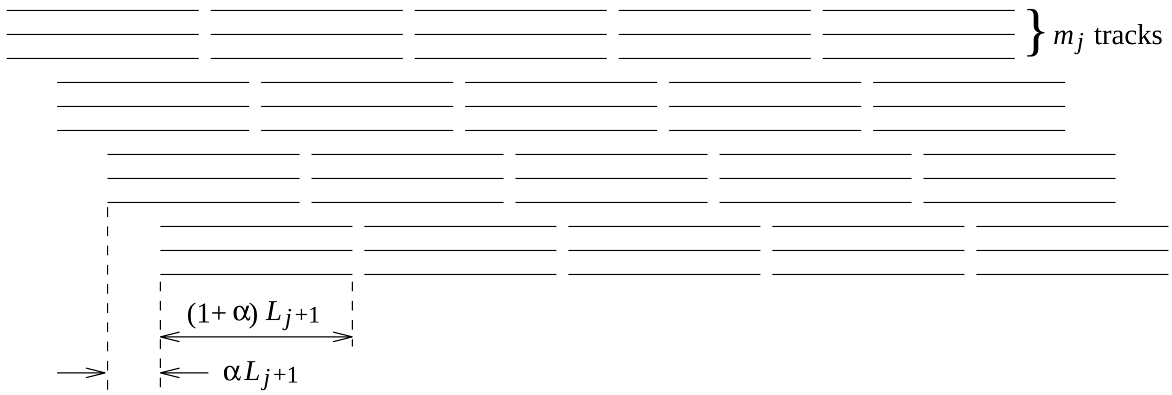


Figure 5.6: Tracks for the nets whose length is in $(L_j,\, L_{j+1}]$

group by αL_{j+1} columns offset, as shown in Figure 5.6. We repeat the construction $\frac{1+\alpha}{\alpha}$ times. The total number of tracks is, thus, $\frac{m_j(1+\alpha)}{\alpha}$. According to the construction, the channel is divided into column groups with αL_{j+1} columns in each group. Consider each group of m_j identical wire segments which are aligned. The bold arrow in Figure 5.7 represents the nets

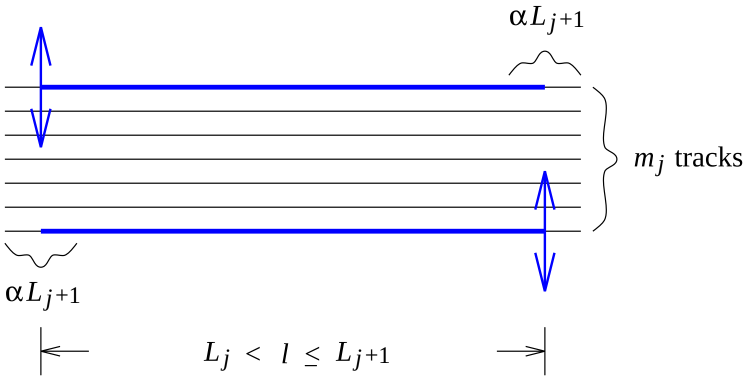


Figure 5.7: Nets assigned to a group of identical wire segments

that will be assigned to these wire segments. These nets satisfy (i) one of the terminals is adjacent to the channel and is located in the first or the last column group; (ii) the horizontal net length is in the range of $(L_j, L_{j+1}]$. It is obvious that if there are at most m_j nets satisfying the above conditions for each of such groups of identical wire segments, then $\frac{m_j(1+\alpha)}{\alpha}$ tracks will suffice.

Consider the lattice points located in the first αL_{j+1} columns, as depicted in Figure 5.8. The bold arrow represents nets with length in the range $(L_j, L_{j+1}]$, one terminal in the first αL_{j+1} columns and the other terminal located to the upper right. The number of nets emanating from

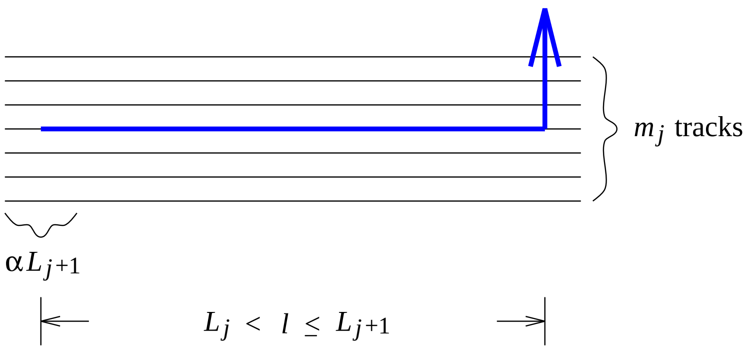


Figure 5.8: Nets assigned to a group of identical wire segments, emanating from the first αL_{j+1} columns

these lattice points with horizontal net length in the range $(L_j, L_{j+1}]$ and ending to the upper right of the emanating point follows a Poisson distribution, with the mean

$$\frac{\alpha L_{j+1} \cdot \lambda \cdot \gamma}{1 - \overline{\gamma}^{2N+1}} \sum_{i=1}^{L_{j+1}-L_j} \sum_{k=L_j+i}^{L_j+i+N} \frac{\overline{\gamma}^k}{8k-4} \tag{5.1}$$

where $\overline{\gamma} = 1 - \gamma$. The number of nets we will assign to the group of m_j identical wire segments in Figure 5.7 has a Poisson distribution, with the mean being 8 times the quantity in (5.1), since we need to take into account (i) the fact that the ending point can be located to the lower right of the emanating point, (ii) both the 1st and the last column groups, and (iii) exchanging the roles of the emanating point and the ending point. We use λ_j to denote 8 times the quantity in (5.1).

Let the random variable X_j be the number of nets that will use the wire segments in Figure 5.7. Then,

$$Pr(X_j > m_j) \quad = \quad \sum_{k=m_j}^{\infty} \frac{e^{-\lambda_j}\lambda_j^k}{k!} \quad = \quad e^{-\lambda_j} \sum_{k=m_j}^{\infty} \frac{\lambda_j^k}{k!}$$

The summation above is the remainder beyond the m_jth term in Taylor's expansion of e^{-t}. Moreover, since $n! \geq \frac{n^n}{e^n}, \forall n \geq 0$, we have

$$Pr(X_j > m_j) \quad \leq \quad (\frac{e\lambda_j}{m_j+1})^{m_j+1} e^{-\lambda_j} \quad \leq \quad (\frac{e\lambda_j}{m_j+1})^{m_j+1}$$

Let $m_j \geq max\{e^2\lambda_j - 1, 2logN - 1\}$. Then, we have

$$Pr(X_j > m_j) \quad \leq \quad \frac{1}{N^2} \tag{5.2}$$

Thus, when $m_j = max\{logN, e^2\lambda_j\} - 1$, the probability of running out of segments in one group shown in Figure 5.7 for nets with length l, $L_j < l \leq L_{j+1}$, is at most $\frac{1}{N^2}$. The probability of running out of tracks for nets with length l, $L_j < l \leq L_{j+1}$, in any of the $\frac{N}{\alpha L_{j+1}}$ segment groups is at most $\frac{1}{N^2} \cdot \frac{N}{\alpha L_{j+1}} = \frac{1}{N\alpha L_{j+1}}$. Summing this for all ranges $0 \leq j \leq j_{max}$, we find that the probability of routing failure in the channel is at most

$$\sum_{j=0}^{j_{max}} \frac{1}{N\alpha L_{j+1}} \quad = \quad \frac{N^{-(1+\rho)}}{\alpha} \sum_{j=0}^{j_{max}} e^{-j} \quad \leq \quad \frac{N^{-(1+\rho)}\frac{e}{e-1}}{\alpha}$$

Therefore, the probability of not being able to successfully complete the routing in all channels is at most

$$2N \cdot \frac{N^{-(1+\rho)\frac{e}{e-1}}}{\alpha} \quad = \quad O(N^{-\rho})$$

It remains to calculate the total number of tracks required by such a scheme. Recall that there are $\frac{m_j(1+\alpha)}{\alpha}$ tracks set aside for each length range. The total number of total tracks for range $j = 0$ is at most $\frac{(1+\alpha)e^2\lambda_0}{\alpha} + O(logN)$, whereas

$$\lambda_0 \quad = \quad \frac{8\alpha L_1 \cdot \lambda \cdot \gamma}{1 - \overline{\gamma}^{2N+1}} \sum_{i=1}^{L_1} \sum_{k=i}^{i+N} \frac{\overline{\gamma}^k}{8k-4} \tag{5.3}$$

Since $L_1 = N^\rho$, $\gamma = 1 - \overline{\gamma}$, and $8k - 4 \geq 4k$ for $k \geq 1$, λ_0 in (5.3) is at most

$$\frac{2\alpha \cdot N^\rho \cdot \lambda \cdot (1-\overline{\gamma})}{1 - \overline{\gamma}^{2N+1}} \sum_{i=1}^{N^\rho} \sum_{k=i}^{i+N} \frac{\overline{\gamma}^k}{k} \leq \frac{2\alpha \cdot N^\rho \cdot \lambda \cdot (1-\overline{\gamma})}{1 - \overline{\gamma}^{2N+1}} \sum_{i=1}^{N+N^\rho} \overline{\gamma}^i = 2\alpha \cdot N^\rho \cdot \lambda \cdot \overline{\gamma} \frac{1 - \overline{\gamma}^{N+N\rho}}{1 - \overline{\gamma}^{2N+1}}$$

Since $\overline{\gamma} < 1$ and $\frac{1-\overline{\gamma}^{N+N\rho}}{1-\overline{\gamma}^{2N+1}} < 1$, m_0 is at most $2e^2(1+\alpha)\lambda \cdot N^\rho + O(logN)$.

The number of tracks for each range $j > 0$ is at most

$$\frac{1+\alpha}{\alpha} \sum_{j=1}^{j_{max}} m_j \quad = \quad \frac{1+\alpha}{\alpha} e^2 \sum_{j=1}^{logN} \lambda_j + O(log^2 N)$$

Since λ_j is eight times the quantity in (5.1), $L_{j+1} = N^\rho e^j$ and $8k - 4 \geq 4k$, for $k \geq 1$, we obtain

$$\frac{1+\alpha}{\alpha} e^2 \sum_{j=1}^{logN} \lambda_j \quad \leq \quad 2 \cdot e^2(1+\alpha)\lambda \cdot N^\rho \frac{1-\overline{\gamma}}{1-\overline{\gamma}^{2N+1}} \sum_{j=1}^{logN} e^j \sum_{i=1}^{L_{j+1}-L_j} \sum_{k=L_j+i}^{L_j+i+N} \frac{\overline{\gamma}^k}{k} \tag{5.4}$$

Consider

$$\sum_{k=L_j+i}^{L_j+i+N} \frac{\overline{\gamma}^k}{k} \leq \sum_{k=L_j+i}^{L_j+i+N} \overline{\gamma}^k/(L_j+i) = \frac{\overline{\gamma}^{L_j+i}}{L_j+i} \cdot \frac{1-\overline{\gamma}^{N+1}}{1-\overline{\gamma}} \leq \frac{\overline{\gamma}^{L_j+i}}{L_j(1-\overline{\gamma})} = \frac{\overline{\gamma}^{L_j+i}}{e^{j-1}}.$$

Therefore, (5.4) is at most

$$2 \cdot e^3(1+\alpha)\lambda \cdot N^\rho \sum_{j=1}^{logN} \frac{1-\overline{\gamma}^{L_{j+1}-L_j}}{1-\overline{\gamma}^{2N+1}} \overline{\gamma}^{L_j+1} \leq 2 \cdot e^3(1+\alpha)\lambda \cdot N^\rho \sum_{j=1}^{logN} \overline{\gamma}^{L_j+1} \qquad (5.5)$$

Since $(1-\gamma)^{L_j} \leq e^{-\gamma \cdot L_j}$, (5.5) is at most

$$2 \cdot e^3(1+\alpha)\lambda \cdot N^\rho \sum_{j=1}^{\infty} e^{-e^{j-1}}$$

The series evaluates to approximately 0.43. This completes the proof. □

In the above analysis, we assumed that the number of nets emanating from each FPGA block follows the Poisson distribution. However, for the case of FPIC, each block represents an I/O pin, which no two nets can share. A more realistic model is to assume that there are M nets whose terminals are located at positions chosen randomly from the $N \times N$ lattice points without repetition. We can prove the following theorem with an analysis similar to that in the previous theorem.

Theorem 7 *For any* N, M, $0 < \rho < 1$, *there is an arrangement of fixed wire segments along each column and row of lattice points such that the probability of failure in attaining a complete routing solution in which each net uses only 2-segment routing is* $\mathrm{O}(\mathrm{N}^{-\rho})$. *Moreover, the number of tracks in each column and row is* $\mathrm{O}(\mathrm{MN}^{\rho-2})$.

The analytic results presented above describe the asymptotic behavior of the proposed routing structures. In reality, the number of tracks in a routing structure is fixed, and the number of switches is also restricted due to area constraint. Thus, we propose an arrangement of wire segments in the horizontal channels and the vertical channels according to the proofs of the above theorems, with some modification for a given number of tracks. The length of the segments in each track is the same (except for the two incident to the boundary) and set to be a power of a constant, for example, 2. Given the number of tracks t in each horizontal channel or vertical channel, the number of tracks with segments of length 2^l is $\frac{t}{2^l}$. That is, the number of tracks with segments of length 2 is $\frac{t}{2}$ and the number of tracks with segments of length 4 is $\frac{t}{4}$, etc. In order to distribute evenly the routing resources, segments in these $\frac{t}{2^l}$ tracks are arranged in a staggered manner, such that, as shown in Figure 5.9, segments in the $(i+1)$st track in a horizontal (or vertical) channel are shifted right (or down) to the ith track by $\frac{2^{2l}}{t}$ vertical (or horizontal) channels. Moreover, the positions of the switches are arranged, such

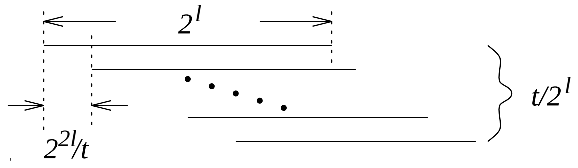


Figure 5.9: The arrangement of wire segments in the routing channels

that, at the intersection of a vertical channel and a horizontal channel, the number of switches located along each horizontal or vertical track is the same. Figure 5.10 shows an example where this number is 3.

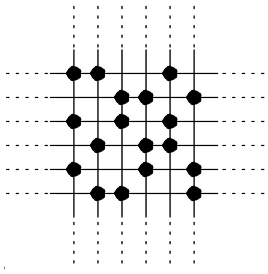


Figure 5.10: An example of the arrangement of switches at the intersection of a vertical channel and a horizontal channel

5.4 A Routing Algorithm

Our routing algorithm is a sequential algorithm that routes the nets one at a time according to the decreasing order of the timing-criticality of the nets. The algorithm is based on the solution of the single net routing problem, which is formulated as that of finding a minimum weight Steiner tree in a graph with weights assigned to both vertices and edges. When routing each net, our algorithm takes into account two factors, namely, (1) routing congestion and (2) the time delay due to both the programmable switches and the wire segments on a connecting path. Specifically, for each segment, we calculate a number that is a measure of how much this segment is in demand for routing, *e.g.*, the number of nets whose bounding boxes of terminals overlap with the segment. This number will be incorporated in the computation of the weight of a node corresponding to the wire segment in a graph-theoretical representation of the routing structure, to be presented below. For timing delay, we assume that a programmable switch introduces a constant delay, the value of which is a function of the resistance and capacitance of a pass transistor. Also, the delay in a wire segment is assumed to be proportional to its length and the number of (used/unused) programmable switches along the wire segment. Consequently, according to [60], the total delay in a connecting path is proportional to the sum of the delays contributed by the metal wire segments and the total number of programmable switches (multiplied by a normalization constant) on the path.

The single net routing problem can be formulated as follows. Given a routing structure R that consists of a set of horizontal and vertical wire segments and a set of programmable switches at some of the intersections of the wire segments, each wire segment is assigned a cost. The cost is a weighted measure of the delay associated with the wire segment and of how much the segment is in demand for routing. Each programmable switch is assigned a fixed cost. We can construct a graph to describe the relationship between the wire segments and the programmable switches as follows. Each wire segment in R is represented by a vertex with weight equal to the cost of the segment. Two vertices are connected by an edge if and only if the corresponding segments intersect and there is a programmable switch at the intersection. The weight of an edge is equal to the cost of a programmable switch. Since no two horizontal or vertical segments intersect, the resultant graph is bipartite. Figure 5.11 shows an example of such a construction where the graph in (b) is constructed according to the routing problem in (a). Note that the bold lines in (a) represent terminal segments, and the solid circles in (b) represent the corresponding demand vertices. $w(e) = 5$, for all e. Let $G = (V, E)$ denote such a bipartite graph, where $V = (X \cup Y)$ and X (Y) is the vertex set corresponding to the set of horizontal (vertical) segments. Let $Z \in V$ be the set of vertices corresponding to the terminal segments of a net. We called Z as a *demand vertex set*. Our problem is to construct a minimum weight tree of G that spans Z. We call a minimum weight tree that spans Z an *optimal tree*. Figure 5.12 shows an optimal tree for the graph in Figure 5.11(b) and the corresponding routing solution. Note that all the vertices in Z must be in a connected component of G; otherwise, the problem will have no solution. Thus, hereafter, we assume that G is connected. Moreover,

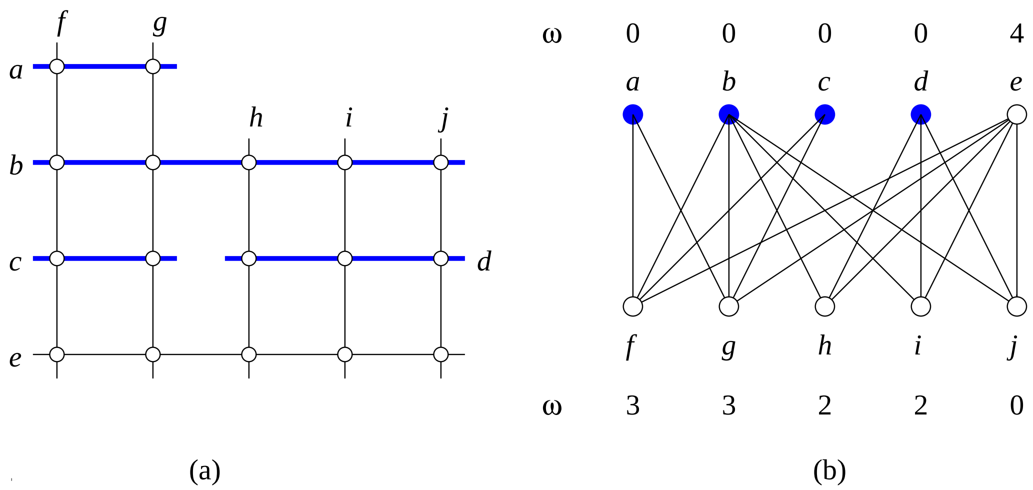


Figure 5.11: An example of the graph construction

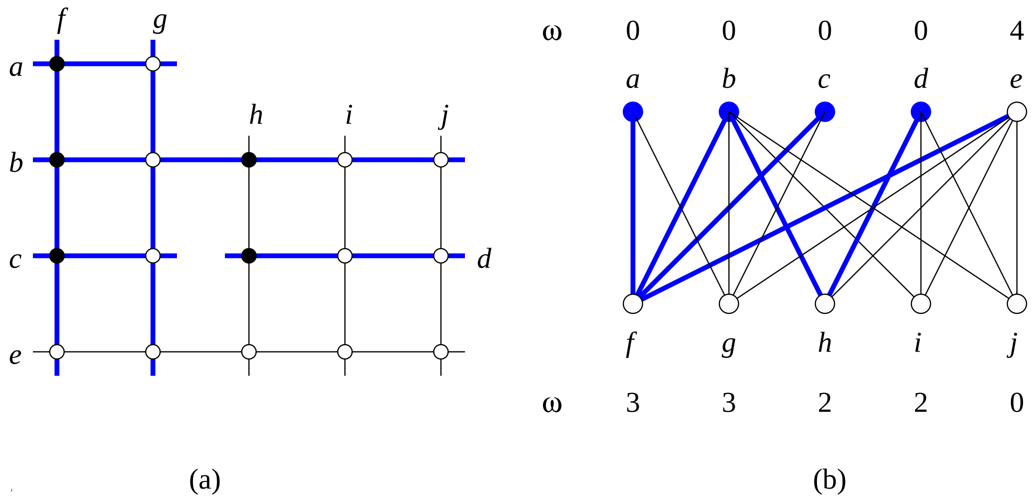


Figure 5.12: An optimal tree shown as bold lines in (b) and its routing in (a)

we assume that there is no articulation point in G, for otherwise we could split the graph into bi-connected components and solve them individually. Since each vertex in Z contributes to the cost of the spanning tree, in order to simplify our presentation we let the weight of each demand vertex be 0 and try to minimize the total cost of the edges and the additional vertices in a tree that spans Z. Given a bipartite graph G, the demand vertex set Z, and the weight function $w : (V \cup E) \rightarrow R^{+} \cup \{0\}$, where $w(e)$ has the same value for all $e \in E$, we use $NWSMT(G, Z, w)$ to denote such a node-weighted instance of the Steiner Minimum Tree (SMT) problem.

Our problem is similar to the conventional SMT problem, with two exceptions : (i) the underlying graph G is always a bipartite graph; (ii) weights are assigned to both vertices and edges, whereas the conventional SMT problem only allows edge weights. Specializing the problem to bipartite graphs with uniform edge weights does not make our problem easier. In fact, we have the following result :

Theorem 8 *Given a bipartite graph* $\mathrm{G} = (\mathrm{X} \cup \mathrm{Y}, \mathrm{E})$*, a demand vertex set* Z*, and a weight function* $\mathrm{w} : (\mathrm{V} \cup \mathrm{E}) \rightarrow \mathrm{R}^{+} \cup \{0\}$, NWSMT(G, Z, w) *is NP-complete even when (i) all vertices and edges are assigned the same weight, (ii)* G *is a planar bipartite graph, (iii) each internal face of* G *is of length 4, and (iv)* $\mathrm{Z} = \mathrm{X}$*.*

Proof : It is obvious that $NWSMT$ is in NP. In what follows, we give a polynomial time reduction from the Planar Vertex Covering [17] problem to the $NWSMT$ problem. Since Planar Vertex Covering is NP-complete [17], the NP-completeness result of our $NWSMT$ problem follows.

The Planar Vertex Covering problem is defined as follows : Given a planar graph $G_{VC} = (V_{VC}, E_{VC})$ and a positive integer K, find a subset $V' \subset V_{VC}$ with $|V'| \leq K$ such that for each edge $(u, v) \in E_{VC}$ at least one of u and v belongs to V'. From an instance of the planar

Vertex Covering problem, we construct an instance of our $NWSMT$ problem as below. First, we obtain a planar embedding of G_{VC}. The graph G in $NWSMT$ is constructed from such a planar embedding as follows. For each interior face of the embedding, we add a vertex inside the interior face. Moreover, for each vertex v on the boundary of this face, we add an edge connecting v and the vertex inside the face. An vertex is added along each edge in E_{VC}. Let G in the $NWSMT$ problem be the resultant graph. Figure 5.13 shows an example of such a construction. If we let Z in the $NWSMT$ problem be the set containing all the vertices added

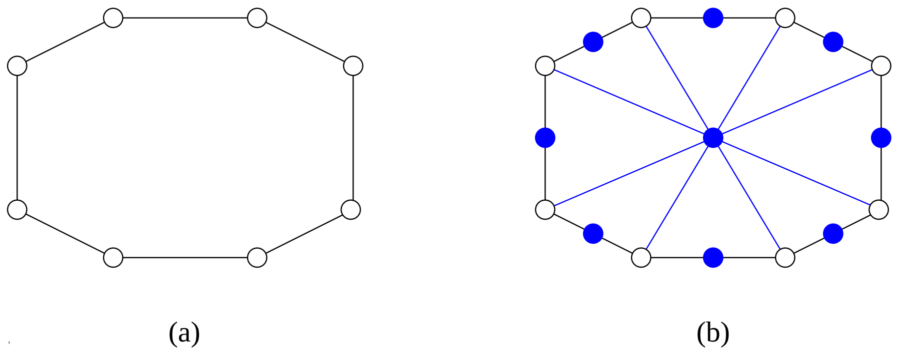


Figure 5.13: (a) G_{VC} (b) The corresponding G in $NWSMT$

during the construction, i.e., $Z = \{v \mid v \in V \setminus V_{VC}\}$ and the weight of each node not in Z and edge in the $NWSMT$ problem be 1, then it is easy to prove that there is a vertex cover of size K in the Planar Vertex Covering problem if and only if there is a Steiner tree of cost $2K + |Z| - 1$. $\square$

Therefore, it is unlikely that there exists a polynomial time algorithm for solving our problem optimally. As indicated by the comprehensive survey given in [23], there is an extensive literature on the conventional SMT problem. However, we know almost nothing about the Node-Weighted SMT problem. We propose an approximation algorithm for the single net routing problem and show that if there are at least two chords in each cycle of length 6 or larger in

G, then our algorithm gives an optimal solution. If G does not satisfy this property, then we prove that the cost of the solution obtained will be at most $min\{\Delta(V\backslash Z), |Z|-1\}$ times the cost of optimal solutions, where $\Delta(S) = max\{degree(v)|v \in S\}$. Moreover, we show that even the problem of finding a solution that is within a constant multiplying factor of the optimal solution is NP-complete.

5.4.1 A Single Net Routing Algorithm

Before describing our algorithm, we need some definitions. For any two sets, P and Q, $P\backslash Q = \{u|u \in P, u \notin Q\}$. Let v be a vertex in G. $G\backslash\{v\}$ denotes the subgraph of G obtained by removing v and its incident edges. Given $NWSMT(G, Z, w)$, a redundant vertex v, and a subset of vertices U, we use $N_U(v)$ to denote the neighbors of vertex v in U, plus those demand vertices which can be reached by v via a path consisting of only demand vertices. That is, $N_U(v)$ $= \{u|(u, v) \in E, u \in U\} \cup \{u|\exists$ a path $(p_1, p_2, \ldots, p_k)$, where $k \geq 1, p_1 = v, p_k = u, p_j \in Z$ for $j = 2, 3, \ldots k\}$. For any subgraph of G, H, we use $V(H)$ to denote the set of vertices in H. A vertex v in $V\backslash Z$ is called *redundant* if $G\backslash\{v\}$ is connected. A vertex v in $V\backslash Z$ is called *essential* if $G\backslash\{v\}$ is not connected.

We have the following lemma.

Lemma 8 *In a given problem instance,* NWSMT(G, Z, w)*, let* u *be a redundant vertex. If* u *satisfies either one of the following two conditions:*

1. *the degree of* u *is one;*

2. *there is a vertex* v $\in$ V *satisfying the condition* $\mathrm{N_V(u)} \subseteq \mathrm{N_V(v)}$ *such that either* v *is redundant and* w(u) $\geq$ w(v) *or* v *is essential;*

then there is an optimal tree that does not contain u.

Proof : If the degree of u is one, the lemma is trivially true. Consider the case in which the second condition is satisfied. Let T^* be an optimal tree. If u is not in T^*, then the lemma is proved. Let $u \in T^*$. If there is an essential vertex v satisfying the condition $N_V(u) \subseteq N_V(v)$, then v must be in T^* and the removal of u from T^* does not disconnect T^*. If there is a redundant vertex $v \in V$ satisfying the condition $N_V(u) \subseteq N_V(v)$ and $w(u) \geq w(v)$, then we construct another tree $\overline{T}$ from T^* as follows : Remove u and its incident edges from T^*. Let T_1^* denote the resultant graph. (In general, T_1^* is a forest.) Add v to T_1^* and connect v to all the vertices in $N_{V(T^*)}(u)$. The cost of $\overline{T}$ must be the same as that of T^*, due to the optimality of T^*. This completes the proof. □

A vertex $u \in V$ satisfying the conditions stated in Lemma 8 is called a *removable vertex*. Our algorithm has two phases. In the first phase, we check if there is a removable vertex. We remove all such vertices, if there are any. The removal of removable vertices usually creates one or more removable vertices in the resultant graph. We repeat the removal process until there is no removable vertex. If every vertex in $V \backslash Z$ in the remaining graph is essential, then we find a Minimum Spanning Tree (MST) of the remaining graph. This MST will be our solution. Otherwise, we continue with the second phase. In the second phase, we apply the MST approach for solving the conventional SMT problem to the remaining graph obtained in the first phase, G_1. In the MST approach, we build a complete graph $G_c = (Z, E_c)$, with the edge weight of edge $(z_1, z_2) \in E_c$ being the distance (the length of the shortest path) between z_1 and z_2 in the graph G_1. Then, an MST of G_c can be constructed easily and each edge in this tree is converted back to the corresponding shortest path in G_1. The resultant subgraph of G_1 spans Z. It is a connected subgraph, but might not be a tree. We find an MST of this subgraph and remove all leaf vertices which are in $V \backslash Z$. The resultant tree is used as our solution to approximate the optimal tree. The time complexity of the first phase is $O(|V|^4)$, since finding the vertices u and v described in Lemma 8 takes $O(|V|^3)$ time, and there are at most $|V|-1$ such vertex pairs. The time complexity of the first phase dominates that of the second phase, which is $|Z|(|E|+|V|log|V|)$.

5.4.2 (6, 2)-graph

In this section, we prove that our algorithm produces an optimal tree if the graph G satisfies a certain chordality property. A chord of a cycle is an edge connecting two non-adjacent vertices in the cycle. A graph G is called an *(s, t)-graph* if and only if each cycle of length at least s in G has at least t chords. An example of a (6, 2)-graph is shown in Figure 5.11(b). We have the following lemma:

Lemma 9 *Given a problem instance* NWSMT(G, Z, w)*, if* G *is a (6, 2)-graph, then either every vertex of* V\Z *in* G *is essential or there is a vertex* u *satisfying the conditions in Lemma 8.*

Proof : Suppose the contrary. There is at least one redundant vertex in $V\backslash Z$. Let Q be the set of redundant vertices. Let q^* be a vertex in Q with the largest weight. q^* must have degree of at least two; otherwise, the lemma is proved. Consider two neighbors of q^*, vertices s and t, as shown in Figure 5.14. Since q^* is redundant and G is bipartite, there is a path P of even length connecting s and t after the removal of q^*. Let s_1 (t_1) be the neighbor of s (t) on path P. Then, either $(s,t_1) \in E$ or $(t,s_1) \in E$, since otherwise in the subgraph $P \cup \{(s,q^*),(t,q^*)\}$ there must be a cycle of length at least 6 with at most one chord. We shall prove that there must be another vertex q such that $N_V(q^*) \subseteq N_V(q)$. If the degree of q^* is two, then based on the above observation, q is either s_1 or t_1, depending on which of (s,t_1) and (t,s_1) is in E. If

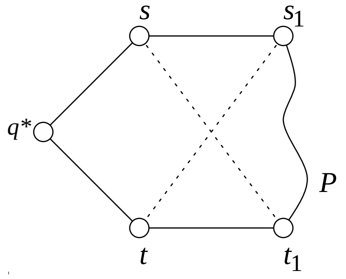


Figure 5.14: Proof of Lemma 9

q is essential, the lemma is proved. Otherwise, according to the definition of q^*, $w(q^*) \geq w(q)$. In both cases, q^* satisfies the conditions in Lemma 8. Suppose the degree of q^* is k, where $k > 2$. Let x be one of the vertices in $G\backslash\{q^*\}$ such that x shares the most common neighbors with q^*. We use $d_1, d_2, \ldots, d_l$ to denote these common neighbors as shown in Figure 5.15, where $l < k$. Let d_{l+1} be adjacent to q^*, but not to x. According to the observation above,

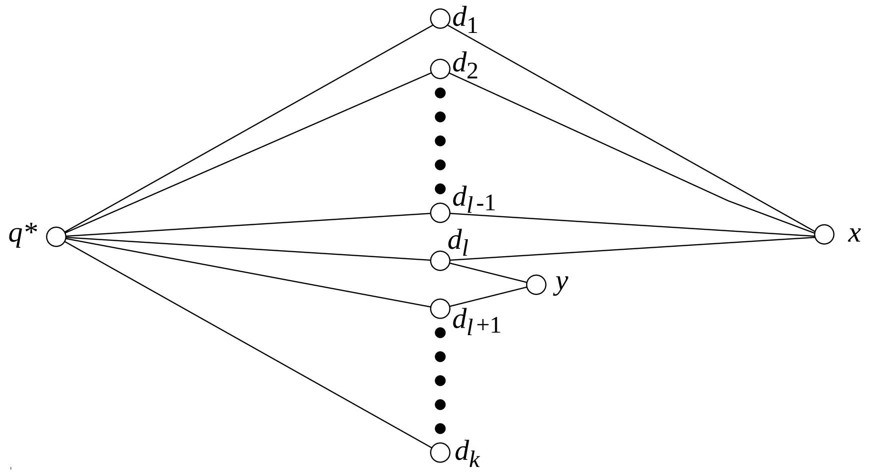


Figure 5.15: A more complicated case

there must be another vertex y, other than x and q^*, adjacent to both d_l and d_{l+1}. Consider the cycle $(q^*, d_{l+1}, y, d_l, x, d_{l-1})$. Since G is a (6, 2)-graph, either (x, d_{l+1}) or (y, d_{l-1}) is in E. It is contrary to the definition of x if $(x, d_{l+1}) \in E$. Therefore, $(y, d_{l-1}) \in E$. Consider the cycle $(q^*, d_{l+1}, y, d_{l-1}, x, d_{l-2})$. By the same token, we have $(y, d_{l-2}) \in E$. Repeating this argument, we conclude that $d_j \in N_V(y)$, for $j = 1, 2, \ldots, l+1$. Again, this is contrary to the definition of x. Therefore, x must be adjacent to all the neighbors of q^*. Moreover, x must be redundant, otherwise the lemma is proved. According to the definition of q^*, $w(q^*) \geq w(x)$. This is contrary to our original assumption, and the lemma is proved. □

From the observation that any induced subgraph of a (6, 2)-graph is also a (6, 2)-graph, it follows immediately that our algorithm produces an optimal tree for any (6, 2)-graph

Theorem 9 *Our algorithm produces an optimal tree for a problem instance* NWSMT(G, Z, w)*, if* G *is a (6,2)-graph.*

A graph is a *(6,1)(8,3)-graph* if and only if each cycle of length at least 6 has at least one chord and each cycle of length at least 8 has at least 3 chords. (A bipartite graph does not contain cycles of length 7.) Using a similar, but more complicated, technique, we can prove:

Theorem 10 *For a given problem instance* NWSMT(G, Z, w)*, if* G *is a bipartite (6,1)(8,3)-graph and the weights of all vertices and edges are the same, then our algorithm produces an optimal tree.*

5.4.3 Performance Bound

Most of the time the MST approach provides a good solution and its cost is guaranteed to be within twice the cost of an optimal Steiner tree. As a result, the MST approach has been used extensively to approximate an optimal Steiner tree in conventional SMT problems encountered in VLSI routing [28]. However, no result has been reported regarding use of the MST approach to approximate Node-Weighted SMT.

Theorem 11 *Given a problem instance* NWSMT(G, Z, w)*, let* T *be the tree obtained by the MST approach presented above and* T^* *be an optimal tree. We have*

$$\frac{\mathrm{w(T)}}{\mathrm{w(T^*)}} \leq \min\{\Delta(\mathrm{V}\backslash\mathrm{Z}), |\mathrm{Z}|-1\}$$

Proof : Note that every leaf of T^* must be in a demand vertex. Choose a leaf vertex and denote it by u_1. Consider a walk W in T^* beginning at u_1, containing all the vertices of T^*, including each edge of T^* at most twice, and ending at a demand vertex. Let $u_1, u_2, \ldots, u_{|Z|}$ be the sequence of demand vertices ordered in accordance with the first appearance in W, as in the example shown in Figure 5.16. W can be decomposed into $|Z|-1$ paths : $u_1 - u_2$ path,

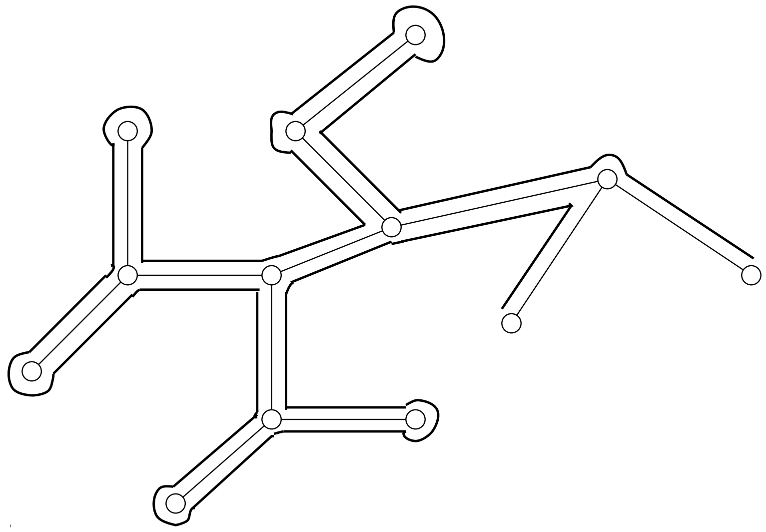


Figure 5.16: An example used in the proof of Theorem 11

$u_2 - u_3$ path, ..., $u_{|Z|-1} - u_{|Z|}$ path. The distance between u_i and u_{i+1}, denoted by $d(u_i, u_{i+1})$, does not exceed the length of $u_i - u_{i+1}$ path, denoted by $w(u_i - u_{i+1})$, for $i = 1, 2, \ldots, |Z|-1$. Let T' be the spanning tree of Z in the complete graph G_c consisting of edges (u_1, u_2), (u_2, u_3),

$\ldots$, $(u_{|Z|-1}, u_{|Z|})$. Then, we have

$$w(T) \leq w(T') = \sum_{i=1}^{|Z|-1} d(u_i, u_{i+1}) \leq \sum_{i=1}^{|Z|-1} w(u_i - u_{i+1}) = w(W) < min\{\Delta(V \backslash Z), |Z| - 1\} w(T^*)$$

$\square$

Figure 5.17 shows that this bound is tight, where the weight of each edge is ϵ.

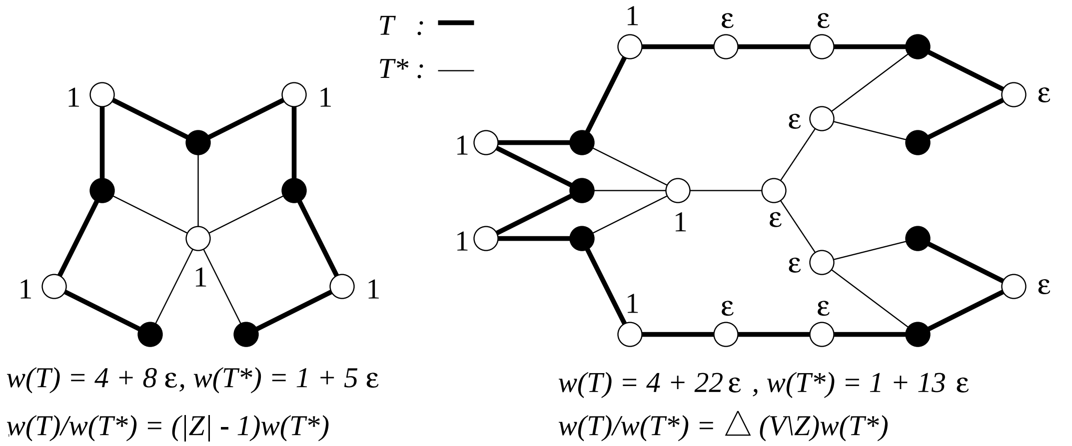


Figure 5.17: Illustration of the tight bound

We shall show that approximating the optimal solution of the NWSMT problem within any constant bound is NP-complete. Therefore, it is unlikely that there exists a polynomial time algorithm that can produce a solution T such that $\frac{w(T)}{w(T^*)} \leq c$, for any constant c. We shall call such an approximating problem the Approximating NWSMT problem.

Theorem 12 *Given a problem instance* NWSMT(G, Z, w)*, let* T^* *be an optimal tree. The problem of obtaining a solution* T *such that* $\frac{\mathrm{w(T)}}{\mathrm{w(T^*)}} \leq \mathrm{c}$*, for any constant* c*, is NP-complete.*

Proof : It is obvious that the Approximating NWSMT problem is in NP. In the following, we shall give a polynomial time reduction from the Approximating Set Covering [17] problem to the Approximating NWSMT problem. Since Approximating Set Covering is NP-complete [3], the NP-completeness result of our Approximating NWSMT problem follows.

The Approximating Set Covering problem is defined as follows : Let S be a set and C be a set of subsets of S. We want to find a subset C' of C such that C' covers[3] S and $\frac{|C'|}{|C^*|} \leq k$, where k is a given positive number and C^* is one of the smallest subset of C that covers S. From an instance of the Approximating Set Covering problem, we construct an instance of our NWSMT problem as follows. Corresponding to each subset in C there is a vertex in X. Corresponding to each element in S there is a vertex in Y. Let x and y be vertices in X and Y, respectively. $(x, y) \in E$, if the element corresponding to y is in the subset corresponding to x. We add a dummy vertex d to Y, and d is connected to all vertices in X. An example is shown in Figure 5.18, where $S = \{s_1, s_2, s_3, s_4, s_5\}$, $C = \{\{s_1, s_2, s_4\}, \{s_2, s_3, s_5\}, \{s_1, s_2, s_3\}, \{s_3, s_4, s_5\}\}$. Let $Z = Y$, $w(x) = w(e) = 0$, for all $x \in X$ and $e \in E$, and $w(y) = 1$, for all

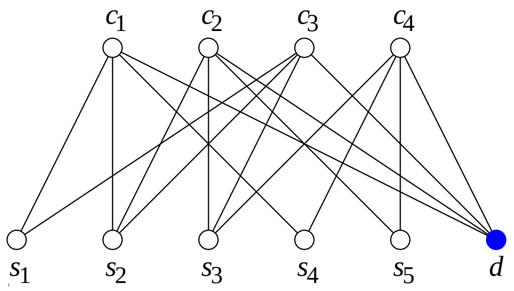


Figure 5.18: The reduction

$y \in Y$. Because of the dummy vertex d, for each subset C'' of C that covers S, the subgraph induced by the vertex set corresponding to C'' union-ed with Y is connected and spans Z. On the other hand, for each Steiner tree T of Z, those subsets corresponding to vertices in $T \cap X$

[3] A collection of subsets of a finite set S, is said to *cover* S, if each element of S belongs to at least one of the subsets in the collection.

form a cover of S. Therefore, there is a subset C' of C such that C' covers S and $\frac{|C'|}{|C^*|} \leq k$ if and only if there is a tree T spanning Z and $\frac{w(T)}{w(T^*)} \leq k$. $\square$

5.5 Experimental Results

Our algorithm was implemented in the C language and executed on a SPARC station. Because of the sequential nature of our algorithm, a rip-up and re-route phase is incorporated. The program was used to route five industrial circuits used in [6]. Segments in each channel are arranged according to the scheme presented in Section 5.3. Let l be the number of tracks in each channel, and let k be the number of programmable switches allocated for each horizontal segment and vertical segment at the intersection of a horizontal channel and a vertical channel. For BUS, a bus controller, we use BUSx_y to denote the case when $l = x$ and $k = y$. For each $l = 12, 13, 14$, we found the minimum value of k, so that our algorithm produces a 100% complete routing solution as shown in the second, third, and fourth columns of Table 5.1. For the other four circuits, we found the minimum number of tracks needed to have a 100% completion rate and the corresponding minimum value of k. The positions of these programmable switches are randomly generated such that the routing resource is spread evenly without any bias. The experimental results are summarized in Table 5.1 and Table 5.2. BUS was routed on an array of 13×12 blocks with 151 nets, or 392 equivalent two-terminal nets. The routing result in [6] used a total of 1616 active switches in the connecting paths. In our case, we used less than 1170 in all these instances. As shown in Table 5.1, the reduction is dramatic. It should be noted, however, that the number of tracks used in the routing structure in [6] is only 10, and that our algorithm cannot produce a 100% complete routing using less than 12 tracks, even if we let $k = l$. The reason is that, in comparison with the conventional FPGA routing structure, our

	BUS12_11	BUS13_9	BUS14_6	BUS14_7	BUS14_8
$\frac{k}{l}$	92%	69%	43%	50%	57%
Blocks	13 × 12				
Nets†	392				
Switches [6]	1616				
Switches(Ours)	1165	1151	1167	1158	1135
S. Reduction	28%	29%	28%	28%	30%
Tracks [6]	10				
Tracks(Maze)‡	15				
Time(min.)	0.79	0.70	0.74	0.75	0.74

Table 5.1: Experimental results for BUS

	DMA14_12	BN15_13	FSM16_7	Z16_15
Blocks	18 × 16	22 × 21	23 × 22	27 × 26
Nets†	771	1257	1422	2135
Switches [6]	3354	5700	6132	10039
Switch(Ours)	2259	3830	4193	5971
S. Reduction	33%	33%	32%	41%
Tracks [6]	10	12	10	13
Tracks(Maze)‡	15	20	18	18
Time(min.)	4.0	19.3	16.9	31.6

Table 5.2: Experimental results for the other four industrial circuits

†Equivalent two-terminal connections

‡Maze routing on the conventional 2-D FPGA routing structure [6]

class of routing structures is less flexible, as, unlike the conventional FPGA routing structure, we do not use switch blocks at all the intersections of horizontal and vertical channels. One should also point out that, since the delays due to the programmable switches dominate those due to the metal wire segments [6], the additional delays in the connecting paths due to the increase in chip size are negligible. The last three columns in Table 5.1 show that increasing k decreases the number of switches used in our routing solution without much effect on the running time. However, a larger number of switches will increase the area of the chip. In Table 5.2, the maximum reduction in the number of active switches is 41%, which happens in the largest circuit, Z16_15.

The routing result for BUS14_6 is shown in Figure 5.19.

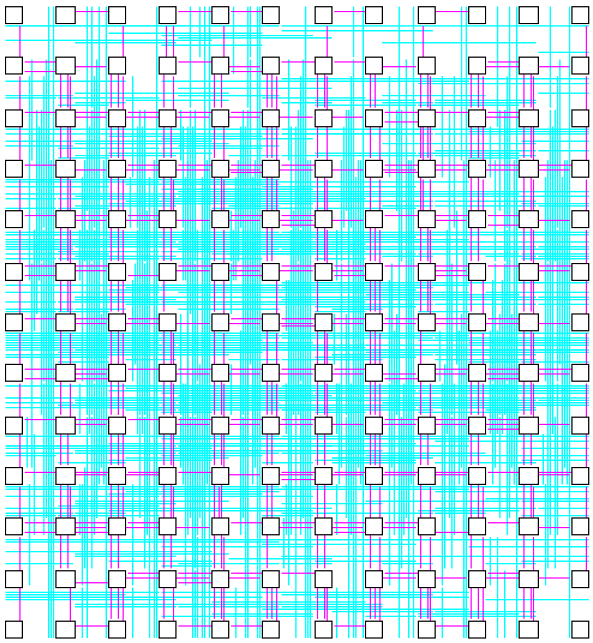


Figure 5.19: B14_6

5.6 Summary

In this chapter, we propose a class of routing structures in which there are fixed orthogonal wire segments. This class of routing structures offers the important benefit of rapid personalization by using programmable switches at some intersections of the orthogonal wire segments. Two examples of this class of routing structures are symmetric FPGA routing structure and FPIC routing structure. Using a probabilistic model, we prove that this class of routing structures is asymptotically as efficient as routing structures without fixed wire segments. We also study the single net routing problem, which is the foundation of a sequential routing algorithm. We take into account the delay in a routing path due to the programmable switches and the wire segments, and formulate the single net routing problem as a Node-Weighted Steiner Minimum Tree problem in a bipartite graph. Then, we propose a polynomial time algorithm to approximate the optimal solution. We prove that our algorithm produces an optimal solution if the graph is a (6,2)-graph. If all the vertices and edges have the same weight, then our algorithm also produces an optimal solution for a (6,1)(8,3)-graph. In general, the solution obtained by our algorithm has a performance bound of $min\{\Delta(V \backslash Z), |Z|-1\}$. In addition, we prove that it is NP-complete to approximate the optimal solution within any constant bound. Experimental results demonstrate that, in comparison with the conventional two-dimensional FPGA routing structure, our class of routing structures significantly reduces the number of switches and, thus, the signal delays.

Chapter 6

Routing in a New FPGA/FPIC Routing Architecture

6.1 Introduction

The conventional two-dimensional FPGA/FPIC routing architecture proposed in [6, 12, 38, 59] is shown in Figure 6.1 which contains only wire segments of unit length. Each square represents a logic block implementing logic functions in the case of FPGA and represents an I/O pin in the case of FPIC. The terminals of a block are located on the boundaries of the block and are connected to wire segments called *terminal segments*. Terminal segments are connected to wire segments in the routing channel through switches at their intersections which are shown as black circles in Figure 6.1. The switch matrix makes connections between wire segments in horizontal and vertical channels. In the switch matrix, each wire segment can be connected to a subset of the wire segments on the other sides of the matrix. A routing example of a 2-terminal net is shown in bold lines. There are eight active switches in the connecting

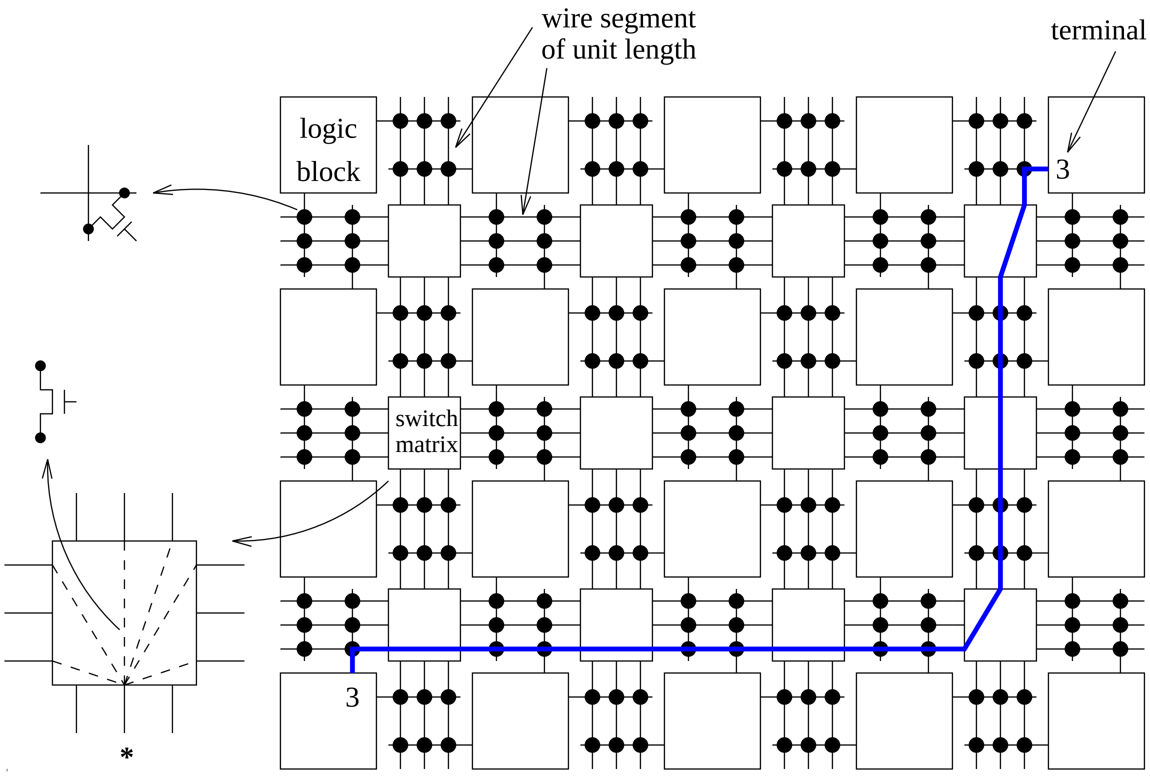


Figure 6.1: Conventional two-dimensional FPGA architecture and a routing example

path, six of which are in switch matrices and each of the remaining two switches connects a terminal of the net to a wire segment in a routing channel.

In earlier Xilinx FPGA chips, interconnection resources were grouped as direct connects, general interconnects, and long lines [60]. A general interconnect, running horizontally and vertically for unit length between two switch matrices, is normally used to route signals within a localized area. Since the number of active programmable switches in a connecting path is equal to the length of the path (minus one), for moderately critical nets or nets of intermediate length, there will be substantial delay when general interconnects are used. To improve upon the situation, the recently-developed Xilinx XC4000 series provides double-length lines which are twice as long as single-length lines. However, the variety of wire segments of different lengths is still very limited.

In Chapter5 and [50], a new FPGA/FPIC routing architecture that allows a larger variety of wire segments of different lengths was proposed. Programmable switches are available only at some of the intersections between horizontal and vertical wire segments. In this routing

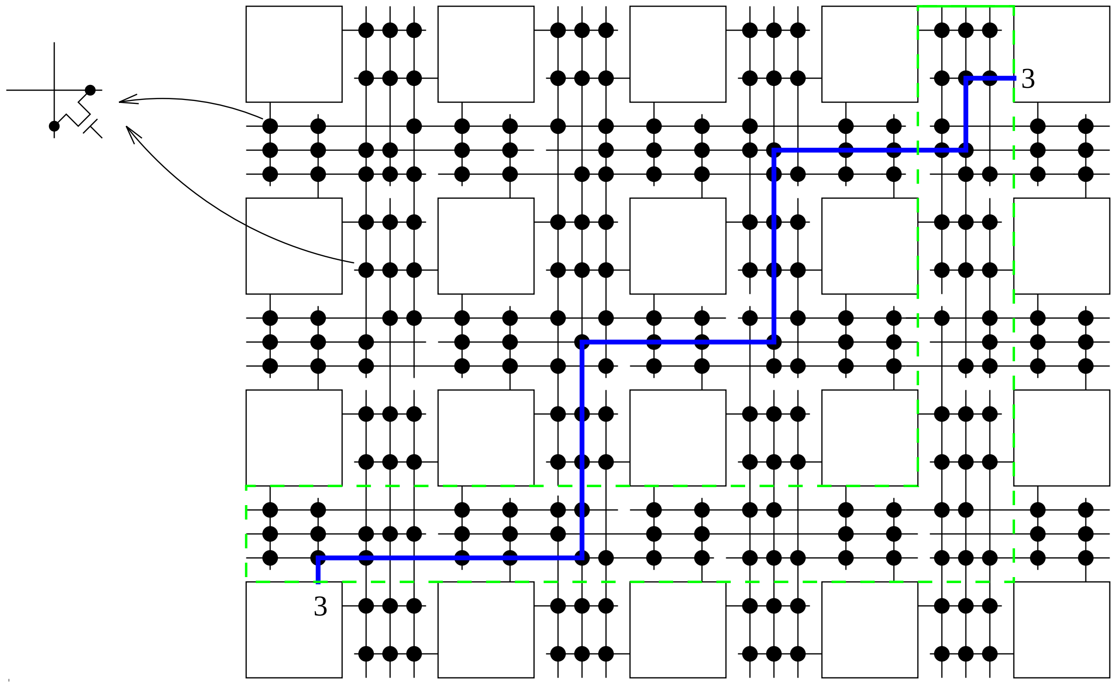


Figure 6.2: Two-dimensional FPGA/FPIC routing architecture in [50]

architecture, a signal passes through an active programmable switch only when making a turn, whereas in the conventional FPGA routing architecture there is an active switch in each unit length. Consequently, signal delay is reduced dramatically. Programmable switches provided to allow routing paths to make turns are called *turning switches*. An example of the routing architecture is shown in Figure 6.2. A routing example for the same 2-terminal net in Figure 6.2 is also depicted. Note that only seven turning switches are active in the connecting

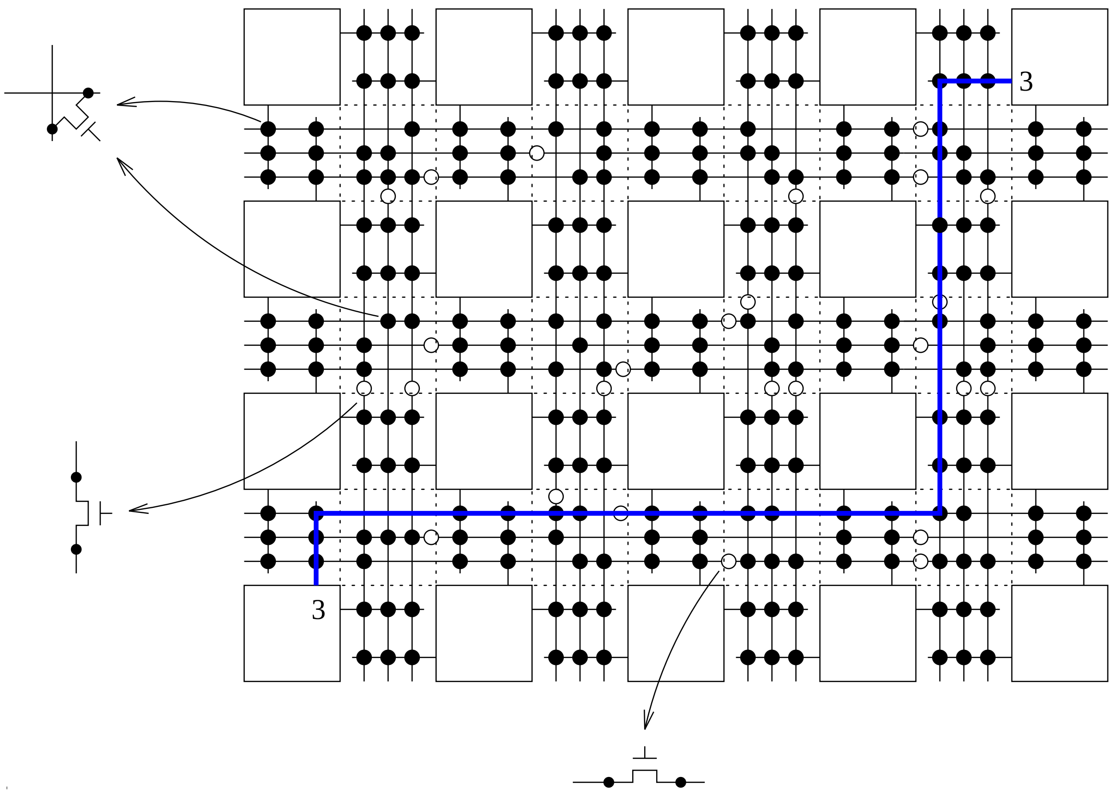


Figure 6.3: Proposed two-dimensional FPGA/FPIC routing architecture and a routing example

path, compared with eight switches used in Figure 6.1. Nevertheless, low routability is a main drawback of this routing architecture, although asymptotically it has been shown to be as efficient as conventional routing architecture with free tracks. In this chapter, we propose a routing architecture which improves the routability by adding programmable switches between two horizontal (or vertical) wire segments which can be programmed to connect two horizontal (or vertical) wire segments to form a longer horizontal (or vertical) segments. Programmable switches for this purpose are called *continuing switches*. Figure 6.3 depicts our new routing architecture corresponding to that in Figure 6.2 with the continuing switches shown as white circles. The dotted squares can be viewed as switch matrices in the conventional FPGA/FPIC architecture, except that segments can pass through them without activating any programmable switch. The same 2-terminal net in Figure 6.1 and 6.2 is routed passing through five switches, two of them are continuing switches and the other three are turning switches. It is easy to see that any routing path in Figure 6.2 can also be found in Figure 6.3, since all routing resource in Figure 6.2 is retained in Figure 6.3. Moreover, the usage of continuing switches usually reduces the number of turns in the routing path and results in either a shorter path or a path with fewer active switches. Therefore, introduction of continuing switches not only enhances the routability, but also decreases the number of active switches in the interconnecting routes, and thus, the routing delay.

6.2 A Routing Algorithm

We now present a routing algorithm for the proposed routing architecture. The first step of our algorithm is to find a conventional two-dimensional global routing solution which satisfies the capacity constraints in the channels. That is, the number of routes passing through any horizontal (or vertical) cut in a vertical (or horizontal) channel is less than or equal to the number of tracks in the channel. A global routing solution not only ensures an even distribution of the nets in the routing channels to avoid congestion but also reduces the complexity of the overall routing problem. Note that the global routing step is not appropriate for the architecture proposed in [50] due to the lack of continuing switches in the routing architecture. The very limited amount of routing resource in this architecture makes the capacity constraint a weak necessary condition for achieving a detailed routing solution in most cases. That is, a global routing solution satisfying the capacity constraint often does not lead to a detailed routing solution. Figure 6.2 shows an example of such a scenario when we try to find a detailed route corresponding to a given global route. The polygon with dashed-line boundary is the given global route for net 3. However, because of the lack of continuing switches, we cannot find a detailed route corresponding to the given global route. However, with the added continuing switches, a detailed route for net 3 can be found easily as shown in Figure 6.3.

Our algorithm for finding a detailed route corresponding to a given global route consists of two phases. In the first phase, we process the nets one at a time and a polynomial time algorithm is used to find an optimal detailed route for each net. When an unroutable net is encountered, a polynomial time rip-up and reroute algorithm is used in the second phase to resolve the conflict. We shall describe these two phases in the remainder of this section. We assume that a route cannot change track in a channel. From a practical point of view, changing track activates more turning transistors which degrades the timing performance. Moreover, the net will occupy more wire segments in the same channel, and thus, increases the possibility of blocking other nets. An example is shown in Figure 6.4, where the route for net 3 in Figure 6.4(a) activates 7 programmable switches and occupies 3 wire segments in the horizontal channel, while the route in Figure 6.4(b) activates only 4 programmable switches and occupies 2 wire segments in the same channel. Note, however, that our algorithm can nonetheless be extended to handle the case in which track changing is allowed.

6.2.1 An Optimal Single-net Routing Algorithm

Suppose we are given a net and its global route as depicted in Figure 6.5(a) where the global route is shown as a dashed polygon. We can represent these information using a two-dimensional grid as follows : Each grid point corresponds to a switch matrix and each grid edge (u, v) corresponds to the channel between the two switch matrices corresponding to u and v. Each

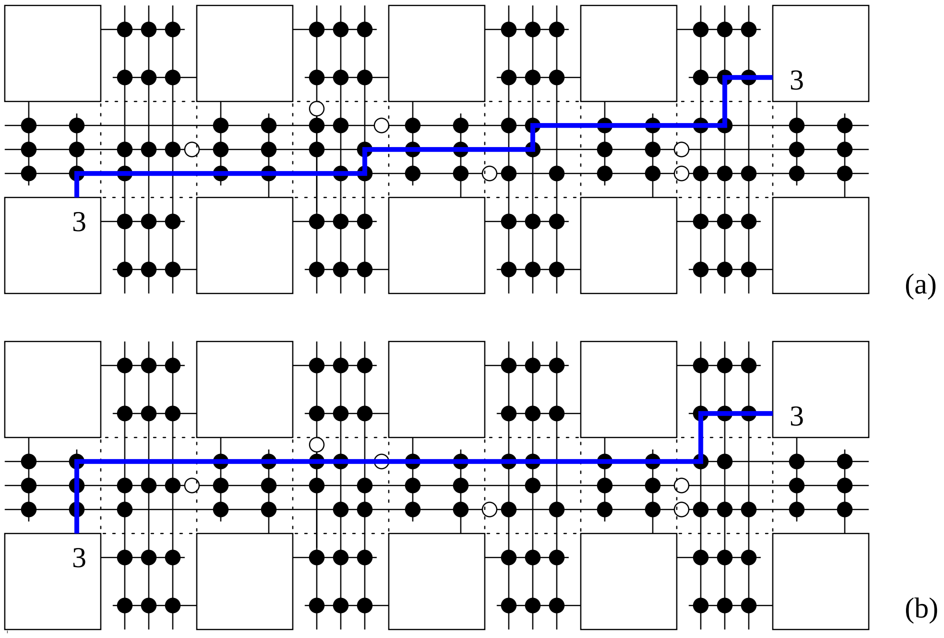


Figure 6.4: Changing tracks

rectangular region corresponds to a logic block, the terminals of which are represented by points in the region. (For each net, there is only one terminal in each logic block.) There is a short edge, corresponding to a terminal segment, which connects a terminal to a grid edge corresponding to a channel. A global route is represented by a tree containing short edges and grid edges. We call such a two-dimensional grid representation of a global route a *channel representation*. Figure 6.5(b) shows the channel representation of the global route of net 3 given in 6.5(a). Given the global route of a net with k terminals, no two of which are connected to the same channel, if the number of tracks to which each terminal segment can be connected is t, then the number of corresponding detailed routes is at least t^k. In other words, the number of detailed routes for a given global route can grow exponentially with respect to the number of terminals. Among these detailed routes, we want to determine one with minimum cost. The cost is defined as a weighted sum of the cost of active programmable switches and the cost of

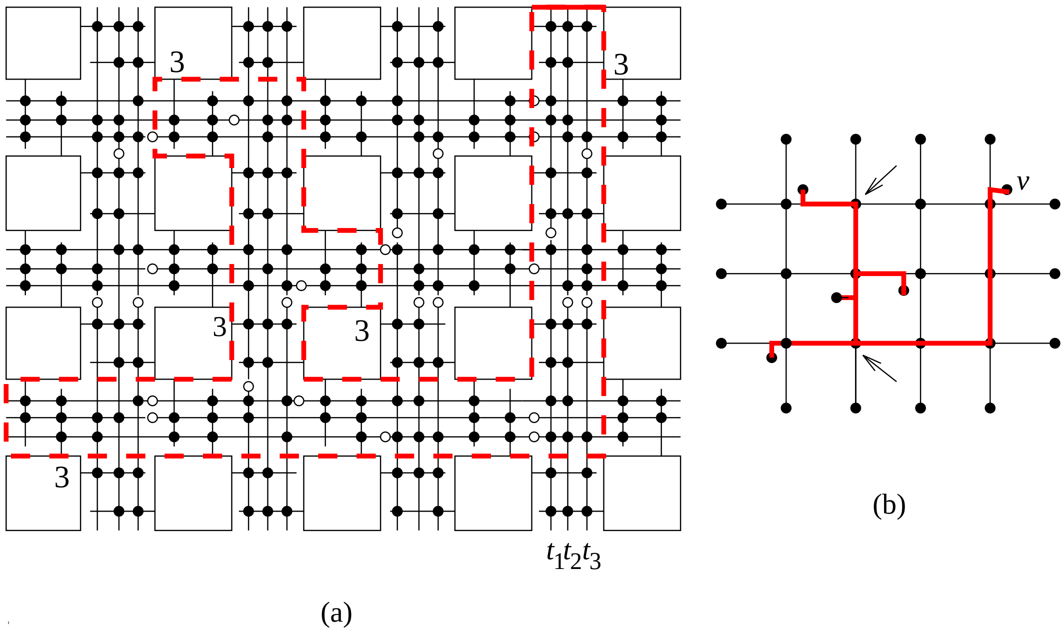


Figure 6.5: A global route of net 3 and its channel representation

wire segments in the route. The cost of a wire segment is a weighted measure of the delay associated with the wire segment and how much the segment is in demand for routing. The cost of a active programmable switch is the time delay it introduces. Since there might not be a turning switch located at the intersection of a horizontal track and a vertical track in a switch matrix, we can conceptually assume that there is a turning switch with infinite cost at such an intersection. Therefore, we shall assume that there is a turning switch at each intersection of a horizontal track and a vertical track in a switch matrix in the rest of this section. Also, we set the cost of any wire segment, turning switch, or continuing switch to be infinity, when it is occupied by a net processed before. In this subsection, we present a polynomial time algorithm to solve this problem optimally using dynamic programming technique.

Given the channel representation of a global route, we decompose the channel representation into maximal line segments. A maximal line segment is a line segment that is not properly contained in any other line segment in the channel representation. For example, the line segment between the two arrows in Figure 6.5(b) is a maximal line segment. We arbitrarily choose a terminal x and build a rooted tree, $T = (V_T, E_T)$, which represents the intersection relationship between maximal line segments as follows: Each maximal line segment has a corresponding vertex in V_T and $(u, w) \in E_T$ if and only if the maximal line segments corresponding to u and w intersect in the channel representation. The root of T is the maximal line segment incident with x. The rooted tree T for the channel representation in Figure 6.5(b) is shown in Figure 6.6. Given a node w in T, we use $l(w)$ to denote the maximal line segment corresponding to w. For a

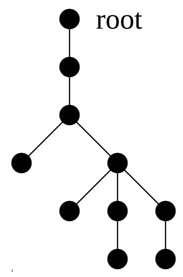


Figure 6.6: T for the channel representation in Figure 6.5

maximal line segment h, we use $r(h)$ to denote the channel that contains h. Also, we use $l^{-1}(h)$ to denote the node w of T such that $l(w) = h$. If w is a leaf in T, then $r(l(w))$ is a terminal segment. If w is an internal node of T, $r(l(w))$ is the channel which contains the maximal line segment $l(w)$.

Given T, the routing problem becomes that of choosing a track in $r(l(w))$ for the detailed route of the maximal line segment $l(w)$, for each $w \in T$. Suppose we have chosen a track for the maximal line segment $l(w)$, for each $w \in T$. Then, we can obtain the cost of a subtree in T, rooted at a node, say s, by adding the costs of the wire segments used in the route for the maximal line segment $l(w)$, for each descendant[1] w of s, and the costs of all turning switches and continuing switches used in the route.

Suppose there are t tracks in each channel. We use C_j^s to denote the minimum cost of the route for the entire subtree rooted at s, with track j, $1 \leq j \leq t$, being chosen for the route of $l(s)$. We call C_j^s, for each j such that $1 \leq j \leq t$, a *C-cost* of s. In the case that s is a leaf, the cost of the subtree rooted at s is simply the cost of a terminal segment. In this case, we can conceptually assume there are t terminal segments the terminal is connected to and then C_j^s, for $1 \leq j \leq t$, is well-defined. In the case that s is an internal node of T, let $v_1, v_2, \ldots, v_n$ denote the children of s in T. Then, we have the following recurrence equation:

$$\begin{aligned} C_k^s = \sum_{i=1}^{n} \min_{m=1,2,\ldots,t} &(C_m^{v_i} + cost\_of\_turning\_switch\_at(k, m, i)) \\ &+ \text{costs of continuing switches on track } k \text{ in } r(l(s)) \\ &+ \text{costs of wire segments on track } k \text{ in } r(l(s)) \quad \forall k \in \{1, 2, \ldots, t\} \end{aligned} \tag{6.1}$$

where $cost\_of\_turning\_switch\_at(k, m, i)$ is the cost of the turning switch located at the intersection of the kth track in $r(l(s))$ and the mth track in $r(l(v_i))$. Figure 6.7 shows an example where $n = 2$, $t = 3$, and the gray circles represent turning switches with infinite cost. Suppose s is the maximal line segment corresponding to the horizontal channel in Figure 6.7, which is denoted by $r(l(s))$. s has two children, α and β. $r(l(\alpha))$ and $r(l(\beta))$ are the middle and the rightmost vertical channels. Let p denote the parent of w. The leftmost vertical channel is $r(l(p))$. C_1^α, the minimum cost of the route for the subtree rooted at α with track 1 being chosen for the route of $l(\alpha)$, is 95. Similarly, $C_2^\alpha = 60$, $C_3^\alpha = 79$, $C_1^\beta = 77$, $C_2^\beta = 85$, and C_3^β

[1] We assume a node in a rooted tree to be a descendant of itself.

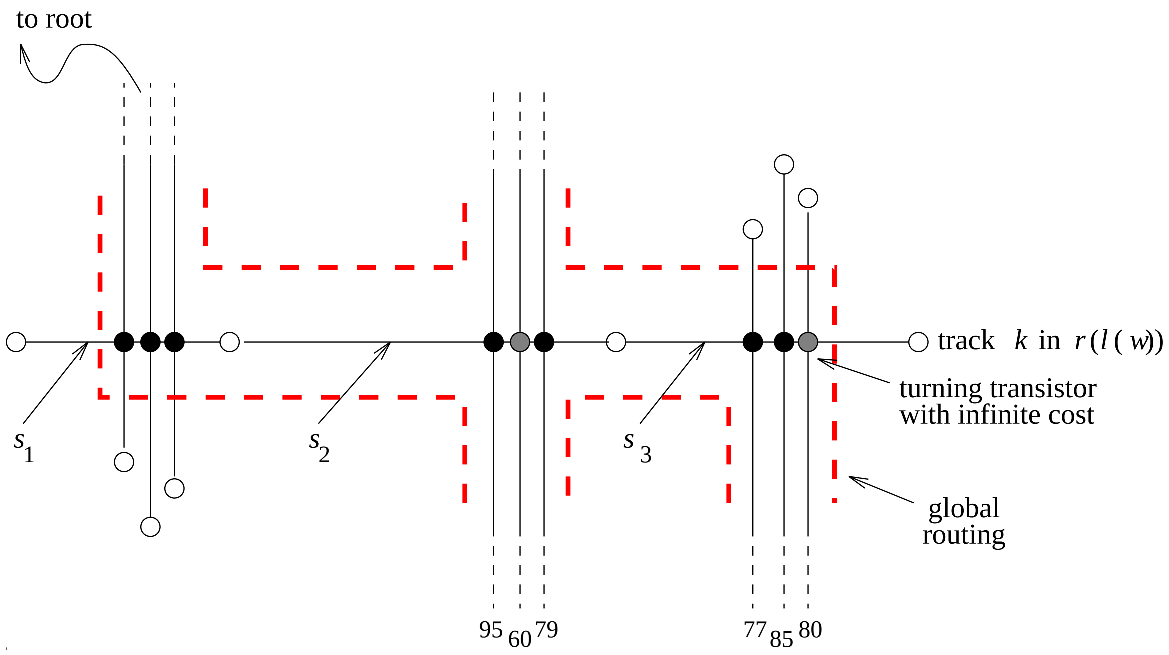


Figure 6.7: Illustration of the cost computation

= 80. The cost of the route for $l(s)$ includes the cost of three wire segments g_1, g_2, and g_3 as well as the two continuing switches, one connecting wire segments g_1 and g_2 and the other connecting wire segments g_2 and g_3. To compute C_k^s, we need to choose a track, say track a, for the route of $l(\alpha)$ and a track, say track b, for $l(\beta)$ such that the sum of the following costs is minimized: C_a^α, the cost of the turning switch at the intersection of track a in $r(l(\alpha))$ and track k in $r(l(s))$, C_b^β, the cost of the turning switch at the intersection of track b in $r(l(\beta))$ and track k in $r(l(s))$, and the cost for the route of $l(s)$. Therefore,

$$\begin{aligned} C_k^s = \min\,&(95 + \text{cost(turning switch)}, 60 + \infty, 79 + \text{cost(turning switch)}) + \\ &\min\,(77 + \text{cost(turning switch)}, 85 + \text{cost(turning switch)}, 80 + \infty) + \\ &2\cdot\text{cost(continuing switch)} + \text{cost}(g_1) + \text{cost}(g_2) + \text{cost}(g_3) \end{aligned}$$

According to formula (6.1), to compute the values of all C_k^w's, we need to compute the C-costs of all children of w. Therefore, our algorithm processes the tree T in a bottom-up fashion and computes the C-costs for each node until the root is reached. Then, we choose one with minimum C-cost and trace back to decide the track for each maximal line segment.

Our algorithm computes an optimal detailed route for a given global route as stated in the following theorem:

Theorem 13 *Our algorithm computes a detailed route with minimum cost in $O(mt(mt+p))$ time, where m is the number of nodes in T, t is the number of tracks in each channel, and p is the maximum number of wire segments among all tracks.*

The fact that our algorithm guarantees a detailed route with minimum cost can be proved easily by induction. To analyze the time complexity of our algorithm, we observe that there are m nodes in T and each node has t C-costs to be computed, one for each track. To compute a C-cost of a node for a track using the formula described above, we need to use the costs of the wire segments along this track and the C-costs of its children for this track. The number of all these variables is $O(p+kt)$, where k is number of children of the node considered and it is at most m. Therefore, the time complexity part of the theorem follows.

6.2.2 Rip-up and Reroute

Since the global router tries to avoid congestion by distributing the global routes evenly in the channels, it is likely that the detailed routing algorithm presented in the previous subsection will succeed in finding a detailed route for each net. In the case that it fails to find a detailed route, we use a rip-up and reroute algorithm to reroute some of the existing routes one track at a time to make room for the route of the current net.

Suppose the global route for net n_7 passes through channel c in which there are existing detailed routes of other nets as shown in Figure 6.8(a), where dashed lines sketch part of the global route of net n_7. Specifically, the detailed route for net n_7 is blocked by nets $n1$, n_2, n_3,

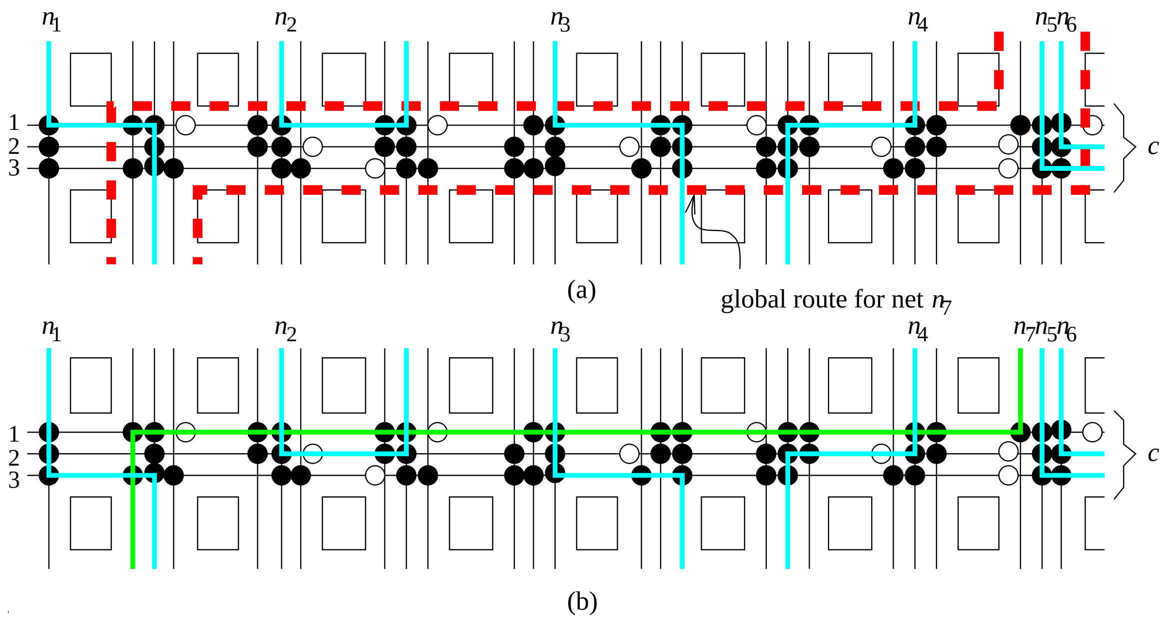


Figure 6.8: Rip-up and reroute

and n_4 in track 1, by net n_5 in track 3, and by net n_6 in track 2. We consider the tracks in the channel one at a time and shall rip-up and reroute all detailed routes existed in the track considered. Let t_1 be the track under consideration. We route the blocked net in track t_1 and then try to reroute all detailed routes which existed in track t_1. If we succeed in rerouting all these nets we then return to the first routing phase for the next net. Otherwise, we consider the next track in the channel until either rerouting is successful in some track or it fails in all tracks. In the latter case, the global route of the net under consideration is modified and the first routing phase of our routing algorithm is repeated.

We now consider the problem of rerouting the detailed routes that existed in track t_1. If we want to reroute all the nets in track 2 for the example in Figure 6.8 , then the answer is no, since it is impossible to reroute the detailed route of net n_6 in track 1 and track 3. So is the case when we try to reroute all the nets in track 3. However, if we try to reroute the nets in track 1, we can reroute the nets n_1, n_2, n_3, and n_4 in tracks 2 and 3 as shown in Figure 6.8(b). Therefore, track 1 can be used to route the blocked net n_7.

Formally speaking, let $\Re$ denote the set of detailed routes in track t_1 of channel c, no two of them overlap. Let U denote the set of unused wire segments in channel c. We want to know if there is a way to reroute the nets in $\Re$ using the unused wire segments in U. First of all, our algorithm labels the detailed routes in $\Re$ from left to right as l_1, l_2, ..., $l_{|\Re|}$. For each detailed route in $\Re$, say l_m, we use $L_i(m)$ to denote the leftmost unused wire segment in track i whose span overlaps with that of l_m and $R_i(m)$ to denote the rightmost unused wire segment in track i whose span overlaps with that of l_m. For example, in Figure 6.8(b), $L_2(2)$ denote the leftmost wire segment in track 2 and $R_2(3)$ denote the third leftmost wire segments in track 2. Suppose we have succeeded in rerouting the detailed routes l_1, l_2, ..., l_m, where $m < |\Re|$. When we try to reroute the detailed route l_{m+1}, we can list all tracks which are available for l_{m+1}. Let t be the number of tracks in the channel. We can use a binary string $(a_1^{m+1}, a_2^{m+1}, \ldots, a_t^{m+1})$ to represent the availability of the tracks for l_{m+1}. If l_{m+1} cannot be rerouted in track i, then $a_i^{m+1} = 1$, and $a_i^{m+1} = 0$, otherwise. We call such a binary string an *availability vector* for l_{m+1}. According to the definition of an availability vector, if the existing detailed route of another net conflicts with the rerouting of l_{m+1} in track i, then $a_i^{m+1} = 1$, no matter how l_1, l_2, ..., l_m are rerouted. Moreover, to reroute l_m in track i, there must be turning switches provided for connecting to the remaining route of the net. For the example in Figure 6.8(b), we cannot reroute net 2 in track 2, if there is no turning switch between the intersection of the middle track in one of the two vertical channels and track 2 in the horizontal channel. Therefore, a_i^{m+1} will be 1 for all such tracks. We use V_{m+1} to denote the set of tracks which are candidates for the rerouting of l_{m+1}. Given an availability vector for l_m, $(a_1^m, a_2^m, \ldots, a_t^m)$, we determine the following five sets of tracks,

$$A = \{i \mid \text{track } i \in V_{m+1},\, a_i^m = 0 \text{ and } R_i(m) \neq L_i(m+1)\},$$

$$B = \{i \mid \text{track } i \in V_{m+1},\, a_i^m = 0 \text{ and } R_i(m) = L_i(m+1)\},$$

$$C = \{i \mid \text{track } i \in V_{m+1},\, a_i^m = 1 \text{ and } L_i(m) \neq L_i(m+1)\},$$

$$D = \{i \mid \text{track } i \in V_{m+1},\, a_i^m = 1 \text{ and } L_i(m) = L_i(m+1)\},$$

$$E = \{i \mid \text{track } i \notin V_{m+1}\}.$$

We will generate $|B| + 1$ availability vectors for l_{m+1} which satisfy the constraints :

1. $a_i^{m+1} = 0$, for each i in Set A,
2. $a_i^{m+1} = 0$, for each i in Set C,
3. $a_i^{m+1} = 1$, for each i in Set D,
4. $a_i^{m+1} = 1$, for at most one i in Set B,
5. $a_i^{m+1} = 1$, for each i in Set E.

Note that there are exactly $|B| + 1$ availability vectors satisfying the five constraints above. The first constraint states that if track i is available for l_m and rerouting l_m does not prohibit l_{m+1} from being rerouted in track i, then track i is available for l_{m+1}. The second constraint states that if track i is occupied by some existing route, and this route does not block the rerouting of l_{m+1} in track i, then track i is available for l_{m+1}. The third constraint states that if some existing route uses the leftmost segment which is needed for l_{m+1}, then track i is not available for l_{m+1}. Note that we can only reroute l_m in track i for i in Sets A and B, since a_i^m = 0. If we choose track i in Set A for rerouting l_m, then all the tracks in both Set A and B are available for rerouting l_{m+1}. If we choose track i in Set B for rerouting l_m, then that track is no longer available for rerouting l_{m+1}. The fourth constraint is imposed based on this observation. The last constraint states that if track i is not in V_{m+1}, then it is impossible to reroute l_{m+1} in track i. Note that there is only one availability vector for l_1, the ith coordinate of which is 0, if track $i \in V_1$, and the ith coordinate of which is 1, otherwise. Based on the relationship between the availability vectors for l_m and l_{m+1} and the availability vector for l_1, we can compute all the availability vectors for l_i, for each i, $2 \leq i \leq |\Re|$. Note that if all the availability vectors are $(1, 1, \ldots, 1)$ for some l_i , then it is impossible to reroute $\Re$ using U. Otherwise, if there is one zero entry in any of the availability vectors for $l_{|\Re|}$, then the way to reroute each l_i, $1 \leq i \leq |\Re|$, in $\Re$ using U can be found by tracing back the computation of that availability vector.

Note that if there are two availability vectors for l_m, (a_1^m, a_2^m, ..., a_t^m) and (b_1^m, b_2^m, ..., b_t^m), such that $a_i^m \leq b_i^m$, for $1 \leq i \leq t$, then the latter is redundant, since the tracks available for l_m specified in the former include those specified in the latter. Therefore, it is unnecessary to keep the redundant availability vector. To store the availability vectors for each route in $\Re$, we use the data structure trie [1]. The number of nonredundant availability vectors is bounded by $\binom{t}{\frac{t}{2}}$, which is the number of binary sequence of length t with $\frac{t}{2}$ ones. If we did not use the data structure tries, the storage size will be $t\binom{t}{\frac{t}{2}}$. However, when a trie is used the storage is at most $2 + 4 + \ldots + 2^t = 2^{t+1} - 2$. For a typical FPGA/FPIC, where $t = 15$, the storage is at most 64K, compared with 96K, if a trie is not used.

6.2.3 Actel Row-based Channel Routing

It is interesting to note that our rip-up and reroute algorithm for two-dimensional FPGA/FPIC routing architecture can be also used as a heuristic method to solve Actel row-based channel routing problem. In Actel row-based channel, the routing problem can be formulated as follows[2]. Given a segmented channel of t tracks and a set of 2-terminal nets M, assign each net to a track such that no segment is occupied by more than one net. An exhaustive search algorithm is proposed in [43] which takes $O(|M| \cdot t^2 \cdot t!)$ time and requires $O(|M| \cdot t \cdot t!)$ memory.

First of all, we use the left edge algorithm [20] to divide the 2-terminal nets in M into d groups, where d is the density of these 2-terminal nets. Note that no two nets in the same group have overlapping spans. We then process the 2-terminal nets in one group at a time during which we use our rip-up and reroute algorithm described in Section 6.2.2 to assign these 2-terminal nets to tracks subject to the constraint that no two nets can share the same wire segment. Note that we do not rip up any existing routes here. Instead, we simply let the group of 2-terminal nets under consideration be the set $\Re$ in Section 6.2.2. If all the nets in $\Re$ can be routed using the remaining unused wire segments, then we proceed to consider the next group of 2-terminal nets. Otherwise, whenever a net whose availability vectors are all $(1, 1, \ldots, 1)$ is encountered, we skip this 2-terminal net and continue the computation of the availability vectors for the remaining 2-terminal nets. After we have considered all the nets in $\Re$, we collect all the 2-terminal nets skipped during the process and run the rip-up and reroute algorithm for these nets by ripping up the existing routes in one track at a time. If the algorithm succeeds in ripping up and rerouting the nets in a track s, then we can assign these unrouted net to track

[2] See [16, 43] for the background information.

s and proceed to consider the next group of 2-terminal nets. If all tracks fails in this step, then our algorithm stops and one more segmented track is needed for the algorithm to proceed.

6.3 Experimental Results

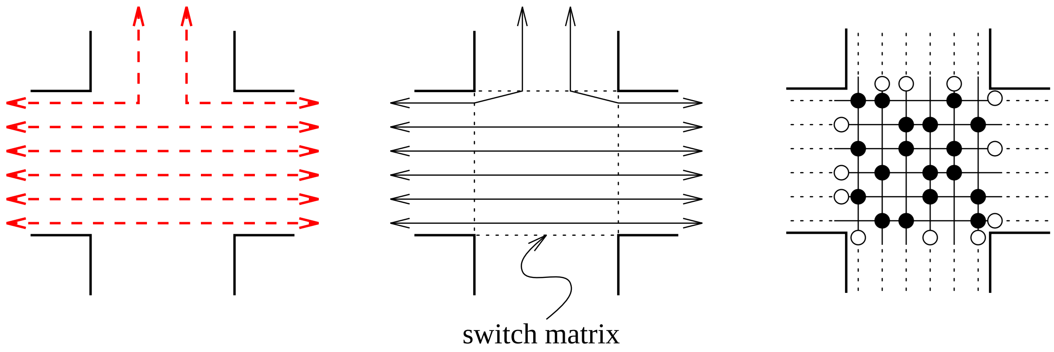
Our algorithm was implemented in the C language and executed on a SPARC-10 station. The program was used to route five industrial circuits used in [6, 50]. Each track is segmented such that the number of segments of length l is proportional to $\frac{1}{l}$, where l is a power of 2. Segments in each track are arranged in a random manner so that there is no bias in the distribution of routing resource. The first circuit, BUS_CNTL, is a bus controller. Let k be the number of turning switches allocated to the horizontal segments and vertical segments at the intersection of a horizontal channel and a vertical channel. For each of the cases in which the number of tracks in a channel is 11, 12, 13, and 14, we determined the minimum value of k such that our algorithm produces a 100% complete routing solution as shown in Table 6.1. The positions of these k turning switches are randomly generated such that routing resource is spread evenly without any bias. Figure 6.9(c) shows an example of the arrangement of turning switches, where the number of tracks in each channel is 6 and $k = 3$. BUS_CNTL was routed on an array of 13×12 blocks with 151 nets, or 392 equivalent two-terminal nets. The routing result in [6] used a total of 1616 active switches in the connecting paths. In [50], fewer than 1170 are used in all these instances. The results in [50] and [6] for BUS_CNTL are summarized in Table 6.2. In our case, we further reduce the number of switches used. Note that in [6], the channel density of the global route is used as a lower bound of the number of tracks required in their algorithm. Their algorithm achieves the lower bound in two benchmark circuits and uses at most two more tracks than the lower bound in the other three circuits. However, in our

# of tracks	min. k	# active switches	$\frac{\text{active sw. (Ours)}}{\text{active sw. [6]}}$	total # switches	$\frac{\text{total sw. (Ours)}}{\text{total sw. [50]}}$	rerouted nets	time (seconds)
11	6	1093	68%	10230	-	0	0.88
12	5	1093	68%	9576	55%	1	0.78
13	3	1115	69%	6942	45%	3	0.92
14	2	1149	71%	5628	51%	1	0.93

Table 6.1: Summary of our results for BUS_CNTL

	# of tracks	min. k	# active switches	total # switches
[50]	11	-	-	-
	12	11	1165	17424
	13	9	1151	15444
	14	6	1167	11088
[6]	10	6	1616	15840

Table 6.2: Summary of results in [50] and [5] for BUS_CNTL



(a) (b) (c)

Figure 6.9: Illustration of the loose bound

circuit name	# of tracks	min. k	# active switches	$\frac{\text{active sw.(Ours)}}{\text{active sw. [6]}}$	total # switches	$\frac{\text{total sw.(Ours)}}{\text{total sw. [50]}}$	rerouted nets	time (sec.)
DMA	13	7	2147	64%	26949	63%	2	2.03
EBNR	15	8	3303	58%	56550	69%	4	3.47
FSM	16	3	3906	64%	29744	57%	1	3.73
Z03	16	8	6092	61%	93808	60%	2	6.73

Table 6.3: Summary of our results for the other four circuits

	circuit name	# of tracks	min. k	# active switches	total # switches
[50]	DMA	13	13	2237	43095
	EBNR	15	13	3830	81900
	FSM	16	7	4193	51744
	Z03	16	15	6054	156000
[6]	DMA	10	6	3354	30600
	EBNR	12	6	5700	60480
	FSM	10	6	6132	55440
	Z03	13	6	10039	101400

Table 6.4: Summary of results in [50] and [5] for the other four circuits

routing architecture, given a global route, the channel density is usually a loose lower bound of the number of tracks required in each channel. Consider the example in Figure 6.9. Given the global route in 6.9(a), if we use the routing architecture in [6], then 6 tracks are enough as shown in 6.9(b). However, for our routing architecture as in (c), 7 tracks are required since each net needs a horizontal wire segment in order to pass through the intersection of the horizontal channel and the vertical channel. Based on this observation, we analyzed the same given global route used in [50] and found that the minimum number of tracks required is at least 11, which was achieved using our algorithm in our architecture. The number of tracks used in [6] is 10, one more than the channel density. The number of tracks for the architecture in [50] is at least 12. It should be emphasized that the minimum value of k for achieving a 100% complete routing solution decreases dramatically in our architecture. This is due to the increase in routability by the introduction of the continuing switches. The total numbers of ripped up and rerouted nets are only 0, 1, 3, and 1, respectively. Therefore, most of the routing is done by our router in Section 6.2.1 and only in very rare cases the rip-up and reroute algorithm is used to resolve the conflicts.

For the other four circuits, we used the same number of tracks as [50] and found the corresponding minimum value of k in order to achieve a 100% complete routing solution. Compared with the results of [50], the dramatic reduction in the values of k demonstrates again that our architecture has much better routability than that proposed in [50]. Also, the number of active switches in the routing solution is substantially smaller than that in [6]. The algorithm is so efficient that it only took less than 7 seconds for each circuit. The results are summarized in Table 6.3 and 6.4.

6.4 Summary

We have proposed a new two-dimensional FPGA/FPIC routing architecture which has better routability and uses fewer active programmable switches than the architecture proposed in [50]. An effective routing algorithm is developed based on the characteristics of the routing architecture. The rip-up and reroute phase of this algorithm can also be used to solve Actel row-based channel routing problem. The experimental results show a great superior performance of both the routing architecture and the routing algorithm.

Chapter 7

Congestion-Balanced Placement for FPGAs

7.1 Introduction

Using programmable switches to provide field programmability, Field-Programmable Gate Arrays (FPGAs) have increased in popularity dramatically in recent years. This advantage is achieved at the expense of a lower gate utilization. One major reason for the degradation in gate utilization is limited availability of routing resources, such as fixed number of tracks in a routing channel and limitations on available interconnections among tracks, in commercial FPGA chips. It is, therefore, important that FPGA physical design tools use strategies to make sure that the resulting layout is always *routable*, that is, there exists a feasible detailed routing under given limitations on routing resources. We illustrate the problem by first giving a brief overview of the physical design for FPGAs next.

In recent years, several layout synthesis strategies that generate a final layout for a circuit description given as a Boolean network have been devised. Most common approach is to divide this task into three steps [5, 53] as shown in Figure 7.1. First, the Boolean network is partitioned

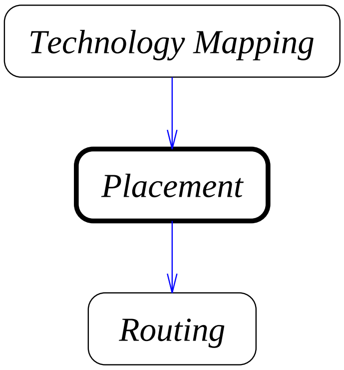


Figure 7.1: The process of FPGA layout synthesis

into simple single-output functions which are mapped to the logic blocks of the FPGA device, subject to the constraints imposed by the architecture of the logic blocks, such as the number of input, the types, and the number of functions that each logic block can implement. This stage is called *technology mapping* and typically consists of solving the subproblems of network decomposition, covering and merging [34, 44]. Technology mapping is followed by the *placement* step which assigns each node in the mapped Boolean network to a specific logic block in the FPGA device. Finally, the detailed interconnections between logic blocks are created in the *routing* step.

This work concerns the problem of block placement such that the resulting solution leads to a feasible final layout. In most CAD tools for FPGAs the placement stage has traditionally employed algorithms originally developed for other technologies such as printed circuit boards and custom integrated circuits. Most placement tools use a min-cut algorithm [4, 29, 52] that employs a recursive top-down bi-partitioning procedure to minimize the number of interconnections crossing the cut in each recursive step [11, 26]. This algorithm runs fast and is easy to implement. The objective of a min-cut based placement algorithm is to optimize the placement by putting communicating blocks close together. Therefore, the total wire length is the dominant component of the cost function in traditional placement algorithms.

However, total wire length alone is not a good metric for architectures with limited routing resource such as FPGA and Complex Programmable Logic Device (CPLD) with partially connected *interconnect matrix*. Since the algorithm tries to place connected blocks close together, it is likely to generate a placement with congested areas where a feasible routing is difficult to find, if not impossible at all.

We propose an improved min-cut based placement algorithm which spreads out the congestion uniformly across the channels, while minimizing the total wire length. Therefore, the placement result maximizes the possibility of finding a feasible routing for FPGA implementations. Further, most traditional placement algorithms use an approximation scheme to estimate the number of interconnections crossing a cut line. An example is to use the bounding box for a multi-terminal net to estimate the probability that the route of the net will cross a cut-line. In contrast, our proposed algorithm minimizes the *actual* number of interconnections crossing a cut-line, as will be explained in Section 7.3.2. The proposed algorithm also obviates the need for a terminal propagation step [10] by using a novel procedure to implement the cut lines. Based on the results of our placement and routing experiments, we also propose a sequential order according to which the cut-lines are applied. Such a sequential order significantly reduces the possibility of net congestion near the center of the chip (as it often happens in the traditional top-down min-cut based placement). Finally, the proposed algorithm treats multi-terminal nets directly without dividing them into equivalent two-terminal nets.

The rest of this chapter is organized as follows: Section 7.2 defines the FPGA layout model and the corresponding placement problem; Section 7.3 describes the proposed algorithm; Section 7.4 demonstrates experimental results compared with the traditional algorithm. Section 7.5 concludes the chapter by noting important observations from our experimental results.

7.2 FPGA Placement Problem

The two-dimensional FPGA layout model proposed in [6, 38, 59] contains an array of logic blocks as shown in Figure 7.2. Each logic block represented by a square implements logic functions.

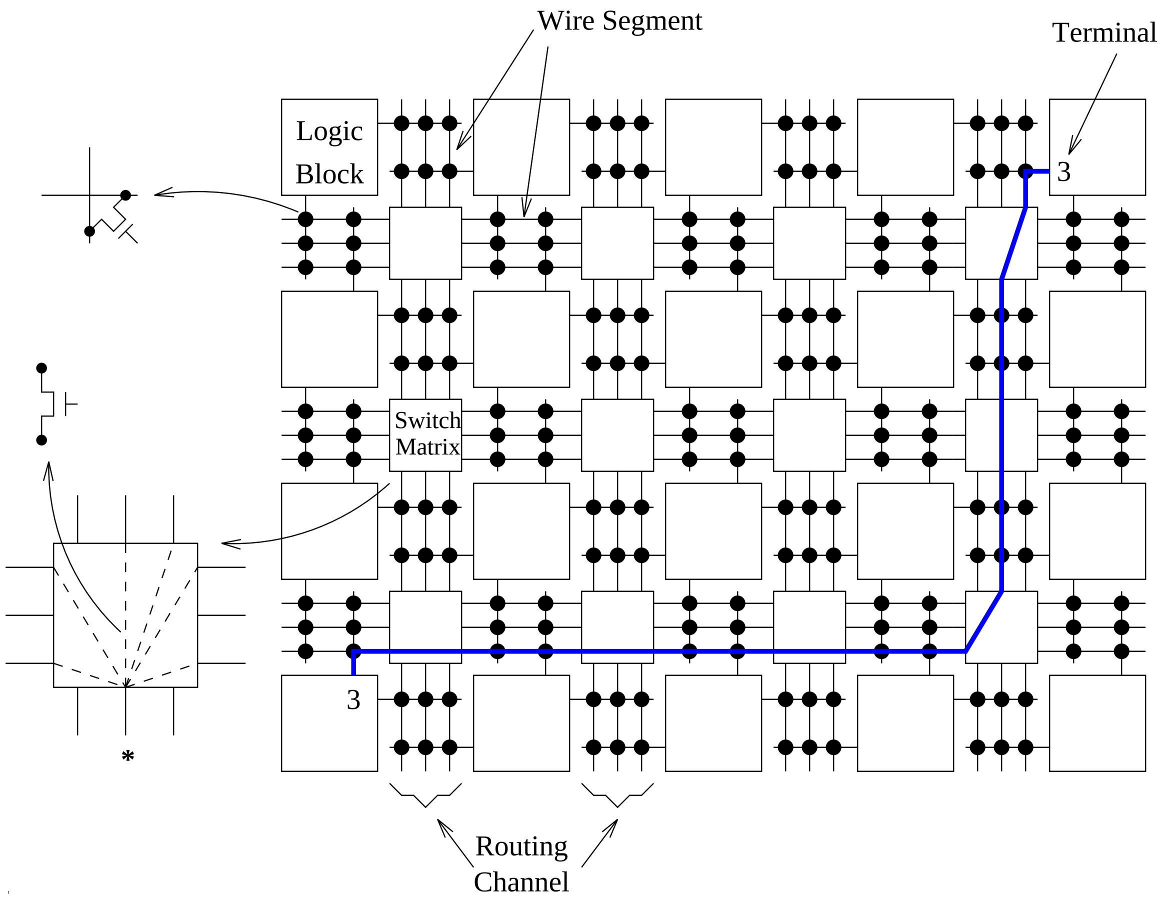


Figure 7.2: Two-dimensional FPGA layout model

The terminals of a block are located on the boundaries of the block and are connected to wire segments called *terminal segments.* Terminal segments are connected to wire segments in the routing channel through switches at their intersections, which are shown as black circles in Figure 7.2. The switch matrix makes connections between wire segments in horizontal and vertical channels. In the switch matrix, each wire segment can be connected to a subset of the wire segments on the other sides of the matrix. Suppose there is a two-terminal net connecting two logic blocks, one at the lower left corner and the other at the upper right corner. A routing example of this two-terminal net is shown in bold lines in Figure 7.2. Logic blocks on the periphery of the FPGA layout model are connected to I/O blocks (not shown in the figure) which serve as primary inputs and primary outputs.

In commercial FPGA products, routing resource is fixed and fairly limited. For instance, there are only five tracks in each routing channel for Xilinx's XC3000 series of FPGAs. The placement problem is especially important in designs using such devices, because a placement is not routable if the number of nets in a channel exceeds the channel capacity. In order to develop an effective placement procedure a metric is needed to measure channel congestion. One way of measuring net congestion is the *cut size*, that is, the number of nets crossing a cut-line in the chip. There are a horizontal cut-lines and b vertical cut-lines in an FPGA with $(a+1)\times(b+1)$ logic blocks, with each cut-line corresponding to a routing channel. We define the cut size of a cut-line to be the number of nets that have terminals on both sides of the cut-line. The cut size is a lower bound on the number of tracks needed for a complete routing solution. Thus, if the cut size of a horizontal (vertical) cut-line is larger than the number of vertical (horizontal) tracks, then it is impossible to have a routable design. Moreover, a cut-line cutting through a connection-congested area usually has a large cut size. If the maximum cut size, the cut size that is the largest among all horizontal cut-lines and vertical cut-lines, decreases, the opportunity of the occurrence of connection-congested area decreases accordingly. Therefore, to avoid the occurrence of connection-congested area, we desire to minimize the maximum cut size. Note that, as was indicated in [30], the sum of the cut sizes over all a horizontal cut-lines and b vertical cut-lines in an $(a+1)\times(b+1)$ layout area gives the exact total wire length when the wire length of a multi-terminal net is estimated by half the perimeter of the bounding box. Thus, there is a close correlation between cut sizes and total wire lengths. In fact, the experimental results in Section 7.4 show that minimizing the maximum cut size produces a placement with a smaller total wire length in most cases, compared with the traditional min-cut based placement algorithm. Also note that the placement problem of minimizing the maximum cut size is NP-complete, since the linear arrangement problem [17] can be reduced to the placement problem when there is only one row or one column of logic blocks.

7.3 Congestion-Balanced Placement

Placement algorithms based on hierarchical min-cut partitioning [30, 40] have been studied and used extensively in both the academia and the industry. Given a layout region and a circuit represented by a graph (or hyper-graph), each node representing a logic block and each edge representing a net, the algorithm recursively bi-partitions both the circuit and the region until the graph is simple enough (for example, a graph with one node) to be placed in the region. The objective of each application of the bi-partitioning step is to minimize the cut size subject to the constraint that the sizes (the number of the nodes) of the two resultant portions are roughly equal. This approach has the advantages of producing placements of good quality, a short running time, and easy implementation. However, the bi-partitioning algorithms based on the Kernighan-Lin algorithm [11, 26] suffers from the lack of control on wire congestion because cut size is the only metric in the cost function to be minimized. Consequently, it is possible to obtain a partition with a small cut size, with one portion being heavily connected and the other being very sparse. In other words, although these two portions have approximately the same number of nodes, the numbers of connections within them can differ significantly. Therefore, such a placement would likely contain unroutable channels. Clearly for FPGA applications, min-cut based placement algorithms must be modified to take into account not only the sizes of the two portions, the size of the network crossing the cut-line but also the distribution of interconnections within the two portions. Figure 7.3 shows an example to illustrate the need of

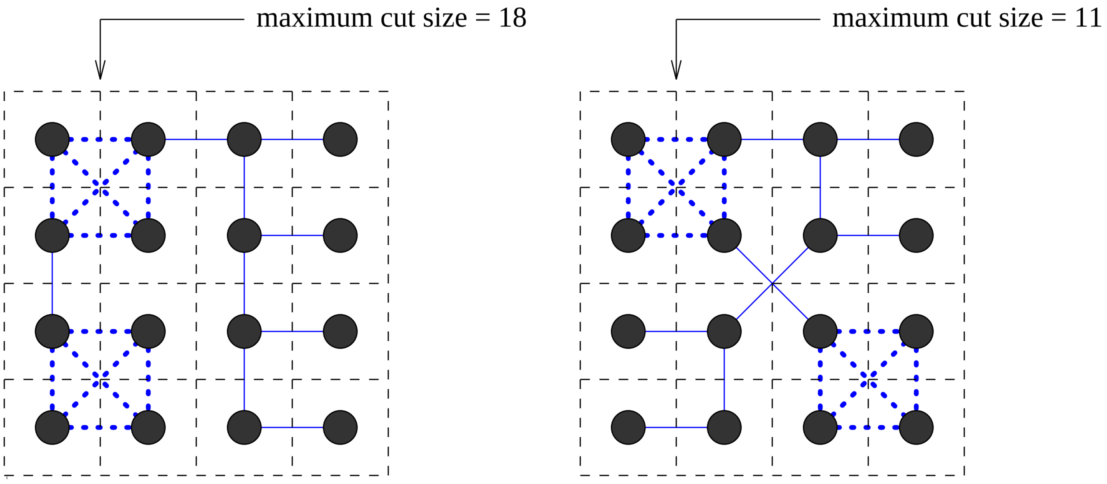


Figure 7.3: The shortcoming of the traditional min-cut based placement algorithm

such modification. A placement obtained by the traditional min-cut based placement algorithm is shown in Figure 7.3(a). There are sixteen nodes in the network to be placed in a 4×4 array. Note that each of the two groups of four nodes connected by six bold dotted edges represent a completely connected 4-node clique. That is, if we label these four nodes by a, b, c, and d, then there are eleven hyper-edges connecting them which are $\{a, b\}$, $\{b, c\}$, $\{c, d\}$, $\{a, d\}$, $\{a, c\}$,

$\{b,d\}$, $\{a,b,c\}$, $\{a,b,d\}$, $\{b,c,d\}$, $\{a,c,d\}$, $\{a,b,c,d\}$. The traditional min-cut based placement algorithm first divides the network into two portions of equal size, one placed in the left half and the other placed in the right half of the array. Since minimizing the cut size is the only objective in the bi-partitioning process, the first bi-partition obtained is shown in Figure 7.3(a), where the cut size is one. Then, the two portions obtained in the first bi-partitioning step are bi-partitioned recursively and the placement obtained is that shown in Figure 7.3(a). The maximum cut size of this placement is eighteen. However, if we consider the distribution of the interconnections, a better placement with the maximum cut size being eleven can be obtained as shown in Figure 7.3(b). In this section, we propose a modified min-cut bi-partitioning algorithm that not only balances the size of the two portions, but also evenly distributes the connections among them.

7.3.1 Congestion-balanced Bi-partitioning

In order to qualify the effect of congestion unbalance between the two portions of a given bi-partition, we define the following terms. Multiple terminal nets are represented by a hypergraph model. For instance, consider a seven-terminal net shown in Figure 7.4. The cut-line currently used is shown as the dotted line. Suppose the bi-partitioning result is such that there are three terminals in the left portion and four terminals in the right portion. This net will contribute a count of one in the cut set. Moreover, we know that two connecting paths are needed to connect those three terminals in the left portion and three connecting paths are needed in the right portion. In general, $max\{k-1,0\}$ connecting paths are needed to connect k terminals. We define the *unbalancing number* of a net to be the number of connecting paths needed to connect all the terminals in the left portion minus the number of connecting paths

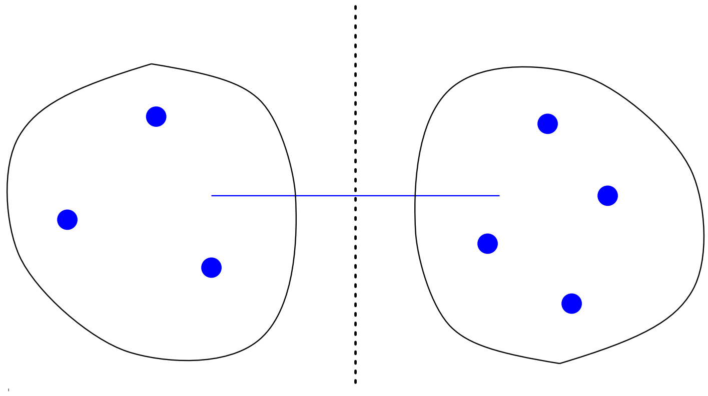


Figure 7.4: A seven-terminal net

needed in the right portion. The unbalancing number of a bi-partition is defined to be the sum of the unbalancing numbers of all nets. *The absolute value of the unbalancing number of a bi-partition counts the difference between the numbers of connecting paths needed in the left portion and the right portion.* Therefore, it is a good measure of the unbalancing situation in a bi-partition.

Given an initial bi-partition, we can compute its unbalancing number in $O(|T|)$ time by examining all the nets, where T is the set of all terminals. Without loss of generality, we assume that there are more interconnecting paths in the left portion than in the right portion, that is, the unbalancing number of this bi-partition is positive. If node v is moved from the left portion to the right portion, then $f(v, e)$, the amount by which the unbalancing number of the net e decreases, is equal to

1 : if either the number of terminals belonging to net e in the left portion is equal to 1 (shown in Figure 7.5 (a)) or there is no terminal of net e in the right portion (shown in Figure 7.5 (b));

2 : otherwise (shown in Figure 7.6);

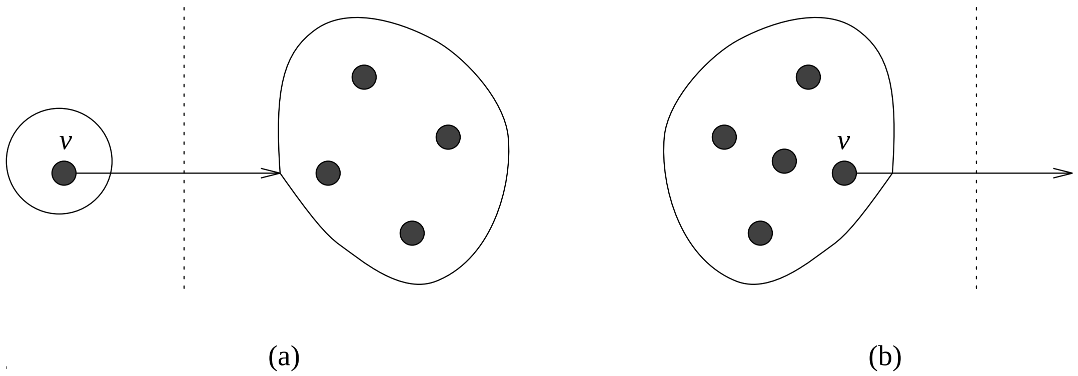


Figure 7.5: Case 1 in updating $f(v, e)$

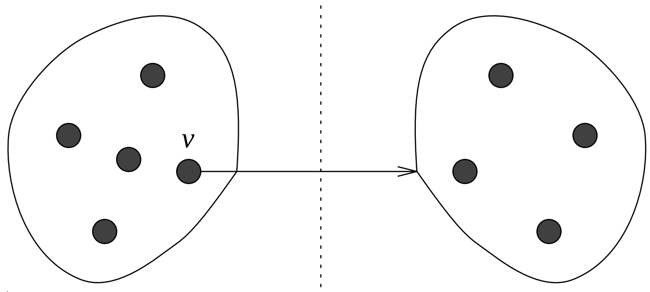


Figure 7.6: Case 2 in updating $f(v, e)$

and the gain in reducing the unbalancing number of the bi-partition is defined as

$$F(v) = \sum_{\forall e, v \in e} f(v, e). \tag{7.1}$$

If node v is moved from one portion to the other portion, it takes only $O(|N(v)|)$ time to update the values of the gain function that are changed by examining $N(v)$, the set of all neighbors of v. When a node v in one portion is swapped with another node u in the other portion, if there is no edge connecting these two nodes, the gain in reducing the unbalancing number of the bi-partition is the sum of $F(v)$ and $F(u)$. If there are edges connecting both nodes, for each of such edges, e, we subtract the amount $f(u, e) + f(v, e)$ from the sum of $F(v)$ and $F(u)$, since the swapping, in fact, contributes no reduction in the unbalancing number of the net e.

The following cost function is used to incorporate the effect of congestion distribution in a given partition

$$\text{CUT\_SIZE} + \text{WEIGHT} \times \text{UNBALANCING NUMBER}$$

where WEIGHT is a constant. If WEIGHT is set to zero, then the algorithm is the same as the conventional min-cut bi-partitioning algorithm. By setting the value of WEIGHT appropriately, we can control the importance of balancing the congestion. For the example in Figure 7.3, if WEIGHT is set to be greater than $\frac{1}{8}$, then the better placement in (b) is obtained. Note that a linear combination of cut size and unbalancing number is chosen to allow incremental computation of the cost function as new bi-partitioning solutions are constructed.

It should be noted that a new definition of the unbalancing number of a bi-partition is needed if the ratio of the sizes of the two resultant portions is required to be r, where r is not equal to one. Suppose we require that the left portion have x nodes and the right portion have y nodes such that $\frac{x}{y} = r$. Given a bi-partition and a net e, if the number of connecting paths needed to connect all the terminals of net e in the left (right) portion is a (b), then the *normalized* unbalancing number of the net e is defined to be

$$a \times \frac{\frac{x+y}{2}}{x} - b \times \frac{\frac{x+y}{2}}{y}$$

That is, we normalize the number of connecting paths needed to connect all the terminals in the left (right) portion by multiplying the ratio of half of the number of nodes to the number of nodes in the left (right) portion. The unbalancing number of the net is defined to be the resultant value for the left portion minus the resultant value for the right portion.

7.3.2 Slicing Line

In conventional min-cut based placement, the layout region is bi-partitioned in a top-down fashion in accordance with a recursive application of the min-cut bi-partitioning. A frequently used strategy is to bi-partition the region into two halves with equal size in alternating directions as shown in Figure 7.7 [30], where the cut-lines depict the sequence in which they are applied. In this figure, dashed cut-lines are vertical cut-lines and solid cut-lines are horizontal cut-lines. Such a hierarchical bi-partitioning process suffers from the problem of the external pin connections [10] as will be explained next. Suppose the layout region is first partitioned

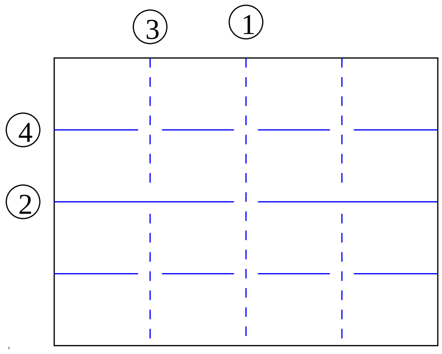


Figure 7.7: Traditional sequence of applying cut-lines

vertically into two equal parts such that each part will accommodate one portion of the bi-partitioned circuit. Then, for these two regions, the bi-partitioning process is repeated, except that horizontal cut-lines are now applied. Suppose that bi-partitioning is first carried out for the right region. The bi-partitioning result for the right region should be taken into account during the bi-partitioning process for the left region. The reason is that the signals entering the logic blocks in the left regions from the logic blocks in the right region might effect where each logic block in the left region should be placed just as much as the internal connections among the logic blocks in the left region. In [10], a technique called *terminal propagation*

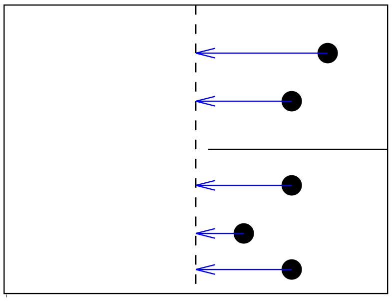
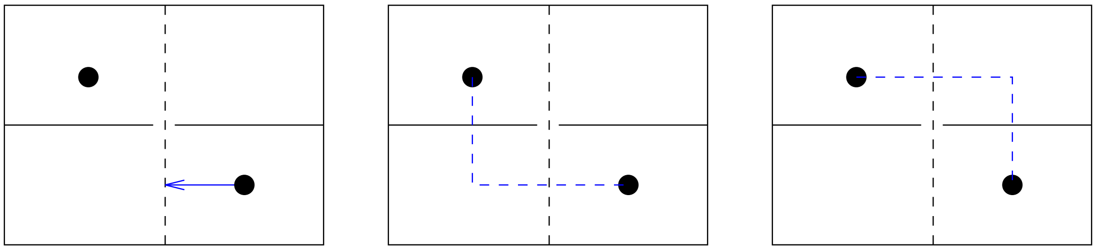


Figure 7.8: Terminal propagation

was proposed. In this approach, all nodes to the right of the vertical cut-line are projected horizontally on the cut-line, as shown in Figure 7.8. We then process the left region, regarding these projected nodes as fixed nodes[1]. This terminal propagation technique has the advantage of taking into account the bi-partitioning result in the right region. However, it suffers from two disadvantages. First, the left and right regions are processed in sequence. There is no clear reason as to which region should be processed first. Secondly, the net in the cut set computed using the terminal propagation technique does not necessarily cross the cut-line, as will be explained below. Consider a two-terminal net with one terminal connected to a logic

[1] Fixed nodes are those that are not allowed to be moved in the bi-partitioning process.

block placed in the lower right region shown in Figure 7.9(a). Assume that the other terminal is connected to a logic block placed in the upper left region. After the process of the right



(a) (b) (c)

Figure 7.9: Two possible routes in (b) and (c) for the two-terminal net in (a)

region and the terminal projection, this net contributes a count of one in the cut size for the left horizontal cut-line. However, there are two possible routes for this net and it does not cross the cut-line if the routing in Figure 7.9(c) is chosen. Therefore, counting this net in the cut set for the left horizontal cut-line is not accurate. One common way of solving this problem is to assume even probabilities for all cut-lines that overlap with the bounding box of the net [18]. That is, in our example of Figure 7.9, the probability for the net to cross each horizontal cut-line is one half. Therefore, one half of a net is contributed to the cut size of each cut-line. We solve this problem by considering only the cut-lines that span the width or the height of the layout region. To distinguish the cut-lines that cut through the entire layout area from the cut-lines used in the traditional min-cut placement algorithm, they are referred to as *slicing lines*. After the first vertical slicing line is applied, we apply one horizontal slicing line, instead of two half horizontal cut-lines in sequence. During the bi-partitioning process, we consider movement of nodes in both the left and the right regions at the same time. We honor the bi-partitioning result for the first vertical slicing line by prohibiting any movement of nodes across the vertical slicing line. It is not hard to see that the cut set obtained in this way constitutes exactly those nets that have terminals in both sides of the slicing line.

7.3.3 Slicing sequence

Based on the observation mentioned previously, slicing lines which cut through the entire layout region are used in our algorithm. There are a horizontal slicing lines and b vertical slicing lines in an FPGA with $(a+1)\times(b+1)$ logic blocks. The sequence in which slicing lines are applied plays an important role. The conventional hierarchical min-cut placement always chooses the cut-line at the center of the region and such a choice is carried out recursively. If we order the slicing lines to be applied in this sequence, then Figure 7.10 (a)-(d) depict the sequence of slicing lines to be applied, where slicing lines currently applied are shown as dashed lines and slicing lines applied earlier are shown as solid lines. Assume that the current slicing line

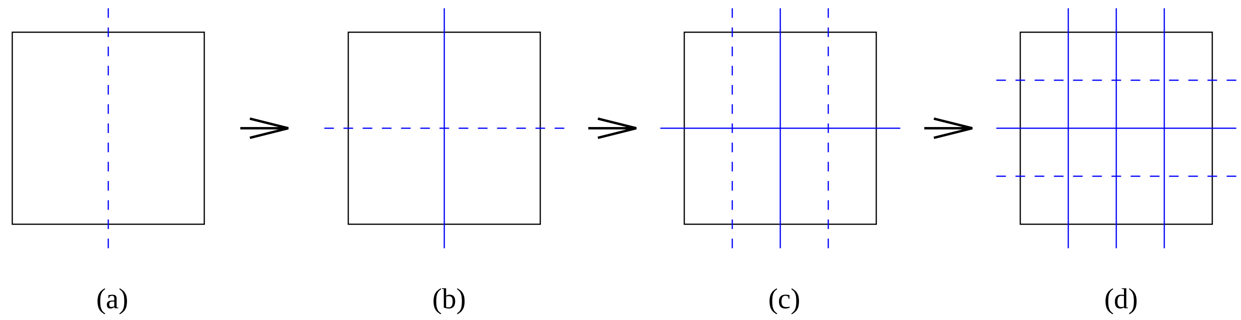


Figure 7.10: Choosing the slicing line at the center

l shown as the dashed line is immediately next to the center slicing line in Figure 7.11, where the remaining solid lines are those slicing lines that are applied earlier. Slicing line l will cut four regions and their corresponding sub-networks into halves. Note that in each move of this min-cut bi-partitioning, we can swap a pair of nodes such that one is in A and the other is in E, one is in B and the other is in F, one is in C and the other is in G, or one is in D and the

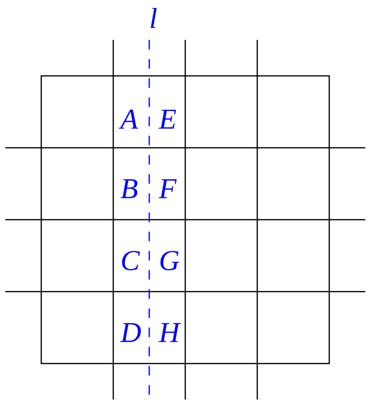


Figure 7.11: A scenario in the traditional min-cut placement

other is in H. However, since we honor the bi-partitioning results for the slicing lines applied earlier, any swapping of nodes that crosses the slicing line applied earlier is not allowed. We know that the number of nodes that reside in a region is proportional to the area of the region. Since the slicing line l is immediately next to the center slicing line, the four regions that l cutting through are fairly small. Therefore, the numbers of nodes that are placed in regions A-H are small. Consequently, The number of possible pairs that we can choose from for a move in this bi-partitioning process is, thus, limited. This usually results in a relatively large cut-size for the slicing lines close to the center due to a small number of possible moves. Especially, for the two horizontal and vertical slicing lines immediately next to the center, a larger cut size is usually observed.

Based on this observation, we propose another sequence of slicing lines in which they are applied to reduce the chance of congestion near the center. The horizontal slicing lines and vertical slicing lines are applied alternatively as before. However, for the slicing lines of the same orientation, those slicing lines that are closer to the center are applied earlier as shown in Figure 7.12, where horizontal slicing lines are not shown for clarity. We compare the vertical

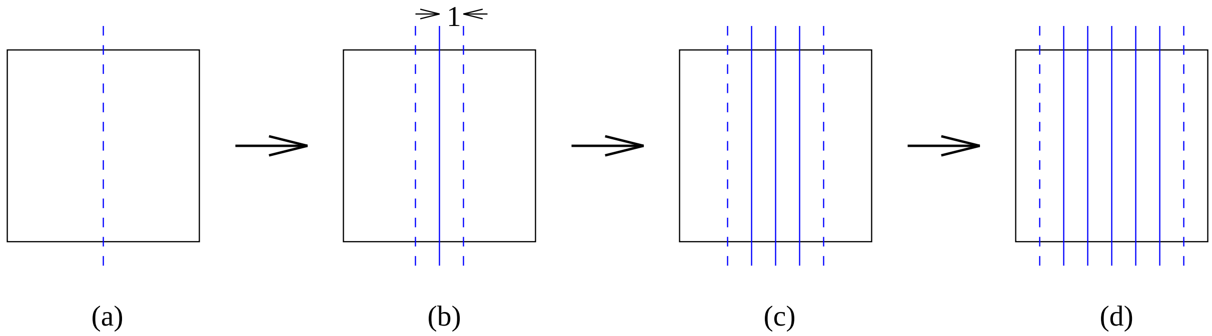


Figure 7.12: Proposed sequence to apply slicing lines

slicing lines, denoted by l, which are immediately next to the center in Figure 7.13 (a) and (b). Since the size of the area of a region is proportional to the number of nodes that reside in the region, we can compute the ratio of the number of possible pairs that we can choose from for a move in (a) to that in (b). By computing the areas of the regions, we can obtain the ratio as $2{\cdot}(9 \times 3 + 3 \times 1) : 4{\cdot}(2 \times 2) = 60 : 16$. There is a substantially larger number of possible pairs of nodes that we can choose from for a move in (a) than in (b). Therefore, it is expected that the cut size for a slicing line close to the center is smaller, if the sequence in (a) is adopted, instead of that in (b).

7.4 Experimental Results

Our congestion-balanced min-cut placement algorithm was implemented in the C language and run on a SPARC-10 Sun workstation. We used eight MCNC circuits to test the efficiency of our algorithm and compare the results with that obtained by the traditional min-cut algorithm when congestion balance and the other features proposed in this chapter were not taken into consideration. We used Xilinx 3000 series chips as the target FPGAs. The circuits were first synthesized and transformed into the XNF (Xilinx Net-list Format) format. Then, the

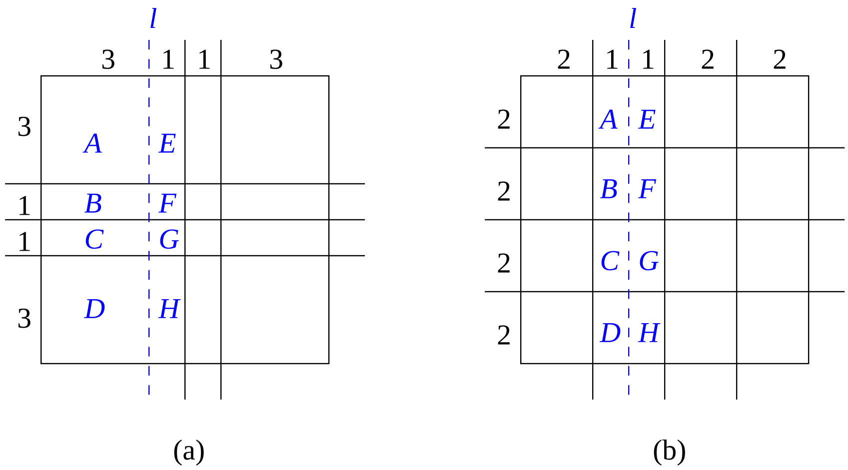


Figure 7.13: (a) The sequence proposed (b) The traditional sequence

circuits were mapped into the 3000-series logic blocks and the results were used as inputs to the placement program. The circuits are listed in the order of increasing number of logic blocks in Table 7.1. The first column of Table 7.1 shows the dimensions of the FPGA chip on which a circuit is implemented. The second, third, and fourth column shows the number of blocks, nets, and terminals in each circuit, respectively. Note that the number of logic blocks shown in the second column is the total number of I/O blocks and logic blocks. The number of logic blocks is shown in parentheses. The results are summarized in Table 7.2. We observe that our algorithm yields a reduction in the maximum cut-size by up to 23%, when compared with the traditional min-cut placement algorithm. We also computed the sum of the cut sizes over all slicing lines. As was mentioned earlier, the sum gives the total wire length according to the estimate of wire length by half of the perimeter of the bounding box. In six of the eight circuits, we obtained a placement with a smaller sum of the cut sizes. For the other two circuits, the sum of the cut sizes in the placement we obtained is roughly the same as that in the placement obtained by the traditional min-cut algorithm. Therefore, we can conclude that reduction in the maximum cut

Circuit Name	Chip Dim.	# Blocks	# Nets	# Terminals
f51m	8×8	58(42)	50	224
misex2	8×8	101(40)	83	309
comp	10×10	101(66)	98	340
9sym	10×10	105(95)	104	518
c499	10×10	139(66)	107	427
term1	12×12	160(116)	150	621
c880	12×12	209(123)	183	677
alu2	16×14	229(213)	223	833

Table 7.1: Summary of the eight MCNC circuits

Circuit	Max. Cut-size			Total Cut-size		CPU Time (seconds)	
Name	Our Alg.	Min-cut	Reduc.	Our Alg.	Min-cut	Our Alg.	Min-cut
f51m	19	24	20.8%	198	234	0.76	0.73
misex2	24	31	22.6%	284	313	2.63	1.72
comp	31	32	3.1%	377	372	1.25	1.48
9sym	23	27	14.8%	370	352	16.4	18.6
c499	40	43	7.0%	540	543	2.38	2.42
term1	38	47	19.1%	692	730	24.2	14.2
c880	48	55	12.7%	803	860	12.5	14.6
alu2	48	59	18.6%	1167	1176	311.6	213.3

Table 7.2: Summary of the Placement Results

size in our algorithm also gives the benefit of reducing the total wire length. The running time for our algorithm is comparable with that for the traditional min-cut placement algorithm.

For the largest circuit, *alu2*, there are 16×14 logic blocks in the array, and thus, 13 vertical and 15 horizontal slicing lines. For the smallest circuit, *f51m*, there are 8×8 logic blocks in the array, and thus, 7 vertical and 7 horizontal slicing lines. Figure 7.14 (a) and (b) show the distribution of the cut sizes of the slicing lines in *alu2*. Figure 7.14 (c) and (d) show the distribution of the cut sizes of the slicing lines in *f51m*. The distribution of the cut sizes in the placement obtained by our algorithm is shown as a solid curve and the distribution of the cut sizes in the placement obtained by the traditional min-cut algorithm is shown as a

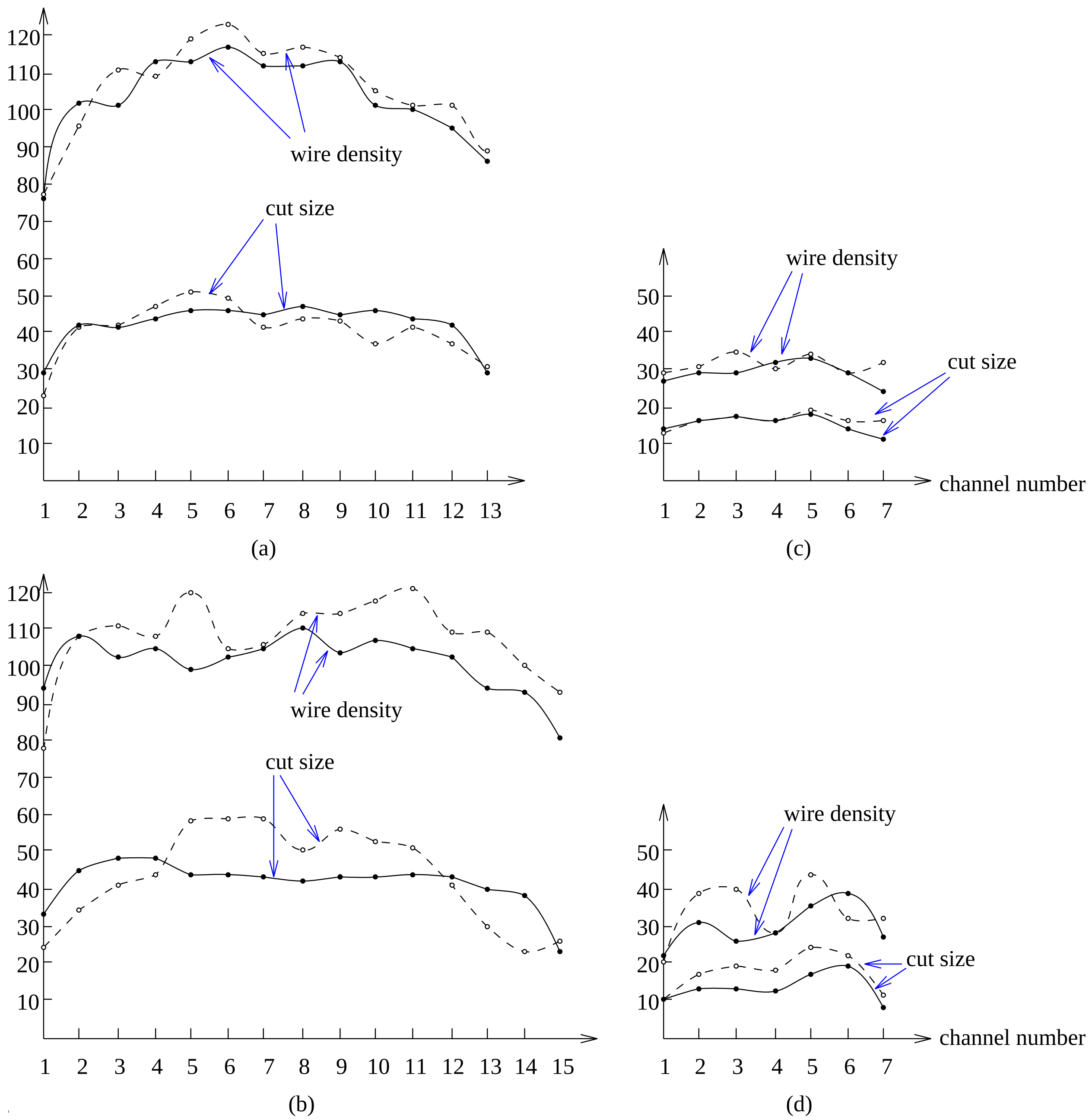


Figure 7.14: (a) The distribution of the cut sizes of the vertical slicing lines in *alu2*(b) The distribution of the cut sizes of the horizontal slicing lines in *alu2* (c) The distribution of the cut sizes of the vertical slicing lines in *f51m* (d) The distribution of the cut sizes of the horizontal slicing lines in *f51m*

Circuit Name	Our Algorithm		Traditional Min-cut	
	Average Ratio	Standard Deviation	Average Ratio	Standard Deviation
f51m	2.17	0.30	2.00	0.20
misex2	2.58	0.45	2.71	0.51
comp	1.81	0.28	1.80	0.39
9sym	2.88	0.29	2.81	0.40
c499	2.62	0.30	2.25	0.27
term1	2.17	0.36	2.31	0.32
c880	1.95	0.25	1.84	0.18
alu2	2.46	0.22	2.67	0.56

Table 7.3: Summary of the average ratio and its standard deviation

dashed curve. In addition to a reduction in the maximum cut size, we also observe that the cut sizes in our placement are distributed more uniformly than those in the placement obtained by the traditional min-cut algorithm. Also, the large cut sizes next to the central horizontal slicing lines illustrate the need of a new slicing sequence in the traditional min-cut placement algorithm. The slicing sequence adopted in our algorithm solves this problem and the values of cut size near the center of the chip are not particularly large. We used the Xilinx Automated Placement and Routing (APR) program for XC3000 series to obtain the routing results. In each routing experiment, we count the number of wire segments passing through each channel and refer it as the *wire density* of the channel. According to Figure 7.14, there is a almost-linear relationship between the cut size and the wire density. In fact, the ratio between wire density and cut size is approximately uniform in our experiments. Table 7.3 lists the average ratios of wire density to cut size over all slicing lines in each circuit and the values of the corresponding standard deviation. The standard deviation is at most between 10%~20%. Consequently, we can conclude that cut size is a good measure of wire congestion. The curves for the wire density in Figure 7.14 also show that the placements obtained by the traditional min-cut algorithm have higher wire congestion than those obtained by our algorithm, which is consistent with the fact that there are 19 unrouted nets in the placement of *alu2* obtained by the traditional min-cut algorithm, whereas there are only 3 unrouted nets in the placement obtained by our algorithm.

7.5 Summary

In this chapter, we propose an important criterion in the FPGA placement problem, balancing the congestion, which used to be overlooked, if not neglected, in the past. We propose an improved min-cut based placement algorithm which spreads out the congestion, while minimizing the total wire length. Therefore, the placement obtained maximizes the possibility of finding a complete routing in FPGA, if one exists. The algorithm balances the numbers of interconnections in the two portions during each application of the min-cut bi-partitioning algorithm. In the meantime, the algorithm minimizes the actual number of interconnections crossing each vertical slicing line and each horizontal slicing line, instead of the probabilistically estimated number of interconnections. We introduce the concept of slicing lines which span the entire chip and are useful for the bi-partitioning process where a larger solution space can be examined, compared with the terminal propagation technique. The sequence of the slicing lines is determined according to the distance from each slicing line to the center of the chip. The slicing line closer to the center is applied earlier. This avoids the possibility of the occurrence of a slicing line with a large cut size close to the center of the layout area (which happens in the traditional top-down min-cut based placement). The experimental results show a reduction of up to 23% in the maximum cut-size and the wire congestion, compared with the traditional min-cut placement algorithm. In the largest circuit, 19 unrouted nets in the traditional min-cut placement are eliminated with only 3 unrouted nets left in our placement which can be manually completed easily.

Bibliography

[1] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[2] Aptix Inc. *FPIC AX1024D, Preliminary Data Sheet*, August 1992.

[3] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximation. In *ACM Symposium on Theory of Computing*, 1993.

[4] M. A. Breuer. Min-cut placement. *Journal of Design Automation and Fault Tolerant Computing*, 1(4):343–362, October 1977.

[5] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays.* Kluwer Academic Publishers, 1992.

[6] S. Brown, J. Rose, and Z. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Transactions on Computer-Aided Design*, 11(5):620–628, May 1992.

[7] H. W. Carter and M. A. Breuer. Efficient single-layer routing along a line of points. *IEEE Transactions on Computer-Aided Design*, 2(4):259–266, October 1983.

[8] K. Chong and S. Sahni. Optimal realizations of floorplans. Technical Report TR 90-20, Computer Science Department, University of Minnesota, April 1990.

[9] S. K. Dong, J. Cong, and C.L. Liu. Constraint floorplan design for flexible blocks. In *International Conference on Computer-Aided Design*, pages 488–491, November 1989.

[10] A. E. Dunlop and B. W. Kernighan. A procedure for placement of standard-cell VLSI circuits. *IEEE Transactions on Computer-Aided Design*, 4(1):92–98, January 1985.

[11] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation Conference*, pages 175–181, 1982.

[12] J. Frankle. Iterative and adaptive slack allocation for performance-driven layout and FPGA routing. In *Design Automation Conference*, pages 536–542, 1992.

[13] Fujitsu Laboratory Ltd. *Layout method of transistors*, January 1992. Japan patent application number 04-00163.

[14] A. El Gamal. Two-dimensional stochastic model for interconnections in master slice integrated circuits. *IEEE Transactions on Circuits and Systems*, 28(2):127–138, February 1981.

[15] A. El Gamal, J. Greene, J. Reyneri, E. Rogoyski, K. A. El Ayat, and A. Mohsen. An architecture for electrically configurable gate arrays. *IEEE Journal of Solid-State Circuits*, 24(2):394–398, April 1989.

[16] A. El Gamal, J. Greene, and V. Roychowdhury. Segmented channel routing is nearly as efficient as channel routing (and just as hard). In *the 13th Conference on Advanced Research of VLSI*, pages 192–211, 1991.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[18] S. Goto and T. Matsuda. Partitioning, assignment and placement. In T. Ohtsuki, editor, *Layout Design and Verification*, chapter 2, pages 55–97. Elsevier Science Publishers B. V., North-Holland, 1986.

[19] R. Guo, H. Nguyen, A. Srinivasan, H. Verheyen, H. Cai, S. Law, and A. Mohsen. A 1024 pin universal interconnect array with routing architecture. In *IEEE Custom Integrated Circuits Conference*, pages 4.5.1–4.5.4, 1992.

[20] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of 8th IEEE Design Automation Workshop*, 1971.

[21] H. Hsieh, K. Dong, J. Ja, R. Kanazawa, L. Ngo, L. Tinkey, W. Carter, and R. Freeman. A 9000-gate user-programmable gate array. In *IEEE Custom Integrated Circuits Conference*, pages 15.3.1–15.3.7, 1988.

[22] C. P. Hsu, R. A. Perry, S. C. Evans, J. Tang, and J. Y. Liu. Automatic layout of channelless gate array. In *IEEE Custom Integrated Circuits Conference*, pages 281–284, 1986.

[23] F. Hwang, D. Richards, and P. Winter. *The Steiner Tree Problem*, volume 53 of *Annals of Discrete Mathematics*. North-Holland, Netherlands, 1992.

[24] M. Igusa, M. Beardslee, and A. Sangiovanni-Vincentelli. ORCA a sea-of-gates place and route system. In *Design Automation Conference*, pages 122–127, 1989.

[25] Integrated Logic Systems, Colorado Springs, Colorado. *Metal programmable gate array, data sheet*, May 1991.

[26] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, February 1970.

[27] E. S. Kuh, T. Kashiwabara, and T. Fujisawa. On optimal single-row routing. *IEEE Transactions on Circuits and Systems*, pages 361–368, June 1979.

[28] E. S. Kuh and T. Ohtsuki. Recent advances in VLSI layout. *Proceedings of IEEE*, 78(2):237–263, February 1990.

[29] U. Lauther. A min-cut placement algorithm for general cell assemblies based on a graph representation. *Design Automation Conference*, pages 1–10, June 1979.

[30] T. Lengauer. *Combinatorial algorithms for integrated circuit layout.* John Wiley & Sons, 1990.

[31] Y. L. Lin, Y. C. Hsu, and F. S. Tsai. Hybrid routing. *IEEE Transactions on Computer-Aided Design*, 9(2):151–157, February 1990.

[32] M. J. Lorenzetti and D. S. Baeder. Routing. In B. Preas and M. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, chapter 5. The Benjamin/Cummings Publishing Co., 1988.

[33] M. Mogaki, C. Miura, and H. Terai. Algorithm for block placement with size optimization technique by the linear programming approach. In *International Conference on Computer-Aided Design*, pages 80–83, November 1987.

[34] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *Design Automation Conference*, pages 620–625, June 1990.

[35] K. Nakamura, Y. Enomoto, Y. Suehiro, and K. Yamashita. Advanced CMOS ASIC design methodologies. In *Microelectronics and Systems*, 1989.

[36] R. H. J. M. Otten. Automatic floorplan design. In *Design Automation Conference*, pages 261–267, June 1982.

[37] R. H. J. M. Otten. Efficient floorplan optimization. In *International Conference on Computer Design*, pages 499–502, October 1983.

[38] M. Palczewski. Plane parallel A* maze router and its application to FPGAs. In *Design Automation Conference*, pages 691–697, 1992.

[39] P. Pan and C. L. Liu. Area minimization for general floorplans. In *International Conference on Computer-Aided Design*, pages 606–609, November 1992.

[40] B. Preas and P. G. Karger. Placement, assignment and floorplanning. In B. Preas and M. Lorenzetti, editors, *Physical Design Automation of VLSI Systems*, chapter 4. The Benjamin/Cummings Publishing Co., 1988.

[41] F. P. Preparata and M. I. Shamos. *Computational Geometry An Introduction.* Springer-Verlag, N.Y., 1985.

[42] E. Rosenberg. Optimal module sizing in VLSI floorplanning by nonlinear programming. *Methods and Models of Operations Research*, 33:131–143, 1989.

[43] V. Roychowdhury, J. Greene, and A. El Gamal. Segmented channel routing. *IEEE Transactions on Computer-Aided Design*, 12(1):79–95, January 1993.

[44] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose. Synthesis methods for field programmable gate arrays. *Proceedings of the IEEE*, 81(7), July 1993.

[45] S. Sastry and A. C. Parker. Stochastic models for wire-ability analysis of gate arrays. *IEEE Transactions on Computer-Aided Design*, 5(1):52–65, January 1986.

[46] E. Shragowitz, J. Lee, and S. Sahni. Algorithms for physical design of 'sea-of-gates' chips. *Computer-Aided Design*, 20:382–397, 1988.

[47] J. N. Song and Y. K. Chen. An algorithm for one and half layer channel routing. In *Design Automation Conference*, pages 131–136, 1985.

[48] L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, 57:91–101, 1983.

[49] Y. Sun and M. Sarrafzadeh. Floorplanning by graph dualization: L-shaped modules. *Algorithmica*, 10:429–456, 1993.

[50] Y. Sun, T. C. Wang, C. K. Wong, and C. L. Liu. Routing on symmetric FPGA/FPIC. In *International Conference on Computer-Aided Design*, pages 486–490, 1993.

[51] T. G. Szymanski. Dogleg channel routing is NP-complete. *IEEE Transactions on Computer-Aided Design*, 4(1):31–40, January 1985.

[52] N. Togawa, M. Sato, and T. Ohtsuki. A simultaneous placement and global routing algorithm for symmetric FPGAs. In *second International Workshop on FPGA*, Berkeley, CA, 1994.

[53] S. Trimberger. A reprogrammable gate array and applications. *Proceedings of the IEEE*, 81(7):1030–1041, July 1993.

[54] S. Tsukiyama, E. S. Kuh, and I. Shirakawa. An algorithm for single-row routing with prescribed street capacities. *IEEE Transactions on Circuits and Systems*, pages 765–772, September 1980.

[55] T. C. Wang and D. F. Wong. An optimal algorithm for floorplan area optimization. In *Design Automation Conference*, pages 180–186, 1990.

[56] S. Wimer, I. Koren, and I. Cederbaum. Floorplans, planar graphs, and layout. *IEEE Transactions on Circuits and Systems*, 35(3):267–278, March 1988.

[57] S. Wimer, I. Koren, and I. Cederbaum. Optimal aspect ratio of building blocks in VLSI. *IEEE Transactions on Computer-Aided Design*, 8(2):139–145, February 1989.

[58] D. F. Wong and C. L. Liu. Floorplan design of VLSI circuits. *Algorithmica*, 4:263–291, 1989.

[59] Xilinx Inc. *XC4000 Logic Cell Array Family, Technical Data*, 1990.

[60] Xilinx Inc. *The Programmable Gate Array Data Book*, 1992.

[61] K. Yamashita, Y. Enomoto, T. Sasaki, S. Kawahara, A. Kumagai, T. Sendo, and A. Okada. A quick turnaround time ASIC 'QCL series'. Technical report, Institute of Electronics, Information and Communication Engineers, June 1989. no. 89-93, pp. 65-69 (In Japanese).

[62] T. Yoshimura and E. S. Kuh. Efficient algorithms for channel routing. *IEEE Transactions on Computer-Aided Design*, 1:25–35, January 1982.

[63] K. Zhu and D. F. Wong. On channel segmentation design for row-based FPGAs. In *Design Automation Conference*, pages 26–29, 1992.

Vita

Yachyang Sun was born on January 24, 1963, in Taipei, Taiwan. He received a Bachelor of Science degree in Electrical Engineering from National Cheng Kung University. After two years of military service, he began graduate study in the fall of 1987 at Northwestern University at Evanston, Illinois. After he obtained a Master of Science degree in Computer Science from Northwestern University in 1989, he joined the doctoral program in the fall of 1989 at the University of Illinois at Urbana-Champaign. He worked as a research assistant there until completing his dissertation in 1994.