Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

1992

Improved I/O pad positions assignment algorithm for sea-of-gates placement

Shyang-Kuen Her
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Electrical and Computer Engineering Commons
Let us know how access to this document benefits you.

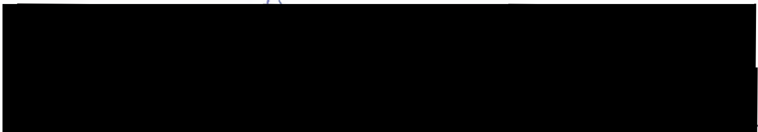
Recommended Citation
Her, Shyang-Kuen, "Improved I/O pad positions assignment algorithm for sea-of-gates placement" (1992). *Dissertations and Theses.* Paper 4316.
https://doi.org/10.15760/etd.6200

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

AN ABSTRACT OF THE THESIS OF Shyang-Kuen Her for the Master of Science in Electrical and Computer Engineering presented February 13,1992.

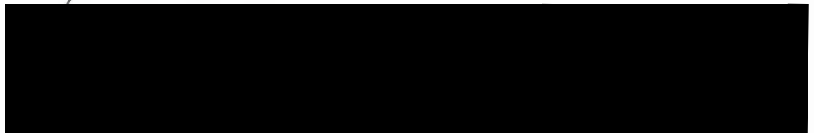
Title: Improved I/O Pad Positions Assignment Algorithm for Sea-of-Gates Placement

APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



Malgorzata Chrzanowska-Jeske, Chair

Marek Perkowski



Andrew Fraser

A new heuristic method to improve the I/O pad assignment for the sea-of-gates placement algorithm "PROUD" is proposed. In PROUD, the preplaced I/O pads are used as the boundary conditions in solving sparse linear equations to obtain the optimal module placement. Due to the total wire length determined by the module positions is the strong function of the preplaced I/O pad positions, the optimization of the I/O pad circular order and their assignment to the physical locations on the chip are attempted in the thesis. The proposed I/O pad assignment program is used as a predecessor of PROUD. The results have revealed excellent improvement.

IMPROVED I/O PAD POSITIONS ASSIGNMENT ALGORITHM FOR SEA-OF-GATES PLACEMENT

by

SHYANG-KUEN HER

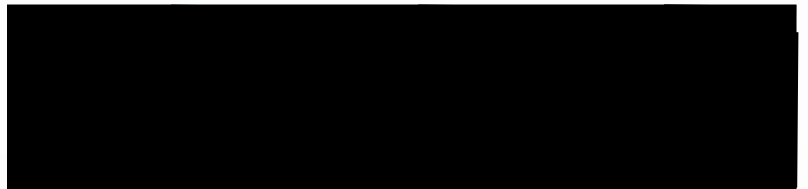
A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

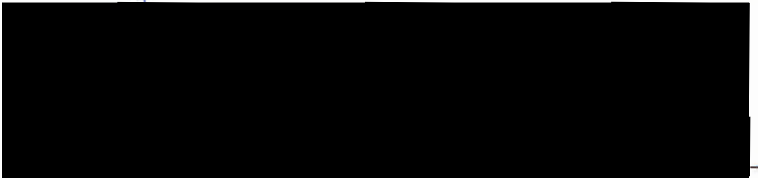
Portland State University
1992

TO THE OFFICE OF GRADUATE STUDIES:

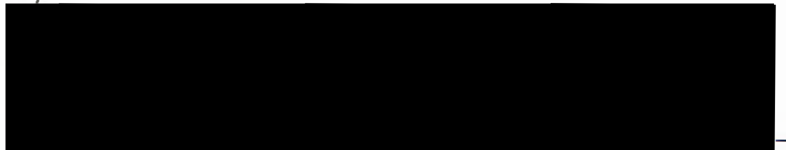
The members of the Committee approve the thesis of Shyang-Kuen Her presented February 13, 1992.



Malgorzata Chrzanowska-Jeske, Chair



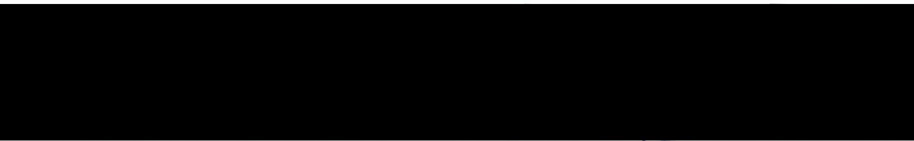
Marek Perkowski



Andrew Fraser

APPROVED:

Rolf Schaumann, Chair, Department of Electrical Engineering



C. William Savery, Vice Provost for Graduate Studies and Research

ACKNOWLEDGEMENT

I am very grateful to my parents for supporting all kinds of help and encouraging me during my entire studying period. I also would like to thank my adviser, Dr. Malgorzata Chrzanowska-Jeske, for instructing my whole thesis work, and Dr. Marek Perkowski and Dr. Andrew Fraser for their valuable comments and corrections. Meanwhile, I must thank my lovely girl friend, Aria Wong, for her kind encouragement and care, and my good friend, Kiswanto Thayib, for his help in the programming task.

I also thank all the faculty in EE department and my schoolmates in PSU for their kind friendship. I appreciate all the people I know in Portland for their hospitality that leaves me a wonderful memory in U. S. A.

TABLE OF CONTENTS

PAGE

ACKNOWLEDGEMENT iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

I INTRODUCTION 1

II SEA-OF-GATES 4

What Is Sea-of-Gates? 4

Layout Tools for Sea-of-Gates 6

III PREVIOUS PLACEMENT ALGORITHMS AND PROUD: AN EFFICIENT SEA-OF-GATES PLACEMENT ALGORITHM 8

Introduction 8

Previous Placement Algorithms 9

PROUD: Efficient Sea-of-Gates Placement Program 12

IV PAD ASSIGNMENT PROBLEM IN PROUD AND SOLUTION APPROACH 13

Introduction 13

General Pad Problems 13

Pad Assignment Problem in PROUD 14

Solution Approach for Pad Assignment 15

V I/O PAD POSITIONS ASSIGNMENT ALGORITHM 18

Introduction 18

Definition 18

Pad Assignment Algorithm 19

Complexity 45

VI TEST RESULTS 47

VII CONCLUSION AND FUTURE WORK 53

REFERENCES 55

APPENDICES

A ONE INITIAL PAD AND TWO WAYS ASSIGNING FUNCTION ALGORITHM 57

B I/O PAD POSITIONS ASSIGNMENT PROGRAM 60

LIST OF TABLES

TABLE		PAGE
I	Benchmark specifications	47
II	Results comparison for PRIMARY1	48
III	Results comparison for PRIMARY2	48
IV	Example specifications	50
V	Results comparison for CIRCUIT_1	50
VI	Results comparison for CIRCUIT_2	51
VII	Results comparison for CIRCUIT_3	51
VIII	Results comparison for CIRCUIT_4	51
IX	Results comparison for CIRCUIT_5	52

LIST OF FIGURES

FIGURE PAGE

1. Sea-of-gates design graph. 5

2. Basic cell enlargement 5

3. Placement results by random I/O arrangement. 15

4. Forces from pads. 16

5. Forces among pads and modules 17

6. Pad assignment algorithm flowchart. 20

7. Cost example. 22

8. The influence of pad on module. 23

9. One initial pad and one way local assignment 24

10. Position switching. 25

11. Total wire length vs allocation for PRIMARY1 25

12. PRIMARY1 results from one initial pad and one way local assigning function. 26

13. PRIMARY1 results from one initial pad and one way global assigning function. 28

14. One initial pad and two ways assignment 29

15. PRIMARY1 results from one initial pad and two ways assigning function. 31

16. PRIMARY2 results from one initial pad and two ways assigning function. 32

17. Two initial pads and four ways assignment 32

18. Two sub-rings concatenate 34

19. PRIMARY1 results from two initial pads and four ways assigning function. 34

20. Rate of PRIMARY1 GP results from one initial pad and one way local assigning function 35

21. Rate of PRIMARY1 GP results from one initial pad and one way global assigning function 35

22. Rate of PRIMARY1 GP results from one initial pad and two ways assigning function 36

23. Rate of PRIMARY1 GP results from two initial pads and four ways assigning function 36

24. Nearer-pad-pairs 38

25. Sums of NWCs vs different rings for PRIMARY1 39

26. Farthest-pad-pairs 40

27. Negative sums of FWCs vs different rings for PRIMARY1 40

28. Sums of NDCs vs different rings for PRIMARY1. 40

29. Negative sums of FDCs vs different rings for PRIMARY1 41

30. Example for block-pad-pairs 41

31. Sums of BWCs vs different rings for PRIMARY1 42

32. Ring's FACTOR_SUM vs different rings for PRIMARY1. 43

33. Ring's FACTOR_SUM vs different rings for PRIMARY2. 43

34. Total wire length of GP vs FACTOR_SUM for PRIMARY1. . . 44

35. Total wire length of GP vs FACTOR_SUM for PRIMARY2. . . 44

36. Results of different I/O pad arrangement for PRIMARY1 49

CHAPTER I

INTRODUCTION

As the design of very large scale integrated (VLSI) circuits becomes more complicated, the conventional design style, like gate arrays and standard cells, with rows of cells separated by fixed-width routing channels is no longer suitable. The conventional design style has worked successfully for circuits with low gate density. But when circuits become more complex and the number of gates is large, some drawbacks become significant in the conventional design style. First, the fixed routing channels use a lot of space. Therefore, the number of gates that can be placed in a chip is limited. Second, sometimes the tracks in a channel are not enough to complete the connections between two sided rows of cells, detours and vias are needed. This will cause undesirable congestion in adjacent neighbor channels and additional metal wire capacitance. Third, the aspect ratio of the cell is always restricted in some area. For circuits with different shapes of cells, like the macro cells, 100 percent automated layout becomes difficult. Therefore a new design style, sea-of-gates, is becoming more and more important, especially in the design of application specific integrated circuits (ASICs).

In sea-of-gates technology, chips are fabricated by adding customized connection layers to a wafer of prefabricated transistor arrays. Sea-of-gates chip features an increased number of gates, up to 250,000 on a chip, and an increased number of wiring layers. The above together with the lack of predefined fixed channels offer more flexible placement and routing options but also increase the complexity of the optimization problems. For high density of gates in a chip, some of the conventional layout algorithms need unberable time to reach an acceptable result if one is possible. Therefore, new, more efficient automatic layout algorithms are needed to overcome the difficulties that the

conventional algorithms met in complex circuits.

A placement algorithm, "PROUD" [1][2], based on the concept of resistive network optimization, was shown to perform very efficiently on large and complex sea-of-gates chips. This algorithm takes the I/O pad positions as input and solves successively linear equations to obtain an optimized module arrangement. As the I/O pad positions determine the boundary conditions needed for this solution method, their arrangement directly influences the final placement solution. Usually the sum of net length, defined as the half-perimeter of the rectangle that encloses all pins in the same net, is commonly used as a measure for placement. Since the total wire length determined by the module positions is the strong function of the preplaced I/O pad locations, a good assignment of the I/O pad circular order and their assignment to the physical locations on the chip should be attempted.

Usually pad problems are solved independently from the placement of interior modules. It means that the placement of the interior modules is solved before the placement of pads. Therefore the problem is an optimization of minimizing the net length between the pad and the modules in the same net. However, when the placement problem is solved by some force-directed algorithms, the fixed I/O pad positions must be known before the interior modules are placed. That is because in the absence of I/O pads, the interior modules collapse to the center of a chip. Thus this kind of pad optimization problem is much concerned to the whole connections between pads and modules.

In this thesis a pad assignment algorithm with a heuristic searching method is proposed. It determines the I/O pad arrangement, which is then used in the sea-of-gates placement program "PROUD". Using the net list information the algorithm determines the relative position of each pad on the circular ring and then assigns them to the physical fixed pad locations on a chip. The preplaced I/O pads are then used as a boundary condition by PROUD. The pad assignment program is written in C and tested on the SUN

SPARC workstation using some benchmark examples from Microelectronic Center of North Carolina (MCNC). A number of tests have been run and good results have been approached.

In Chapter II a brief introduction of sea-of-gate is presented and some layout tools suitable for the sea-of-gates chip are described. Chapter III reviews some commonly used conventional placement algorithms and an efficient sea-of-gates placement algorithm "PROUD" is presented. Due to the superior characteristics of PROUD over other conventional placement algorithms, we were motivated to engage in the study of its pad problem. The pad problem in PROUD is defined in Chapter IV. In Chapter V the whole I/O pad positions assignment algorithm is presented. The results are presented in Chapter VI. Finally, a brief conclusion of this research work and some possible problems for future study are suggested at the end of the thesis.

CHAPTER II

SEA-OF-GATES

WHAT IS SEA-OF-GATES?

The sea-of-gates design style is shown in Figure 1. It does not have pre-defined routing channels like the conventional design style. Gates are placed close together all over the chip. Therefore, high density on the order of 250,000 gates per chip becomes possible. For example, recently IBM presented a 300,000 gates ultralarge-scale integrated (ULSI) CMOS structured sea-of-gates array [3]. The features of this design are quadruple-level metalization, 0.45μm effective channel lengths, 0.8μm drawn gates, 0.18ns gate delays typically. The size of the chip is only one-ninth of a square inch, yet it contains 2 million transistors. Since channels are not provided in the sea-of-gates chip, gates are connected by routing through existing gates as shown in Figure 2, and by adding more metals or polysilicon interconnection layers. However, if interconnections are completed by routing through existing gates, then the overall usable gate count is reduced. For a typical sea-of-gates chip a fraction around 30 to 70 percent of gates are used with current technology [4]. Though the utilization of usable gates is low in the sea-of-gates chip, it is still the best method to obtain higher density order of gates per chip. For example, a 250,000 gates of sea-of-gates chip with 40 percent utilization can obtain 80,000 gates more than a 20,000 gates of conventional gate array with 95 percent utilization [4]. Yet, there is a tradeoff between gate utilization and layout complexity. Higher gate utilization will cause less space for routing. Therefore, layout becomes difficult. In addition to above features, cells in sea-of-gates can grow two dimensionally and the number of routing tracks is adjustable. This is because the concept of the routing channel

has been changed from hardware to software domain. The channel size is defined variably by the layout algorithm according to the shape of cells and available routing space. Moreover, sea-of-gates design can be easily adopted into different kinds of circuit designs like gate arrays, standard cells or with macro cells combined circuits.

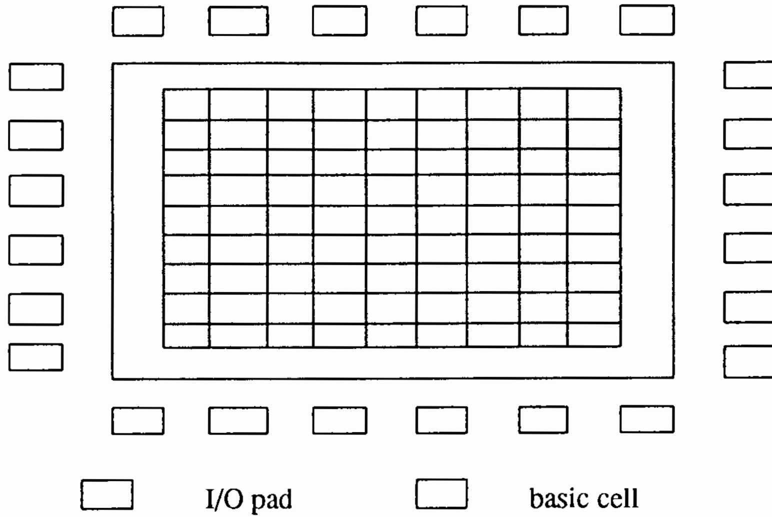


Figure 1. Sea-of-gates design graph.

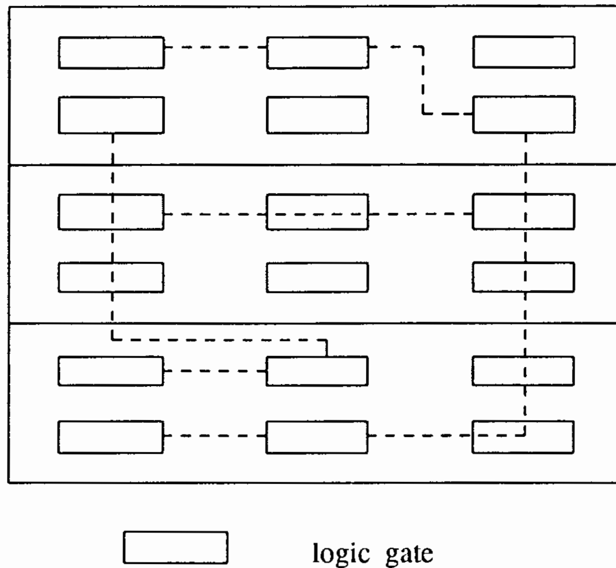


Figure 2. Basic cell enlargement.

LAYOUT TOOLS FOR SEA-OF-GATES

Since the number of gates in the sea-of-gates design is large, more efficient placement and routing tools are needed to reach an acceptable result in reasonable time. Several methods have already been proposed and research efforts are still continuing.

SoGOLaR (Sea-of-Gates Optimized Layout and Routing) [5] is a program to generate functional cells for static CMOS circuits in the sea-of-gates layout style. It is used on two applications. One application is to regenerate a fixed cell library in corresponding to the change in fabrication technology. The other application is to generate customized cells as part of a more flexible approach to automatic layout. In addition to these two applications, it can be used as a framework for evaluating the quality of the base array templates used in different sea-of-gates layout systems by measuring the area utilization. A floorplanner FOLM [6] for sea-of-gates design style with a frame overlapping floorplan model was proposed by Toshiba Corporation. This floorplanner is based on a model which uses frames for placement without any constraint on inter-block channels. A frame is defined as a region to restrict the placement of a specific cell group. A forced directed method is used to control the shape and movement of the frames. Meanwhile the frames are allowed to overlap in order to reduce the possibility of causing area with extreme aspect ratio. The objective of this floorplanner is to minimize the net length among frames in order to efficiently use the chip area. By using this frame model, FOLM-planner demonstrated the capabilities of making the cell density uniform and of minizing the net length. Improvement has been approached comparing to those floorplan models with inter-channels restriction like the building block model. Another sea-of-gates layout tool ORCA [7] is a place and route system, which provides standard-cell-like, macro-cell-like and porous macro-cell layout styles. By using the characteristic of the over-cell routing more flexible cells are generated and good results are achieved. ORCA had demonstrated its good ability to solve problems on row-based gate arrays as well as sea-

of-gates. VLSI Technology Incorporates proposed a hierarchical floor-planning, placement and routing tool for sea-of-gates design. This tool is designed to handle 250K gates gate array with RAM and ROM function blocks. It features concurrent processing, timing-driven layout, special clock distribution and power distribution. Yet, for complex sea-of-gates gate arrays this tool is still on the testing stage. Moreover a sea-of-gates placement algorithm "PROUD" [1] was proposed by R. Tsay, E. Kuh and C. Hsu from University of California, Berkeley. The concept of resistive network optimization was used to solve the placement optimization problem. By solving successive analogous linear equations together with hierarchical partitioning and iteration, the global placement optimization is approached. By considering actual pin position, module rotation, I/O pad position adjustment and module swap, further improvement of the placement quality can be obtained. Experiments demonstrate that a good result for total wire length can be obtained in shorter time than the latest simulated annealing approach. Due to its superior performance in solving circuits with large number of gates, it is suitable for sea-of-gates chips. This program can also be applied to different kind of design styles like gate arrays and standard cells.

The benefit of large number of gates and the flexibility of design options make sea-of-gates applicable to current ASIC's. Sea-of-gates has demonstrated its superior performance in a lot of applications. By using sea-of-gates technology a system implementation with 177k raw gates and flexibility of accommodating RAM and ROM in a single die was achieved by NEC Corporation [8]. Hitachi Semiconductor have announced his 250k available gates achievement [3]. NCR Microelectronics announced a CMOS gate array family with 0.7µm effective channel length and up to 100k usable gates in a double-level metal (DLM) process [3]. The exploration of higher gates utilization is continuing.

CHAPTER III

PREVIOUS PLACEMENT ALGORITHMS AND PROUD: AN EFFICIENT SEA-OF-GATES PLACEMENT ALGORITHM

INTRODUCTION

The placement problem for VLSI design is to place modules on a specified geometrical plane based on the given module interconnection specification, netlist. The final purpose of placement is to provide a facility for 100 percent routing. Usually the quality of placement is hard to measure until the routing is completed. For the convenience of comparison, the area of the resulting chip, the roughly estimated wire length, and the execution time are often used to compare the performance of various placement algorithms. Especially when circuits are complicated and the number of modules is large, the execution time becomes a much concerned factor. A lot of methods to solve the placement optimization problems like: minimum cut, branch and bound [9], force directed [1], simulated annealing, simulated evolution and more have been proposed. As the chips have become increasingly complex, larger in dimension and larger in module number, more efficient algorithms are being proposed continuously. Most of these methods perform well only for some specific placement problems. Due to the complexity and larger number of cells in the sea-of-gates design style, a good algorithm suitable for this kind of circuit design should possess the ability of reaching an acceptable solution in a bearable time. In this chapter several often used conventional placement algorithms are reviewed, and a new, efficient placement algorithm "PROUD" suitable for the sea-of-gates design is introduced.

PREVIOUS PLACEMENT ALGORITHMS

Numerous placement algorithms have been proposed. Most of them are used in some specific placement problems. No general placement algorithms exist which can solve all the placement problems. Some of these algorithms possess significant characteristics in solving placement problems with conventional design styles. But when circuits becomes complex and the number of gates on a chip becomes large, these methods present some drawbacks in the consideration of efficiency.

Min-Cut

The min-cut algorithm is a widely used heuristic algorithm for circuit placement. It starts with an initial bisection and exchanges pairs of modules across the cut of the bisection if the objective performance is improved. The objective is to minimize the module connections across the cut. This procedure allowed tightly interconnected cells to be placed together. The cutting and minimizing processes continue vertically and horizontally until each block contains a small specified amount of cells. This top-down hierarchical algorithm avoids the wiring congestion which is found usually in the center of the layout. However, a final result is possible to be stuck at a local optimum. A basic min-cut algorithm based on pairwise exchange was proposed by Kerninghan and Lin [10]. Fiduccia and Mattheyse [11] have made a modification to this Kerninghan-Lin min-cut heuristic. Also, a multiway partitioning method was proposed [12]. Practically, this greedy, recursive bipartitioning heuristic has demonstrated its ability to generate satisfactory solutions for many applications. But, due to the unavoidable large computation, the efficacy of this method becomes a problem for large circuits.

Simulated Annealing

Simulated annealing algorithm came from the concept of crystal growing processes. From experiments, a perfect crystal can be reached by applying the process of

annealing. By using this concept into combinatorial optimization problems for the determination of global minimum, good results can be obtained with a good annealing schedule. This method features exploring high cost move in order to avoid being trapped at local optimum. TimberWolf [13], an integrated set of placement and routing program, is based on this simulated annealing idea. Its basic algorithm can be presented as follows. Given a combinatorial optimization problem specified by a finite set of configurations and a cost function. A generation function is applied to generate a new configuration, and a random acceptance function is used to decide acceptance or rejection of new configuration. A parameter T, in analogy with temperature in annealing process, controls acceptance rule. A stopping criterion is reached when the cost remains the same after several annealing process. Good result with chip area saving and wire length reduction can be obtained, and the trap of local optimum can be avoided [14]. But the computation complexity is high for simulated annealing algorithm. It means tremendous CPU time is required. When the number of cells becomes large an unberable time will be needed to reach an acceptable result. Another difficult problem is how to build a good and efficient annealing schedule.

Simulated Evolution

Like simulated annealing this method is based on an analogy to the natural selection process in biological environments. According to natural evolution theory the superior characteristics of creatures will be kept and the ill-suited will be eliminated from one generation to next generation. The other way by a small rate of mutation, an unpredictable process that changes the characteristics, nature can prevent the developments of species from getting stuck at local optimum. The purpose of the evolution is to create stable structures which are finally perfectly adapted to the given constraints. By applying the idea of natural evolution to combinatorial optimization problems, some approaches have been announced by R. M. Kling and P. Banerjee [15][16][17]. The simulated evolu-

tion algorithm starts from an initial placement as a seed. After precomputation, according to wire length cost function, the already well placed cells are kept at the original locations and try to improve the other cells. Next step is the mutation process. Two modules are selected and exchanged randomly without regard to the placement value. Then evaluation is operated to the current placement based on specific cost function. By comparing each cell's placement value with a random number in the range from 0 to 100%, the decision is made to which cell will retain its current position in the next generation and which one should change to new position. The process will stop when no cells need to change its current position. This algorithm performs well only on small size circuits. When cell count becomes large the placement quality tends to get worse.

GORDIAN

Due to the complexity of circuit and large size of cell count some of the placement tools are preceded by dividing and iterations in order to reduce the problem size. This kind of algorithms have been claimed to solve optimization problems locally. Recently, a new placement optimization program GORDIAN [18] was proposed to solve the placement problem globally. By using the connectivity information of the circuit this program formulates the placement problem into a sequence of quadratic programming problems. Instead of dividing the whole problem into independent subproblems it adds more constraints to restrict the movement of cells on the chip space by partitions. This program is proceeded by an iteration of global optimization and partitioning steps. By using global optimization in the first stage all cells can be placed in the whole region of the chip. Then the partitioning step is provided to cut the whole space into subregions and cells into sub-cells. During next global optimization step each group of modules can only be placed in the specific sub-region. The processes will continue until each region contains only a specified number of cells. Experiments show its ability of solving circuits with large number of gates.

PROUD: EFFICIENT SEA-OF-GATES PLACEMENT PROGRAM

"PROUD" is an automatic circuit placement program written in C programming language[1]. It is designed for high complexity row-structured sea-of-gates, gate array and standard cell designs. It comprises of constructive phase and iterative improvement phase. Successive Over-Relaxation method and hierarchical partitions are used in constructive phase. In iterative improvement phase, local perturbations, I/O pad position adjustment, module swap or insertion are performed to achieve detail placement. The objective function of squared wire length is analogous to the power dissipation. The predefined I/O pad positions are analogous to the fixed voltage sources, and the interior movable modules are analogous to the node voltages. By using the concept of resistive network optimization, successively sparse linear equations are solved. The efficient sparse matrix technique is applied to solve these linear equations. Followed by a series of partitions and iterations, the interior module arrangement is optimized to result in shorter total wire length. In each partition some of the modules are allowed to represent fixed I/O pad positions. The total wire length has been improved and the execution time of PROUD is an order of magnitude faster than simulated annealing. This algorithm performs well in solving circuits with large number of gates. In this program the I/O pad positions are used as initial conditions. Experiments show that the final result of interior module arrangement is influenced by the boundary conditions, and a good arrangement of the I/O pad arrangement will result in an optimized placement of interior modules.

CHAPTER IV

PAD ASSIGNMENT PROBLEM IN PROUD AND SOLUTION APPROACH

INTRODUCTION

The pad problem is to find a set of ordered I/O pads in a ring facilitate on the outside edge of the chip, and to complete interconnections with interior modules specified by the net list. The objective is to minimize the total wire length and the chip area. The I/O pad problems are considered to be NP-complete combinatorial problems from the view of computational complexity. Usually, the I/O pad problem can be solved in three ways. One is to solve it after the interior modules have been already placed. Another way is to solve it with the interior modules together. Also it can be solved before the interior modules are placed. No matter which method is used, no one can guarantee an optimal solution for the real life large scale problems. Hence, algorithms based on the heuristic method are employed to reach good answers.

GENERAL PAD PROBLEMS

Several methods have been proposed to solve the pad assignment problem. For some placement algorithms, the pad assignment is determined independently after the optimization of module arrangement is already solved. This kind of pad assignment algorithm starts with an initial pad arrangement and exchanges the pad-pairs which will result in shorter total wire length. The exchange process will stop when there is no more improvement. Due to the fact that the interior modules are all fixed, the contribution of wire length reduction all comes from the change of pads' locations. More possible reduction of wire length contributed from the movement of modules in large core space is prohibit-

ed. The improvement is assumed limited. Furthermore, the quality of final total wire length is heavily influenced by the initial pad arrangement. D. C. Wang has proposed another pad assignment method [19] based on a bipartite graph. This graph contained two sets of nodes. One represents the pads and the other the physical pads' positions on the chip. The edge represents a cost to assign a pad to a specific physical location. The objective is to have the assignment of a pad to a position increase the wire length of that net as small as possible.

Recently, an I/O pad assignment method based on the analysis of circuit structure was proposed by M. Pedram, k. Chaudhary and E. S. Kuh [20]. This method uses a directed acyclic graph, which is represented as a Boolean network or a directed net list, to determine the relative pad positions. This pad assignment is obtained before the interior modules are placed, and the module arrangement will be determined after the pad positions are specified. Therefore, the circuit placement is not solved independently for pads and modules respectively, thus the whole interconnections among pads and modules need to be considered.

PAD ASSIGNMENT PROBLEM IN PROUD

The sea-of-gates placement algorithm "PROUD" is based on the concept of resistive network optimization. The transformed objective function $X^T BX$, the sum of the squared wire length, is analogous to the power dissipation of a n-node linear resistive network, where the X represents the coordinate matrix of modules and B is a modified connectivity matrix. By using the two Kirchhoff Laws the resistive network equations are written as:

$$B_{11}X_1 + B_{12}X_2 = 0$$

$$B_{21}X_1 + B_{22}X_2 = i_2$$

The I/O pad positions are analogous to fixed voltage sources X_2 used as the boundary

conditions. The movable modules, to be determined, are analogous to the node voltages X_1. The matrices B_{11}, B_{12}, B_{21} and B_{22} are the modified connectivity submatrices determined from netlist information. By solving the sparse linear equations with successive over-relaxation method, the optimal module placement is determined. Because the modules are specified as point modules, partition and iteration methods are used to solve the overlapping problem when the point modules are replaced by real modules. We have made a number of tests using the "PROUD" program to determine the influence of different pad arrangements on the total wire length. Figure 3 shows the variable global placement results versus different pad arrangements for the standard cell example PRIMARY1.

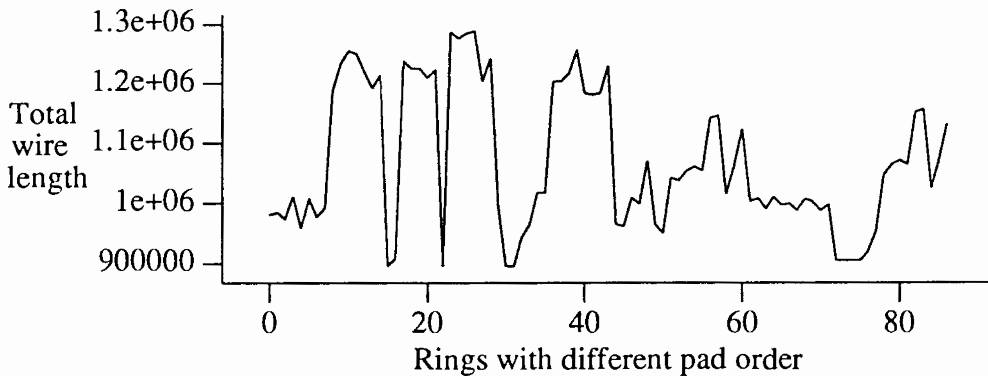


Figure 3. Placement results by random I/O arrangement.

The horizontal axis represents different arrangements of pads. A resulting deviation of 16.3% above and below the average result was observed for the standard cell benchmark example "PRIMARY1" from the MCNC.

SOLUTION APPROACH FOR PAD ASSIGNMENT

At the beginning we run the PROUD program with different pad arrangements used as inputs. The results of the total wire length change variably. We noticed that in every test some interior modules will be placed closely to the pads that have connections

with them. We assume that there are pulling forces among pads and modules. If a module connects to more than one pad, every pad will try to pull this module closer to itself. We assume that the strength from each pad to this module is the same, a position for this module in the gravity center of these pads will cause these forces from pads into a balance status as shown in Figure 4. The module in Figure 4 is pulled to the center of the chip by four pads.

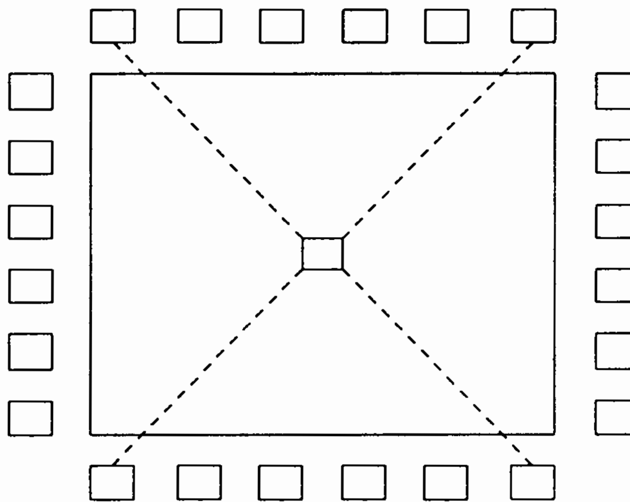


Figure 4. Forces from pads.

The balance status is influenced not only by the forces from pads but also by the forces existing among the modules themselves. The boundary condition represented by fixed I/O pad positions could be interpreted as the source forces, and the connections among modules could be interpreted as transmitted forces as shown in Figure 5. In addition to the source force trying to pull the connected modules closer to the pad, the transmitted forces try to pull the connected modules closer to themselves. Therefore, the final location of a module depends on which direction the total pulling forces are stronger and whether the slots to place modules are available or not. It is assumed that if the most related pads can be placed closely then the modules most related to these pads can be forced into available slots close to these pads, which would result in shorter total wire

length. The words "most related" means the strongest connectivity strength among pads and modules. According to this force idea a heuristic searching algorithm for pad assignment is proposed in Chapter V.

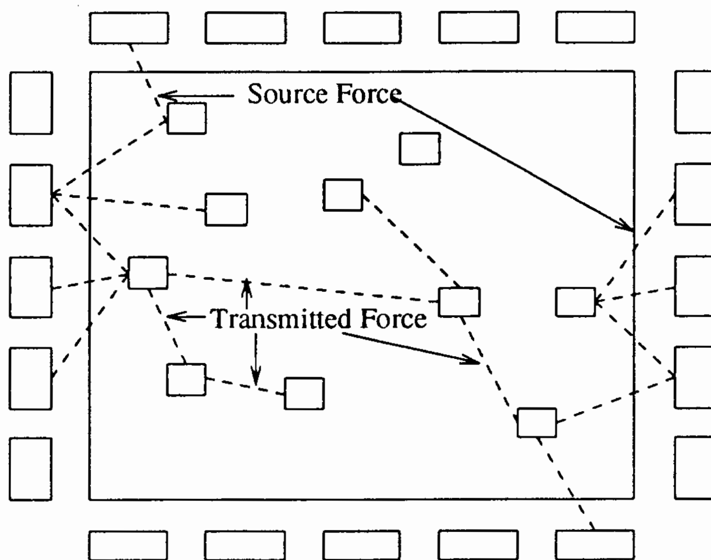


Figure 5. Forces among pads and modules.

CHAPTER V

I/O PAD POSITIONS ASSIGNMENT ALGORITHM

INTRODUCTION

In this chapter a pad positions assignment algorithm with a heuristic searching method to determine the I/O pad arrangement is proposed. This algorithm determines the relative position of each pad on the ring and then assigns this ring of pads to the physical locations on a chip. The preplaced I/O pads are then used as a boundary condition for the sea-of-gates placement program "PROUD". The pad assignment program comprises of five stages: input data, cost factors, assigning function, selection function and output. In the cost factors stage, two cost factors are defined to determine the connectivity strength between pads. In the assigning function stage, four different pad assigning methods have been studied. One of them was chosen to be used in our pad assignment program. The output of the program is a design file with specified pad positions, which is then used as the input to PROUD. The pad assignment program has been tested using two benchmark examples: PRIMARY1 and PRIMARY2. Excellent improvement has been approached.

DEFINITION

The physical chip has two parts: the interior area for placing movable modules and the outside area for fixed I/O pads. The cell is used as a general name for the pad and the module. The modules represent the cells in the interior area of a chip, netlist is represented as an undirected graph $G = (V, E)$, in which the vertices $V = \{ c_1, ..., c_n \}$ represent cells and the edges $E = \{ e_1, ..., e_m \}$ represent the connections between cells. Several definitions are stated.

Definition 1: Path P_{xy} is an undirected path between pad x and pad y.

Definition 2: Cell's weight W_x is defined as the number of cells that are connected to cell x. W_x = degree of c_x

Definition 3: Weight cost $WC(x,y)$ is the sum of all the cells' weights along the path between pad x and pad y. $WC(x,y) = \sum_{c_i \in P_{xy}} W_i$

Definition 4: Depth cost $DC(x,y)$ is the number of cells along a path between pad x and pad y. $DC(x,y) = \sum_{c_i \in P_{xy}} c_i$

Definition 5: The shortest path between two pads is the path with the smallest weight cost.

Definition 6: An initial pad is the first assigned pad.

Definition 7: A pad-ring $RING(x)$ is defined as a ring of pads with pad x as the initial pad.

Definition 8: A pad already assigned to its position and waiting for the other pad to be assigned beside it is called the host pad.

Definition 9: A pad chosen to be possibly placed beside the host pad is called the candidate pad.

Definition 10: A neighbor pad is a pad which has been determined to be placed beside the host pad.

PAD ASSIGNMENT ALGORITHM

A heuristic searching method to improve the I/O pad assignment for the placement algorithm based on the resistive network optimization model is introduced. The input data are: the original design file used by PROUD and the physical pad-location-file. The original design file contains the description of I/O pads and modules. The output of this algorithm is the same design file but with the physical pad locations specified for each pad.

The pad assignment algorithm comprises of 5 stages: input data, cost factors calculation, assigning function, selection function and output. The algorithm flowchart is shown in Figure 6. The blocks at the right side of Figure 6 represent the contents for each stage. These assigning functions squared by dashed line in Figure 6 are used to compare with the other assigning function which is squared by solid line. They are not used in the I/O pad assignment program.

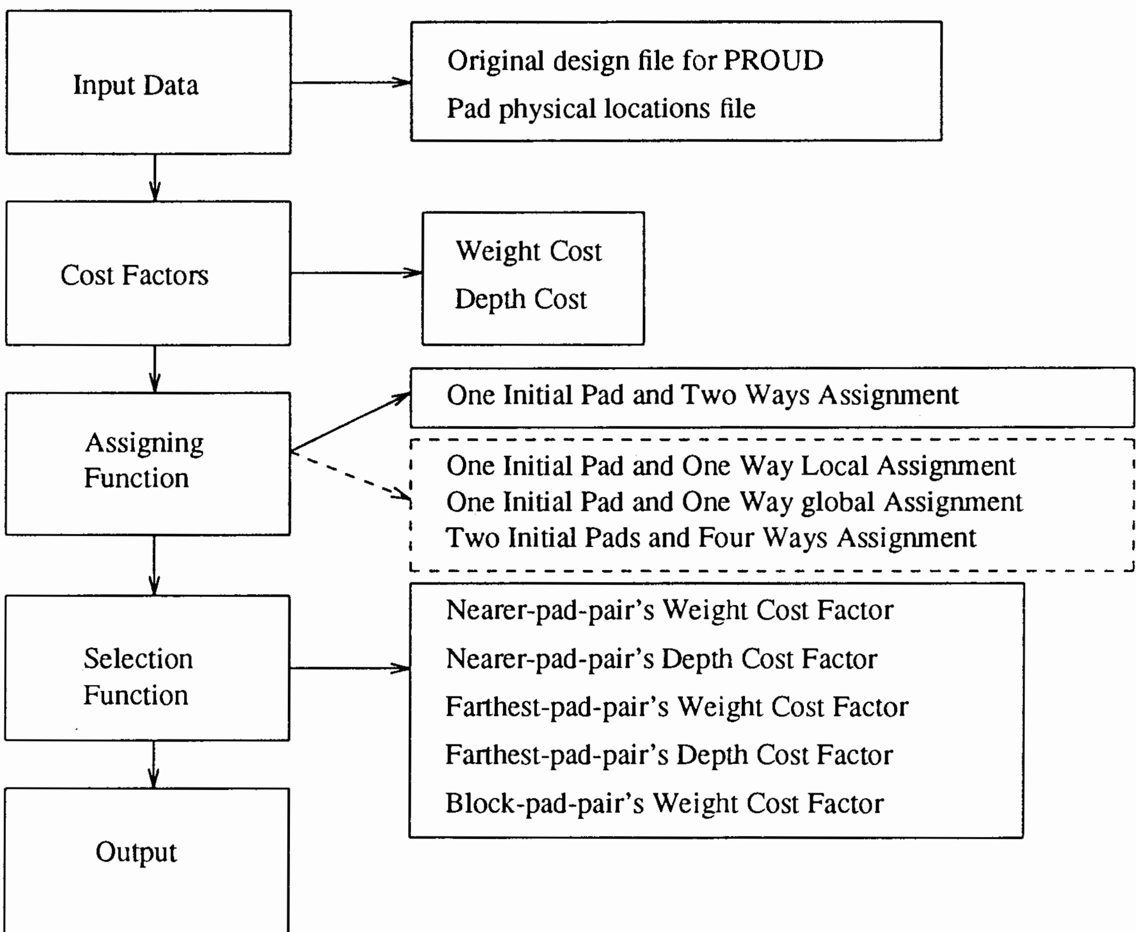


Figure 6. Pad assignment algorithm flowchart.

The cell adjacent list is built by reading the input file. Once the cell adjacent list is known the cost factors are calculated. Then the pad assigning process is executed to build pad-rings with different initial pads. After the pad assigning process, a selection

function is used to choose good pad-rings from a group of rings with different initial pad, which are formed by assigning function. Five factors are used to determine the selection function. At last the design file with specified pad positions is printed out. This pad assignment program is used as a predecessor for PROUD, and was tested on the standard cell benchmark examples the same examples as in [1]: PRIMARY1 and PRIMARY2.

Input Data

The netlist information can be obtained from the pad and module descriptions in the design file. The netlist is then transformed into the cell adjacent list. Later this cell adjacent list is used to search for highly related pads. The other input file contains the coordinate values of the physical pad locations on a chip. The format of this file is pre-defined by assigning the coordinate values of the available top middle pad position or bottom middle pad position of a chip to the first order of the pad-location-file if the first partition in PROUD is specified as a vertical cut. The first coordinate values are used as physical location of the initial pad. Following either the clockwise or counterclockwise direction on a chip, the other coordinate values are assigned in successive order. If the first partition in PROUD is specified as a horizontal cut, then the coordinate values of the available right middle or left middle position are used as physical location of the initial pad.

Cost Factors

For each pad-pair, there are a lot of different paths to connect them. But only the shortest path between each pad-pair can best express the connectivity relationship between them. Therefore, in the following text, the weight cost $WC()$ and the depth cost $DC()$ will represent only the costs along the shortest paths between two pads. The smaller values of these two factors indicate stronger connectivity strength between pads.

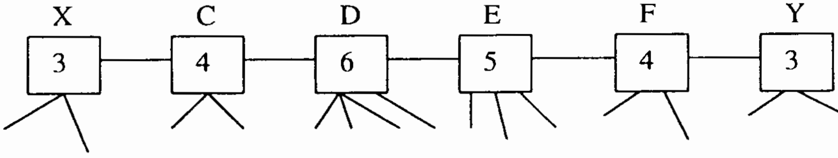


Figure 7. Cost example.

For example, in Figure 7 $X-C-D-E-F-Y$ is the shortest path from pad X to pad Y. The number in the vertex is the weight of each cell. The cost factors are calculated as follows:

$$WC(X,Y) = W_X + W_C + W_D + W_E + W_F + W_Y = 3+4+6+5+4+3 = 25$$

$$DC(X,Y) = The\ number\ of\ cells\ along\ the\ shortest\ path = 6$$

Dijkstra's algorithm [21] is used to find the shortest path between any two pads. The reason we use only the shortest path values as the cost factors is because that the forces applied by fixed pads on modules are significant when the weight cost is small. The influence of the forces applied by fixed pads on the modules is weak when there are many other forces competing for the same modules. In other words, the modules along the shortest path between two pads are easier to be pulled close to the pads, compared to the modules in other paths. Therefore, the connectivity strength is represented by the weight cost along the shortest path between two pads. Pad-pair's connectivity strength along different paths is explained graphically in Figure 8. In the figure there are two paths from pad A to pad B. The number in the vertex is the weight of a cell. In path I, three forces are pulling module $m1$. There are two forces applied to module m2 and four applied to module m3. If pad A and pad B are placed together, they have to compete with other five forces in order to pull $m1$, $m2$, and $m3$ closer to themselves. In path II if pad A and pad B are placed together, they have to compete with other 9 forces in order to pull $m4$, $m5$, $m6$, $m7$ and $m8$ closer to themselves. Therefore it is easier for pad A and pad B to pull $m1$, $m2$ and $m3$ closer to themselves. The influence of the forces from

pad A and pad B on the modules in path II is not as significant as the influence on the modules in path I.

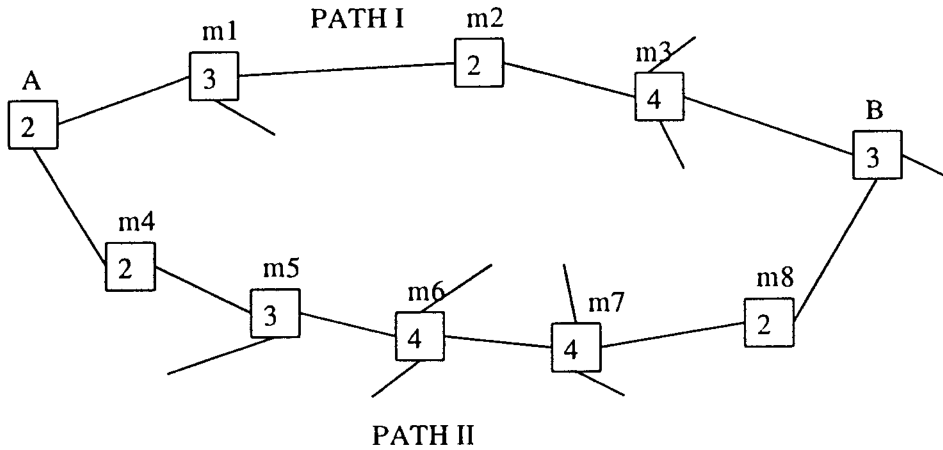


Figure 8. The influence of pad on module.

Assigning Function

The objective of the assigning function is to form a ring of pads with the most related pads located as close to each other as possible and the most unrelated pads located as far from each other as possible. The most related pads are the pad-pair with the smallest weight cost and the smallest depth cost along the shortest path between the two pads. After the ring of pads is formed, its components are assigned to the physical locations on a chip. Four different assigning functions have been studied. Each one forms I/O pad arrangements of different quality. One of these methods, which can obtain better I/O pad arrangements than the others is used in our I/O pad positions assignment program. Two MCNC layout benchmarks PRIMARY1 and PRIMARY2 were used for testing. The results will be presented in Chapter VI.

One Initial Pad and One Way Local Assignment. This assigning function uses the simplest method to construct a ring of pads. This method starts with an initial pad A as shown in Figure 9.

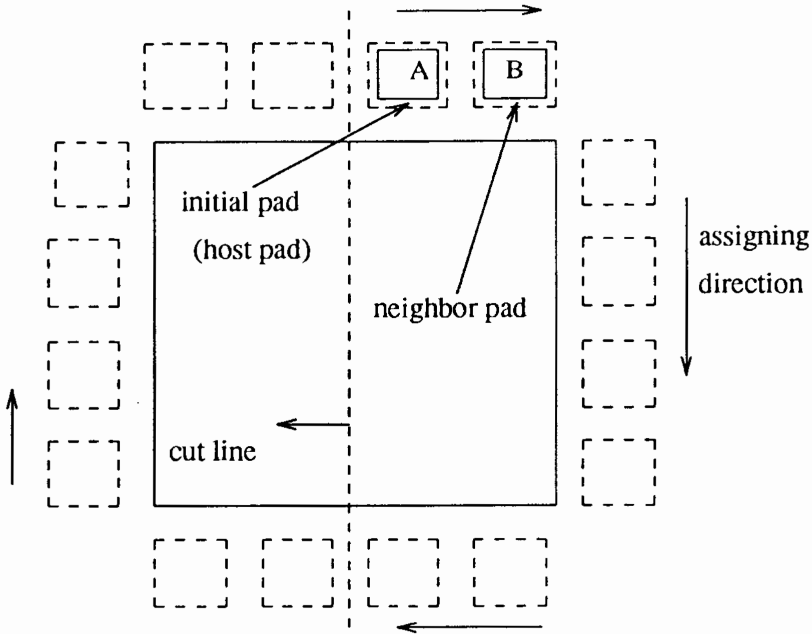


Figure 9. One initial pad and one way local assignment.

The initial pad A is called a host pad. Next, a neighbor pad will be chosen to be placed beside this host pad. It will be pad B with the smallest weight cost $WC(A, B)$ and the smallest depth cost $DC(A, B)$ in respect to pad A. Now, pad B becomes the host pad. This assigning process continue until all pads have been assigned. The forming of the ring is much influenced by the choice of the initial pad. Different initial pads will result in different circular orders. In Figure 9, the allocation of this pad-ring on the chip is only one of the ways. By rotating the ring, there are as many ways as the number of pads to assign the ring to a physical chip. For example in Figure 10, pad 1 can be moved from position A to position B and all the other pads will be rotated by one position at the same time. Therefore, there are 22 allocations for the ring on the chip for this example in Figure 10. Figure 11 shows the total wire length versus different allocations of $RING(x)$ on a chip for PRIMARY1. PRIMARY1 has 81 I/O pads, therefore, 81 different allocations of $RING(x)$ on a chip were found.

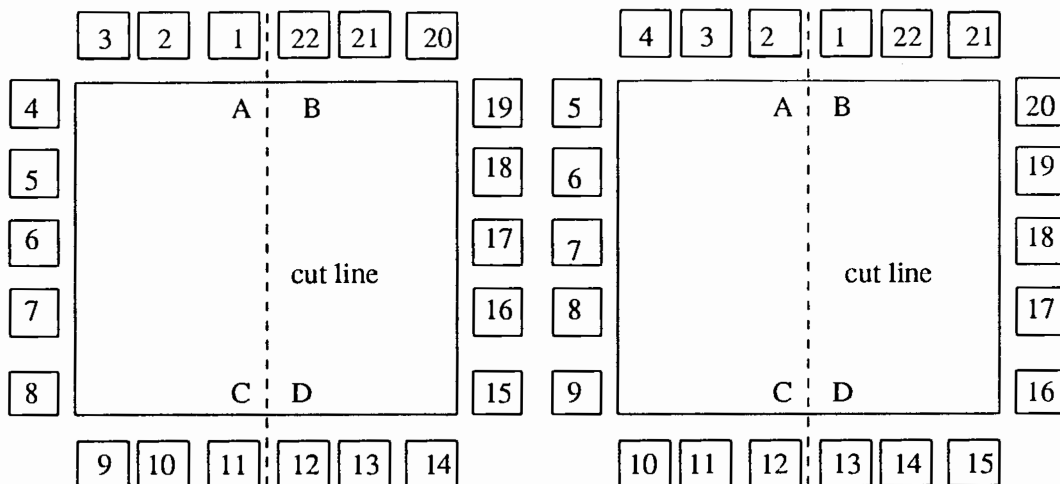


Figure 10. Position switching.

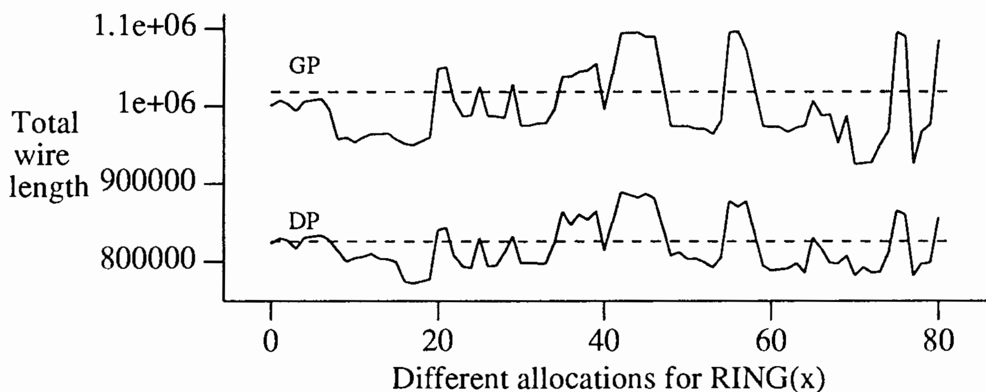


Figure 11. Total wire length vs allocation for PRIMARY1.

In Figure 11 the dashed lines represent the results of global placement (GP) and detail placement (DP) from [1]. We use them as references to compare with our results. For different allocations of a $RING(x)$ on a chip, the results are different. The partitioning algorithm in PROUD could be partially responsible for this diversity.

As previously mentioned, different initial pads will result in different rings of pads. Additionally, rotating the physical assignment of pads in every ring different total wire lengths can be obtained. For PRIMARY1, the number of I/O pads is 81. Therefore, by using the pad assigning function, 6561 results of total wire length can be obtained. To

simplify the problem, first we test rings with different initial pads for PRIMARY1 using a method described below to assign pads to the physical locations on a chip. The initial pad is assigned to the top middle or bottom middle position of the physical chip if the first partition in PROUD is specified as a vertical cut as shown in Figure 10. The other pads are located sequentially after the initial pad in either clockwise or counterclockwise direction. On the other hand, if the first cut in PROUD is specified as a horizontal one, the initial pad is assigned to the right middle or left middle position of the physical chip. The results of total wire length versus rings with different initial pads are presented in Figure 12.

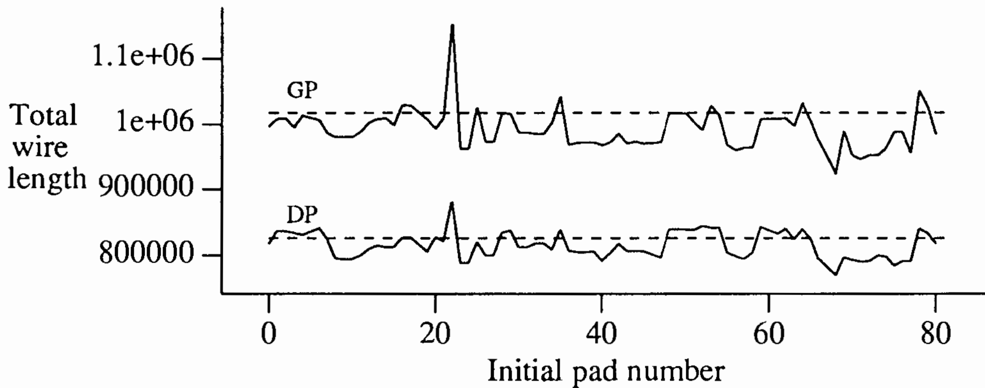


Figure 12. PRIMARY1 results from one initial pad and one way local assigning function.

Figure 12 shows that most of the rings show better results of total wire length than the results from [1], the dashed lines. It demonstrates that a good arrangement of pads can be obtained by using this specific allocation method. Second, we rotate the ring on the chip for each ring with different initial pads and test them on PRIMARY1 respectively. The experiment shows that the improvement of the total wire length by rotating the ring is limited to a small amount around 0.17%. Therefore, the rotation of a ring on the chip is not considered in the assigning function. Two properties have been observed for this assigning function. The chosen neighbor pad is a pad with the smallest weight

cost and depth cost in respect to the host pad. The relationships between this chosen neighbor pad and the other unassigned pads are not considered. Therefore, the search for the neighbor pad is restricted to local relations. Moreover, another important assignment factor to keep unrelated pads away from each other is not considered either. The global consideration for searching the neighbor pad will be discussed in the next assigning function.

One Initial Pad and One Way Global Assignment. This method starts with an initial pad as the host pad. Next, a pad with the smallest cost in respect to the host pad is chosen as the candidate pad. This candidate pad is not assigned to be the neighbor of the host pad directly. The costs between the candidate pad and the other unassigned pads are compared with the cost between the candidate pad and the host pad. If there exists more than one unassigned pads with smaller costs in respect to the candidate pad than the the cost between the candidate pad and the host pad, then this candidate pad is reserved and not assigned as a neighbor pad. The connectivity strengths between the candidate pad and at least two other unassigned pads are stronger than the strength between the candidate pad and the host pad. The reserved pad will not be chosen as a candidate pad for this host pad again. If the candidate pad is reserved, the host pad will continue to look for another candidate pad with the smallest cost to it and the relations of the candidate pad with the other unassigned pads are checked. This process will continue until a candidate pad is found to meet the connectivity constraint. If no pads satisfy the constraint, then the first reserved pad is assigned beside the host pad. After the neighbor pad has been assigned, the reserved pads are released and can be chosen as a candidate pad for the next host pad. The purpose of this additional constraint is to place pad-pairs with stronger connectivity closer together in the view of global relations. This one initial pad and one way global assigning function is tested on PRIMARY1 using the same allocation for assigning pads to the physical chip as previous assigning function. The results of the total wire length versus rings with different initial pads are shown in Figure 13.

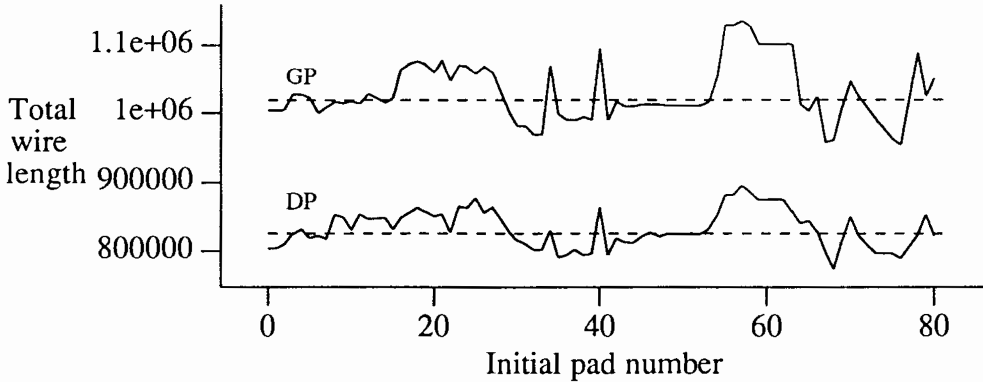


Figure 13. PRIMARY1 results from one initial pad and one way global assigning function.

In Figure 13 some rings show better results than [1], but a number of rings demonstrate worse results than [1]. From the results in Figure 13, it could be observed that although the much related pads are assigned closer to each other, the result of the total wire length did not been improved. Therefore, in addition to assigning much related pads close to each other, some other factors should be considered during the assigning process. Another possible factor that could be included, is to keep the unrelated pads away from each other.

One Initial Pad and Two Ways Assignment. As previous assigning functions, the one initial pad and two ways assigning function starts with choosing an initial pad randomly. Two functions are used to choose the strongly related pad-pairs. Function $NEXT_CANDIDATE(x)$ chooses a candidate pad which is placed next to the host pad x. It finds a candidate pad y with the smallest weight cost $WC(x,y)$ in respect to the host pad x. The other function $SEARCH(x,y)$ finds the number of unassigned pads such that $WC(y,z) < WC(x,y)$, in which z is any unassigned pad, y is the candidate pad and x is the host pad. If the returned value from function $SEARCH(x,y)$ is 0, it means that there are no other unassigned pads with smaller costs to the candidate pad y than the cost between the candidate pad y and the host pad x. Therefore the candidate pad y and the host pad x have the strongest connectivity to each other, and they can be assigned closer

to each other. If the returned value is 1, it means that there is another unassigned pad z which possesses the smaller cost value to the candidate pad y than the cost between the candidate pad y and the host pad x. In this situation the host pad and the candidate pad can still be assigned together. This is because the unassigned pad could be assigned to the other side of the candidate pad on next assignment, if it is acceptable. When the returned value from function $SEARCH(x,y)$ is greater than 1, it means that two or more other unassigned pads have stronger connectivity with the candidate pad y than the candidate pad and the host pad. Therefore, the candidate pad y should be assigned close to the other unassigned pads rather than to the host pad x. In this situation, the candidate pad y is reserved and not to be chosen as the candidate pad for this host pad x again.

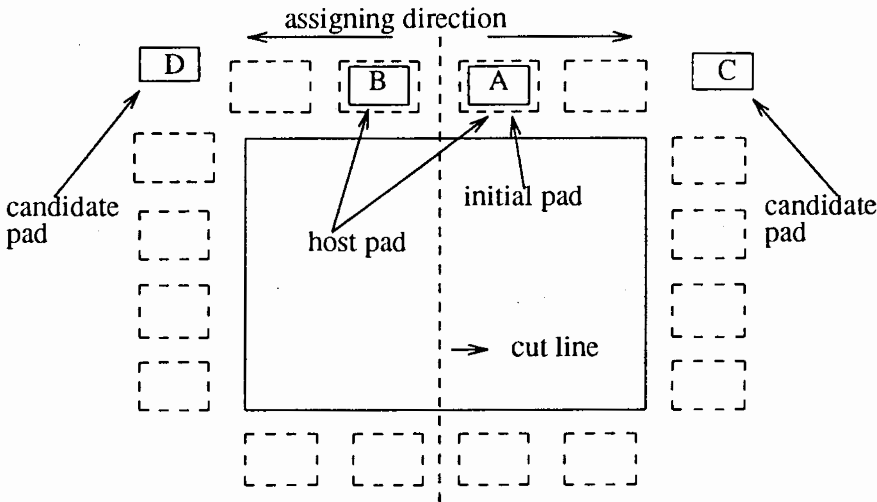


Figure 14. One initial pad and two ways assignment.

After the initial pad A has been assigned as shown in Figure 14, the second assigned pad next to the initial pad will be a pad B with the smallest cost in respect to the initial pad. Pad B is determined by using $NEXT_CANDIDATE(A)$ and $SEARCH(A,B)$. If no pads can satisfy the constraint in function $SEARCH(A,B)$, then the first reserved pad is assigned next to the host pad. The other reserved pads are released and can be chosen as the candidate pads in next stage of assignment. Now pad A and pad B are

called host pads. Next, two candidate pads, pad C and pad D, are chosen simultaneously for two host pads, pad A and pad B, respectively. The $NEXT_CANDIDATE(A)$ and $NEXT_CANDIDATE(B)$ processes are used. Since these two candidate pads are chosen at the same time, one of two possible cases will occur. These two candidate pads are the same pad or they are different pads. On the situation of two host pads competing for the same candidate pad, the candidate pad is assigned next to one of the host pads, which has smaller cost value to the candidate pad than the other, and the returned value from function $SEARCH()$ for this pad-pair must be less than 2. Otherwise the candidate pad is reserved and two other candidate pads will be chosen. If the cost values for the candidate pad to each host pad are the same, the candidate pad can be assigned to either host pad. At the other case, whether pad C will be assigned next to the host pad A and whether pad D will be assigned next to the host pad B depend on the returned values from $SEARCH(A,C)$ and $SEARCH(B,D)$. No matter what condition, once one candidate pad is assigned to its position, then all the reserved pads are released and can be chosen as a candidate pad in the next stage of assignment. The whole assigning process will stop when all pads have been assigned to their positions. The algorithm of the one initial pad and two ways assigning function is presented in APPENDIX A.

It can be noticed that the forming of the pad-ring is strongly influenced by the choice of the initial pad like previous functions. Moreover, because the assigning process progresses in two directions, the conditions for two host pads competing for the same candidate pad will occur. In this situation, no matter to which host pad the candidate pad is assigned, there is always another host pad which looses the chance to stay close to the candidate pad.

By analyzing the partitioning algorithm in PROUD, we noticed that the chip is bipartited into two equal size blocks for the first partition. The modules in the same blocks possess stronger connectivity among themselves than the connectivity between

the modules in different blocks. In the two ways assigning function, two sub-sections of a ring are formed with the most related pads in the same section. We assume that if the pads in each section are assigned to each block on a chip respectively, then the strongly related modules should be forced into the same block. Therefore, the assignment for the ring to the chip is completed by assigning the initial pad at the top middle or bottom middle position of the chip when the first cut is specified as a vertical cut in PROUD as shown in Figure 14. The other pads are located sequentially either clockwise direction or counterclockwise direction. On the other hand, when the first cut is specified as a horizontal cut, the initial pad is assigned at the right middle or left middle position of the chip.

Two standard cell benchmark examples PRIMARY1 and PRIMARY2 are tested using this one initial pad and two ways assigning function as predecessor for PROUD. Figure 15 and 16 show the results of total wire length versus rings with different initial pads for PRIMARY1 and PRIMARY2 respectively. The dashed flat lines are the results from [1]. The solid lines are the results of global placement and detail placement by using the two ways assigning function. Most of the rings demonstrate excellent improvement, compared to the results from [1].

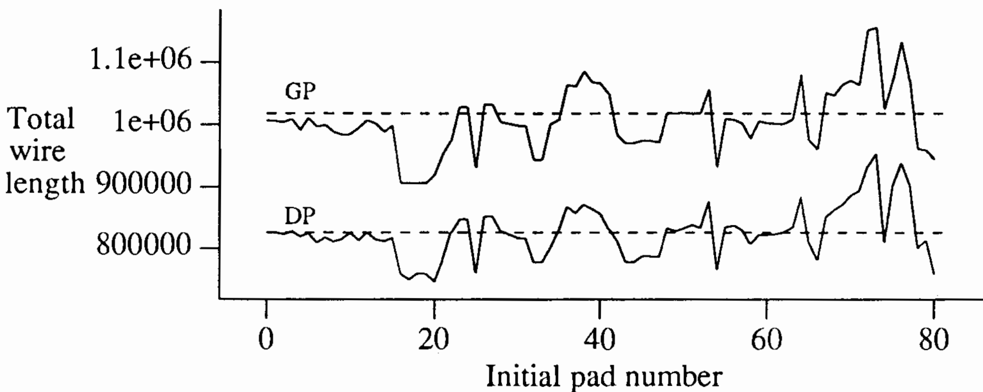


Figure 15. PRIMARY1 results from one initial pad and two ways assigning function.

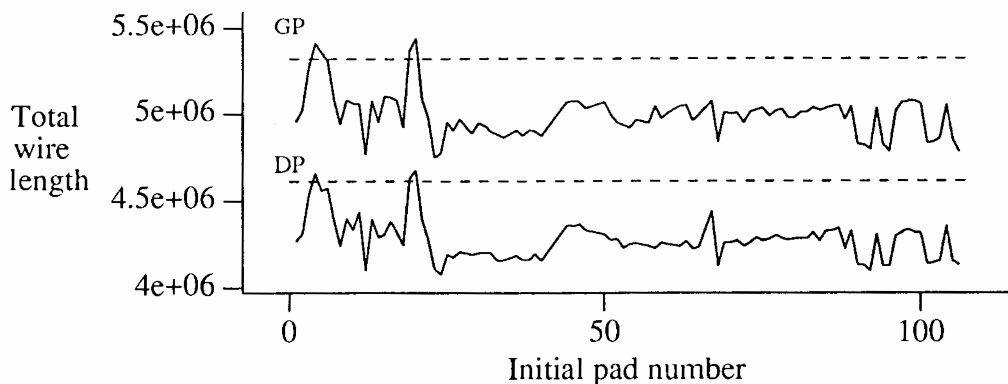


Figure 16. PRIMARY2 results from one initial pad and two ways assigning function.

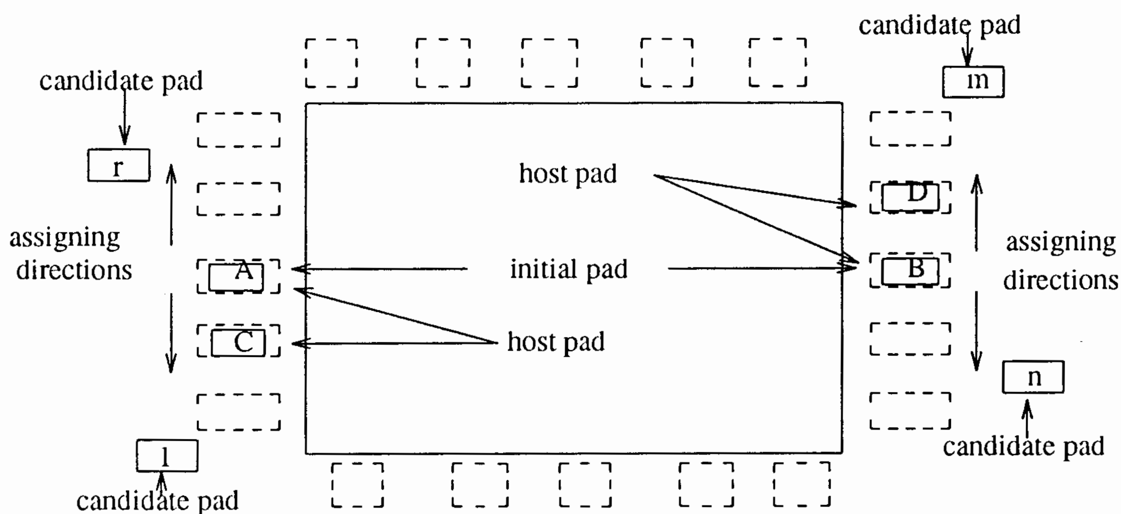


Figure 17. Two initial pads and four ways assignment.

Two Initial Pads and Four Ways Assignment. Since the two ways assigning method has shown good improvement in the total wire length, will the four ways, six ways or even more ways assignments perform better? A two initial pads and four ways assigning method is discussed. First, the function assigns two initial pads with the largest cost in respect to each other as host pads. Let these two host pads be pad A and B as shown in Figure 17. Next, two pads, pad C and pad D, with the smallest cost to each host pads respectively are assigned next to each host pad. Now, four host pads exist and four candidate pads are chosen at the same time. Let the four chosen candidate pads be

pad r, pad l, pad n and pad m. The situation for which candidate pad should be assigned next to which host pad becomes complicated. The possible occurring cases are:

(1) $r \equiv l$ and $r \neq m$ and $r \neq n$ and $m \neq n$

(2) $m \equiv n$ and $m \neq r$ and $m \neq l$ and $r \neq l$

(3) $r \equiv l$ and $m \equiv n$ and $r \neq m$

(4) $r \equiv l$ and $r \equiv m$ and $r \neq n$

(5) $r \equiv l$ and $r \equiv n$ and $r \neq m$

(6) $m \equiv n$ and $m \equiv r$ and $m \neq l$

(7) $m \equiv n$ and $m \equiv l$ and $m \neq r$

(8) $r \neq l$ and $r \neq m$ and $r \neq n$ and $l \neq m$ and $l \neq n$ and $m \neq n$

(9) $r \equiv l$ and $m \equiv n$ and $r \equiv m$

(10) *the other possibilities*

In each condition the assigning process is the same as the two ways assigning process. The *NEXT_CANDIDATE* () function is used to find the available candidate pad for the host pad locally and the *SEARCH* () function is used to verify up the global constraint. Due to the complex relationship between the candidate pads and the host pads, the probability of several host pads competing for the same candidate pad is high. In this competing situation, no matter what decision is made, some host pads will always loose the chance to be placed close to some candidate pads which have small costs in respect to them.

After all pads have been assigned, these two sub-rings are concatenated end to end as shown in Figure 18. The concatenated ring is assigned to the physical locations on the chip by placing either one of the initial pads at the right middle or left middle position on a chip when the first cut in PROUD is specified as a vertical cut. The other pads are located sequentially after the initial pad in either clockwise or counterclockwise direction. On the other hand, when the first cut is specified as a horizontal cut in PROUD,

then either one of the initial pads is placed at the top middle or bottom middle position on a chip.

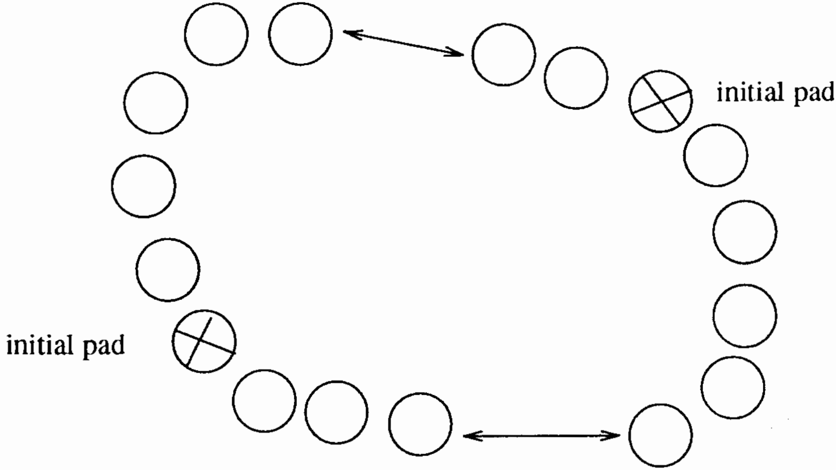


Figure 18. Two sub-rings concatenate.

Again this function is tested on the benchmark example PRIMARY1. The results of the total wire length versus rings with different initial pads are presented in Figure 19. The figure shows that although some rings result in improved total wire length, but most rings result in worse total wire length than the reference results.

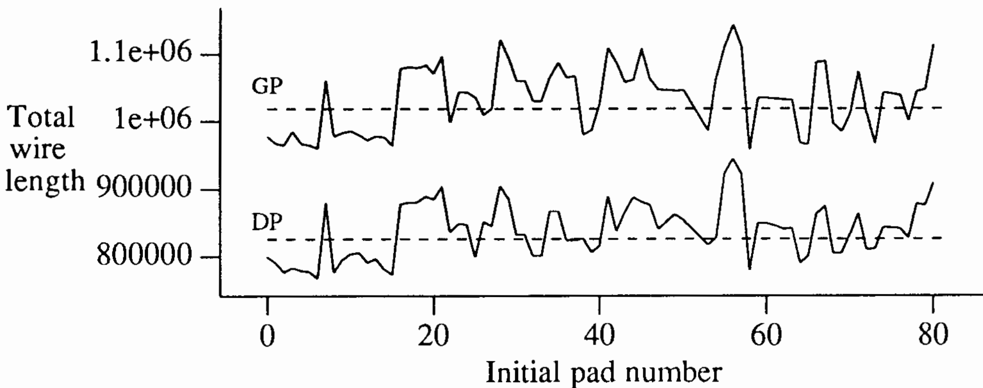


Figure 19. PRIMARY1 results from two initial pads and four ways assigning function.

Four different assigning methods have been tested on PRIMARY1. Although all these assigning methods can result in good arrangement of pads, but each of them performs different qualities. For the GP result of PRIMARY1, the smallest total wire length from the one initial pad and one way local assignment is 923560, 953611 for the one initial pad and one way global assignment, 905225 for the one initial pad and two ways assignment, and 957398 for the two initial pads and four ways assignment. The distribution of all the results obtained from above four assigning functions are presented in Figure 20, 21, 22 and 23 respectively.

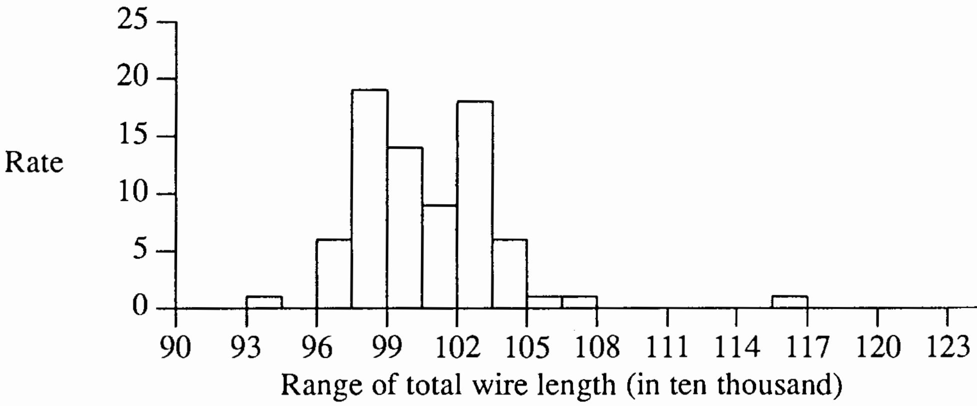


Figure 20. Rate of PRIMARY1 GP results from one initial pad and one way local assigning function.

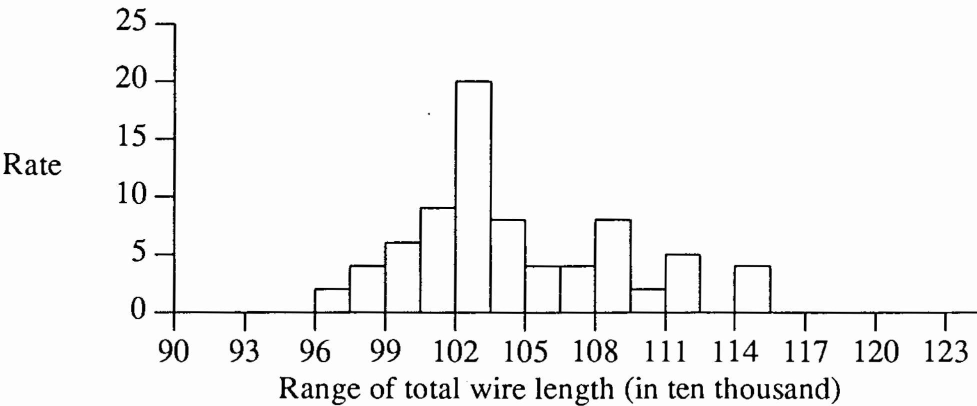


Figure 21. Rate of PRIMARY1 GP results from one initial pad and one way global assigning function.

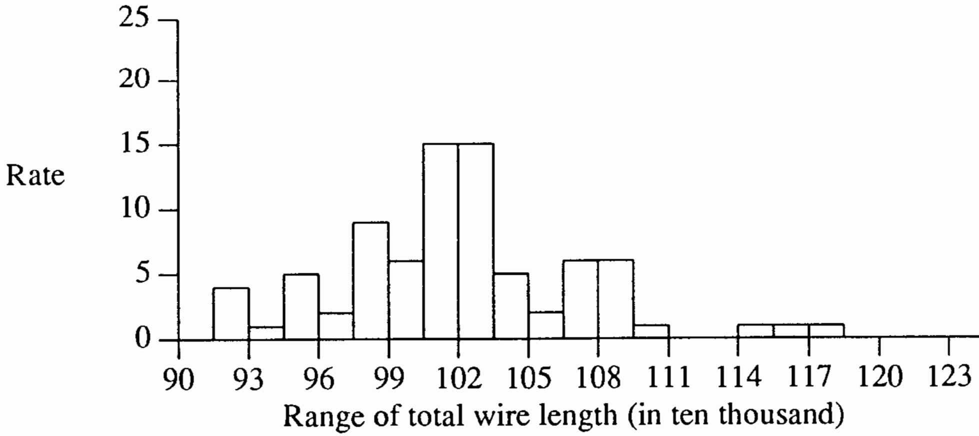


Figure 22. Rate of PRIMARY1 GP results from one initial pad and two ways assigning function.

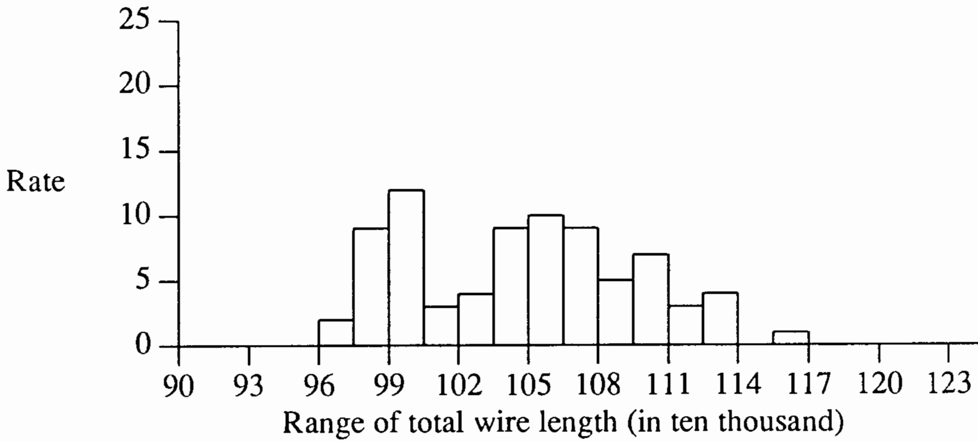


Figure 23. Rate of PRIMARY1 GP results from two initial pads and four ways assigning function.

The horizontal axis in the above four figures represents the range of total wire length and the vertical axis represents the rate of rings which result in the answers in the range. It is observed that the one initial pad and two ways assigning function can result in the shortest total wire length and most of the rings locate in the ranges of small value of the total wire length. Therefore, we use it in our I/O pad positions assignment program.

Selection Function

With as many as the number of pads, the two ways assigning function forms the same number of rings with different initial pads. So the question arrives: which ring is better? Due to the complicated mutual relationship among pads, the influential factors, which reflect the quality of arranged I/O pad positions, are hard to determine exactly. A selection function with five factors as measures is used to select better rings. These five factors are: nearer-pad-pair's weight cost, nearer-pad-pair's depth cost, farthest-pad-pair's weight cost, farthest-pad-pair's depth cost and block-pad-pair's weight cost.

Definition 11: Nearer-pad-pair's weight cost $NWC(p_i, p_j)$ is the weight cost between pad p_i and pad p_j, where pad p_j represents the pads which are close neighbors of pad p_i in a ring.

Definition 12: Farthest-pad-pair's weight cost $FWC(p_i, p_j)$ is the weight cost between pad p_i and pad p_j, where pad p_j represents the pad that locates farthest away from pad p_i in a ring.

Definition 13: Nearer-pad-pair's depth cost $NWC(p_i, p_j)$ is the depth cost between pad p_i and pad p_j, where pad p_j represents the pads which are close neighbors of pad p_i in a ring.

Definition 14: Farthest-pad-pair's depth cost $FWC(p_i, p_j)$ is the depth cost between pad p_i and pad p_j , where pad p_j represents the pad that locates farthest away from pad p_i in a ring.

Definition 15: Block-pad-pair's weight cost $BWC(p_i, p_j)$ is defined as the weight cost of pads in the same block and the weight cost of pads in the opposite diagonal blocks after the partition corresponding to the partitioning algorithm in PROUD.

The first factor is the sum of weight costs between every nearer-pad-pair. The nearer-pad-pair is defined as two pads located close to each other on a physical chip. But how far the pads can be away from each other to be considered close pads? It is hard to determine. From experiments, the nearer-pad-pair is defined according to the number of partitions in PROUD.

$$\textit{The farthest distance} \text{ for } \textit{nearer pads} \equiv 2^{\textit{number of cuts}}$$

All the pads inside the farthest distance are considered strongly related pads. In Figure 24, pad 2, pad 3, pad 4 and pad 5 are the nearer pads for pad 1 when the number of cuts in PROUD is 2.

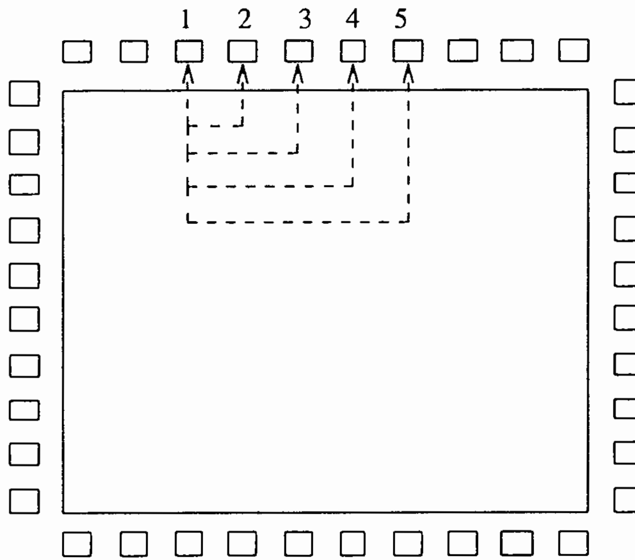


Figure 24. Nearer-pad-pairs.

Since the objective of the assigning function is to assign the strongly related pads as close to themselves as possible, the smaller sum of *NWC* s reflects better arrangement of pads. The sums of *NWC* s versus rings with different initial pads for PRIMARY1 is presented in Figure 25.

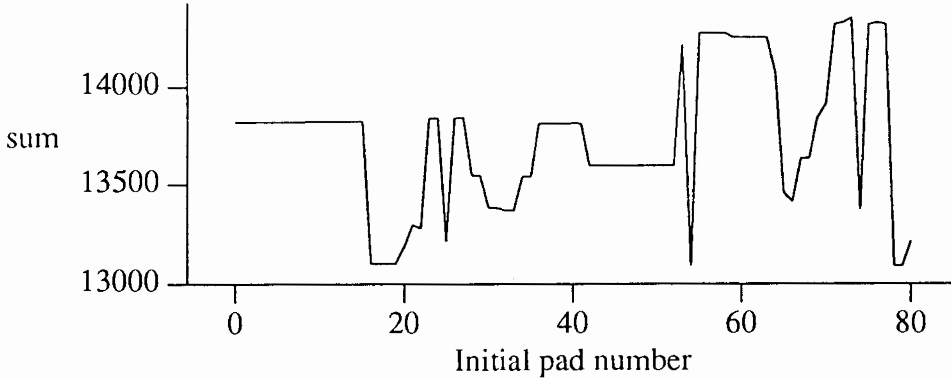


Figure 25. Sums of NWCs vs different rings for PRIMARY1.

Comparing Figure 25 to Figure 15, it is observed that some smaller values of the cost sum correspond to better results of the total wire length. In addition to assigning strongly related pads close to each other, another important objective for assigning function is to keep weakly related pads away from each other. Farthest-pad-pair's weight cost $FWC(p_i, p_j)$ represents the relation. Due to the rectangle shape of a chip, the farthest distance for every farthest-pad-pair is different. A rough estimation for the farthest distance is defined as:

The farthest-pad-pair is two pads located half of the number of pads away from each other.

Figure 26 shows the farthest-pad-pairs, which are specified by two arrows directed lines. The sum of all farthest-pad-pairs' weight costs should be large to keep the unrelated pad-pairs far away from each other. Therefore, the more negative sum of FWCs reflects better arrangement of pads. Figure 27 shows the negative sums of FWCs versus rings with different initial pads for PRIMARY1. The definitions used for NWC and FWC are applied to NDC and FDC respectively by substituting the weight cost to depth cost. The sums of these two factors versus rings with different initial pads are presented in Figure 28 and Figure 29 for PRIMARY1 respectively.

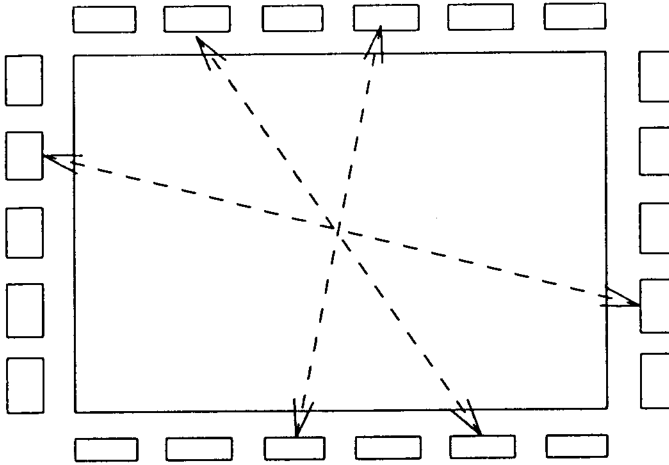


Figure 26. Farthest-pad-pairs.

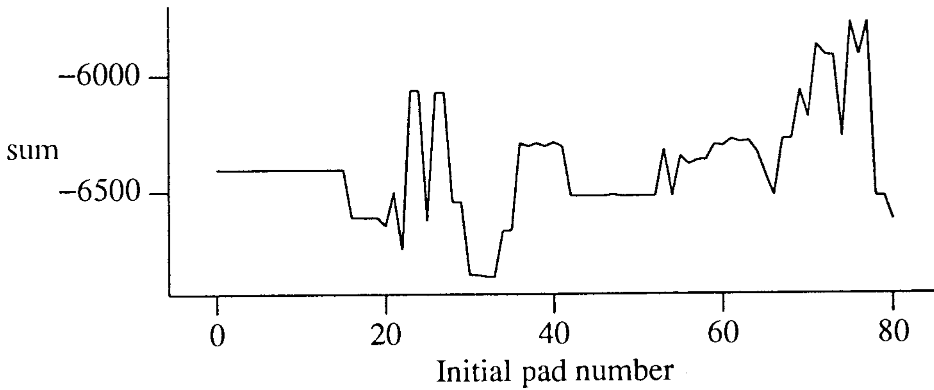


Figure 27. Negative sums of FWCs vs different rings for PRIMARY1.

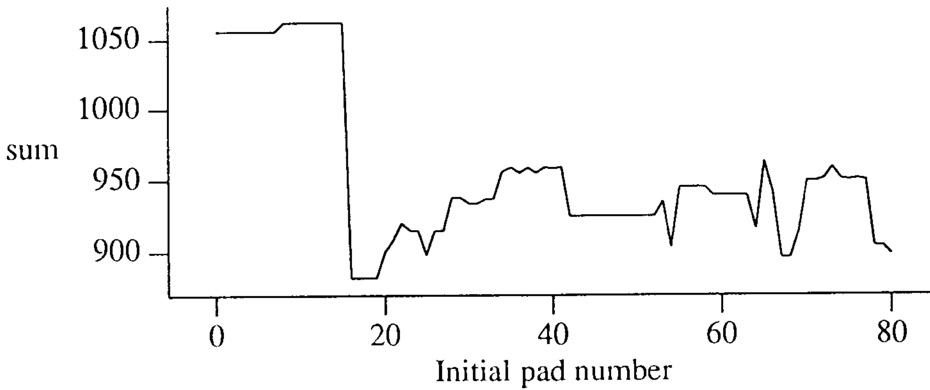


Figure 28. Sums of NDCs vs different rings for PRIMARY1.

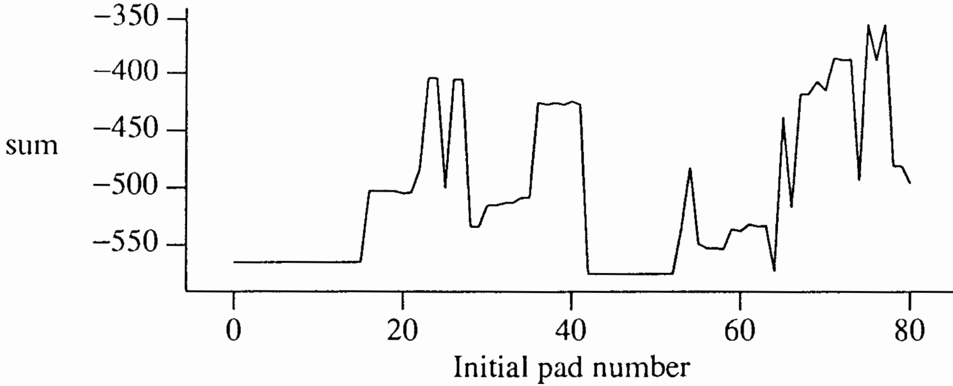


Figure 29. Negative sums of FDCs vs different rings for PRIMARY1.

Furthermore, due to the partitioning schedule in PROUD and the way we assign pads to physical locations on the chip, the pads in the same partition block should be much strongly related. It means that the weight cost between each pad-pair in the same block needs to be small. The pads in the same partition block I and block II are shown in Figure 30. On the other hand, the pad-pairs in the different diagonal blocks should be weakly related. So, the weight cost between each diagonal pad-pair, like the pads in block I with the pads in block II, needs to be large. When the partition continues and the blocks become smaller, like block III and IV, the same condition is still applied to each smaller blocks until the end of the partition.

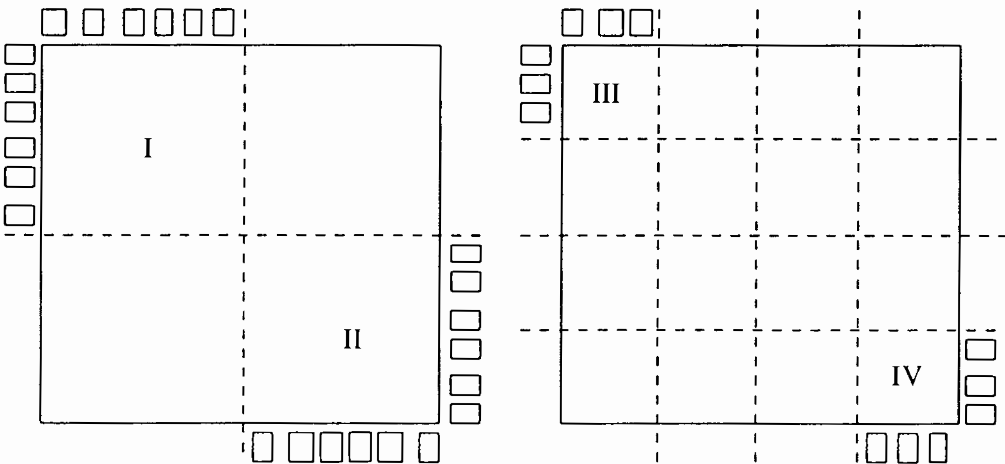


Figure 30. Example for block-pad-pairs.

The sum of BWC s of pads in the same block contributes positive reflection to the results of the total wire length. The sum of BWC s of pads in different diagonal blocks contributes negative reflection. Therefore, the smaller sum of positive BWC s and negative BWC s reflects better arrangement of pads. Figure 31 shows the sum of block-pad-pairs' weight costs versus rings with different initial pads for PRIMARY1.

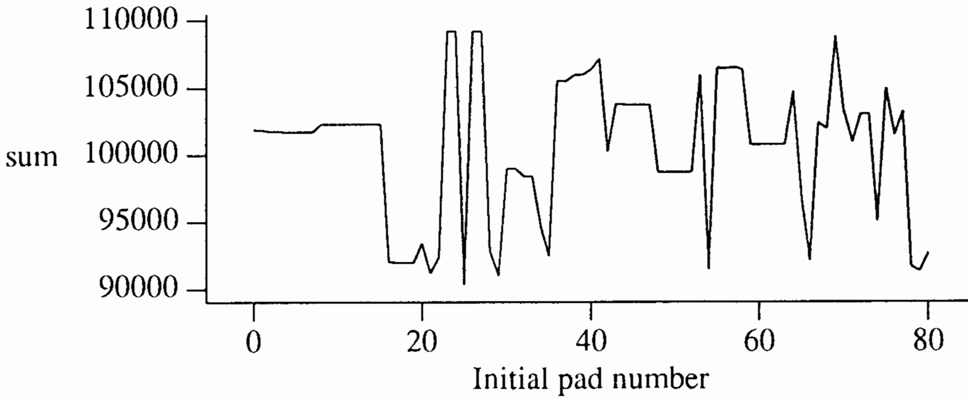


Figure 31. Sums of BWCs vs different rings for PRIMARY1.

A combination of these five factors is used to predict the goodness of a ring. The contribution of nearer-pad-pairs' weight cost $NWC(p_i,p_j)$, nearer-pad-pairs' depth cost $NDC(p_i,p_j)$ and the block-pad-pairs' weight cost $BWC(p_i,p_j)$ to the results of total wire length is positive. The other factors are negative. The weighted sum of these five factors is formulated as below:

$$FACTOR_SUM(ringx) = \sum NWC(p_i,p_j)_{ring(x)} + \frac{A}{B} \sum NDC(p_i,p_j)_{ring(x)} - \frac{A}{C} \sum FWC(p_i,p_j)_{ring(x)} - \frac{A}{D} \sum FDC(p_i,p_j)_{ring(x)} + \frac{A}{E} \sum BWC(p_i,p_j)_{ring(x)}$$

in which

$$A = Max\,[\sum NWC(p_i,p_j)]_{ring(a)} - Min\,[\sum NWC(p_i,p_j)]_{ring(b)}$$

$$B = Max\,[\sum NDC(p_i,p_j)]_{ring(c)} - Min\,[\sum NDC(p_i,p_j)]_{ring(d)}$$

$$C = Max\ [\ \sum\ FWC\ (p_i, p_j)\ ]_{ring(e)} - Min\ [\ \sum FWC\ (p_i, p_j)\ ]_{ring(f)}$$

$$D = Max\ [\ \sum FDC\ (p_i, p_j)\ ]_{ring(g)} - Min\ [\ \sum FDC\ (p_i, p_j)\ ]_{ring(h)}$$

$$E = Max\ [\ \sum BWC\ (p_i, p_j)\ ]_{ring(g)} - Min\ [\ \sum BWC\ (p_i, p_j)\ ]_{ring(h)}$$

Due do the different ranges of each factor's value, the coefficient A, B, C and D are used to normalize these factors. The sum of the above described five factors is used as the selection function and tested on benchmark examples PRIMARY1 and PRIMARY2 for rings with different initial pads obtained from assigning function. The results of FACTOR_SUM versus rings with different initial pads for PRIMARY1 and PRIMARY2 are shown in Figure 32 and Figure 33 respectively.

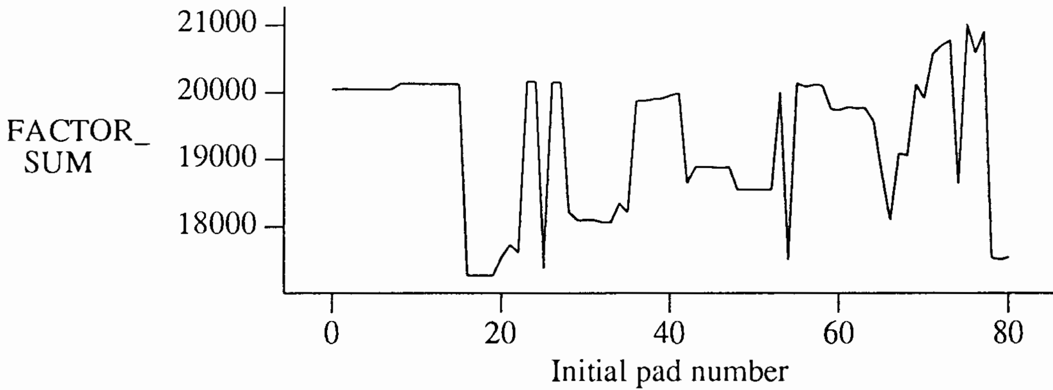


Figure 32. Ring's FACTOR_SUM vs different rings for PRIMARY1.

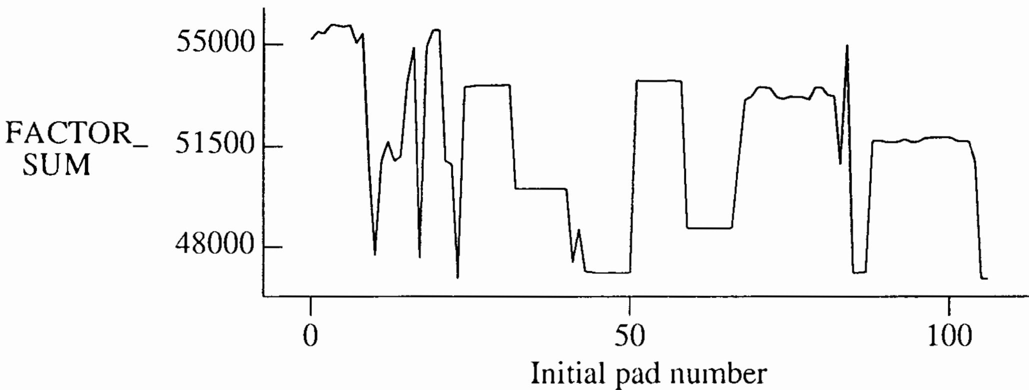


Figure 33. Ring's FACTOR_SUM vs different rings for PRIMARY2.

Comparing Figure 32 and 33 to Figure 15 and 16 respectively, the smaller values of *FACTOR_SUM* correspond to the shorter total wire lengths. For example, the ring with initial pad 17 has the smallest value of FACTOR_SUM in Figure 32, and the same ring in Figure 15 corresponds to the second shortest total wire length. The ring with initial pad 75 in Figure 32 has the largest FACTOR_SUM, and the same ring in Figure 15 corresponds to the much longer total wire length than the others.

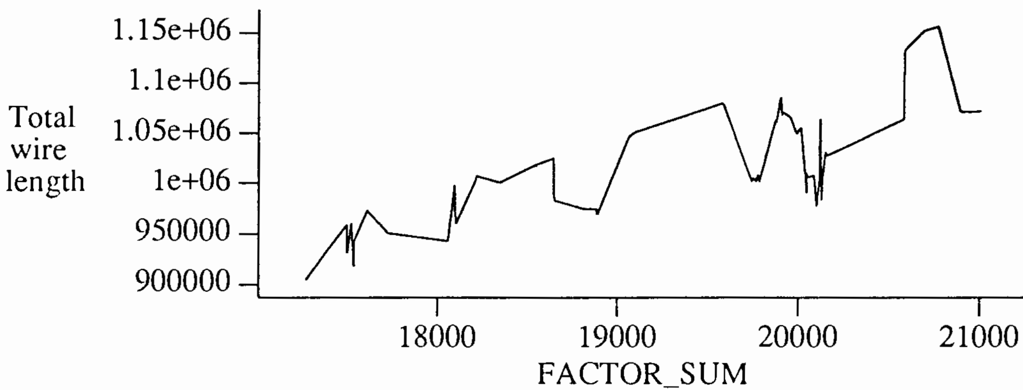


Figure 34. Total wire length of GP vs FACTOR_SUM for PRIMARY1.

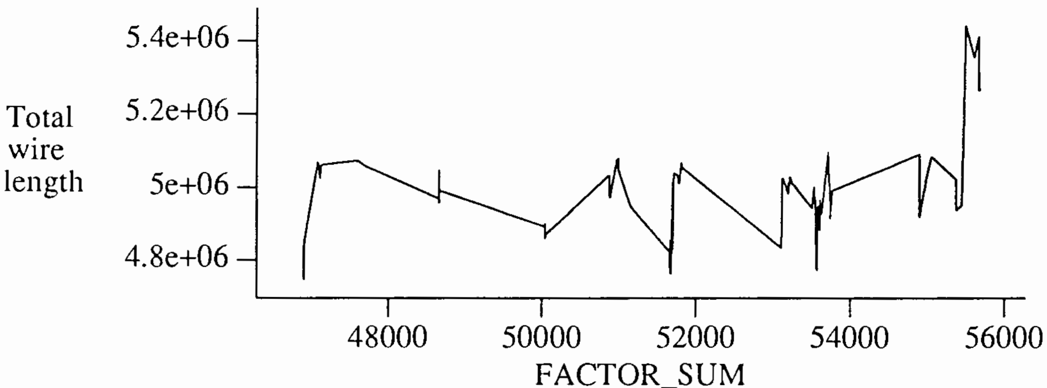


Figure 35. Total wire length of GP vs FACTOR_SUM for PRIMARY2.

Figure 34 shows the total wire length versus the FACTOR_SUM of each ring for PRIMARY1, and Figure 35 is for PRIMARY2. The above situation of the smaller FACTOR_SUM corresponding to the smaller total wire length is not so significant in

Figure 35 because the values of the total wire length for most of the rings are very close and most of them can be assumed as good answers. However, the rings with much larger values of FACTOR_SUM, which will result in the longer total wire length, still can be distinguished from the rings with smaller values of FACTOR_SUM, which will result in the shorter total wire length. Above two figures have demonstrated that good arrangements of pads can be found from a group of rings, which are formed by assigning function, by choosing rings with smaller values of FACTOR_SUM in the selection function. One thing is observed from the comparison of these figures that not every ring with smaller value of FACTOR_SUM reflects a better result of the total wire length. Some rings with larger values of FACTOR_SUM can result in shorter total wire lengths than the rings with smaller values of FACTOR_SUM. This is because there are still other factors should be considered. The factors that have been used in the selection function are only a rough estimation. Therefore, in the selection function, several rings with the smaller values of FACTOR_SUM are selected, and the best arrangement of pads will be one of these selected rings, which will result in the shortest total wire length.

Output

After the good arrangement of pads has been determined by the selection function, a design file with the I/O pad positions specified is produced. Then this design file is used as the boundary condition for PROUD.

COMPLEXITY

The complexity of the I/O pad positions assignment program can be described in four parts: input and output, calculation of cost factors, pad assigning function and ring selection function. The complexity for the input and output is linear to the number of pads and modules. The runtime for input and output is 7.82% of the total runtime for PRIMARY1, and 3.3% for PRIMARY2. In the calculation of cost factors, the shortest

paths for every pad to the other pads have to be found. The complexity for searching the shortest path on the sparse graph is $O((E+V)\log V)$ [21], where E is the number of connections among modules and pads, and V is the number of modules and pads. Since this search has to be repeated for P pads, the total complexity for the calculation of cost factors is $O(P(E+V)\log V)$. The runtime for the calculation of cost factors in PRIMARY1 is 70.60% of the total runtime, and 88.94% for PRIMARY2. In the assigning function, every pad is chosen as the initial pad and P different rings are formed. The complexity for each function: *NEXT_CANDIDATE*() and *SEARCH*(), is linear to Q, where Q is the number of unassigned pads. Each pad is assigned to its position by using above two functions, then the complexity for each pad assignment is $O(Q^2)$. Or it could be expressed as $O(P^2)$. Since each ring has P pads, the complexity for each ring is $O(P^3)$. Therefore, for the formation of P rings, the total complexity is $O(P^4)$. Five factors are used in the selection function to calculate the FACTOR_SUM for each ring. The complexity for the factor of NWC and NDC is $O(P)$. The complexity for the factor of FWC and FDC is $O(P)$. The complexity for the factor BWC is $O(P^2)$. Since P rings' FACTOR_SUMs are calculated, the total complexity for the selection function is $O(P^3)$. The runtime for the assigning and selection function is 21.3% of the total runtime for PRIMARY1, and 7.66% for PRIMARY2. Due to $P << V+E$ for large circuits, the complexities of input and output, pad assigning function and ring selection function are much smaller than the complexity of the cost calculation. It is observed that the calculation of cost factors use most of the runtime in the benchmark examples. When the number of modules becomes large, the above situation becomes more significant, then the complexity of this assignment program is dominated by the calculation of cost factors.

CHAPTER VI

TEST RESULTS

The I/O pad positions assignment program is written in C programming language. It is used as the predecessor of PROUD to locate the I/O pad positions on the chip. The source code is presented in APPENDIX B. We tested our pad assignment program on MCNC layout benchmark examples PRIMARY1 and PRIMARY2 on the SUN SPARC workstation. The chip area for PRIMARY1 is 5420μm x 4320μm and the modules are placed in 17 rows. For PRIMARY2, the chip area is 9240μm x 9080μm and the modules are placed in 29 rows. The specifications of these two benchmark examples are tabulated in Table I.

TABLE I

BENCHMARK SPECIFICATIONS

Example	No. of pads	No. of modules	No. of nets	No. of pins
PRIMARY1	81	752	1239	3303
PRIMARY2	107	2907	3773	12014

From the output of our pad assignment program, we chose the first three rings with the smallest FACTOR_SUM and then used PROUD to calculate the total wire length for each example. The shortest total wire length resulted from one of these three rings was chosen as the best result. The results of total wire length and execution time from our pad assignment program as predecessor, the results from [1] and the bad results that we could find are tabulated in Table II for PRIMARY1 example and Table III for PRIMARY2 example.

TABLE II

RESULTS COMPARISON FOR PRIMARY1

PRIMARY1 Example				
Algorithm	Total Wire Length (μm)		Runtime (seconds)	
	Global Placement	Detail Placement	GP	DP
Original PROUD	1017824	826096	10.44	89.45
With pad predecessor	905596	749311	26.26	115.71
Bad result	1286258	1022676		

TABLE III

RESULTS COMPARISON FOR PRIMARY2

PRIMARY2 Example				
Algorithm	Total Wire Length (μm)		Runtime (seconds)	
	Global placement	Detail placement	GP	DP
Original PROUD	5319366	4610701	74.23	915.34
With pad predecessor	4746728	4106832	217.9	1060.04
Bad result	6247443	5243377		

For PRIMARY1 a range of 11.02% total wire length reduction has been reached, compared to the result from [1] without the I/O pad assignment predecessor. A 29.59% reduction in the total wire length was observed when comparing the best result determined by our method with the bad result that could be obtained by choosing bad relative pad locations. For PRIMARY2 the ranges are 10.76% and 24.02% respectively. The runtime that has been specified in the above tables for the pad predecessor indicates the time needed to run our pad assignment program and PROUD program once. Since we selected three rings and tested them, the total runtime for using our pad predecessor should be three times of the specified value in the table. However, the runtime for only our pad assignment program is 14.84 seconds for PRIMARY1, and 142.27 seconds for PRIMARY2. If the parallel computer is available, then we can use the three chosen rings as the three different boundary conditions and run PROUD simultaneously for these three

cases. The runtimes for the benchmark examples will be the values specified in the above tables. In that case, one thing is observed from Table III that the global placement result obtained by using our pad assignment program as predecessor for PROUD is close to the detail placement result obtained by the original PROUD program. But the execution time to obtain the detail placement result is four times the execution time needed to obtain the similar global placement result with our pad predecessor. We believe that for circuits with large number of pads and modules, we do not have to run the most time consuming process of detail placement, and still a good result can be obtained at the global placement stage using our assignment program as predecessor. Also we test the all possible I/O pad arrangements from the one initial pad and two way assigning function for PRIMARY1. With 81 different rings, due to different initial pads, and 41 switching possibilities for every ring, the total number of possible arrangements of pads is 3321. The possible results together with the best results, dashed lines, found by using our final I/O pad positions assignment program are presented in Figure 36. This figure demonstrates that the I/O pad arrangement, found by our I/O pad positions assignment program, will lead to very good results.

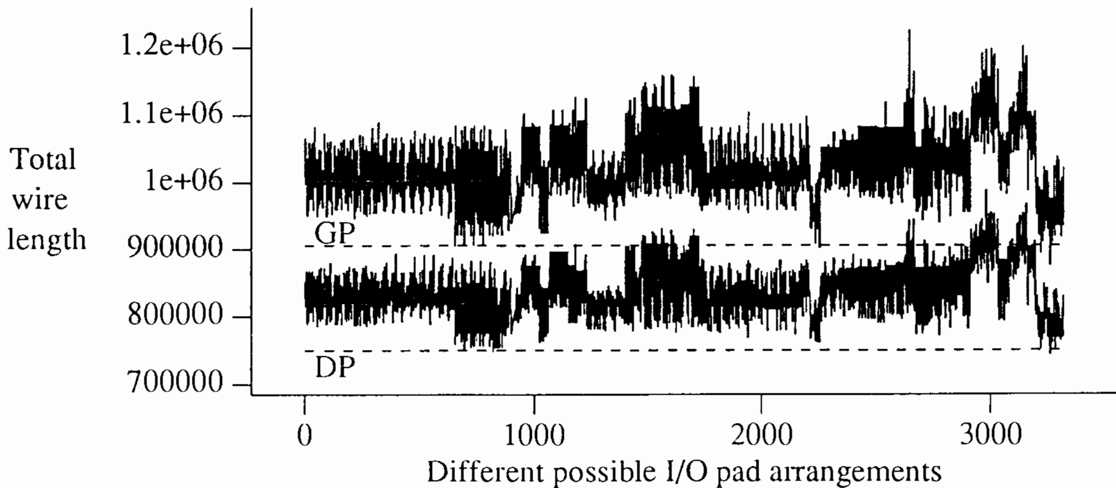


Figure 36. Results of different I/O pad arrangement for PRIMARY1.

In addition to the above two benchmark examples, we tested our algorithm on other circuits. Their specifications are presented in Table IV.

TABLE IV

EXAMPLE SPECIFICATIONS

Example	No. of pads	No. of modules	No. of nets	No. of pins
CIRCUIT_1	8	16	20	51
CIRCUIT_2	24	125	144	478
CIRCUIT_3	53	1404	1660	5535
CIRCUIT_4	60	2721	3033	11169
CIRCUIT_5	188	3659	3212	15317

The results of total wire length and execution time from our pad assignment program as predecessor are compared to the average results and bad results that we could obtained. They are shown in Table V, VI, VII, VIII and IX for CIRCUIT_1, CIRCUIT_2, CIRCUIT_3, CIRCUIT_4 and CIRCUIT_5 respectively. The average result is obtained by random pad assignment without pad predecessor. Like the benchmark example, the runtime shown in these tables only indicates one execution of PROUD program with pad predecessor or without.

TABLE V

RESULTS COMPARISON FOR CIRCUIT_1

CIRCUIT_1 Example				
Algorithm	Total Wire Length (µm)		runtime (seconds)	
	GP	DP	GP	DP
With pad predecessor	760	737	0.26	0.29
Average result (PROUD)	823.7	799.1	0.09	0.12
Bad result	900	883		

TABLE VI

RESULTS COMPARISON FOR CIRCUIT_2

CIRCUIT_2 Example				
Algorithm	Total Wire Length (μm)		Runtime (seconds)	
	GP	DP	GP	DP
With pad predecessor	39515	34155	1.33	4.16
Average result (PROUD)	44295.4	36587.5	0.53	3.36
Bad result	49392	38939		

TABLE VII

RESULTS COMPARISON FOR CIRCUIT_3

CIRCUIT_3 Example				
Algorithm	Total Wire Length (μm)		Runtime (seconds)	
	GP	DP	GP	DP
With pad predecessor	2129526	1945374	42.78	62.97
Average result (PROUD)	2337163.2	2122328.8	18.52	63.02
Bad result	2791816	2511614		

TABLE VIII

RESULTS COMPARISON FOR CIRCUIT_4

CIRCUIT_4 Example				
Algorithm	Total Wire Length (μm)		Runtime (seconds)	
	GP	DP	GP	DP
With pad predecessor	5752991	4767442	183.62	949.05
Average result (PROUD)	6108668.9	5051713.1	84.72	765.43
Bad result	6583660	5333168		

TABLE IX

RESULTS COMPARISON FOR CIRCUIT_5

CIRCUIT_5 Example				
Algorithm	Total Wire Length (μm)		Runtime (seconds)	
	GP	DP	GP	DP
With pad predecessor	8622457	7273966	686.2	1891.44
Average result (PROUD)	9392480.3	7718809.3	123.97	1329.17
Bad result	9870649	8026015		

For CIRCUIT_1 a range of 7.77% total wire length reduction has been reached, compared to the average result, and a 16.53% reduction, compared to the bad result, is observed. For CIRCUIT_2, CIRCUIT_3, CIRCUIT_4 and CIRCUIT_5 the ranges are 6.65% and 12.28%, 8.34% and 22.54%, 5.62% and 9.37%, 5.76% and 9.37% respectively.

CHAPTER VII

CONCLUSION AND FUTURE WORK

The proposed I/O pad positions assignment algorithm is applied to determine the I/O pad positions for the sea-of-gates placement program PROUD. A heuristic searching method is used to find good solutions. Two cost factors have been defined to represent the connectivity strength between pads. By assigning strongly related pads close to each other, we have achieved an excellent improvement in the total wire length for a number of examples. The results obtained by using our pad assignment program have been 11.02% and 10.76% better than the results from [1] on benchmark examples, PRIMARY1 and PRIMARY2. For circuits with large number of modules and pads, we noticed that most of the rings formed by the assigning function have good arrangement of pads, and some rings with larger values of the FACTOR_SUM in the selection function can result in shorter total wire length. The above situation can be observed from the results of PRIMARY2 example. However, the good arrangement of pads still can be determined by choosing the ring with smaller value of FACTOR_SUM, although it is possible that the chosen ring does not have the best arrangement of pads. In PROUD program, the detail placement process will take most of the execution time for large circuits. However, when the number of modules and pads is large, the result of total wire length obtained at the global placement stage in PROUD using our pad assignment as predecessor is also a good answer. Therefore, we believe that a good result can be obtained at the global placement stage in shorter time for circuits with extremely large number of modules and pads by using the pad assignment predecessor if the parallel computer is available. Our pad assignment algorithm used as the predecessor to PROUD, which solves the sea-of-gates placement problem, has revealed excellent im-

provement on the total wire length.

In this algorithm the assignment of the pad to its position is determined by the cost among pads. The cost factors, weight cost and depth cost, represent the connectivity strength between pads in our algorithm. In the one initial pad and two ways assigning function, the chances for two host pads compete for the same candidate pad are high. This means that these two cost factors are not enough to distinguish the connectivity relationship clearly between pads. Besides, in the assigning function, the assigning decision depends on how close the pad-pairs are related. Another possible assigning decision of keeping weakly related pads away from each other is not included into consideration. Some other cost factors, which represent the weak relationship between pads, should be explored. We expect that using the cost factors, which represent the strongly related pads and the cost factors which represent the weakly related pads, the assigning function can not only place the most related pads close together but also at the same time keep the most unrelated pads away from each other.

This pad assignment program does not consider the allocation of the ring's components to the physical locations on the chip. A little improvement is assumed to be reached by finding a good solution to the allocation problem. Moreover, the five factors used in the selection function are only roughly estimated. Maybe some other factors which can reflect the quality of a ring could be explored to determine the ring with the best arrangement of pads. Another extension of this pad assignment program is to add the path-delay constraints into consideration. Especially in the sea-of-gates designs with large number of gates, the timing constraint becomes more important. Possibly the predefined locations for some specific pads, the restriction for some pads to be placed close to one another or not evenly distributed pad positions on the chip could be added as the initial constraints. We hope to extend our work to include these considerations.

REFERENCES

[1] Ren-Song Tsay, Ernest S. Kuh, Chi-Ping Hsu, "PROUD: A Sea-of-Gates Placement Algorithm", *IEEE Design & Test of Computers*, December 1988, pp. 44-56.

[2] C. K. Cheng and Ernest S. Kuh, "Module Placement Based on Resistive Network Optimization", *IEEE Trans. Computer Aided Design*, vol. CAD-3, No. 3, July 1984, pp 218-225.

[3] Ernest Meyer, "Jumbo Arrays Scale 250,000gates", *Computer Design Journal,* March 1, 1990, pp. 28-36.

[4] Jon Gabay, "A Burst of Activity in Sea-of-Gates Arrays", *Semicustom Design Guide,* No. 26, 1989, pp. 26-33.

[5] G. D. Adams, C. H. Sequin, "Template Style Consideration for Sea-of-gates Layout Generation", *26th ACM/IEEE Design Automation Conference*, 1989, pp. 31-36.

[6] M. Murofushi, M. Yamada, T. Mitsuhashi, "FOLM-planner: A New Floorplanner with a Frame Overlapping Floorplan Model Suitable for SOG (Sea-of-gates) Type Gate Arrays", *IEEE Custom Integrated Circuit Conference*, 1990, pp. 140-143.

[7] M. Igusa, M. Beardslee, A. Sangiovanni-Vincentelli, "ORCA: A Sea-of-gates Place and Route System", *26th ACM/IEEE Design Automation Conference*, 1989, pp. 122-127.

[8] M. Murayama, Y. Matsuda, K. Yoshida, H. Ooka, "A 177k Gate 150 ps CMOS SOG with 1856 I/O Buffers", *IEEE Custom Integrated Circuits Conference,* 1989, pp. 8.1.1-8.1.4.

[9] Hidetoshi Onodera, Yo Taniguchi, Keikichi Tamaru, "Branch-and-Bound Placement for Building Block Layout", *28th ACM/IEEE Design Automation Conference*, 1991, pp. 433-439.

[10] B. W. Kerninghan, S. Lin "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell Systems Technical Journal,* 49(2):292-307, 1970.

[11] C. M. Fiduccia, R. M. Mattheyses, "A Linear Time Heuristic for Improving network partitions", *ACM/IEEE Proceedings of the 19th Design Automation Conference,* 1982, pp. 175-181.

[12] Ching-Wei Yeh, Chung-Kuan Cheng, "A General Purpose Multiple Way Partitioning Algorithm", *28th ACM/IEEE Design Automation Conference*, 1991, pp. 421-426.

[13] C.Sechen, Alberto Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package", *IEEE Journal of Solid-State Circuits*, vol. sc-20, No. 2, April 1985. pp. 510-522.

[14] Abhijit Chatterjee, Richard Hartley, "A New Simultaneous Circuit Partitioning and Chip Placement Approach Based on Simulated Annealing", *27th ACM/IEEE Design Automation Conference*, 1990, pp. 36-39

[15] Ralph-Michael Kling, Prithviraj Banerjee, "ESP: A New Standard Cell Placement Package Using Simulated Evolution", *in Proc. 24th ACM/IEEE Design Automation Conference*, June 1987, pp. 60-66.

[16] Ralph-Michael Kling, Prithviraj Banerjee, "Optimization By Simulated Evolution With Applications To Standard Cell Placement", *in Proc. 27th ACM/IEEE Design Automation Conference*, 1990, pp. 20-25.

[17] Ralph-Michael Kling, Prithviraj Banerjee, "Empirical and Theoretical Studies of the Simulated Evolution Method Applied to Standard Cell Placement", *IEEE Transactions on Computer-Aided Design*, vol. 10, No. 10, October 1991. pp. 1303-1315.

[18] J. M. Kleinhans, G. Sigl, F. M. Johannes, K. J. Antreich, "GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization", *IEEE Transaction on Computer-Aided Design*, vol. 10, No. 3, March 1991, pp. 356-365.

[19] D. C. Wang, "Pad Placement and Ring Routing for Custom Chip Design", *27th ACM/IEEE Design Automation Conference*, 1990, pp. 193-199.

[20] Massoud Pedram, Kamal Chaudhary, Ernest S. Kuh, "I/O Pad Assignment Based on the Circuit Structure", *IEEE International Conference on Computer Design: VLSI in Computer* and Processor", 1991, pp. 314-318.

[21] Robert Sedgewick, "Algorithm In C", *Addison-Wesley Publishing Company*, 1990.

APPENDIX A

ONE INITIAL PAD AND TWO WAYS ASSIGNING FUNCTION ALGORITHM

```
BEGIN
{

   assign the initial pad i randomly;

   assign the second pad j with the smallest cost[i][j] and depth[i][j];

   iteration until all pads have been assigned{

      r = NEXT_CANDIDATE ( i );   l = NEXT_CANDIDATE ( j );

      if( r ≡ l ){

         if( cost[r][i] < cost[l][j] && SEARCH(r,i) < 2) assign r next to i;

         else if( cost[r][i] > cost[l][j] && SEARCH(l,j) < 2 ) assign l next to j;

         else if( cost[r][i] == cost[l][j] && SEARCH(r,i) < 2 ){

            if( depth[r][i] < depth[l][j] ) assign r next to i;

            else if( depth[r][i] > depth[l][j] ) assign l next to j;

            else assign to each one by turn;
         }
         else RESERVE(r);

      }

      else if( r != l ){

         if( SEARCH(r,i) < 2 ) assign r to i;
         else RESERVE(r);

         if( SEARCH(l,j) < 2 ) assign l to j;
         else RESERVE(l);
      }

      if( r or l has been assigned ) RELEASE();
   }

} END


NEXT_CANDIDATE(x) {
    return ( pad y with the smallest cost[x][y] to pad x );
}


SEARCH(x,y) {
```

```
    search whole available pads{
      find pad z with cost[z][x] < cost[x][y]; amount++;
    }
    return ( amount );
}

RESERVE (x,y) {
    put x and y into list which is not used as next candidate;
}

RELEASE () {
    release all the pads reserved before;
}
```

APPENDIX B

I/O PAD POSITIONS ASSIGNMENT PROGRAM

```
/*****************************************************************/

I/O pad positions assignment program

INPUT: Original design file for PROUD and pad position file.

/*****************************************************************/

#include <stdio.h>
#include <string.h>
#include "global.h"
#include <sys/time.h>
#include <sys/resource.h>

main( argc, argv )
int   argc;
char **argv;
{

FILE      *fp;
int   i;
int   j;
int   good_ring;
float Begintime;
float temptime;
float Time();

 printf("0    Pad Assignment Program       by Shyang-Kuen Her       1/7/920);
 printf("0);

 Begintime = Time();
 temptime = Time();

 Readin( argv[1] );               /* read in data */

 printf("Readin:                  %f seconds.0, Time() - temptime);
 temptime = Time();

 Weight();                    /* cell's weight */

 Listpfs();                 /* cost factors */

 printf("Listpfs:                 %f seconds.0, Time() - temptime);
 temptime = Time();
```

```
 good_ring = Select();              /* Selecting a good ring */

 printf("Selection:              %f seconds.0, Time() - temptime);
 temptime = Time();

 Print_out(good_ring, argv[1], argv[2]);  /* new input for PROUD */

 printf("Print_out:              %f seconds.0, Time() - temptime);
 printf("0);
 printf("Pad Assignment:              %f seconds.0, Time() - Begintime);

}


Time( )
{

 struct rusage rusage;
 float  time;

 getrusage(RUSAGE_SELF, &rusage);

 time = (float) rusage.ru_utime.tv_sec +
      (float) rusage.ru_utime.tv_usec/1.0e6;

 time += (float) rusage.ru_stime.tv_sec +
       (float) rusage.ru_stime.tv_usec/1.0e6;

 return( time );

}
```

```
/*******************************************************************/

Read Input Function

Description: Read in original design file for PROUD and format it
             into cell adjcent list format. Net number can not be "0".

Note: The nets are valued from 1 to NrPin, when nets are valued 0 or
      character it means floating pins.

/*******************************************************************/

#include<stdio.h>
#include "global.h"

Readin(ExampleFile)
char ExampleFile[30];
{

FILE  *fp;
int     i;
int     j;
char   buffer[40];
struct node   *temp_list;
struct node   *net_adj[Nrnets], *temp_adj;
struct node   *z;

  /* read in the original design file for PROUD */

  fp = fopen(ExampleFile, "r");

  fgets(buffer, 40, fp);
  sscanf(buffer, "%s %d0, name1, &nrCols);
  fgets(buffer, 40, fp);
  sscanf(buffer, "%s %d0, name2, &nrRows);
  fgets(buffer, 40, fp);
  sscanf(buffer, "%s %d0, name3, &bcX);
  fgets(buffer, 40, fp);
  sscanf(buffer, "%s %d0, name4, &bcY);
  fgets(buffer, 40, fp);
  sscanf(buffer, "%s %d0, name5, &xGrid);
  fgets(buffer, 40, fp);
  sscanf(buffer, "%s %d0, name6, &yGrid);
  fgets(buffer, 40, fp);
  sscanf(buffer, "%s %s0, name7, name8);
```

```
fgets(buffer, 40, fp);
sscanf(buffer, "%s %d0, name9, &nrIos);
fgets(buffer, 40, fp);
sscanf(buffer, "%s %d0, name10, &nrMods);
fgets(buffer, 40, fp);
sscanf(buffer, "%s %d0, name11, &nrNets);
fgets(buffer, 40, fp);
sscanf(buffer, "%s %d0, name12, &nrPins);

z = (struct node *)malloc(sizeof *z);
z->next = z;

for( i=0; i< Nrnets; i++) net_adj[i]=z;

/* read in pad data */

for(j=0; j< Nrpad; j++){
 pad[j] = (struct nodes *)malloc(sizeof(struct nodes));
 fgets(buffer, 40, fp);
 sscanf(buffer, "%d %s %d %d %d %d %d %d0, &pad[j]->id,
      pad[j]->name,&pad[j]->type, &pad[j]->obstacle,
      &pad[j]->nrpin,&pad[j]->x,&pad[j]->y, &pad[j]->pattern);

 for(i=0; i<pad[j]->nrpin; i++){
  nets = (struct NETS *) malloc(sizeof *nets);
  pad[j]->net[i] = nets;
  fgets(buffer, 40, fp);
  sscanf(buffer, "%d %d %d %d0,&pad[j]->net[i]->pin,
       &pad[j]->net[i]->order,&pad[j]->net[i]->id,
       &pad[j]->net[i]->ntype);

  if( pad[j]->net[i]->id > 0 ) {
   temp_adj = net_adj[pad[j]->net[i]->id];
   net_adj[pad[j]->net[i]->id]=(struct node *)malloc(sizeof(struct node));
   net_adj[pad[j]->net[i]->id]->v = pad[j]->id;
   net_adj[pad[j]->net[i]->id]->next = temp_adj;
} } }


/** read in module data **/

for(j=0; j< Nrmodule; j++){
 module[j] = (struct nodes *)malloc(sizeof(struct nodes));
 fgets(buffer, 40, fp);
 sscanf(buffer, "%d %s %d %d %d %d %d %d0, &module[j]->id,
```

```
        module[j]->name, &module[j]->type, &module[j]->obstacle,
        &module[j]->nrpin,&module[j]->x,&module[j]->y,&module[j]->pattern);

  for(i=0; i<module[j]->nrpin; i++){
   nets = (struct NETS *) malloc(sizeof(struct NETS));
   module[j]->net[i]=nets;
   fgets(buffer, 40, fp);
   sscanf(buffer, "%d %d %d %d0,&module[j]->net[i]->pin,
        &module[j]->net[i]->order,&module[j]->net[i]->id,
        &module[j]->net[i]->ntype);

   if( module[j]->net[i]->id > 0 ) {
    temp_adj = net_adj[module[j]->net[i]->id];
    net_adj[module[j]->net[i]->id] = (struct node *)malloc(sizeof
                              (struct node));
    net_adj[module[j]->net[i]->id]->v = module[j]->id+Nrpad;
    net_adj[module[j]->net[i]->id]->next = temp_adj;
} } }


/* build adjcent list */

y = (struct node *)malloc(sizeof(struct node));
y->next = y;

for( i=0; i < Maxv; i++) adj[i]=y;

/* pad adjcent list */

for(j = 0; j < Nrpad; j++){
 for(i=0; i < pad[j]->nrpin; i++){
  if( pad[j]->net[i]->id > 0 ) {
   for(g = net_adj[pad[j]->net[i]->id]; g != z; g = g->next){

     if(g->v != pad[j]->id) {
      temp_list = adj[pad[j]->id];
      adj[pad[j]->id] = (struct node *) malloc(sizeof (struct node));
      adj[pad[j]->id]->v = g->v;
      adj[pad[j]->id]->next = temp_list;
} } } } }

/* module adjcent list */

for(j = 0; j < Nrmodule; j++){
 for(i=0; i<module[j]->nrpin; i++){
```

```
    if( module[j]->net[i]->id > 0 ) {
     for(g = net_adj[module[j]->net[i]->id]; g != z; g = g->next){

       if(g->v != module[j]->id+Nrpad) {
         temp_list = adj[module[j]->id+Nrpad];
         adj[module[j]->id+Nrpad] = (struct node *)malloc(sizeof
                                 (struct node));
         adj[module[j]->id+Nrpad]->v = g->v;
         adj[module[j]->id+Nrpad]->next = temp_list;
  } } } } }

  fclose(fp);

}


/* calculate cell's weight */

Weight()
{

int     i;
int     count;
int     weight;

  for( i=0; i<Maxv; i++)
  {
    count=0;
    for( t = adj[i]; t != y; t=t->next)
      count = count+1;
    adj_weight[i] = count;
  }

}
```

/***/

Priority-First search is used to find the shortest path between two pads and determine the weight cost "cost[][]" and depth cost "level[][]".

/***/

```
Listpfs()
{

int     source, i;

  for(source = 0; source < Nrpad; source++) {
   Initialize();
   for(i = 0; i < Nrpad + Nrmodule; i++) val[i] = -2;
   val[source] = adj_weight[source];
   Pf_visit(source);
  }

}


Pf_visit(k)
int     k;      /* source pad */
{

int     m;
int     count;
int     depth;
int     father;

 Insert(k);              /* insert cells into priority queue */

 dad[k][k]=k; count=0;

 while ( Priq != pfs_end ) {   /* check up queue is not empty */

  m = Getdata();

  if(m < Nrpad)
  {
   cost[k][m] = val[m];     /* weight cost along the shortest path */
   depth = 0; count++;
   father = dad[k][m];
   while(father != k){
```

```
      depth++;
      father = dad[k][father];
     }

    level[k][m] = depth;     /* depth cost along the shortest path */

    if(count >= Nrpad) return;
   }

   for(t = adj[m]; t != y; t = t->next)
   {
    if(val[t->v] < 0){
      dad[k][t->v] = m;
      val[t->v] = adj_weight[t->v] + val[dad[k][t->v]] ;
      Insert(t->v);
     }
   }
  }

}


/****** utility functions for Priority first search *****/

Initialize()          /* initialize the priority queue */
{

 pfs_end = (struct node *) malloc(sizeof(struct node)) ;
 pfs_end->v = Maxv;
 val[Maxv] = REFERENCE_MAX;
 pfs_end->next = pfs_end;

 Priq = pfs_end;

}


Insert(p)           /* add the cell into the priority queue */
int     p;
{
struct node  *temp;
struct node  *former;
struct node  *new;

  if(Priq == pfs_end) {
```

```
    new = (struct node *) malloc(sizeof(struct node)) ;
    new->v = p ;
    new->next = pfs_end;
    Priq = new;
   }

  else if(val[p] < val[Priq->v]) {
    new = (struct node *) malloc(sizeof(struct node)) ;
    new->v = p ;
    new->next = Priq ;
    Priq = new;
   }

  else{
    temp = Priq;

    while( val[p] >= val[Priq->v] ) {
      former = Priq;
      Priq = Priq->next;
     }

    new = (struct node *) malloc(sizeof(struct node)) ;
    new->v = p ;
    new->next = Priq ;
    former->next = new;

    Priq = temp;
  }

}


int Getdata()              /* pick the first order of queue into tree */
{

int      data;
struct node   *temp;

  temp = Priq;
  data = Priq->v;
  Priq = Priq->next;
  free(temp);
  return data;

}
```

```
/****************************************************************/

One initial pad and two ways assigning function

/****************************************************************/

#include "global.h"
#include <stdio.h>

Assign(start)
int start;                    /*  first initial pad  */
{
FILE  *fp;
int   i;
int   l;
int   r;
int   count;
int   id;
int   end;
int   w;
int   sensor = 0;
int   sensor_ref = 2;
int   left_signal = 1;
int   l_timer;
int   r_timer;
int   r_counter = 0;
int   l_counter = 0;

 repeat = 0;                  /* counter for already assigned pads */

 for(i = 0; i<Nrpad; i++) value[i] = 0;

 right[0] = start;            /* assigning first initial pad */
 value[right[0]] = -1; repeat = 1;

 end = 0; id = 0;
 while(end == 0)              /* choose for the second host pad */
 {
  if(id >= Nrpad-1){
   left[0] = temp[0];
   repeat++;
   for(w = 0; w < id; w++) value[temp[w]] = 0;
   value[left[0]] = -1;
   break;
  }
```

```
 l = alc_next(right[r_counter]);
 sensor = 0; count = 0;
 while(count < Nrpad)
 {
  if(value[count] < 0) count++;
  else if(count != l && cost[l][count] < cost[right[0]][l] ){
   sensor++; count++;
  }
  else if(count != l && cost[l][count] == cost[right[0]][l] ){
   if(level[l][count] < level[left[l_counter]][l] ){
    sensor++; count++;
   }
   else count++;
  }
  else count++;
 }

 if(sensor >= sensor_ref){ value[l] = -1; temp[id++] = l; }
 else{
  left[0] = l; repeat = 2;
  for(w = 0; w < id; w++) value[temp[w]] = 0;
  end = 1; value[left[0]] = -1;
 }
}

/* two ways assigning process */

while(repeat < Nrpad)
{
 end = 0; id = 0; count = 0;
 while(end == 0)
 {
  if( id >= Nrpad-repeat && left_signal == 0 ){
   right[r_counter+1] = r; repeat++; r_counter++;
   for(w=1; w<id; w++) value[temp[w]]=0;
   value[r] = -1; left_signal = 1;
   break;
  }
  if( id >= Nrpad-repeat && left_signal == 1 ){
   left[l_counter+1] = l;repeat++;l_counter++;
   for(w = 1; w < id; w++) value[temp[w]] = 0;
   value[l] = -1; left_signal = 0;
   break;
  }
```

```
r = alc_next(right[r_counter]);
l = alc_next(left[l_counter]);

/* condition 1: two host pads compete for one candidate pad */

if(r == l && cost[right[r_counter]][r] < cost[left[l_counter]][l]){
 sensor = 0; count = 0;
 while(count < Nrpad)
 {
  if(count = Nrpad-1) count++;
  else if(value[count] < 0) count++;
  else if(count != r && cost[r][count] < cost[right[r_counter]][r]){
   sensor++; count++; }
  else if(count != r && cost[r][count] == cost[right[r_counter]][r]){
   if(level[r][count] < level[right[r_counter]][r] ){
    sensor++; count++; }
   else count++;
  }
  else count++;
 }

 if(sensor >= sensor_ref){ value[r] = -1; temp[id++] = r; }
 else{
  condition_1++;
  right[r_counter+1] = r; repeat++; r_counter++;
  for(w = 0; w < id; w++) value[temp[w]] = 0;
  end = 1; value[r] = -1;
 }
}

else if(r == l && cost[right[r_counter]][r] > cost[left[l_counter]][l])
{
 sensor = 0; count = 0;
 while(count < Nrpad)
 {
  if(count = Nrpad-1) count++;
  else if(value[count] < 0) count++;
  else if(count != l && cost[l][count] < cost[left[l_counter]][l]){
   sensor++; count++; }
  else if(count != l && cost[l][count] == cost[left[l_counter]][l]){
   if(level[l][count] < level[left[l_counter]][l] ){
    sensor++; count++; }
   else count++;
  }
  else count++;
```

```
    }

    if(sensor >= sensor_ref){ value[l] = -1; temp[id++] = l; }
    else{
      condition_1++;
      left[l_counter+1] = l; repeat++; l_counter++;
      for(w = 0; w < id; w++) value[temp[w]] = 0;
      end = 1; value[l] = -1;
    }
  }

  else if(r == l && cost[right[r_counter]][r] == cost[left[l_counter]][l])
  {
    sensor = 0; count = 0;
    while(count < Nrpad)
    {
      if(count = Nrpad-1) count++;
      else if(value[count] < 0) count++;
      else if(count != r && cost[r][count] < cost[right[r_counter]][r]){
        sensor++; count++; }
      else if(count != r && cost[r][count] == cost[right[r_counter]][r]){
        if(level[r][count] < level[right[r_counter]][r] ){
          sensor++; count++; }
        else count++;
      }
      else count++;
    }

    if(sensor >= sensor_ref){ value[r] = -1; temp[id++] = r; }
    else
    {
      if(level[right[r_counter]][r] < level[left[l_counter]][l]){
        condition_2++;
        right[r_counter+1] = r; repeat++;
        for(w = 0; w < id; w++) value[temp[w]] = 0;
        value[right[r_counter+1]] = -1; r_counter++; end = 1;
      }
      else if(level[right[r_counter]][r] > level[left[l_counter]][l]){
        condition_2++;
        left[l_counter+1] = l; repeat++;
        for(w = 0; w < id; w++) value[temp[w]] = 0;
        value[left[l_counter+1]] = -1; l_counter++; end = 1;
      }
      else{
        if(left_signal == 1){
```

```
      condition_3++;
      left[l_counter+1] = l; repeat++;
      for(w = 0; w < id; w++) value[temp[w]] = 0; end = 1;
      value[left[l_counter+1]] = -1; left_signal = 0; l_counter++;}
     else{
      condition_3++;
      right[r_counter+1] = r; repeat++;
      for(w = 0; w < id; w++) value[temp[w]] = 0; end = 1;
      value[right[r_counter+1]] = -1; left_signal = 1;r_counter++;
     }
    }
   }
  }

/* condition 2: no competetion between two host pads */

else
{
 sensor = 0; count = 0;
 while(count < Nrpad)
 {
  if(count=Nrpad-1) count++;  /* exit loop */
  else if(value[count] < 0) count++;
  else if(count != r && cost[r][count] < cost[right[r_counter]][r]){
   sensor++; count++; }
  else if(count != r && cost[r][count] == cost[right[r_counter]][r]){
   if(level[r][count] < level[right[r_counter]][r] ){
    sensor++; count++; }
   else count++;
  }
  else count++;
 }
 if(sensor >= sensor_ref){ value[r] = -1; temp[id++] = r; }
 else{
  condition_4++;
  right[r_counter+1] = r; repeat++; r_counter++;
  for(w = 0; w < id; w++) value[temp[w]] = 0;
  end = 1; value[r] = -1;
 }

 sensor = 0; count = 0;
 while(count < Nrpad)
 {
  if(count = Nrpad-1) count++;
  else if(value[count] < 0) count++;
```

```
        else if(count != l && cost[l][count] < cost[left[l_counter]][l]){
          sensor++; count++; }
        else if(count != l && cost[l][count] == cost[left[l_counter]][l]){
          if(level[l][count] < level[left[l_counter]][l] ){
            sensor++; count++; }
          else count++;
        }
        else count++;
      }
      if(sensor >= sensor_ref){ value[l] = -1; temp[id++] = l; }
      else{
        condition_4++;
        left[l_counter+1] = l; repeat++; l_counter++;
        for(w = 0; w < id; w++) value[temp[w]] = 0;
        end = 1; value[l] = -1;
      }
    }
  }

  /* check up the end of assignment */

  if(repeat == Nrpad-1)
  {
    condition_4++;
    r = alc_next(right[r_counter]);
    right[r_counter+1] = r; repeat++; r_counter++;
    right[r_counter+1] = -1; left[l_counter+1] = -1;
  }

  else if(repeat == Nrpad)
  {
    right[r_counter+1] = -1; left[l_counter+1] = -1;
  }
}

/* print out circular-ordered pads into file "pad_order" */

fp = fopen("pad_order", "w");

i = 0;
while(right[i] != -1) i++;
r_timer = i;
for(i = 0; i < r_timer; i++){
  pad_order[start][i] = right[i];
  fprintf(fp,"%d0, right[i]);
```

```
  }

  i = 0;
  while(left[i] != -1) i++;
  l_timer = i;
  for(i = l_timer-1; i >= 0; i--){
    pad_order[start][r_timer + l_timer-1-i] = left[i];
    fprintf(fp,"%d0, left[i]);
  }

  fclose(fp);
}


/* choosing candidate pad with the smallest cost to host pad */
alc_next(x)
int x;                /** counter **/
{
int     i;
int     j;
int     cost_ref = REFERENCE_MAX;
int     level_ref;
int     candidate;

  for( i = 0; i < Nrpad; i++)
  {
    if( value[i] < 0 ) continue;

    else if(x != i && cost[x][i] < cost_ref){
      cost_ref=cost[x][i];
      candidate = i;
      level_ref = level[x][i];
    }

    else if(x != i && cost[x][i] == cost_ref && level[x][i] < level_ref){
      cost_ref = cost[x][i];
      candidate = i;
      level_ref = level[x][i];
    }
  }

  return(candidate);
}
```

```
/****************************************************************/

Selection Function

Description: Select a good ring from a group of rings formed by
             assigning function.

/****************************************************************/

#include <stdio.h>
#include <string.h>
#include "global.h"
#include <math.h>

Select( )
{

int    i,j;
int    target[Nrpad+1];
int    ncost_max, nlevel_max, far_ncost_max, far_nlevel_max, mincost_max;
int    ncost_min, nlevel_min, far_ncost_min, far_nlevel_min, mincost_min;
float  parameter_sum[Nrpad], min_cost[Nrpad];
float  ncost[Nrpad], nlevel[Nrpad],far_ncost[Nrpad], far_nlevel[Nrpad];
float  total_ncost,total_nlevel, total_fcost, total_flevel;
float  ncost_ref1, nlevel_ref1, far_ncost_ref1, far_nlevel_ref1;
float  ncost_ref2, nlevel_ref2, far_ncost_ref2, far_nlevel_ref2;
float  mincost_ref1, mincost_ref2, total_min;
float  ncost_dif, nlevel_dif, far_nlevel_dif, far_ncost_dif, min_dif;
struct node  *order, *new, *temp, *former, *order_end;


 /* each factor's value for a ring */

 for(i=0; i<Nrpad; i++)
 {
  Assign(i);
  ncost[i] =  (float) Neighbor_cost(i);
  nlevel[i] = (float) Neighbor_level(i);
  far_ncost[i] = (float) Far_neighbor_cost(i);
  far_nlevel[i] = (float) Far_neighbor_level(i);
  min_cost[i] = (float) Mincut_cost(i);
 }

 /*  initialization  */
```

```
ncost_ref1 = REFERENCE_MIN;
far_ncost_ref1 = REFERENCE_MIN;
nlevel_ref1 = REFERENCE_MIN;
far_nlevel_ref1 = REFERENCE_MIN;
mincost_ref1 = REFERENCE_MIN;
ncost_ref2 = REFERENCE_MAX;
far_ncost_ref2 = REFERENCE_MAX;
nlevel_ref2 = REFERENCE_MAX;
far_nlevel_ref2 = REFERENCE_MAX;
mincost_ref2 = REFERENCE_MAX;

total_ncost = 0.0;
total_nlevel = 0.0;
total_min = 0.0;
total_fcost = 0.0;
total_flevel = 0.0;

/* Find the maxium and mininum values for each factors */

for(i = 0; i < Nrpad; i++)
{
  /* nearer-pad pair weight cost */
  total_ncost = total_ncost + ncost[i];
  if(ncost[i] > ncost_ref1){
    ncost_ref1 = ncost[i];
    ncost_max = i;
  }
  if(ncost[i] < ncost_ref2){
    ncost_ref2 = ncost[i];
    ncost_min = i;
  }

  /* farrest-pad pair weight cost */
  total_fcost = total_fcost + far_ncost[i];
  if(far_ncost[i] > far_ncost_ref1){
    far_ncost_ref1 = far_ncost[i];
    far_ncost_max = i;
  }
  if(far_ncost[i] < far_ncost_ref2){
    far_ncost_ref2 = far_ncost[i];
    far_ncost_min = i;
  }

  /* nearer-pad pair depth cost */
  total_nlevel = total_nlevel + nlevel[i];
```

```
  if(nlevel[i] > nlevel_ref1){
    nlevel_ref1 = nlevel[i];
    nlevel_max = i;
  }
  if(nlevel[i] < nlevel_ref2){
    nlevel_ref2 = nlevel[i];
    nlevel_min = i;
  }

  /* farrest-pad pair depth cost */
  total_flevel = total_flevel + far_nlevel[i];
  if(far_nlevel[i] > far_nlevel_ref1){
    far_nlevel_ref1 = far_nlevel[i];
    far_nlevel_max = i;
  }
  if(far_nlevel[i] < far_nlevel_ref2){
    far_nlevel_ref2 = far_nlevel[i];
    far_nlevel_min = i;
  }

  /* block-pad pair weight cost */
  total_min = total_min + min_cost[i];
  if(min_cost[i] > mincost_ref1){
    mincost_ref1 = min_cost[i];
    mincost_max = i;
  }
  if(min_cost[i] < mincost_ref2){
    mincost_ref2 = min_cost[i];
    mincost_min = i;
  }
}


/*  value range between maximum value and minimum value */

ncost_dif =  (ncost[ncost_max] - ncost[ncost_min] );
far_ncost_dif = ( far_ncost[far_ncost_max] - far_ncost[far_ncost_min]);
nlevel_dif =  (nlevel[nlevel_max] - nlevel[nlevel_min]);
far_nlevel_dif = ( far_nlevel[far_nlevel_max] - far_nlevel[far_nlevel_min]);
min_dif = (min_cost[mincost_max] - min_cost[mincost_min]);

/* choose the rings with smaller values of FACTOR_SUM */

order_end = (struct node *) malloc(sizeof(struct node)) ;
order_end->v = Nrpad;
```

```
parameter_sum[Nrpad] = REFERENCE_MAX;
order = order_end;

target[0] = Nrpad;
for(i=0; i<Nrpad; i++)
{
 /* FACTOR_SUM */
 parameter_sum[i] = ncost[i] +
          (ncost_dif/far_ncost_dif) * far_ncost[i] +
          (ncost_dif/min_dif) * min_cost[i] +
          (ncost_dif/nlevel_dif) * nlevel[i] -
          (ncost_dif/far_nlevel_dif) * far_nlevel[i] ;

 /* sorting */

 if(order == order_end){
  new = (struct node *) malloc(sizeof(struct node)) ;
  new->v = i;
  new->next = order_end;
  order = new;
 }

 else if( parameter_sum[i] < parameter_sum[order->v] ){
  new = (struct node *) malloc(sizeof(struct node)) ;
  new->v = i;
  new->next = order;
  order = new;
 }

 else{
  temp = order;
  while( parameter_sum[i] >= parameter_sum[order->v] ){
   former = order;
   order = order->next;
  }
  new = (struct node *) malloc(sizeof(struct node)) ;
  new->v = i ;
  new->next = order ;
  former->next = new;
  order = temp;
 }

}

/*printf("*** The first 5 smaller value of FACTOR_SUM * ");
```

```
 i = 0;
 for(temp = order; temp != order_end; temp = temp->next){
  i++;
  if( i > 5 ) break;
  else printf("%d  %f0, temp->v, parameter_sum[temp->v]);*/
 }

 return order->v;   /* the smallest value of FACTOR_SUM */

}


/*
**   Selection Factors:
**
**   Neighbor_cost()        nearer-pad pair weight cost
**   Neighbor_level()       nearer-pad pair depth cost
**   Far_neighbor_cost()    farrest-pad pair weight cost
**   Far_neighbor_level()   farrest-pad pair depth cost
**   Mincut_cost()          block-pad pair weight cost
*/

Neighbor_cost(start)
int start;           /* initial pad */
{

int    i;
int    j;
int    n_cost;
int    RANGE;        /* farthest distance for nearer-pad pair */

 n_cost=0;
 RANGE = Power(2, Nrcut);

 for(i = 1; i <= RANGE; i++) {
  for(j = 0; j < Nrpad-i; j++)
   n_cost = n_cost + cost[pad_order[start][j]][pad_order[start][j+i]];
 }

 for(i = 1; i <= RANGE; i++){
  for(j = 0; j < i; j++)
   n_cost = n_cost +
         cost[pad_order[start][Nrpad-j-1]][pad_order[start][i-j-1]];
 }
```

```
  return n_cost;

 }


Neighbor_level(start)
int start;
{

int    i;
int    j;
int    n_level;
int    RANGE;

  n_level = 0;
  RANGE = Power(2, Nrcut);

  for(i = 1; i <= RANGE; i++) {
   for(j = 0; j < Nrpad-i; j++)
     n_level = n_level + level[pad_order[start][j]][pad_order[start][i+j]];
  }

  for(i = 1; i <= RANGE; i++){
   for(j = 0; j < i; j++)
     n_level = n_level +
             level[pad_order[start][Nrpad-j-1]][pad_order[start][i-j-1]];
  }

  return n_level;

 }


Far_neighbor_cost(start)
int start;
{

int    i;
int    j;
int    f_cost;
int    RANGE = 0;

  f_cost = 0;

  for(i = Nrpad/2; i <= Nrpad/2+RANGE; i++) {
```

```
    for(j = 0; j < Nrpad-i; j++)
      f_cost = f_cost + cost[pad_order[start][j]][pad_order[start][j+i]] ;
  }

  for(i = Nrpad/2; i <= Nrpad/2+RANGE; i++){
    for(j = 0; j < i; j++)
      f_cost = f_cost +
              cost[pad_order[start][Nrpad-j-1]][pad_order[start][i-j-1]];
  }

  return f_cost;

}


Far_neighbor_level(start)
int start;
{

int     i;
int     j;
int     f_level;
int     RANGE = 0;

  f_level = 0;
  for(i = Nrpad/2; i <= Nrpad/2+RANGE; i++) {
    for(j=0; j<Nrpad-i; j++)
      f_level = f_level + level[pad_order[start][j]][pad_order[start][j+i]];
  }

  for(i = Nrpad/2; i <= Nrpad/2+RANGE; i++){
    for(j = 0; j < i; j++)
      f_level = f_level +
              level[pad_order[start][Nrpad-j-1]][pad_order[start][i-j-1]];
  }

  return f_level;

}


Mincut_cost(start)
int     start;
{
```

```
int    i;
int    j;
int    k;
int    cutcost = 0;
int    Nrsec;
int    cut;       /* number of partitions in PROUD */
int    p;


 /* for the block-pad pair in diagnol blocks */
 for( p = 1; p <= Nrcut; p++ )
 {
  Nrsec = Power(2,Nrcut)*4-4;
  for( i = 0 + Nrpad/Nrsec*p; i < Nrpad/4 - Nrpad/Nrsec*p; i++ ){
   for( j = Nrpad/2 + Nrpad/Nrsec*p; j <
       Nrpad/2 + Nrpad/4 - Nrpad/Nrsec*p; j++ ){
     cutcost = cutcost - cost[ pad_order[start][i] ][ pad_order[start][j] ];
   }
  }
 }

 /* for the block-pad pair in diagnol blocks */
 for( p = 1; p <= Nrcut; p++ )
 {
  Nrsec = Power(2,Nrcut)*4-4;
  for( i = Nrpad/4 + Nrpad/Nrsec*p; i < Nrpad/2 - Nrpad/Nrsec*p; i++ ){
   for( j = Nrpad/2 + Nrpad/4 + Nrpad/Nrsec*p; j <
       Nrpad - Nrpad/Nrsec*p; j++ ){
    cutcost = cutcost - cost[ pad_order[start][i] ][ pad_order[start][j] ];
   }
  }
 }

 /* for the block-pad pair in the same blocks */
 for(cut = Nrcut; cut > 0; cut-- )
 {
  Nrsec = Power(2,cut)*4-4;
  for( k = 0; k < Nrsec; k++ ){
   for( i = k*Nrpad/Nrsec; i < (k+1)*Nrpad/Nrsec; i++ ){
    for( j = k*Nrpad/Nrsec; j < (k+1)*Nrpad/Nrsec; j++ ){
     if( i != j )
      cutcost = cutcost + cost[pad_order[start][i]][pad_order[start][j]];
    }
   }
  }
```

```
 }

 return cutcost;

}


Power(base,y)
int    base;
int    y;
{

int    i;
int    p;

 p = 1;
 for( i = 1; i <= y; ++i) p = p * base;
 return p;

}
```

```
/*******************************************************************/

Output Function

Description: print out a formated design file with pad locations
        specified used as the input file for PROUD

/*******************************************************************/

#include<stdio.h>
#include "global.h"

Print_out( start, ExampleFile, PadPosition )
int start;
char ExampleFile[30];          /* input data file */
char PadPosition[30];          /* physical pad locaion on chip */
{

FILE  *fp;
FILE  *position;
int    i;
int    j;
int    site_x;
int    site_y;
int    initial = 0;
char   buffer[40];
struct location      *pad_position[Nrpad];

   fp = fopen(ExampleFile, "w");

   position = fopen( PadPosition, "r" );

   for(i = 0; i < Nrpad; i++){
    pad_position[pad_order[start][i]] =
       (struct location *)malloc(sizeof(struct location));
    fgets(buffer, 40, position);
    sscanf(buffer, "%d %d", &pad_position[pad_order[start][i]]->x,
       &pad_position[pad_order[start][i]]->y);
   }


   /* print out the new design file  */

   fprintf(fp, "%s %d0, name1, nrCols);
```

```
  fprintf(fp, "%s %d0, name2, nrRows);
  fprintf(fp, "%s %d0, name3, bcX);
  fprintf(fp, "%s %d0, name4, bcY);
  fprintf(fp, "%s %d0, name5, xGrid);
  fprintf(fp, "%s %d0, name6, yGrid);
  fprintf(fp, "%s %s0, name7, name8);
  fprintf(fp, "%s %d0, name9, nrIos);
  fprintf(fp, "%s %d0, name10, nrMods);
  fprintf(fp, "%s %d0, name11, nrNets);
  fprintf(fp, "%s %d0, name12, nrPins);

  for(i=0; i<nrIos; i++){
   site_x = pad_position[i]->x;
   site_y = pad_position[i]->y;
   fprintf(fp,"%d %s %d %d %d %d %d %d0, pad[i]->id,
     pad[i]->name,pad[i]->type, pad[i]->obstacle,
     pad[i]->nrpin,site_x,site_y, pad[i]->pattern);
   for(j=0 ; j<pad[i]->nrpin; j++){
    fprintf(fp,"%d %d %d %d0,initial++,pad[i]->net[j]->order,
         pad[i]->net[j]->id,pad[i]->net[j]->ntype);
   }
  }

  for(i=0; i<nrMods; i++){
   fprintf(fp,"%d %s %d %d %d %d %d %d0, module[i]->id,
     module[i]->name, module[i]->type, module[i]->obstacle,
     module[i]->nrpin,module[i]->x,module[i]->y,
     module[i]->pattern);
   for(j=0; j<module[i]->nrpin; j++){
    fprintf(fp, "%d %d %d %d0,module[i]->net[j]->pin,
      module[i]->net[j]->order,module[i]->net[j]->id,
      module[i]->net[j]->ntype);
   }
  }

  fclose(fp);
  fclose(position);

}
```

```
/********************************************************************/

Global variable definitions

/********************************************************************/

#define Maxv 833              /* total cell number */
#define Nrpad 81              /* pad number */
#define Nrmodule 752          /* interior module number */
#define Nrnets 1239           /* net number */
#define Nrpins 3303           /* pin number */
#define Nrcut 2               /* number of cuts */
#define REFERENCE_MAX 3.0e+35
#define REFERENCE_MIN -3.0e+35

struct node{
  int v;
  struct node *next;
};

struct new_node{
  int v1;
  int v2;
  struct new_node *next;
};

struct NETS{
  int pin;
  int order;
  int id;
  int ntype;
}*nets;

struct nodes{
  int id;
  char name[8];
  int type;
  int obstacle;
  int nrpin;
  int x,y; /** pad position **/
  int pattern;
  struct NETS *net[50];
};

struct adj{
```

```
  char name[8];
  struct adj *next;
};

struct location{
  int x;
  int y;
};

struct nodes *pad[Nrpad], *module[Nrmodule];

/* for function Listpfs */
int     adj_weight[Maxv];
int     cost[Nrpad][Nrpad];
int     connect[Maxv][Maxv];
int     level[Nrpad][Nrpad];
int     dad[Nrpad][Maxv];
int     val[Maxv];
struct node   *Priq;
struct node   *pfs_end;

/* for Assigning Function */
int     repeat,a[Nrmodule];
int     value[Nrpad];
int     pad_order[Nrpad][Nrpad];
int     temp[Nrpad];
int     right[Nrpad];
int     left[Nrpad];

/* for function Readin and Print_out */
int     nrCols, nrRows, bcX, bcY, xGrid;
int     yGrid, nrIos, nrMods, nrNets, nrPins;
char    name1[20], name2[20], name3[20], name4[20], name5[20], name6[20];
char    name7[20], name8[20], name9[20], name10[20], name11[20], name12[20];
struct node  *y;
struct node  *g;
struct node  *t;
struct node  *s;
struct node  *adj[Maxv];
```