A Merge Algorithm for Circuit Partitioning

Go-An Rau

A thesis submitted for the degree of

Doctor of Philosophy

to the Faculty of Science and Engineering of the University of Edinburgh.

1995

Abstract

Digital systems continue to increase in size and complexity, and the associated design process has grown lengthy and expensive. A method for partitioning these designs into smaller sub-units is required for board and integrated circuit levels of implementation. In addition, rapid checks of the functionality of a particular design will require a digital system to be partitioned into the set of programmable logic devices that form a system emulator.

A merge algorithm for circuit partitioning is presented in this thesis. Results are presented illustrating the performance of a software implementation of this algorithm. These results show that successful circuit partition can be efficiently achieved.

The merge algorithm is based on the simple concept that cells having the maximum number of connections should be the first to be merged. Merging starts with a predefined initial size constraint on circuit groups, and it is implemented in several stages. In each stage, the size constraint on groups is enlarged to keep the merge operation active. A free competitive merge strategy followed by a leading groups merge strategy is used to ensure a good size balance between the finally partitioned groups.

A pseudo-parallel merge algorithm is presented to reduce the processing time when the design to be partitioned is large. This facilitates rapid exploration of possible partition solutions. A data parallelism approach is adopted which distributes data to a number of processors. Each processor contains the same merge algorithm program operating on a different segment of the circuit netlist. Results are presented showing that the pseudo-parallel merge algorithm reduces the time to partition a circuit while maintaining the same quality of result. The predicted performance of a fully parallel implementation of the merge algorithm is also investigated. Practical and computer generated netlist are used to investigate the performance of the experimental partitioning software system.

Declaration of Originality

Except where noted in the text, the research recorded in this thesis is the original and sole work of the author.

Go-An Rau

May 1995

Acknowledgements

I wish to thank Professor Jim Jordan and Doctor John Hannah for their advice and direction during the course of the research recorded in this thesis. The support of the Science and Engineering Research Council and the Government of Taiwan, R.O.C. is gratefully acknowledged.

I also would like to thank InCA in general for supporting the use of their VA software, and in particular Bijan Kiani, John Dunn, Shaun Lytollis and Alison Ahmed for their valuable discussions during the development of this research.

Contents

CHAPTER 1

Introduction

Partitioning has numerous definitions in different scientific and engineering application areas. In this thesis partitioning is aimed at dividing a large circuit into two or more small circuits to suit various technologies for implementing a digital system. This research project started in collaboration with InCA (Integrated Circuit Applications Limited). This company was interested in implementing a digital design by means of arrays of FPGAs. Visits were made to InCA during this project to discuss ideas for solving practical partitioning problems with industrial experts working for InCA. Unfortunately, InCA has been taken over by a much larger American company and it has been difficult to continue with this collaboration.

There are a wide variety of implementation approaches for a digital system. Generally speaking, system designers have the options of prototyping their designs by using either dedicated hardware or software simulation. To provide an early working prototype, a dedicated hardware implementation consisting of available standard parts can be created to give designers a real feel of how the system will work and a chance to try out many functions and locate the hidden problems in the system. Unfortunately, designers who have tried this method have been diverted from real task of testing function and have spent considerable time on issues not directly related to the final design, such as wire-wrapping and soldering errors and defective components. In addition, the circuit overhead and long development time make the resultant cost increase dramatically when the system is large and complex.

Thus, its usage is limited.

Over the years digital circuit designers have used software algorithms for simulation of their designs at logic gate level. Simulation of a design involves the execution of an algorithm that models the behaviour of the actual design. Simulation provides the ability to analyse and verify a design without actually constructing the design and has many benefits in design process. However, simulation suffers from three major limitations:

(1) the speed of simulation.

(2) the need for simulation models.

(3) the inability to actually connect a simulation of one part of a design to actual physical implementation of another part of the design.

Therefore, the results of using software algorithms for simulation have nearly always been poor, with gate level simulation consuming large quantities of time and computing resources. There also have been many examples of designs that were simulated properly but have failed to operate correctly after committing designs to silicon.

A digital emulator, which is constructed with programmable logic devices, can solve many of the problems of dedicated hardware and software simulation. Like a conventional breadboard, the digital emulator provides a hardware model of the design, letting the designer test and debug a system that is operating at or close to real-time speeds. Unlike a conventional breadboard, however, changes may be made to a design by editing a schematic diagram instead of changing wires.

The resulting system provides functional and timing verification at speeds thousands of times faster than the fastest hardware simulation accelerators, which are

themselves thousands of times faster than software simulation. Because the digital emulator is a user-programmable hardware test bed, it can easily be configured for fault emulation as well, giving the same acceleration factors for the test generation process as for the design verification process. Furthermore, it is easy to accommodate arbitrary catalog components within the emulation system, and to interface the digital emulator with other emulation system(such as microprocessor emulators for firmware and software development).

A digital emulator helps in the development of integrated circuit and system design by quickly and automatically generating a hardware prototype of the integrated circuit or system to be designed from user's schematics or net list. The prototype is electrically reconfigurable and may be modified to represent an infinite number of designs with little or no manual wiring changes or device replacement. The prototype runs at real time or close to real time speed and may be plugged directly into a larger system.

1.1. Programmable logic devices

1.1.1. Using programmable logic device for emulation

As a broad definition, a programmable logic device is an integrated circuit capable of having its function defined by the user at the point of design rather than during IC production. A complex ASIC design can be implemented by an array of PLDs which are programmed individually in advance and wire-wrapped on a board according to the pre-defined connections. This board can be further plugged into the ASIC socket in a target system. Then designers may execute a large number of verification cycles. System software and some peripheral devices can be developed and tested on the target hardware. PLDs have obvious advantages since initial ASIC production is avoided and design faults can be located in the early design phase.

PLDs have become increasingly important to system manufacturers and designers for use in prototyping early ASIC production and low volume manufacture. For prototyping, PLD's are often used as hardwired emulators to test out designs and state machines before they are committed to silicon.

Each PLD is usually based around the concept of a single repeatable cell which consists of basic gates and can be instanced from one to many times over the silicon. With PLDs now reaching higher gate densities it should be possible to emulate entire VLSI designs on an array of devices connected through a standard PCB. The design could then be down loaded using silicon compilation technology to place and route the gates over the whole array.

1.1.2. The criteria for assessment of PLDs

There are several PLD ranges which might be used for ASIC emulation. One of the most obvious differences between the various PLDs on the market is the functionality which is attached to each of the repeatable cells. The other main difference is the method which each manufacturer uses to program his device. There are three types of PLD.

(1) The fuse programmable device, in which the connectivity and function of each logic block is defined by removing links using a "programmer".

(2) The anti-fuse programmable device, in which the connectivity and function of each logic block is defined by creating links, again using a "programmer".

(3) Configurable logic in which the function and connectivity of each block is defined by static memory.

In the first two cases the devices may be either re-usable or non re-usable whereas all the configurable logic can be redefined. The Xilinx FPGA [17], [36] is one of the

most popular and widely used configurable logic device. Its user-programmable array architecture is made up of three types of configurable elements: input/output blocks (IOBs), logic blocks and interconnect. The array of Configurable Logic Blocks (CLBs) provides the functional elements from which the user's logic is constructed. The logic blocks are arranged in a matrix within the perimeter of IOBs. The user can define individual I/O blocks for interface to external circuitry and define interconnection network to compose larger scale logic functions.

In order to select suitable PLDs to optimise a design, there are some factors which should be taken into consideration [36-42]:

(1) What percentage of the available gates are used once the design is downloaded onto the chip? This must be considered, in conjunction with software provided by the manufacture, for placing and routing gates onto the PLDs.

(2) What functions are the particular cells capable of fulfilling? All the PLDs on the market provide the normal boolean operations. However they differ according to the size of the equations which can be placed within each cell and the number of inputs and outputs allowed.

(3) It is desired to program an array of PLDs through an interface connected to a host without having to place individual chips in special programmers. Thus dynamic programmability allows the user to be disassociated from the physical construction of the hardware.

(4) The number of inputs and outputs which can be supplied to each PLD or logical block is important. I/O characteristics can restrict the use of a particular area of cells. For example, if the function requires only one input more than the number supplied by a particular PLD an extra package will be required.

(5) The lower and higher level software available for programming and placing particular functions within an array of PLD must be considered. Each PLD chip has to be programmed independently and the high level partitioning software will have to know how much functionality to assign each cell and chip. In part, the "chunk" sizes assigned to each chip will be determined by the efficiency of the low level minimisation and placement software (the lower level software).

1.2. Digital Emulator

There are already some digital emulators available, two of which are introduced in the following. One is called the VA (Virtual ASIC) developed by InCA (Integrated Circuit Applications Limited) which is contained in a single cabinet, suitable for either desktop or stand-alone use and capable of emulating up to 80,000 gates. The other one is called the Anyboard Rapid Prototyping System created at NCSU (North Carolina State University) for the development and rapid implementation of digital hardware designs.

1.2.1. The VA

The VA emulator hardware [7] is based on arrays of Xilinx FPGAs (XC3090). A six FPGA architecture shown in Figure 1.1 and a twelve FPGA ring architecture shown in Figure 1.2 have been developed. The components in the centre of the ring (FPGA 5/6 and FPGA 9/10/11/12 for the six FPGA and twelve FPGA designs respectively) are used as interconnect components and the FPGAs on the outer ring provide a logic and interconnect resource. The numbers associated with the connecting lines shown in Figures 1.1 and 1.2 indicate the programmable interconnects available for linking to FPGAs. The six FPGA design includes an 11-way bus that provides a direct connection to all FPGAs. This bus is driven from the front plane and connects

to I/O pins on the FPGAs. It is used for clock lines, critical input nets and high fan-out nets. The twelve FPGA design includes a 3-way bus, driven from the front plane, that connects to all of the FPGAs. The unconnected lines in both figures indicate the external I/O capability of the emulator cards.

Each of the centre (interconnect) FPGAs used in the twelve FPGA design can be viewed as the hub of a wheel with eight spokes corresponding to the 16 interconnect lines from each of the eight rim FPGAs. The four hub FPGAs provide a total interconnect from a rim FPGA to the centre of 4 x 16 i.e. 64 lines. This design ensures that each rim FPGA is a single hop (i.e. one interconnect link across an FPGA) from any other rim FPGA.

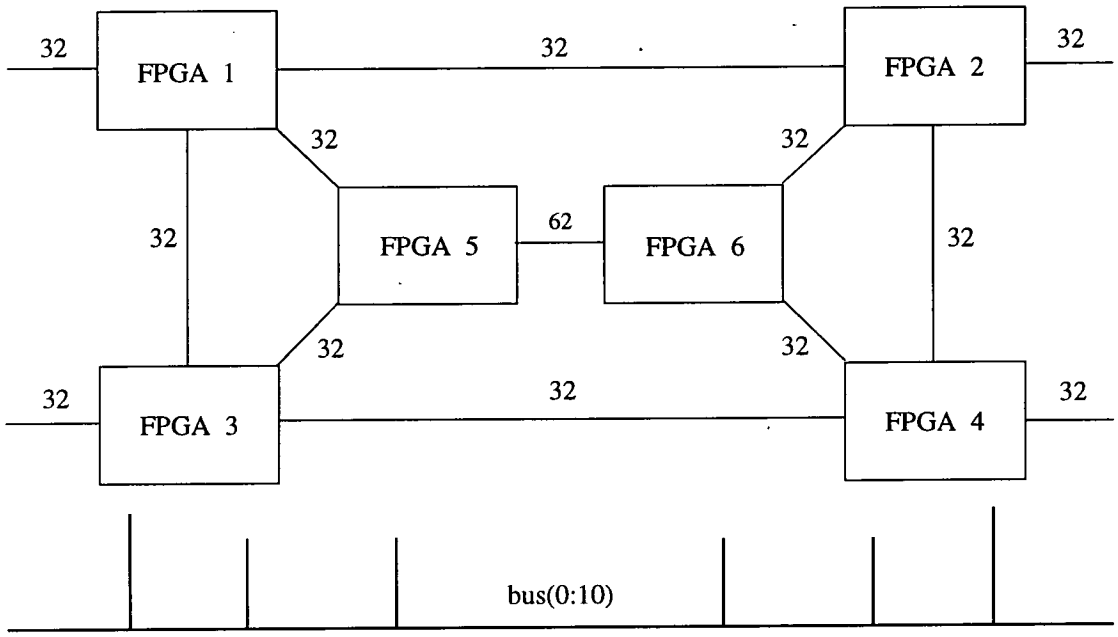


Figure 1.1 Six FPGA emulator board

1.2.2. The Anyboard

The structure of Anyboard [21-24], [84-86] is shown in Figure 1.3. The heart of the Anyboard is an array of five Xilinx FPGAs (XC3090) that provide a large collection of uncommitted logic gates. The usable gate count is approximately 25,000 logic

gates.

Adjacent Xilinx chips in the array are connected by local buses that provide communication between function blocks in systems too large to fit on one chip. High-fan-out signals (such as clocks) can be allocated to the global bus that connects to all the FPGAs.

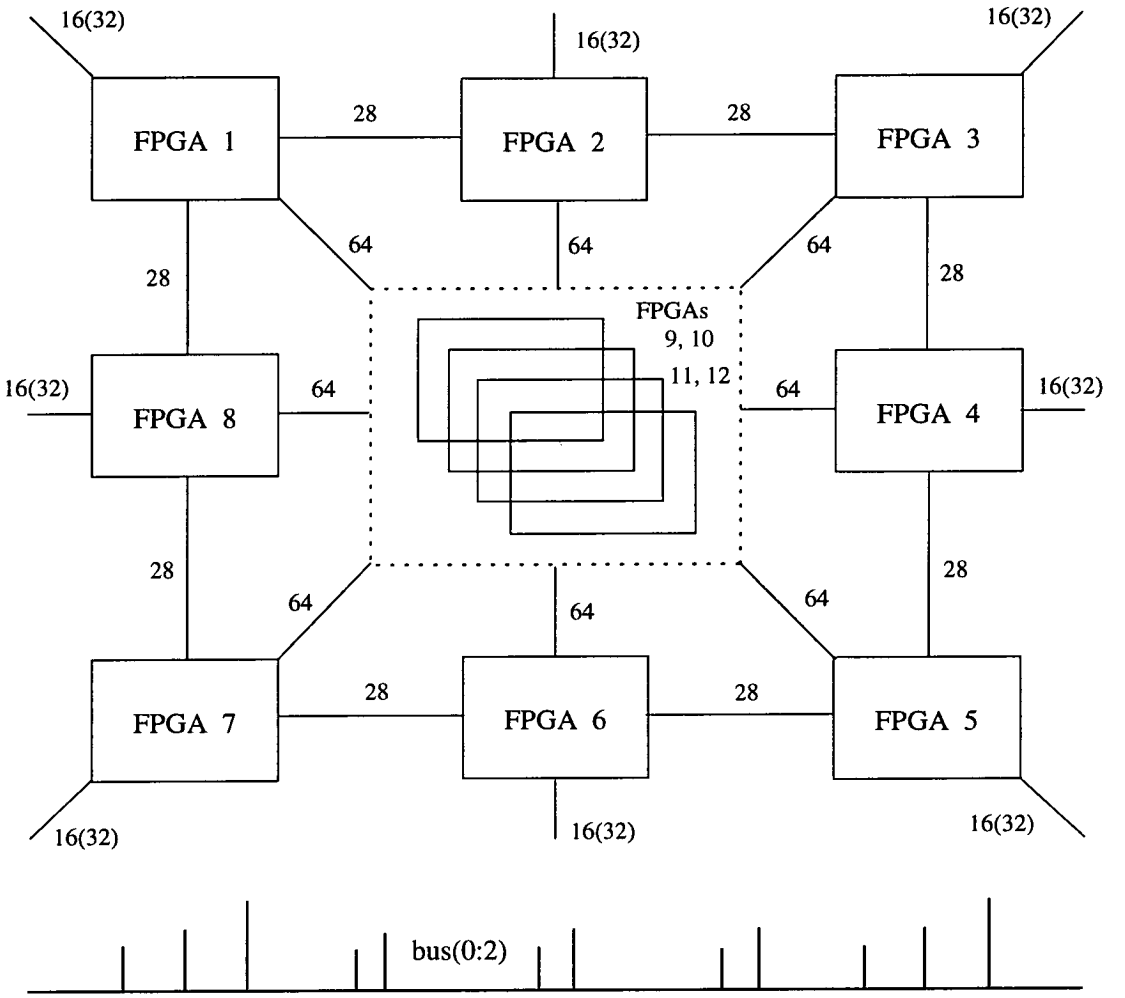


Figure 1.2 Twelve FPGA emulator board

RAMs are attached to the FPGAs in the middle of the array because the Xilinx chips do not provide sufficient storage for memory-intensive designs. The leftmost Xilinx chip serves as an address generator for all the RAMs. This limits the flexibility of RAM addressing, but saves a large number of I/O pins on each FPGA.

The Anyboard communicates with external systems through its system interface, an extension of the global bus with dedicated I/O lines from each FPGA. With appropriate connectors and level-conversion circuitry attached, the Anyboard can emulate an ASIC.

The Anyboard hardware is built on a single 13-by-4-inch card and housed in an ordinary PC. Interfacing to the PC's ISA (Industry Standard Architecture) bus allows the Anyboard to access the hardware resources of the PC and act as a simple coprocessor.

System configuration data cascades through the Xilinx chips on a single wire. Each FPGA picks off its own configuration data and passes along the remainder to the chips downstream.

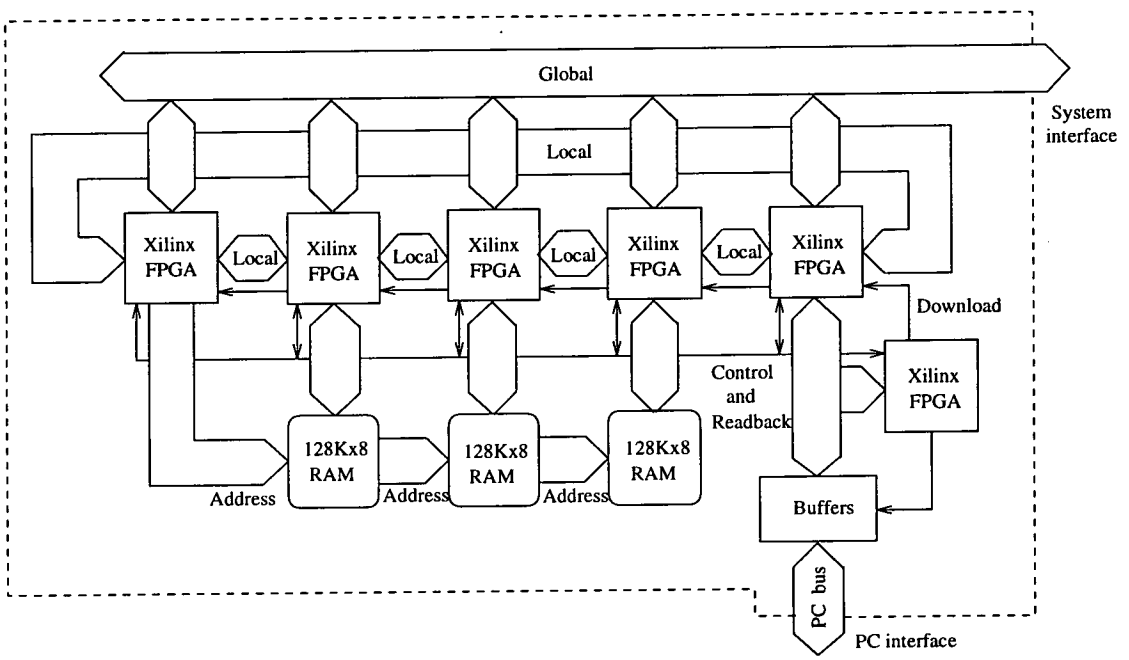


Figure 1.3 Anyboard

1.3. The need for partitioning

Semiconductor circuit technology continues to advance. Moore's law [15] states

that the number of discrete components that can be placed on a single substrate will be doubled every five years. Fair [16] has speculated that, provided no fundamental limits are encountered and microfabrication equipment to operate with a 0.3μ feature size can be economically developed, the ULSI (Ultra Large Scale Integration) era will, at the end of this century, lead into gigantic scale integration (GSI) with a capability of 2^{26} or 67 million components per chip. It is clear that levels of silicon system complexity will continue to increase and a stronger need will emerge for rapid prototyping methods that will enable the functionality of product specifications to be evaluated in a realistic setting.

As FPGAs become a mainstream technology to be considered for board, system and application specific integrated circuit (ASIC) design processes, design complexity will continue to increase more rapidly than the availability of larger, faster devices [19], [20]. Board-level designers find that consolidating random logic into FPGAs saves valuable board real estate and can often improve reliability. System-level ASIC designers are turning to FPGAs for design verification due to their lower cost and the advantages of more rapid prototyping. Complex designs with FPGAs can require multiple iterations in order to achieve a successful design implementation. If automatic design tools cannot provide a solution, the designer is forced to obtain expert level architectural knowledge to support the manual intervention required to complete the design. To effectively use multiple FPGAs, while enjoying the benefit of shorter design time, an automatic partitioning method is strongly required to partition a large design among multiple devices.

Many partitioning approaches have been proposed for attacking circuit partitioning problem, such as clustering [1], [8], [66-68], eigenvector decomposition [25], network flow [26], [35], [93-98], group swapping [3-5], [27-29], [101] and simulated annealing [30], [33], [34], [70]. The clustering method is limited by the lack of a

global view. The eigenvector decomposition requires the transformation of every multi-terminal net into a two-terminal net in real circuits before establishing the matrix. The network flow method usually produces two unevenly sized partition. The group swapping approach needs a good initial partition to start with. Simulated annealing requires a long running time. These available partitioning techniques cannot suit the requirements of the increasingly circuit complexity and the rapid growth of the circuit size. The merge algorithm presented in this thesis possesses a number of features that will lead to its preferred use as circuit size and complexity continue to increase:

- A well-defined data structure that can easily incorporate the required information to suit various circuit requirements.
- A multi-way partitioning algorithm.
- A parallelizable algorithm which can be executed on a multi-processor computer to reduce processing time as circuit size increases.

1.4. Thesis structure

Chapter 2 describes the terminologies of graph theory which are related to the partitioning problems, and reviews the partitioning techniques, such as the Kernighan-Lin based heuristics, constructive, simulated annealing and ratio cut methods. The Kernighan-Lin based heuristics which includes the Kernighan-Lin heuristic, the Fiduccia-Mattheyses heuristic and the Krishnamurthy heuristic, are intensively discussed.

A new method for merging cells to achieve a desired partition is introduced in Chapter 3. The basic merging concept and operation are described and some merging strategies are discussed. The flow chart of the merge algorithm is

described.

In Chapter 4 the data structure and important routines for implementing the merge algorithm are described and Chapter 5 presents the results of partitioning two test circuits which have been created by a random circuit generator.

When the design becomes large, the execution speed of the available partitioning approaches tends to be slow. Chapter 6 presents a pseudo-parallel merge algorithm which is developed to cope with increasing circuit size and complexity. The merge algorithm has the parallelizable feature which can be implemented by data-parallelism method. The design can be evenly distributed to several processors in a computer. Each processor contains the same program operating on a different portion of the design. This concept is realised in a serial way by splitting the design into several small size pieces, the procedures residing in a single processor sequentially work on them. This provides a better speed performance while maintaining the same quality of result. Distributing the data to several processors can be predicted to give even better speed performance than a single processor.

Finally, chapter 7 summarises the work presented in this thesis, and suggests future work.

CHAPTER 2

Partitioning Techniques

The partitioning problem is usually formalized as an operation on graphs and hypergraphs which are briefly described in this chapter. Some terminologies and definitions related to the partitioning problem are illustrated. Three related partitioning approaches which have been used in many applications are discussed. The common idea among these approaches is that an initial partition is given and the algorithm improves the quality of the partition by modifying the partition iteratively.

2.1. Graph and hypergraph

Graphs and hypergraphs appear in many areas, such as in electrical engineering, computer science, chemistry and geography. Graphs find their importance as models for many kinds of problems and processes. The components on a circuit board connected by wires form a graph, as do cities connected by highways. An organic chemical compound can be considered a graph with the atoms as vertices and the bonds between them as edges. Graph theory has long become recognised as an important and useful mathematical background in these areas [63-65].

The basic definitions in the area of graphs are described in the following. The definitions included here will be related to the partitioning problems later.

2.1.1. Graph

A graph G(X, U) consists of a set of vertices X = { x_1, x_2, $\cdots$ } and a set of edges

U = { u_1, u_2, $\cdots$ }, the edges are pairs of distinct vertices from X. If u = (x_1, x_2) is an edge with vertices x_1, and x_2, then x_1 and x_2 are said to lie on u, and u is incident to x_1 and x_2.

The intuitive way to picture a graph is to represent vertices as points, squares or circles and edges as line segments or arcs connecting the vertices. Figure 2.1 shows an example of a graph. Here X ={x_1, x_2, x_3, x_4, x_5, x_6}, U = {u_1, u_2, u_3, u_4, u_5, u_6, u_7}. The edge u_7 = (x_1, x_5) incident to x_1 and x_5 which are called its endpoints. The edge u_3 and u_4 have the same endpoints and therefore are called parallel edges. The degree of a vertex x , d(x), is the number of times x is used as an endpoint of the edges. Thus, in our example d(x_2)=4, d(x_4)=1 and d(x_5)=2. Also, a vertex x whose degree is zero is called isolated ; in this example x_6 is isolated since d(x_6)=0.

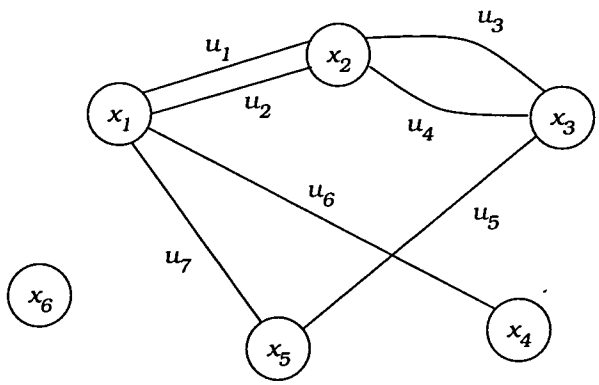


Figure 2.1 A graph

2.1.2. Hypergraph

A hypergraph H = (V, E) consists of a finite set of vertices V = {v_1, v_2, $\cdots$ } and a set of hyperedges E = {e_1, e_2, $\cdots$, e_i, $\cdots$} where e_i is the subset of V, $|e_i| \geq 2$ and $\cup e_i$ = V, where i $\in$ I. If $|e_i| = 2$, then a hypergraph becomes a graph.

A hyperedge with two endpoints is sometimes called a two-terminal edge; a hyperedge with more than two endpoints is sometimes called a multi-terminal edge.

If $e_1 = (v_1, v_2, v_3)$ is a hyperedge with vertices v_1, v_2, and v_3, then v_1, v_2, and v_3 are said to lie on e_1 , and e_1 is said to be incident to v_1, v_2, and v_3. The degree of a vertex v is the number of hyperedges incident to v.

A hypergraph is shown in Figure 2.2, which consists of V = $\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$, and E = $\{e_1, e_2, e_3, e_4, e_5\}$. The vertices are drawn as points. An edge e_i with $|e_i| > 2$, is drawn as a curve encircling all the vertices of e_i. An edge e_i with $|e_i| = 2$, is drawn as a curve or a line segment connecting its two vertices.

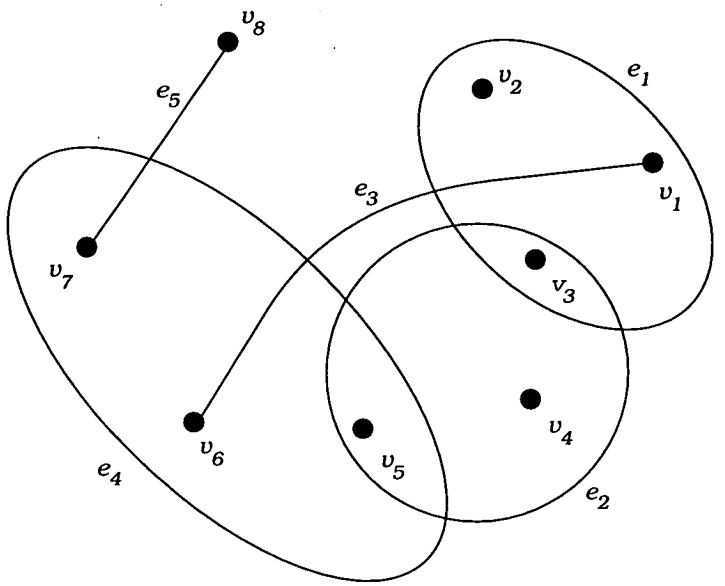


Figure 2.2 A hypergraph

2.2. Partitioning

Partitioning is the task of decomposing a design into parts so that a given objective function is optimised. The objective function may be quite complex according to the application. For example, system designers have to decompose the circuitry they wish to implement, into components that can be realised with standard parts such as TTL logic or with custom and semi-custom chips. The cost function in this case includes the design time, performance of the system, the reliability of the design, etc.. To simplify the objective function, circuit partitioning problems are

concentrated on the links (wires) between components. Before discussing the partitioning techniques, some definitions and terminologies are described in the following.

2.2.1. Definition of terms

Definition 2.1: Network

A network (circuit or system), which is denoted by S, is a set of cells (components, modules) which are interconnected.

Definition 2.2: Partition

A partition of a network S separates the network into two or more parts, i.e. S = { $s_1; s_2; \cdots; s_t$ }, $s_i \subset S$, $\bigcup s_i = S$, $s_i \cap s_j = \varnothing$, $i \neq j$, and $i,j,t \in I$.

Definition 2.3: Net

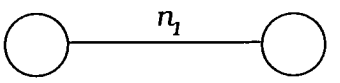
A net, which is denoted by n, may be viewed as the connection that links a set of cells together. Examples of nets are shown in Figure 2.3.

Definition 2.4: Cutset

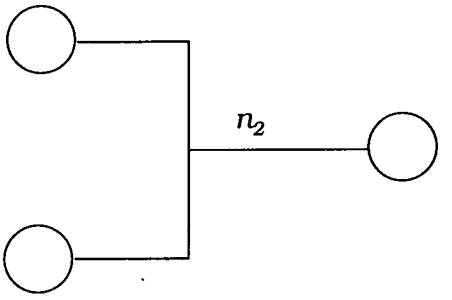
A cutset is the set of nets that interconnects cells in different parts of a partition, for instance, of a network S = { A; B } shown in Figure 2.4. In this case, cutset = $\{n_1, n_2, n_6\}$.

Definition 2.5: Cutsize

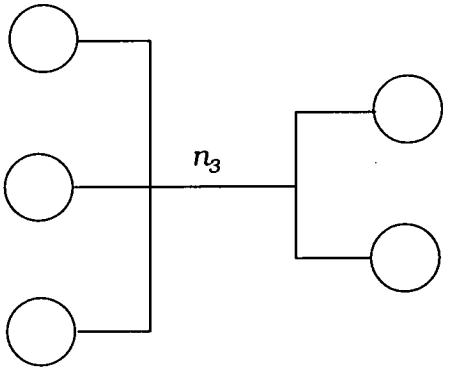
A cutsize is the number of nets in the cutset. The cutsize of the partition in Figure 2.4 is 3.



n_1 is a net which connects two cells.
and is a hyperedge with $|n_1| = 2$



n_2 is a net which connects three cells
and is a hyperedge with $|n_2| = 3$



n_3 is a net which connects five cells.
and is a hyperedge with $|n_3| = 5$

Figure 2.3 Examples of nets

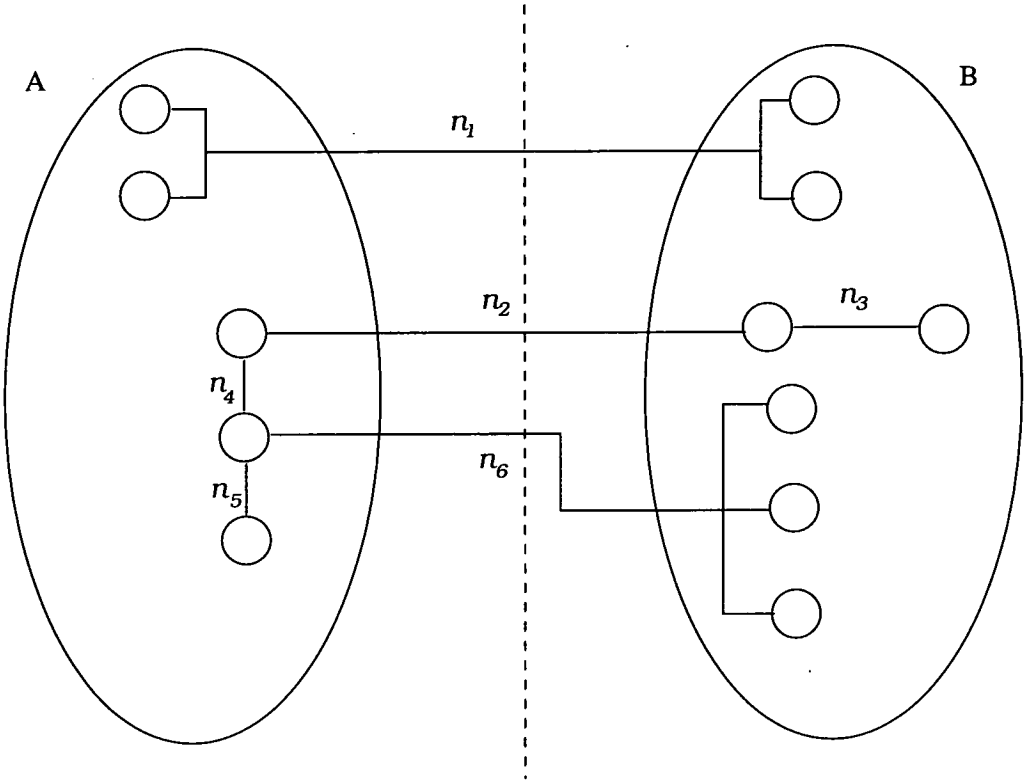


Figure 2.4 An example of a cutset

A circuit contains some elements and their interconnecting wires. The elements are often called components, modules, or cells, and the wires are called nets. Use will be made of cells as the general term for elements in a circuit, and nets as wires. Cells can be viewed as vertices, and nets as hyperedges in a hypergraph. From now on, these names will be used interchangeably depending on the problems under investigation.

2.2.2. Cost Function

A formal description of partitioning problem is given as follows: For any network S = (C,N), let C = { $c_1, c_2, \cdots, c_i, \cdots, c_n$} be a set of q cells, interconnected by a set of nets, N = { $n_1, n_2, \cdots, n_j, \cdots, n_k$}, for instance, n_j may interconnect several number of cells, i.e., n_j = { (c_j, c_1), (c_j, c_2), $\cdots$,(c_j, c_m) }, where $1 \leq m \leq q$ and

$m \neq j$. Nets with m=1 are sometimes called two-terminal nets and nets with $m \geq 2$ are sometimes called multi-terminal nets. Net n_j has at least a pin on each of the cells it connects. The number of pins on a cell c_i is denoted as $p(i)$, and the total number of pins in the circuit as P. The problem is to find a partition of S, S = { s_1, s_2,$\cdots$, s_t }, $s_i \subset$ S, $\bigcup s_i$ = S, $s_i \bigcap s_j = \varnothing$, $i \neq j$, subject to the size constraints to each part of the partition , which minimises the cost function:

$$ct(\mathrm{S}) = \sum_{c_i \in s_h,\ c_j \in s_k,\ h \neq k} ct(i, j)$$

where $ct(i, j)$ is the cost of nets that connect c_i to c_j, i.e. the number of nets between c_i and c_j.

This problem is NP-hard [13], [31-32], [99] and there are various ways in which the solution of the NP-hard problem can be approached. The heuristic partitioning algorithms is by far the most widespread method in practice today [2]. In these algorithms iterative improvement techniques have been used most in many applications. The common concept among these methods is that an initial solution (partition) is given and the algorithms improve the quality of the solution by making local changes to the initial partition.

The simplest form of iterative improvement algorithm is the so called random interchange algorithm [69]. In this algorithm, a given partition, is modified by first selecting a pair of cells, one in each element of the partition, and then evaluating the cost of the partition by interchanging them. These algorithms use randomly selected pair of cells to swap and accept an interchange only if the cost function decreases. If the cost function increases the interchange is rejected and the cells remain in their previous positions.

Cell swapping was sugested by Kernighan and Lin who proposed a two-way partitioning algorithm [3]. Subsequently, many improvements have been made to

this method. Fiduccia and Mattheyses [4] improved this algorithm by moving one cell at a time. Krishnamurthy [5] further added a lookahead ability. The details of these algorithms are discussed in the next section.

For the sake of simplicity the various iterative algorithms will be discussed, assuming that the bi-partition problem is to be solved, i.e., that S = {A;B}, where A and B are two subset of S and |S|=2. Assume further that the weights of all the nets and the sizes of all the cells to be partitioned are the same.

2.3. Iterative improvement algorithms

2.3.1. The Kernighan-Lin heuristic

The basic idea of this algorithm is again to interchange pairs of cells among the two elements of the partition to obtain a better solution. Instead of randomly selecting pairs of cells to swap, a scoring function is used to evaluate the interchanges.

Before the algorithm for the Kernighan-Lin approach is explained in detail, some definitions are required as follows:

Definition 2.6: (External and Internal Two-Terminal Net Cost)

For any partition S = {A;B} which is a network S = (C,N), the external two-terminal net cost of a cell $c_i \in$ A is defined as

$$E(i) = \sum_{\substack{n=(c_i,c_j)\in N \\ c_j \in B}} ct(n)$$

where $ct(n)$ is the cost of nets that connect c_i to c_j, i.e. the number of nets between c_i and c_j.

Similarly, the internal two-terminal net cost is defined as

$$I(i) = \sum_{\substack{n=(c_i,c_j)\in N \\ c_j \in A}} ct(n)$$

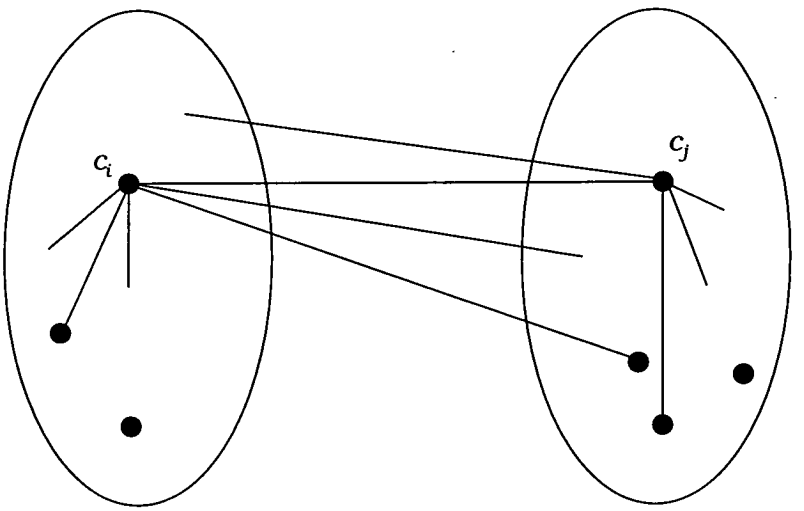
The definitions for $c_i \in B$ are made in the same way.

Definition 2.7: Gain

The gain of c_i is defined as

$$D(i) = E(i) - I(i)$$

A simple example shown in Figure 2.5 illustrates the basic calculations of these functions.



$$E(i) = 3 \quad I(i) = 3 \quad D(i) = 0$$

$$E(j) = 2 \quad I(j) = 3 \quad D(j) = -1$$

Figure 2.5 Cost calculation

Lemma 2.1:

Consider any $c_i \in A$, $c_j \in B$. If c_i and c_j are interchanged, the gain (that is, the reduction in cost) is precisely $D(i) + D(j) - 2ct(i, j)$.

proof: Let z be the total cost due to all connections between A and B that do not

involve c_i or c_j. Obviously the cost including these two cells is

$$T = z + E(i) + E(j) - ct(i, j)$$

Exchange c_i and c_j, let T' be the new cost. The following result is obtained.

$$T' = z + I(i) + I(j) + ct(i, j)$$

and then

$$\begin{aligned} gain &= old\ cost - new\ cost = T - T' \\ &= E(i) - I(i) + E(j) - I(j) - 2ct(i, j) \\ &= D(i) + D(j) - 2ct(i, j) \quad \square \end{aligned}$$

$D(i)$ is the amount by which the cutsize decreases if c_i changes sides in the partition. Then obviously an exchange of the cell pair $\{c_i, c_j\}$ is associated with a decrease in the cutsize of $D(i) + D(j) - 2ct(i, j)$, where $ct(i, j)$ is the cost of the net (c_i, c_j) if that net exists, otherwise $ct(i, j) = 0$. In fact, if the two cells are interchanged, the nets that were connecting c_i to cells in B, become internal interconnections and do not contribute to the cost function. However, the nets that interconnect c_i to cells in A, become external interconnections after the interchange and they now contribute to the cost of the partition. Note that the term $2ct(i, j)$ is subtracted from the cost function since the connections between c_i and c_j are counted twice as external connections in $E(i)$ and $E(j)$.

Consider a simple example given in Figure 2.6. This network has 8 cells interconnected by 18 nets. A partition of a network is denoted by PT. Obviously the optimal solution of this network is PT_{opt}, i.e.,

$$PT_{opt}: S = \{A;B\}$$

where A = {1,2,3,4}, B = {5,6,7,8}. The cutsize of this partition is equal to 2.

Suppose a partitioning procedure starts with the initial partition PT_{init} whose cutsize is equal to 8. To reduce the danger of being trapped in local minima, Kernighan and Lin select the pairs of cells whose exchange results in the largest decrease of the cutsize or the smallest increase, if no decrease is possible.

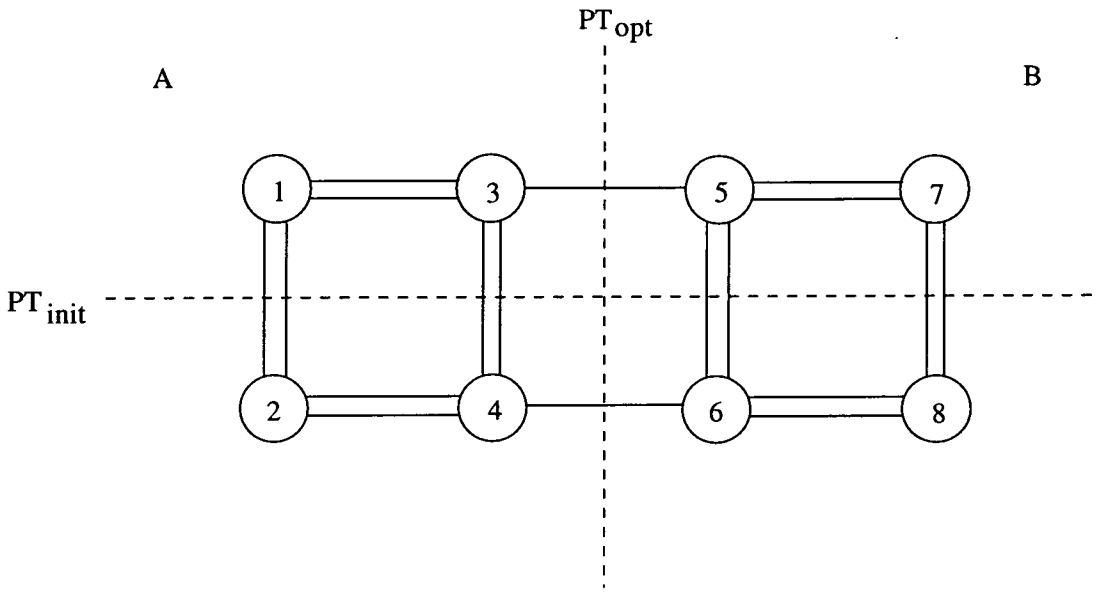


Figure 2.6 A graph to be partitioned

In Figure 2.6, such a pair is, for instance, the pair {4,5}, the corresponding exchange increases the cutsize by 2. The gain calculation is as follows:

$$E(4) = 2,\ I(4) = 3$$

$$D(4) = E(4) - I(4) = -1$$

$$E(5) = 2,\ I(4) = 3$$

$$D(5) = E(5) - I(5) = -1$$

$$ct(4,5) \doteq 0$$

$$D(4,5) = E(4) + E(5) - I(4) - I(5) - 2ct(4,5)$$

$$= 2 + 2 - 3 - 3 - 0 = -2$$

The new configuration of this network after exchange is shown in Figure 2.7.

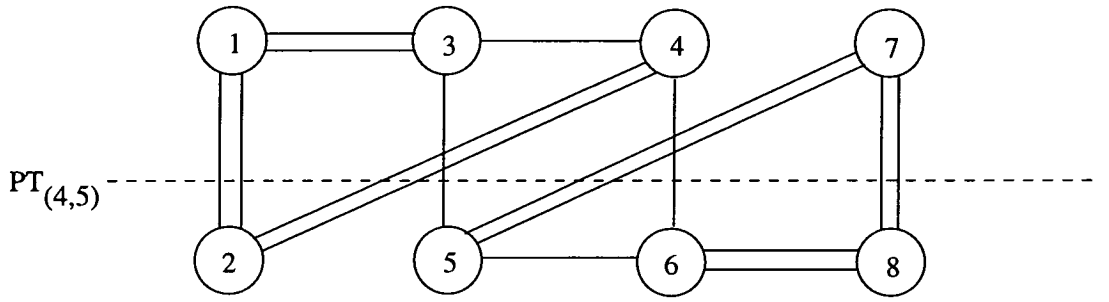


Figure 2.7 A new partition after cells swap

This exchange is made only temporatory, however. The exchanged cells are now locked. This locking prohibits them from taking part in any further tentative exchanges. A search is then made for a second pair whose exchange improves the cut cost. In this example, a suitable pair is {2,7}. This procedure is continued, keeping a record of all tentative exchanges and the resulting cutsizes. The procedure finishes when all cells are locked. At this time, both side of the partition have been exchanged and are back to the original cutsize. Table 2.1 shows a possible choice of pairs of the example in Figure 2.7. The fifth column in the table displays the change in the cutsize, if the first corresponding exchanges in the table are performed. The third column of the table shows the gain with respect to the previous result and the fourth column contains the respective new cutsize.

The final procedure is exchanging a sequence of pairs of cells from step 1 to the step that results a best cutsize. In this example, step 2 reaches a smallest cutsize, so that the cells in pairs {4,5} and {2,7} are exchanged, and this produces an optimal partition in this example.

Step No.	Selected pair	Gain	Cutsize	Change
0	—	—	8	—
1	{4,5}	-2	10	2
2	{2,7}	8	2	-6
3	{1,8}	-8	10	2
4	{3,6}	2	8	0

Table 2.1 The record of cell exchanges for Figure 2.6

2.3.2. The Fiduccia-Mattheyses heuristic

Fiduccia and Mattheyses improved Kernighan-Lin heuristic by introducing the following new elements:

(1) Only a single cell is moved across the cut in a single move, then it is locked following the move.

(2) The "cell gain" concept is introduced to help select the cell to be moved from one block of the partition to the other.

(3) The concept of the D value is extended to hypergraphs, this makes the algorithm able to handle hypergraphs.

(4) A minimum balance condition is maintained throughout the process.

As before, the moves in a pass are tentative and are followed by locking the moved cell. They may increase the cutsize. However, just as in Kernighan-Lin heuristic, at the end of a pass, when no more moves are possible, the sequence of moves is realised only if it decreases the cutsize. Otherwise, the pass is ended.

The concept of the algorithm will now be detailed. The extension of the D value to hypergraphs is quite straightforward. It is only necessary to redefine the internal and external hyperedge costs as follows.

Definition 2.8: (External and Internal Net Cost)

For any partition S = {A,B} which is a network S = (C,N), the external net cost of cell c_i is defined as

$$E(i) = \sum_{e \in N_{ext,i}} ct(n)$$

where

$$N_{ext,i} = \{n \in N \mid \{c_i\} = n \cap A\}$$

Analogously, the internal net cost of cell c_i is defined as

$$I(i) = \sum_{e \in N_{int,i}} ct(n)$$

where

$$N_{int,i} = \{n \in N \mid c_i \in n \text{ and } n \cap B = \varnothing\}$$

With these provisions, the gain $D(i)$ is again defined as

$$D(i) = E(i) - I(i)$$

Intuitively, $N_{ext,i}$ is the set of nets that is removed from the cut if c_i changes sides, and $N_{int,i}$ is the set of nets that is added to the cut if c_i changes side. With these observations, it is obvious that, when moving c_i from A to B, the cutsize changes by an amount of $-D(i)$. The nets in $N_{ext,i} \cup N_{int,i}$ are also called critical nets for c_i. This will be explained later.

By the definition above, the cell gain of a cell, which is denoted by $D(i)$, means what results can be obtained due to moving a cell from one part to the other, i.e. , the

number of nets this moving reduces from or adds to the cutsize. Figure 2.8 shows an example of cell gain.

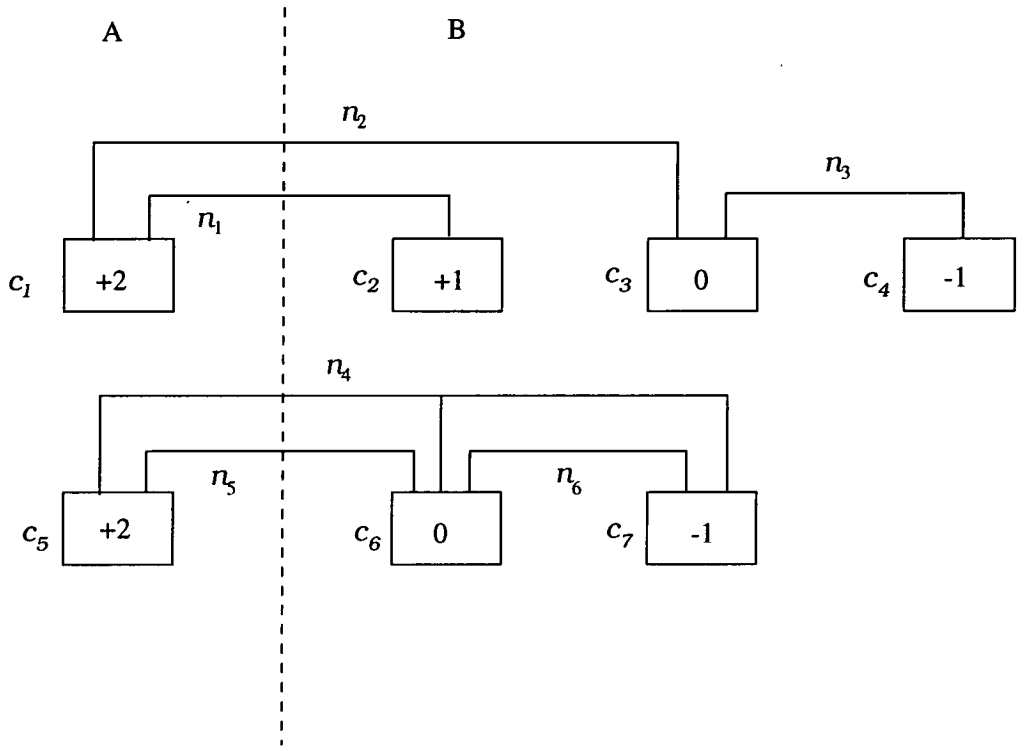


Figure 2.8 An example of the cell gain

$D(c_1)$=2, after moving c_1 to B side, two nets, n_1 and n_2, are removed from the cutset.

$D(c_2)$=1, after moving c_2 to A side, one net, n_1, is removed from the cutset.

$D(c_3)$=0, after moving c_3 to A side, a net n_2 is removed from the cutset, but another net n_3 is added to the cutset.

$D(c_4)$=-1, after moving c_4 to A side, one net n_3 is introduced to the cutset.

$D(c_5)$=2, after moving c_5 to B side, two nets, n_4 and n_5, are removed from cutset.

$D(c_6)$=0, after moving c_6 to A side, net n_5 is removed from the cutset, but another net n_6 is added to the cutset. The net n_4 incident to c_6 is still in the

cutset.

$D(c_7)$=-1, after moving c_7 to A side, net n_6 is added to the cutset.

2.3.2.1. Data structure — Bucket list structure

Clearly, the gain $D(i)$ of cell(i) is an integer in the range $-p(i)$ to $+p(i)$, where $p(i)$ is the number of pin on cell(i), so that each cell has its gain in the range $-p_{max}$ to $+p_{max}$, where $p_{max} = \max\{p(i) \mid \text{cell}(i)\}$.

To choose the next free cell to move, a sorted list of cell gains is maintained, this is done using an array Bucket[D], where D corresponds to the range $-p_{max}$ to $+p_{max}$, and whose entry contains a doubly-linked list of free cells with gains equal to current D. Figure 2.9 shows the structure of this bucket list. Two such arrays are set up for each side of partition. Each array is maintained by quickly moving a cell to the appropriate bucket whenever its gain changes due to the movement of one of its neighbours. Also a cell array is required which allows each cell to be directly addressed in the bucket array. In addition, two lists, LockedA and LockedB, are maintained to accommodate locked cells which are moved from one side to the other side.

For each Bucket array, a MAXGAIN index is maintained which is used to keep track of the bucket having a cell of highest gain. This index is updated by decrementing it whenever its bucket is found to be empty and resetting it to a higher bucket whenever a cell moves to a bucket above MAXGAIN.

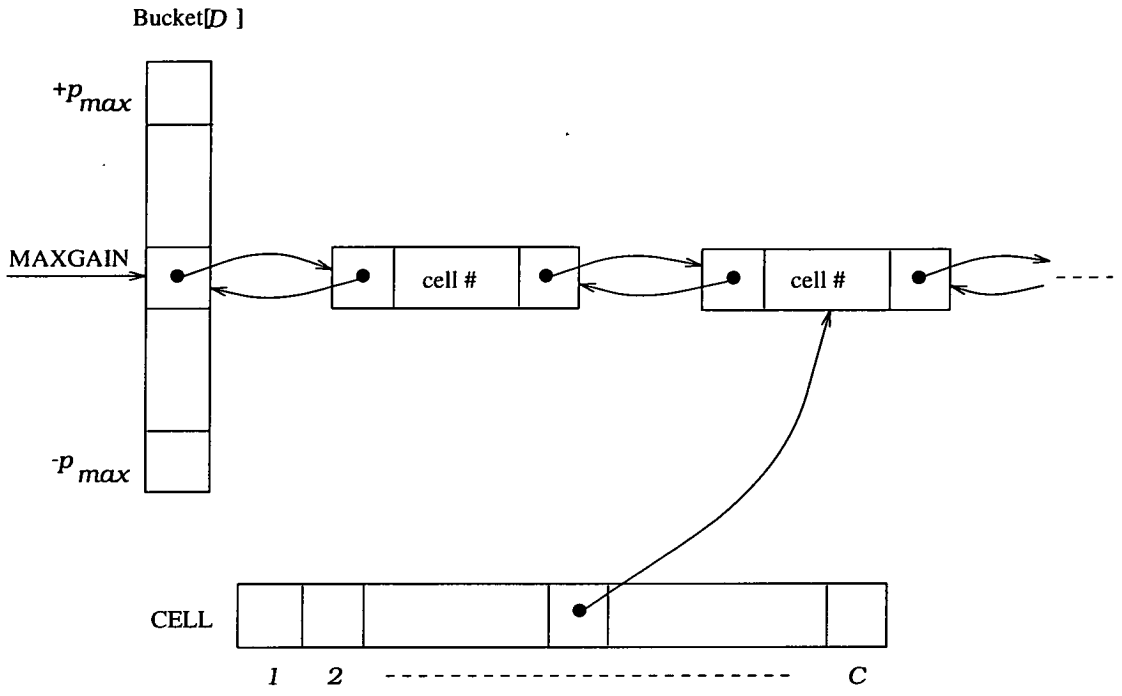


Figure 2.9 Bucket list structure

2.3.2.2. Initialising the data structure

Some input routines are needed to deal with real networks whose interconnections between cells are described in a text format. The principal function performed by the input routine is to construct two data structures from the text input which represents the network. The first structure is a CELL array, which for each cell contains a linked list of the nets that are incident to the cell. The second structure is a NET array, which for each net contains a linked list of the cells on the net.

To compute and maintain cell gain, the notion of a critical net must be introduced. Consider an arbitrary net n_j. Given a partition S = {A;B}, define the number of cells on net n_j in side A as A(n_j), likewise the number of cells on net n_j in side B as B(n_j). A net is critical if there exists a cell on it which if moved would make the cutset increase or decrease. It is easy to see that net n_j is critical if and only if either A(n_j) or B(n_j) is equal to 0 or 1. Figure 2.10 shows an example of critical nets, and

$A(n_1) = 1$, $B(n_1) = 2$, $A(n_2) = 2$ $B(n_2) = 1$, $A(n_3) = 4$, $B(n_3) = 0$ $A(n_4) = 2$, $B(n_4) = 3$. In this example n_1, n_2, and n_3 are critical nets, and n_4 is not a critical net.

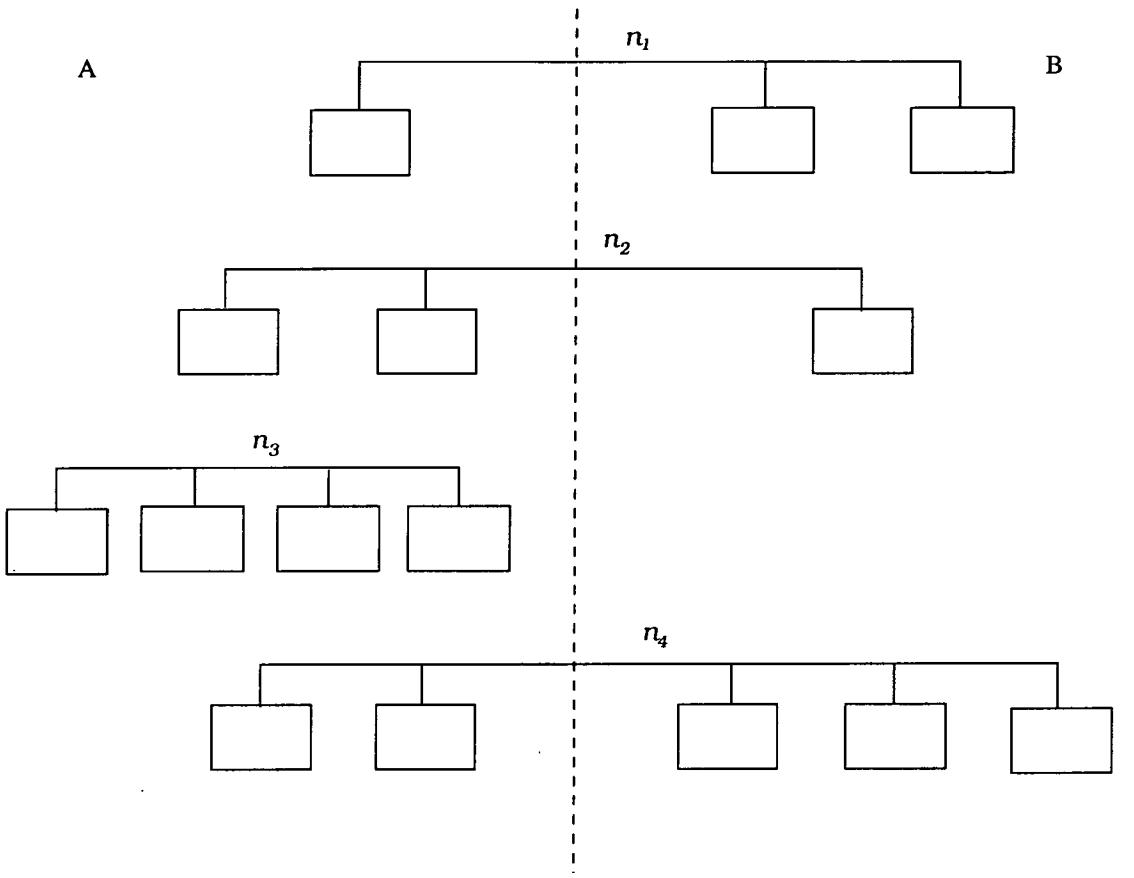


Figure 2.10 An example of critical nets

It is now clear that the gain of a cell, previously defined in terms of its effect on the cutset, depends only on its critical nets. This means that if the net is not critical, its cutset cannot be affected by a move. What is more important, a net which is not critical either before or after a move cannot possibly influence the gains of any of its cells.

To compute and manipulate D values, two more array $A[n_j]$ and $B[n_j]$ are constructed. $A[n_j]$ contains the number of cells on net n_j on side A. $B[n_j]$ is defined analogously. Moreover, two net lists Unlocked$A[n_j]$ and Unlocked$B[n_j]$ are provided for each net containing unlocked cells of the net on side A and side B,

respectively. These two lists will be used for updating the D values in the next section. Then, D values are computed using a scan of the Net array. The following algorithm computes the initial gains of all free cells.

```
FOR each free cell c_i DO
    D[i] = 0 ;
    FOR each net n_j on cell c_i DO
        IF A[n_j] = 1 THEN D[i] = D[i] + 1 ;
        IF B[n_j] = 0 THEN D[i] = D[i] - 1 ;
    END FOR
END FOR
```

2.3.2.3. Updating the bucket data structure

Assume now that a cell c_i on side A has been chosen. The cell is then locked— that is, its item is removed from its bucket list and is attached to the list LockedB, where it awaits the next iteration.

Then, adjustments of D values become necessary, since during the move of cell c_i the nets incident to cell c_i may become or cease to be critical, or their contribution changes from positive to negative or vice versa. Let us call a net n_j for which this changing happens. For a net n_j to be changing in the move of cell c_i, it is necessary that cell c_i is on net n_j, otherwise the values A[n_j] and B[n_j] do not change. However, not all such nets are changing. A convenient way of viewing the changes is to think of the move of cell c_i as two-phase procedure. First, cell c_i is deleted from side A, then it is added to side B. Figure 2.11 details the respective algorithm for changing the D values. In line 3, cell c_i, is effectively deleted from side A. Through this action, some nets become critical, but none of them cease to be critical. Line 4 to 6 process the nets that make a new negative contribution to the D values. Lines 8 to 10 process the nets whose new contribution to D values are positive. In line 14, cell c_i is effectively added to side B, which causes some nets to cease to be critical. Lines 15 to 17 make the respective positive changes, and lines 19 to 21

make the respective negative changes, to D values.

```
(1) FOR net n_j such that cell c_i on net n_j DO
(2)     remove cell c_i from UnlockedA[n_j]
(3)     A[n_j] = A[n_j] - 1 ;
(4)     IF A[n_j] = 0 THEN
(5)        FOR cell c_k ∈ UnlockedB[n_j] DO
(6)            D[k] = D[k] - 1 ;
(7)        END FOR
(8)        ELSE IF A[n_j] = 1 THEN
(9)                FOR cell c_k ∈ UnlockedA[n_j] DO
(10)                   D[k] = D[k] + 1 ;
(11)               END FOR
(12)       END IF
(13)    END IF
(14)    B[n_j] = B[n_j] + 1 ;
(15)    IF B[n_j] = 1 THEN
(16)       FOR cell c_k ∈ UnlockedA[n_j] DO
(17)           D[k] = D[k] + 1 ;
(18)       END FOR
(19)       ELSE IF B[n_j] = 2 THEN
(20)               FOR cell c_k ∈ UnlockedB[n_j] DO
(21)                   D[k] = D[k] -1 ;
(22)               END FOR
(23)       END IF
(24)    END IF
(25) END FOR
```

Figure 2.11 Updating the D values

2.3.3. The Krishnamurthy heuristic

One disadvantage that Fiduccia-Mattheyses heuristic share with the Kernighan-Lin heuristic is that there is a large amount of unresolved nondeterminism. The heuristics choose arbitrarily between cells that have equal gain. Krishnamurthy introduced a more look-ahead approach into heuristic. With this it is possible to distinguish among such cells with equal gain.

If a net contains more than two cells, and if the net is in the cutset of the current partition, then moving one of the cells on this net will not necessarily remove the net from the cutset; however, it may make it possible to remove the net in a future iteration when another cell on that net is moved. Figure 2.12 shows a simple

example which has a net N connecting cells c1, c2, c3, c4, and c5, c1 and c2 are in partition A while c3, c4, c5 are in partition B. All these five cells have equal conditions. Moving any one of these cells will not remove N from the cutset. However, moving cell c1 or c2 would be better than moving the other cells, since in the next iteration c2 or c1 might be moved to remove N from the cutset. Although this example is rather simplified (because in general the other nets the cells are connected to will also play a role in the choice of cell to be removed), it provides the motivation for the concept of Krishnamurthy's level gain.

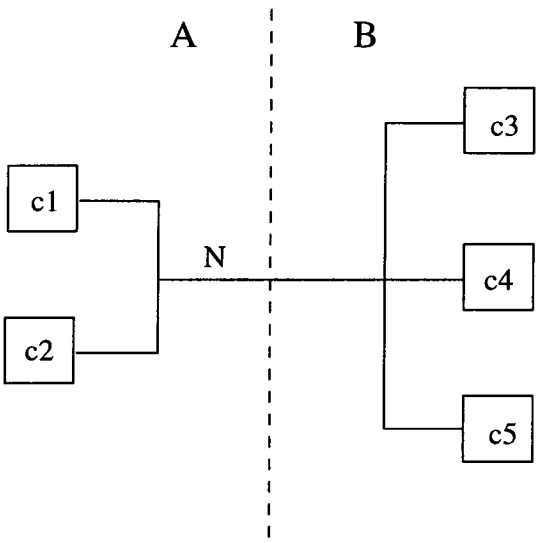


Figure 2.12 An example to show the concept of Krishnamurthy's level gain

Specifically, Krishnamurthy extends the cell gain value to the level gain which is a sequence of number $<D_1(i),\cdots, D_k(i)>$. This sequence number is referred to as the gain vector of order k of a cell i, denoted by Γ_k(i), as

$$\Gamma_k(i) = < D_1(i), \cdots, D_k(i) >$$

Here, $D_1(i)$ is the first level gain of cell i, i.e. the original cell gain of cell i, and the following numbers D_2(i),$\cdots$, D_k(i) define the second level gain and higher level gain for distinguishing cells with the same D_1 value. The gain vector $<D_1(i),\cdots, D_k(i)>$ is compared according to the lexicographic ordering. This provision places the

highest priority on the value of $D_1(i)$, and successively lower priorities on the following components of the level gain.

What could be the second level gain criterion for distinguishing cells with the same D_1 value? Consider Figure 2.13. Here, c1 and c2 have the same gain $D_1(1) = D_1(2) = 0$. However, moving c2 increases the gain of c3 from 0 to 1, whereas moving c1 does not change any gains. Therefore c2 should be preferred to c1 as the next cell to be moved. Thus, $D_2(2)$ should be larger than $D_2(1)$.

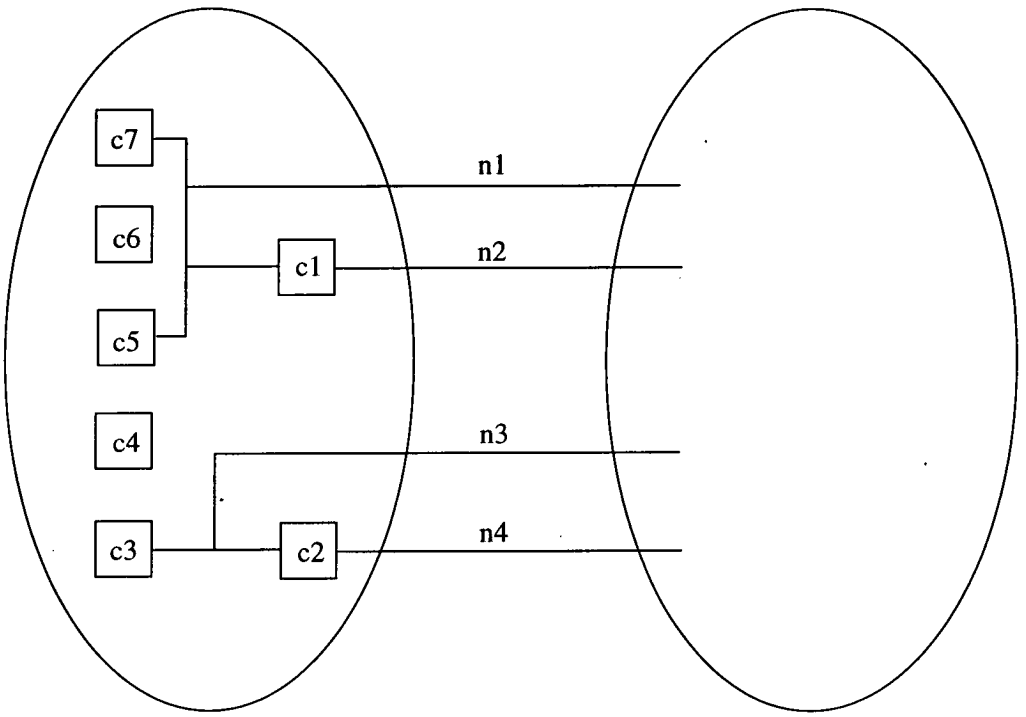


Figure 2.13 An example that the cell gain concept cannot distinguish

A generalisation of this concept to higher degrees of look ahead is based on the following definitions of gain.

Definition 2.9: Binding Number

The binding number $\beta_A(n)$ of a net n on side A of a partition is defined to be the number of unlocked cells of net n on side A, unless there is a locked cell of net n on

side A, in which case $\beta_A(n) = \infty$.

Definition 2.10: The *k*th-level Gain

The *k*th-level gain of a cell c_i on side A of a partition is defined as

$$D_k(i) = \sum_{n \in N_{pos,k}(i)} ct(n) \; - \sum_{n \in N_{neg,k}(i)} ct(n)$$

where

$$N_{pos,k}(i) = \{n \in N | c_i \in n,\ \beta_A(n) = k,\ \beta_B(n) > 0\}$$

$$N_{neg,k}(i) = \{n \in N | c_i \in n,\ \beta_A(n) > 0,\ \beta_B(n) = k - 1\}$$

Figure 2.14 shows a concrete example of how to calculate the gain vector of order k=3 of a cell.

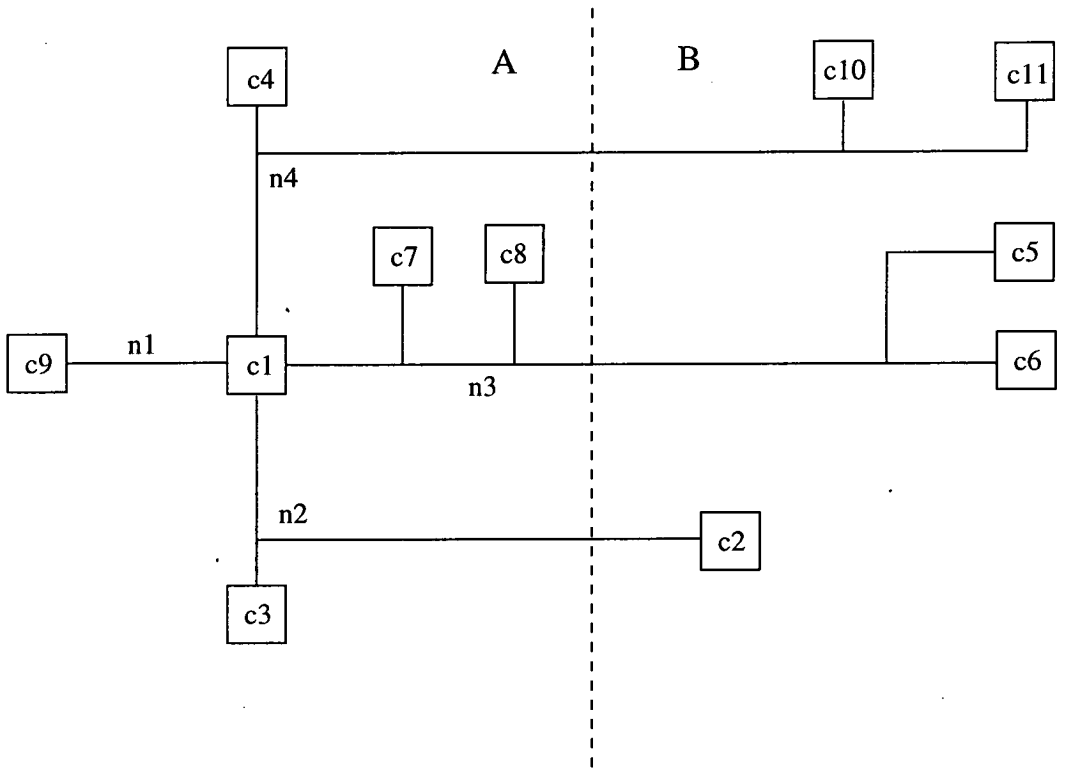


Figure 2.14 An example to calculate the gain vector

$k = 1$, the number of nets with binding number equal to 1 in side A is naught, and the number of nets with binding number equal to k-1 = 1-1 = 0 in side B is 1 (i.e. n1), therefore, $D_1(1) = 0 - 1 = -1$.

$k = 2$, the number of nets with binding number equal to 2 in side A is two (i.e. n2 and n4), and the number of nets with binding number equal to k-1 = 2-1 = 1 in side B is 1 (i.e. n2), therefore, $D_2(1) = 2 - 1 = 1$.

$k = 3$, the number of nets with binding number equal to 3 in side A is one (i.e. n3), and the number of nets with binding number equal to k-1 = 3-1 = 2 in side B is 1 (i.e. n3), therefore, $D_2(1) = 1 - 1 = 0$.

The gain vector of cell c1 for order 3 is $\Gamma(c1) = < -1, 1, 0 >$.

2.4. Other partitioning techniques

Constructive methods do not need an initial partition to be given. The starting point is in general the un-partitioned set. One of the most commonly used methods in this class follows an aggregation strategy, i.e., assigns one module at a time to a partition. Several versions of clustering techniques [61-62], [9-10] have been proposed over the years. This constructive algorithm is very fast, but the quality of the result is not good in general. It is in fact mostly used as a starting point for other methods such as iterative improvement.

Simulated annealing [30], [33], [34] is another technique of the iterative improvement variety. This algorithm starts at a random solution and makes stochastically chosen moves to modify that solution. Initially the moves which are accepted include a high proportion of moves which increase the solution's cost. As the algorithm progress, the proportion of such move is decreased until finally almost no moves that increase are accepted. Simulated annealing usually needs much more

running time than the Kernighan-Lin heuristic, but has a smaller cutsize [2].

Ratio cut [6], [100] partitioning method adopts an approach termed "ratio cut" as a metric in order to locate natural clusters in the circuit. This approach removes the constraint on subset sizes, and lets the ratio cut produce the subsets which are natural clusters in the circuit. It is difficult to use when tight control on the subset size is required.

2.5. Summary

The partitioning techniques presented in this chapter do solve some partitioning problems in certain application areas. But these techniques do not satisfactorily handle a large, complex circuit which has buses, timing critical paths, structural architecture and long shift registers, for example. A merge algorithm which has a flexible data structure and can be readily modified to suit various circuit requirements, is introduced in the next chapter.

CHAPTER 3

The Merge Algorithm

The previous chapter reviewed the techniques available to produce a required partition. This chapter introduces a novel partitioning approach referred to as the "Merge Algorithm" which obtains a desired partition by merging cells into groups.

The basic merge operations on two-terminal nets and multi-terminal nets are described. During the merge operations, some new cells will contain others cells and the nets incident to the new cells are changed. The configuration of a design will also be changed and the number of cells is reduced.

The data structure needed for representing a design is briefly described. Following this description, some merging strategies are discussed. Finally, the merge algorithm for partitioning a design is detailed.

3.1. Merge operation

3.1.1. The primitive merge operation

Before explaining the details of the primitive merge operation, it is worth emphasising some basic terms which were mentioned in the previous chapter and are illustrated below.

A cell which does not contain any other cell is called a basic cell. All single cells including a basic cell can be viewed as a group. This means that a group consists of at least one basic cell. A basic cell contains itself, so it forms a group itself.

A net with only two cells on each end is called a two-terminal net, whereas a net with more than two cells on it is called a multi-terminal net. Figure 3.1 shows these two kinds of nets.

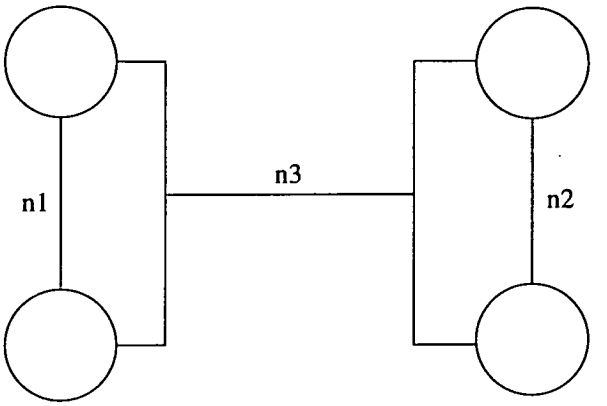
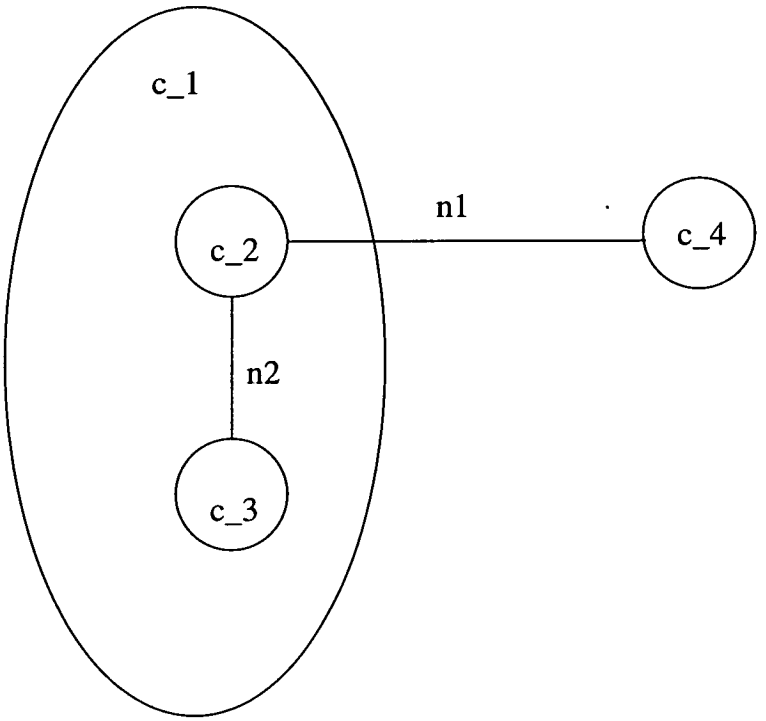


Figure 3.1 Two-terminal and multi-terminal nets

A net is called an external net to a cell when there are still other cells outside of this cell on the other ends of this net, otherwise it is called an internal net. Examples of external and internal nets are shown in Figure 3.2.

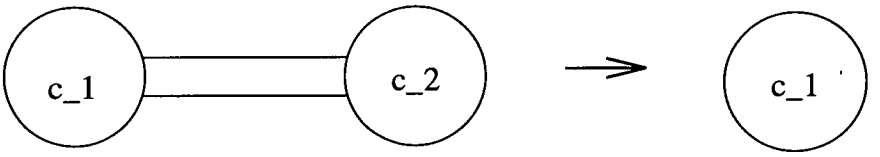
The basic function of the merge operation is to make one cell contain another cell. The cell covered by another cell is called an implicit cell, whereas the containing cell is called an explicit cell. The property of the implicit cell will be represented by the explicit cell. In practice, after merging, the cell name of the explicit cell is used as the new cell name and the cell name of the implicit cell is not used anymore. An example shown in Figure 3.3 illustrates how the primitive merge operation proceeds.



— n1 is an external net to cell c_1

— n2 is an internal net to cell c_1, but is an external net to cell c_2 and c_3

Figure 3.2 The external and internal net



— c_1 is an explicit cell.

— c_2 is an implicit cell.

Figure 3.3 The primitive merge operation

3.1.2. The basic merge operation on two-terminal nets

A graph with two-terminal nets only is shown in Figure 3.4. There are 6 cells and 15 nets. It is obvious that the optimal solution for this graph will be, when it is divided into two groups, a cutsize of 1.

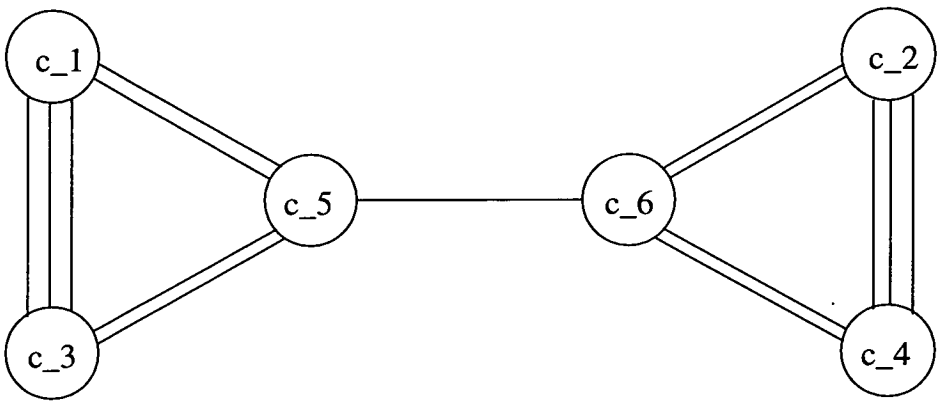


Figure 3.4 A graph with two-terminal nets

The main objective in merging cells is to look for the cells with the maximum inter-connections and merge them. This will be demonstrated by the following merge steps:

step(1)

There are four possibilities in selecting cells to merge, which are

(1) merge c_3 to c_1.

(2) merge c_1 to c_3.

(3) merge c_4 to c_2.

(4) merge c_2 to c_4.

For example, the possibility (1), i.e. merge c_3 to c_1, is taken. After merge, 3 nets between c_1 and c_3 are buried inside the new cell c_1, i.e., these 3 nets become internal nets of c_1. The result of this merge is shown in Figure 3.5.

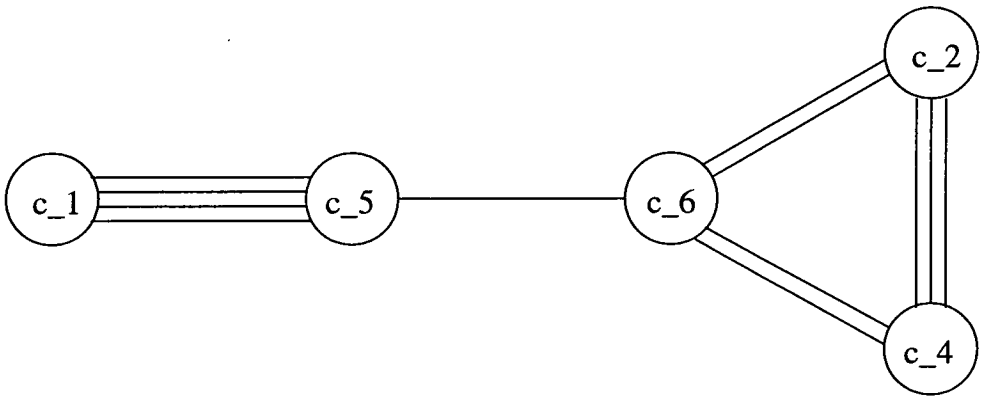


Figure 3.5 The result of merge c_3 to c_1

step(2)

From the graph above, the next merge operation is going to be

(1) merge c_5 to c_1 or

(2) merge c_1 to c_5.

Taking the first choice of merging c_5 to c_1, this time four nets become internal nets of c_1. The result is shown in Figure 3.6.

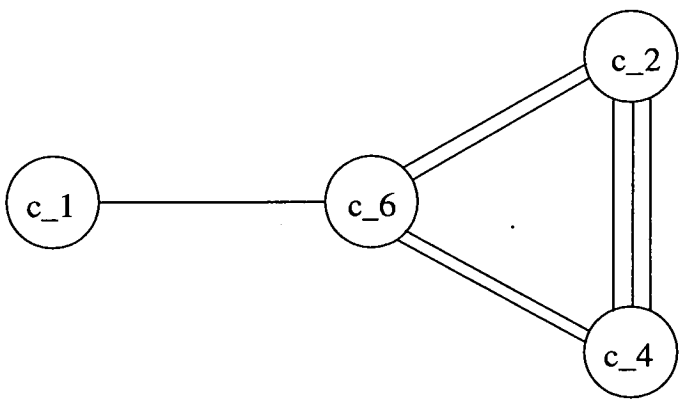


Figure 3.6 The result of merge c_5 to c_1

step(3)

Following the same approach, the operation of merging c_4 to c_2 takes place. There are 3 nets which become internal nets of c_2. The resulting graph is shown in Figure 3.7.

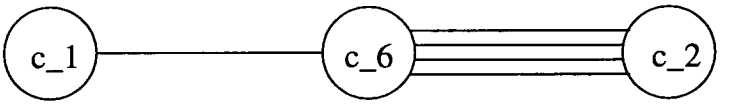


Figure 3.7 The result of merge c_4 to c_2

step(4)

Finally, the operation of merging c_6 to c_2 is processed. In this case four nets become internal nets of c_2. The final graph is a partition with an optimal solution, the cutsize of which is equal to one. In this partition group c_1 contains c_1, c_3, c_5 and group c_2 contains c_2, c_4, c_6. The result is shown in Figure 3.8.

Figure 3.8 The final partition

3.1.3. The merge operation on multi-terminal nets

A graph with mixed multi-terminal nets and two-terminal nets is shown in Figure 3.9. There are 6 cells and 11 nets, two of which are multi-terminal nets that are n1 and n2. It can be easily identified that in this graph there is an optimal partition with cutsize equal to one if it is divided into two groups.

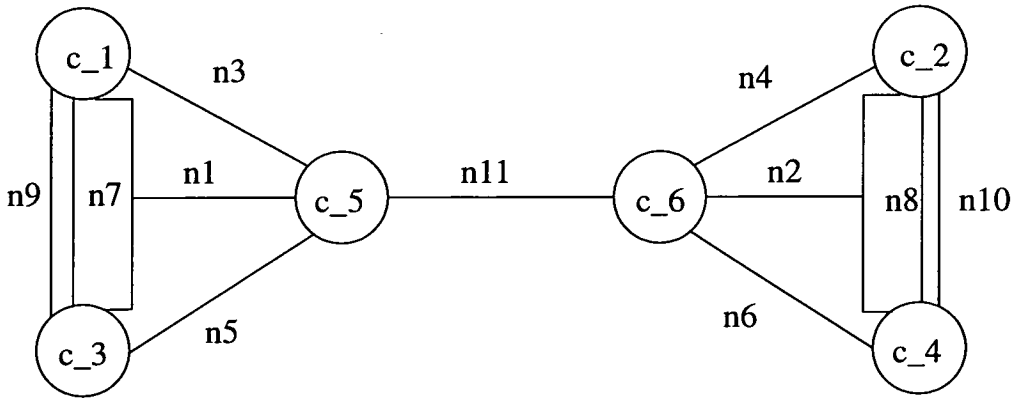


Figure 3.9 A graph with multi-terminal nets and two-terminal nets

The same approach to merging cells introduced in the above section is applied again to this graph. The step by step demonstrations are as follows:

step(1)

Merge c_3 to c_1. The two two-terminal nets (n9 and n7) between c_1 and c_3 become internal nets to c_1. The net n1, a multi-terminal net , still exists and is an external net to c_1 and becomes a two-terminal net. The resulting graph is shown in Figure 3.10.

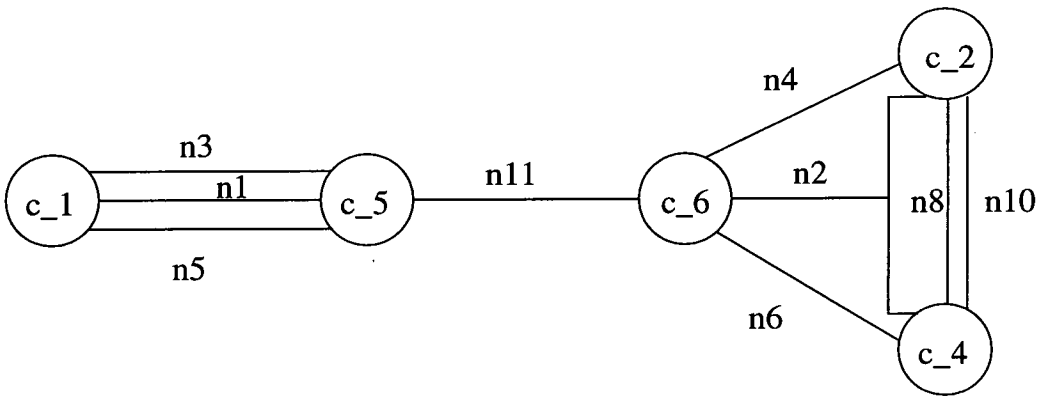


Figure 3.10 The resulting graph after merge c_3 to c_1

step(2)

Merge c_4 to c_2. The two two-terminal nets (n8 and n10) between c_2 and c_4 become internal nets of c_2. The net n2, a multi-terminal net, still exists

and is an external net to c_2, and becomes a two-terminal net. The resulting graph after this merge is shown in Figure 3.11.

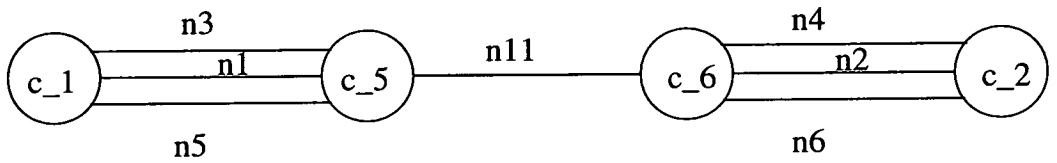


Figure 3.11 The graph after merge c_4 to c_2

step(3)

Merge c_5 to c_1. There are three two two-terminal nets (n3, n1, n5) between c_1 and c_5 which become internal nets to c_1. The resulting graph is shown in Figure 3.12.

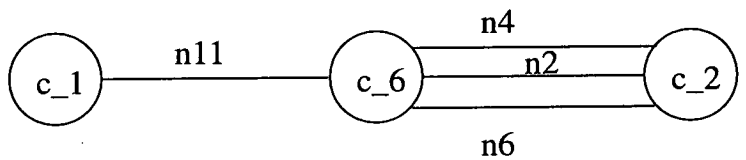


Figure 3.12 The resulting graph after merge c_5 to c_1

step(4)

In this final step, merge c_6 to c_2. The nets n4, n2, and n6 become internal nets to c_2. The resulting graph shown in Figure 3.13 comes to an optimal partition with the cutsize equal to one. In this partition group c_1 contains cells c_1, c_3 and c_5 and group c_2 contains cells c_2, c_4 and c_6.

Figure 3.13 The resulting graph after merge c_6 to c_2

When the merging sequence presented above is analysed, it is not difficult to observe two properties about nets which are explained below:

(1) For two-terminal nets, after merging the nets between cells become internal nets to the explicit cell. Hence benefits arise from burying the number of nets between a pair of cells in the explicit cell indirectly reducing the cutsize.

(2) For multi-terminal nets, after merging the nets between cells are still external to the explicit cell. The merge does not imply the reduction of the number of nets between cells. This is a significant difference from two-terminal nets which needs further attention.

3.2. Graph representation

3.2.1. Using a matrix to represent a graph

The most straightforward representation for graphs is the so-called matrix representation. The matrix representation for the graphs in Figure 3.4 and Figure 3.9 is shown in Figure 3.14. Although these two graphs have different configurations, they have exactly the same matrix representation which shows the number of connections between each pair of cells in each entry of the matrix.

	c_1	c_2	c_3	c_4	c_5	c_6
c_1	X	0	3	0	2	0
c_2	0	X	0	3	0	2
c_3	3	0	X	0	2	0
c_4	0	3	0	X	0	2
c_5	2	0	2	0	X	1
c_6	0	2	0	2	1	X

Figure 3.14 A matrix representation for the graphs in Figure 3.4 and Figure 3.9

This is a symmetric matrix where the data in the upper triangle contains all the connection information and the lower triangle can be considered as redundant. The matrix representation is satisfactory, only if the graphs to be processed are dense and of a reasonable size. When circuits become large, the matrix is usually large and sparse.

3.2.2. Using a linked list to represent a graph

Using a matrix to represent a graph is inefficient for large, sparse matrices since a huge amount of memory is wasted.

There is another representation that is more suitable for graphs. This representation has both a list of cells (vertices) and, for each cell, another list of cells (edges) related to each cell. This can be easily implemented with linked lists. Figure 3.15 shows the linked list for the above matrix. Note that at each cell in the list of vertices only cells having higher index number are shown as being connected to that cell. This corresponds to ignoring the lower triangle of the matrix representation.

It is often necessary to associate other information with the vertices or edges of a graph to allow it to model more complicated objects or to save bookkeeping information in complicated algorithms. Extra information associated with each vertex and edge can be put in adjacency list nodes.

The foremost advantage of using a linked list is its flexibility. Memory usage is efficient in a computer implementation. With dynamic allocation of memory for linked lists, they are much better suited to a computing solution than matrices.

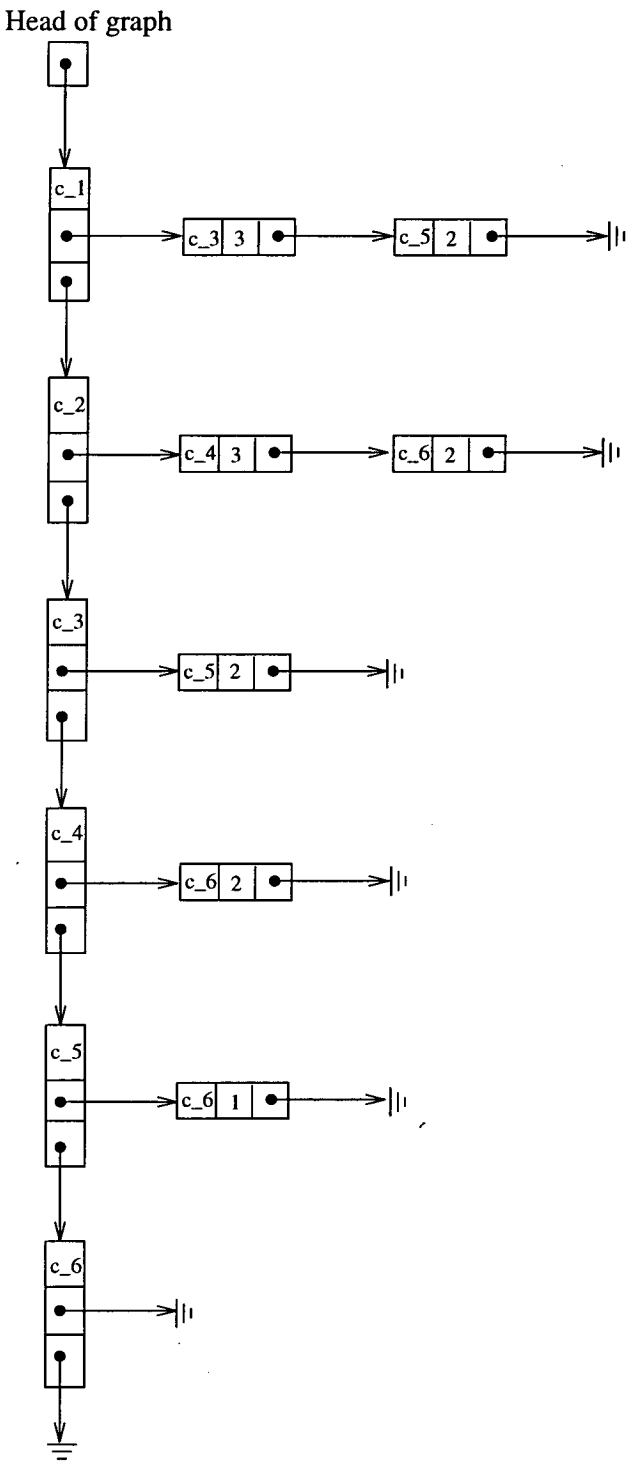


Figure 3.15 The linked list for a graph

3.3. Merging strategy

The most common method of partitioning a system is to divide it into groups so that the total number of interconnections between groups is minimised. Conversely, maximising the number of connections between cells inside groups can achieve the same objective of minimising the interconnections between groups.

The objective of merging in a system with a large number of cells is to obtain a successful partition by means of maximising the number of connections which are inside groups. The methods used to guide the merging operation into a partition are based on many factors including the size constraints (the maximum number of cells in the groups), the required number of groups in the final partition, and the balance of the partition. Partitioning techniques can be classified into four categories which are free merge, free merge with size constraint, merge in-turn, and merge in stages as will be discussed below.

3.3.1. Free merge

In this method the merge operation has the most degrees of freedom to make the choices of pairs of cells to merge. The merge will proceed according to the number of connections between cells, the pair of cells with the maximum connections is chosen to merge into a group which is viewed as a new cell in the system. No size limitation is applied to each group which may therefore grow large so long as the merge condition is satisfied. The same procedures are continued until the number of groups matches the number of blocks in a partition desired.

In the final result, some blocks in a partition could contain a large number of cells while some others may contain a few cells. This is because, during the merge procedure, the groups containing more cells tend to have stronger relations with other cells, so that the larger groups keep absorbing cells and growing continuously

without giving opportunities to other groups to choose candidates to merge. This can easily happen if there is no constraint on the size of each group.

3.3.2. Free merge with size constraint

To solve the unbalanced partition problem arising from the free merge method described in the last section, a size limit to groups is applied. Once a group reaches this limit, no more cells are merged which gives a chance for the rest of the groups to select candidates to merge. This can prevent a group from dominating the whole merging process and produce a better balanced partition.

3.3.3. Merge in-turn

A perfectly balanced partition is when the cells are evenly distributed between each individual group. To achieve this goal, it is necessary to decide the number of groups, denoted by *ng*, which a system will be partitioned into, select the first *ng* different pairs of cells with the first *ng* maximum connection number to form the first *ng* leading groups, and then merge the rest of cells with closest relations with each of leading group in turn until no cells are left. A fairly evenly distributed partition can be obtained by using this technique. The drawback of this approach is that it constrains cells to be merged to leading groups and does not give other cells chances to establish other groups which could have good structures for later merging and so achieve a good final partition.

3.3.4. Merge in stages

In the real world, a system is usually organised in a hierarchical manner. It can be made up of several functional blocks which can be decomposed into more functional blocks in the next lower level, each of which can be further divided into even more functional blocks until the bottom of the system is reached. In such a hierarchical

system, cells inherently possess some degree of group-oriented characteristics. A number of cells may form a small group in the lowest level of a system, some small groups may construct a functional block in the intermediate level and a few functional blocks may form a partition of a system. This implies that a small size constraint may be applied to each group in the beginning, then the size constraint on each group is enlarged in each successive stage, until the desired partition is achieved. Figure 3.16 shows a merge in stages procedure. The merge can progress through several operating stages, each stage denoted by i, where $1 \leq i \leq L$, and L is the maximum possible number of stages valid in a partition.

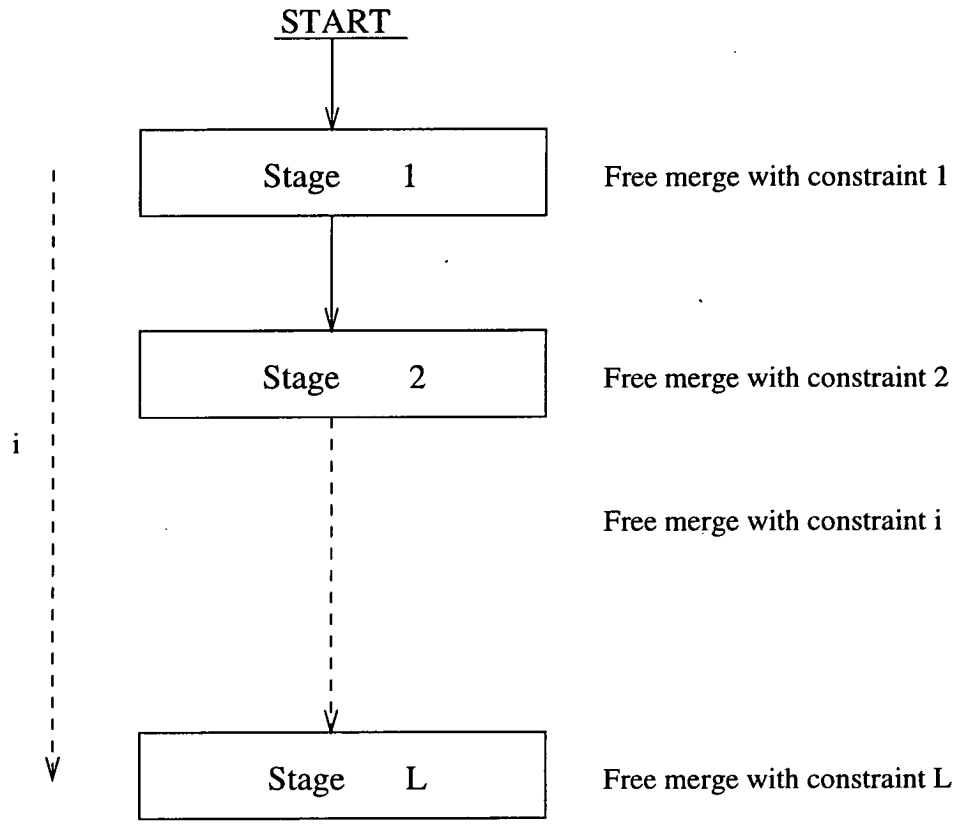


Figure 3.16 Diagram illustrating the merge in stages method

Due to the small size constraint in the beginning, a lot of small groups will be formed, and the system arrives at a new configuration in which the small groups in the next merging stage have equal chances to compete with one another to decide

which pair of groups will be the next pair to be merged under the enlarged size constraint. This is the main advantage of the merge in-stages method which prevents a group from growing unimpeded to the final size constraint, and allows closely related cells to be grouped in the early stages.

3.4. Partitioning with a merging algorithm

3.4.1. Data structures

A linked list, called the cell-net list, is used. It lists nets connected to an individual cell and shows the number of cells on a net. This number is separated into two fields, one of which gives the number of cells included in this cell, and the other gives the number of cells outside of this cell. This cell-net list is used to control and calculate the interconnections between cells/groups during the merge operation and the merge operation includes merging cells in this list to keep the interconnections between cells correct. Figures 3.17 and 3.18 give the cell-net list for the circuit shown in Figure 3.9.

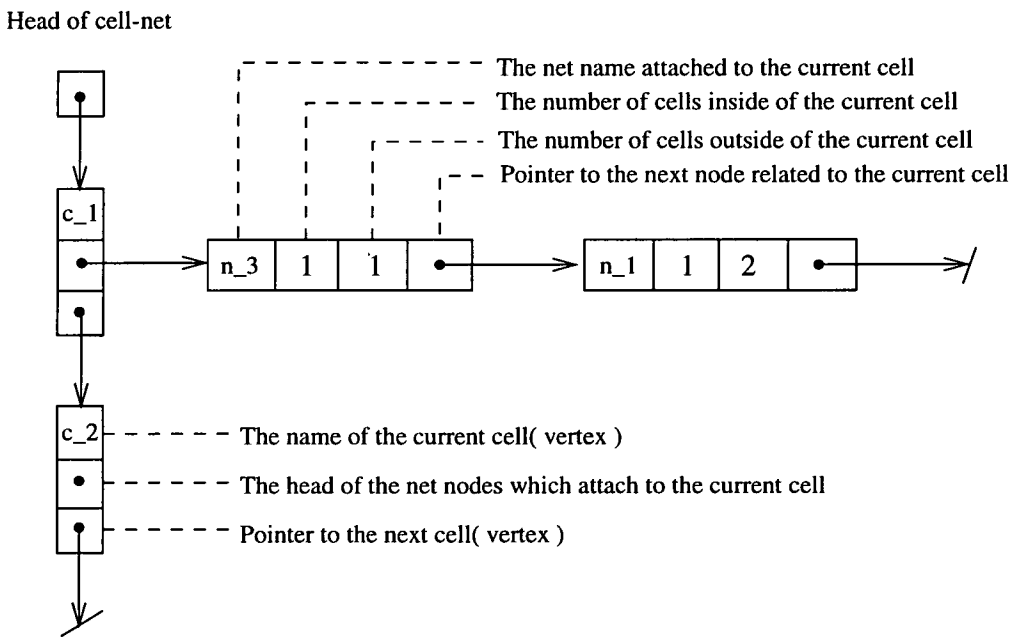


Figure 3.17 Diagram defining the cell-net list

A software representation of a circuit, called a graph, is established to define the connections between cells. Then a merge sequence is set up by listing cells in descending connection order from the maximum value. This descending order in connection number establishes the merging priority of cells. Figures 3.19 and 3.20 give the graph and merging sequence for the circuit shown in Figure 3.9.

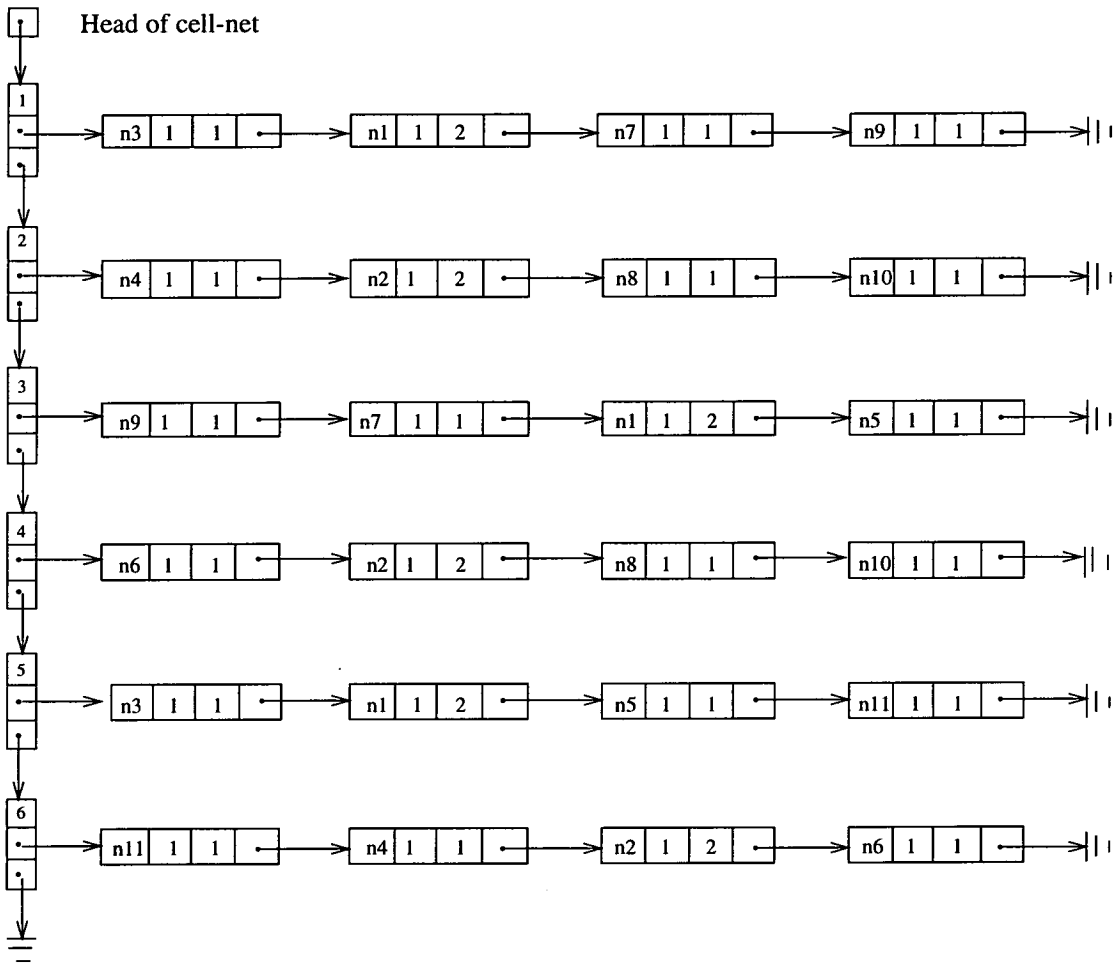


Figure 3.18 Cell-net list for the circuit shown in Figure 3.9

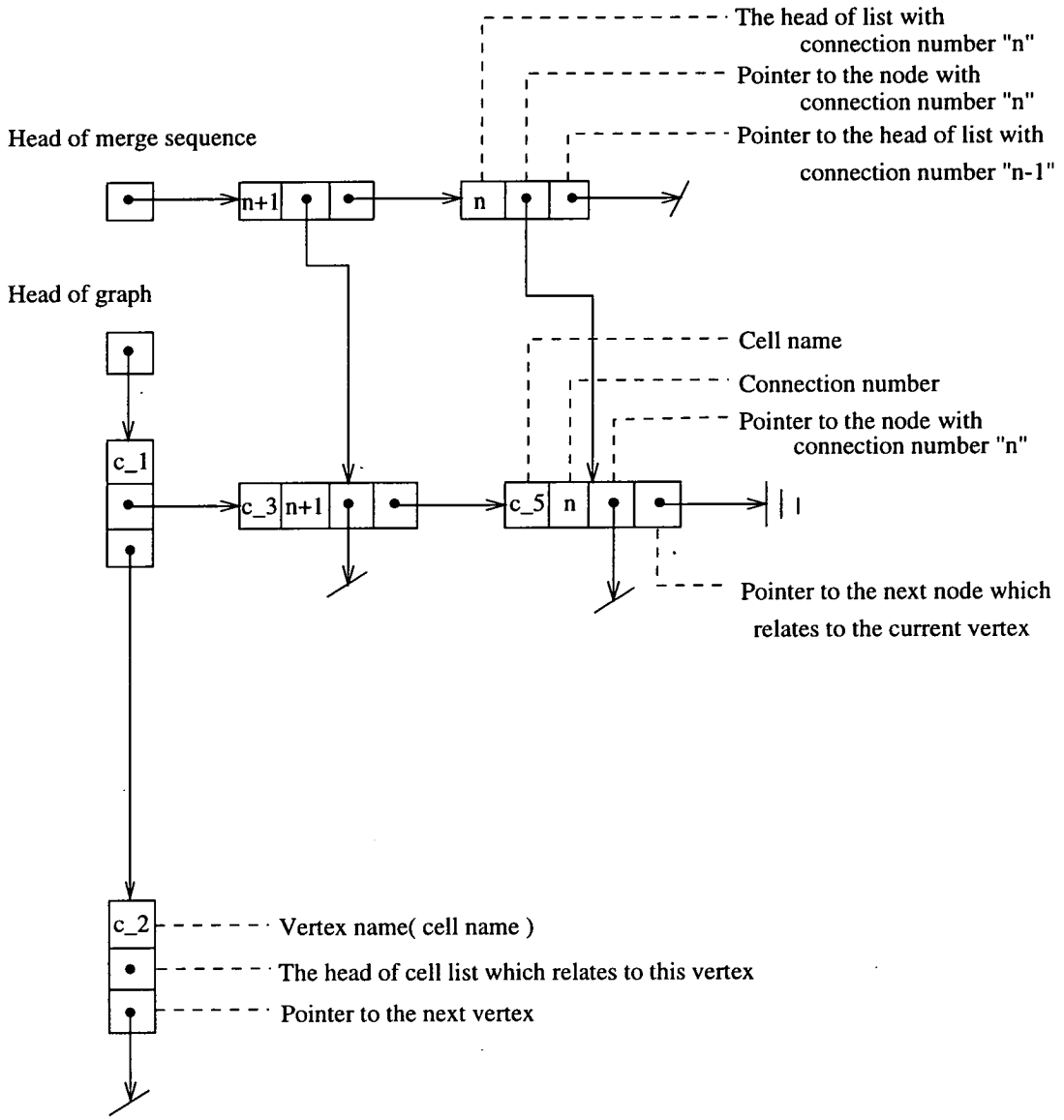


Figure 3.19 Diagram defining a graph and the merge sequence

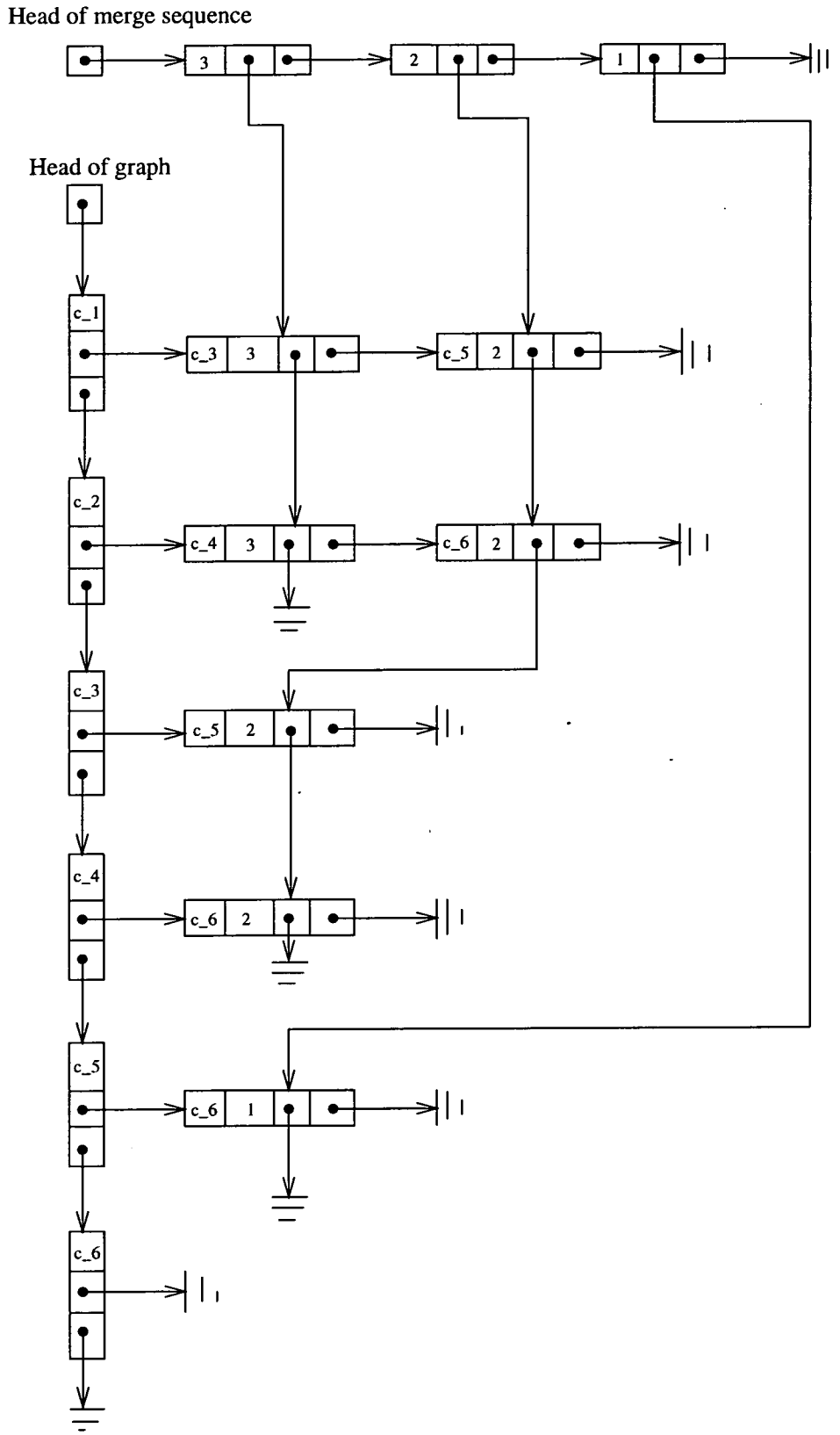


Figure 3.20 Graph and merge sequence for the circuit shown in Figure 3.9

3.4.2. Merging algorithm

The merging algorithm uses circuit information in the form of a netlist with a total number of basic circuit cells, C. The algorithm can be considered as four separate linked sections. The first section implements a free merging operation with size constraint which allows merging to proceed with a constraint only on the size of the resulting group of cells. The free merging operation progresses in stages with the size constraint on the merged group increasing as the stage number increases. An initial size constraint, S_0, and the number of stages to be used must be selected and L must be set. This is discussed further after the results illustrating the performance of the algorithm are presented below. An iteration at each stage is complete once all available cells have been merged. Figure 3.21 shows a flow chart of free merging with size constraint, the algorithm of which is defined in pseudo C code as follows:

```
/*
 * C : The number of cells in the current circuit
 * ng: The required number of groups in the partition
 * S0 : Initial size constraint
 * Si : The ith-stage size constraint
 * Cx : The number of cells in cell_x
 * Cy : The number of cells in cell_y
 * Note : A merging stage corresponds to the merging of
 *        all available listed cells. Let i be the stage
 *        index and let imax be L. Cells linked by
 *        the largest number of connections are selected
 *        for merging.
 */
read_data() ;
create_cell-net() ;
create_graph() ;
setup_merge_sequence() ;
for( i = 1; i ≤ L; i++ ){
  Si = 2Si-1 ;
  while( select_candidate( cell_x, cell_y ){
    if( C ≤ ng )
      break ;
    if( Cx + Cy < Si )
      merge( cell_x, cell_y ) ;
  }
}
output() ; /* output a reduced graph */
```

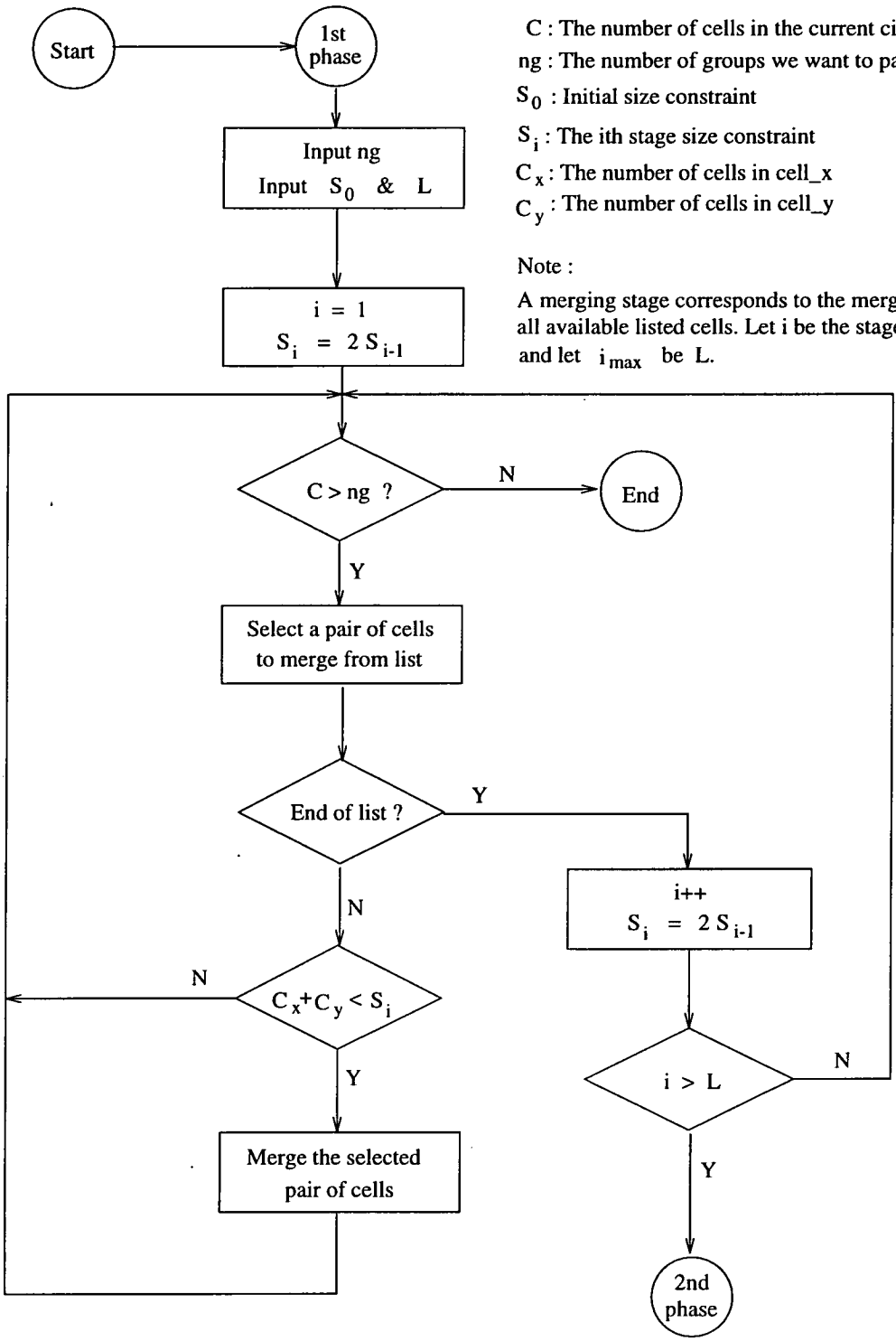


Figure 3.21 The flow chart of free merging in stages

A merge operation starts by merging a pair of cells into a group; the initial size constraint S_0 on this group will be at least 2. To test all possible constraints on a

partition it is necessary to set $S_0 = 2$. The subsequent size constraints are applied by using $S_i = 2S_{i-1}$ where i starts at 1. The maximum i (i_{max}) is the total number of stages (L) to be used. To set an appropriate L the following inequality;

$$C > 2 * 2^L \tag{3.1}$$

is used to give;

$$L < \frac{\ln C}{\ln 2} - 1 \tag{3.2}$$

In practice the maximum integer value of L satisfying this inequality is used.

The netlist resulting from the free merge operation is presented to the second section and will be much shorter than the original circuit netlist. The second section of the merge algorithm proceeds in a similar way except that leading groups are selected to ensure that results from the next section will give an approximately equal share of cells in each of the groups in the final partition. A leading group is defined as being the first *ng* pair of cells selected on the basis of maximum number of connections. A flow chart for the selecting leading groups operation is shown in Figure 3.22 and a listing illustrating the second section algorithm is shown below:

```
/* k : Leading groups index, S_f : Final size constraint */

 input() ; /* using the output of section 1 as input data */
 for(k = 1; k ≤ ng; k++ ){
   while( select_candidate( cell_x, cell_y ) ){
     if( C ≤ ng )
       break ;
     if( C_x + C_y < S_f ){
       merge( cell_x, cell_y ) ;
       leading_group[] = cell_x ;
       break ;
     }
   }
 }
 output() ;
```

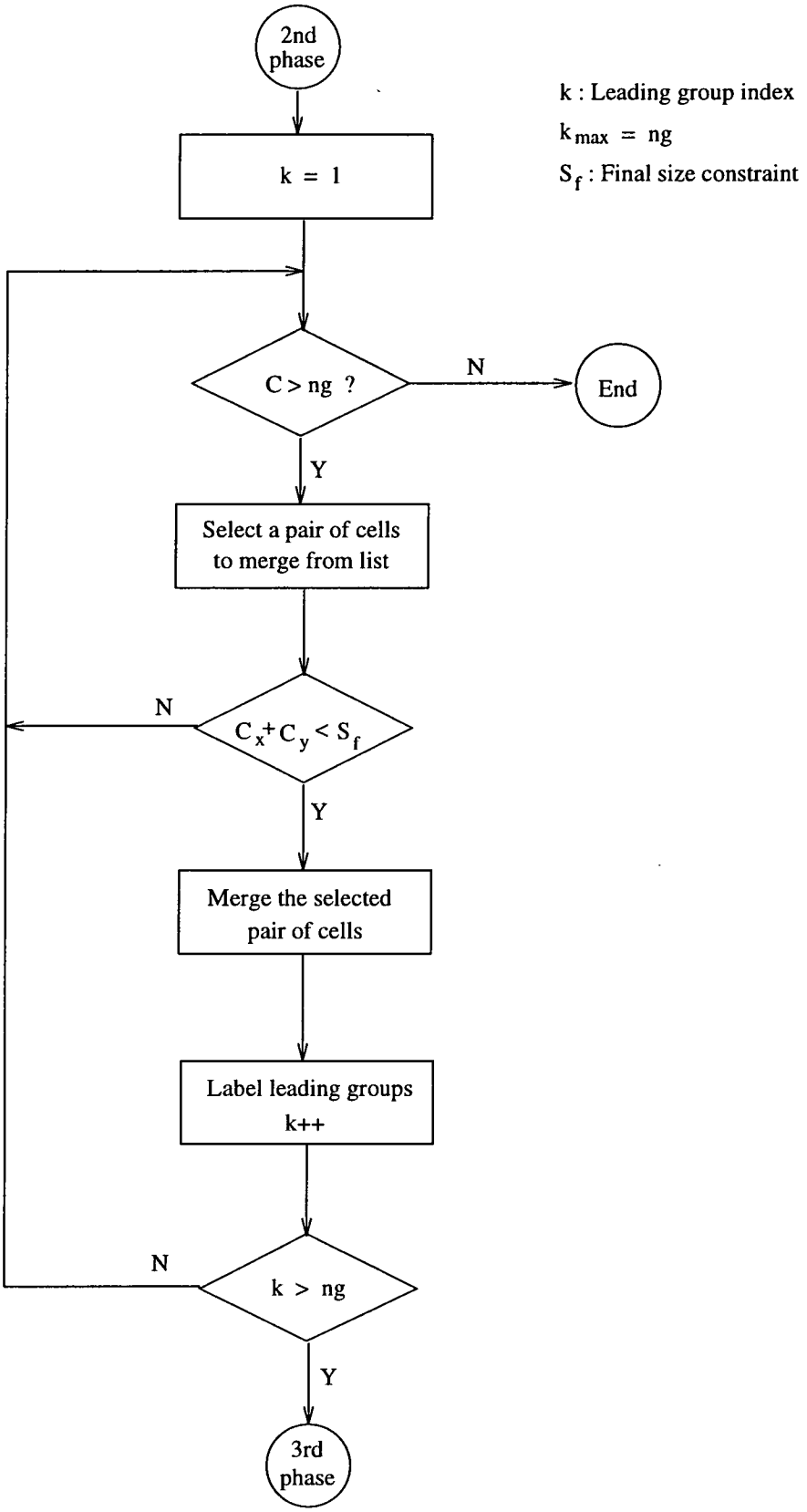


Figure 3.22 The flow chart of the selecting leading groups operation

The leading groups are used in the third section. In this section merging proceeds similarly except that the pair of cells to be merged must include a leading group. The flow chart of merging with leading groups is shown in Figure 3.23 and the listing illustrating the third section of the merge algorithm is shown below:

```
input() ;
while( select_candidate( cell_x, cell_y ) ){
  if( C ≤ ng )
    break ;
  if( C_x + C_y > S_f )
    break ;
  if( cell_x = leading_group[] ⊕ cell_y = leading_group[] )
    merge( cell_x, cell_y ) ;
    /* "⊕" stands for exclusive OR */
}
output() ;
```

A final section has been included to account for unconnected cells where the number of connections is zero. The flow chart of merging the rest of cells is shown in Figure 3.24 and the algorithm for this section is shown below:

```
input() ;
while( C > ng )
  if( cell_x != leading_group[] )
    merge( leading_group[], cell_x ) ;
output() ;
```

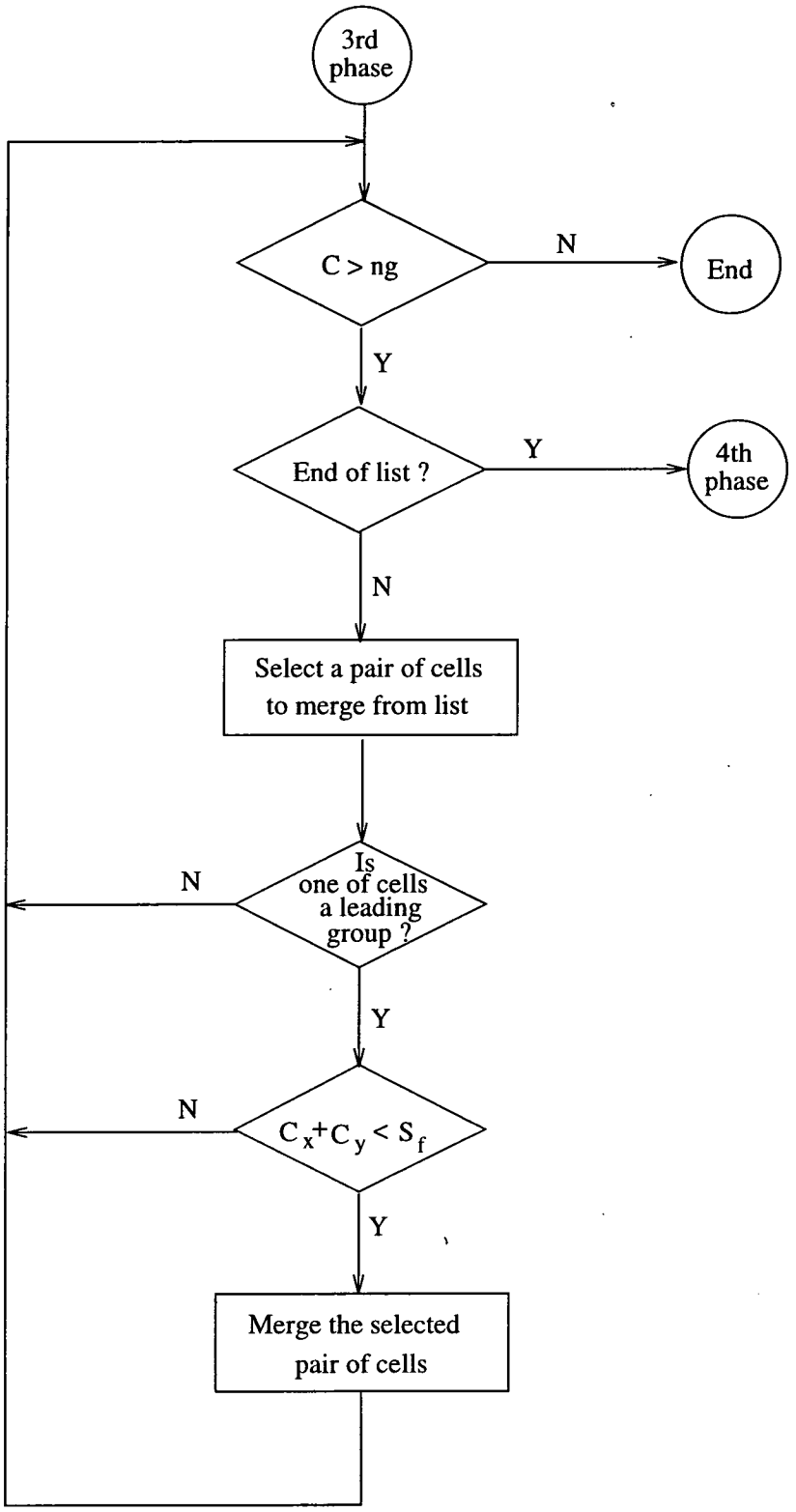


Figure 3.23 The flow chart of merging with leading groups

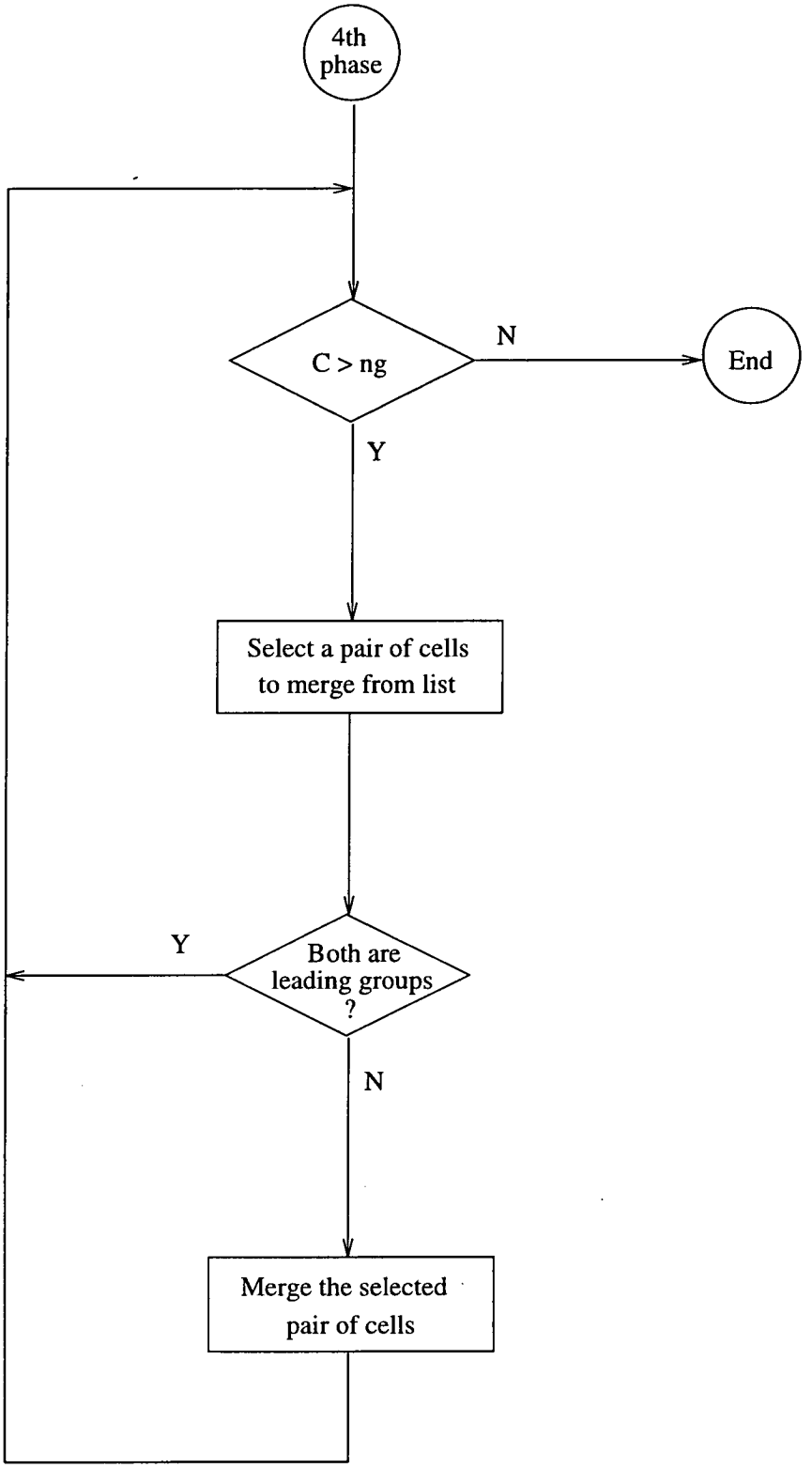


Figure 3.24 The flow chart of merging the remaining cells

3.5. Constraints on merge algorithm

The operation of the algorithm can be further constrained by other design parameters such as the number of pins and timing requirements. In the case of the number of pins a check is made before each merge operation to determine whether the resulting group of cells can be realised within the input and output pin constraints of the target architecture. To satisfy timing requirements it is necessary to know the critical paths in the source design and then ensure that cells on critical paths are merged. This is achieved in practice by an initial grouping operation before the merge algorithm is used. If area limitations prevent all critical paths from being dealt with in this way then any paths between partitioned groups that are in fact critical paths should be reserved for the fastest board wiring of the target system.

If function blocks are defined for the target system the designer of the source circuit could formulate his design in terms of these function blocks. A drawback of this approach is that a circuit designed for ASIC implementation may perform in a different way if reformulated to suit available function blocks. Alternatively the initial design could be flattened and then formulated in terms of function blocks. In this case the originally defined critical paths will remain as defined but new critical paths will appear in the flattened circuit and must be discovered to ensure the best performance can be achieved. An automatic technique for identifying critical paths is required.

3.6. Parallel process on merge algorithm

A feature of this merge algorithm is that its modularity can be used to create a pipeline of parallel operating algorithms with smaller graphs. The parallel algorithm can be implemented by a parallel language that enables the algorithm to execute in parallel in a multi-processor environment. Searching large amounts of data at one

time can be very slow. Processing data in smaller pieces helps to reduce searching time. Therefore it is necessary to write a data parallel program to allow smaller amounts of data to be distributed to each individual processor. This will result in a higher partitioning speed for large circuits. The details of a parallel implementation of the merge algorithm will be discussed in chapter 6.

3.7. Summary

This chapter has presented a set of merging strategies, namely: free merge, free merge with size constraint, merge in-turn, merge in stages. These strategies are all based on the simple concept that cells having the maximum number of connections should be the first to be merged. Applying different size constraints or giving the merge algorithm a different free degree of selecting candidates to merge can produce different results.

The next chapter presents how the merge algorithm is implemented by means of a programming language.

CHAPTER 4

Implementing the Merge Algorithm

The merge algorithm discussed in the previous chapter has been implemented by means of the C programming language (see Appendix D). This software was written to interface with the InCA Virtual ASIC [7]. All source netlists were initially converted to InCA netlist format referred to as "CIF" file format (see Appendix A for detail) which describes a circuit in a hierarchical manner and is different from the Caltech Intermediate Form (CIF) which is a standard machine readable form for representing integrated system layouts.

The software for implementing the merge algorithm is separated into two parts. The first part is called "Data Preparation" (see Appendix C) which will flatten the circuit to acquire the detail connections between the basic gates, the flattened circuit will be passed to the parser to create a cell list and a net list which are used to create the cell-net list and the cell-net list is further used to generate the graph which represents the circuit. Figure 4.1 shows the data preparation flow chart. The second part is the implementation of the merge algorithm itself. The cell-net list and the graph will be used by the merge program under the size constraint on groups of cells, and the merge operations will continue until the desired partition is reached. Figure 4.2 shows the flow chart of the implementation of the merge algorithm.

The data structure for implementing the merge algorithm was introduced in chapter 3. In this chapter the detailed data structure for implementing the merge algorithm will be described in Section 4.1, and the procedures for implementing the merge algorithm follow in Section 4.2.

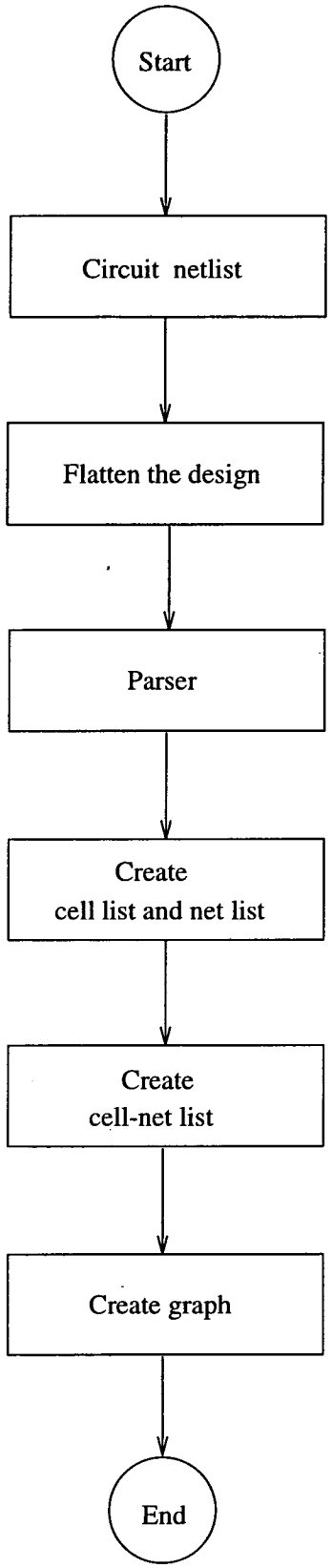


Figure 4.1 The data preparation flow chart

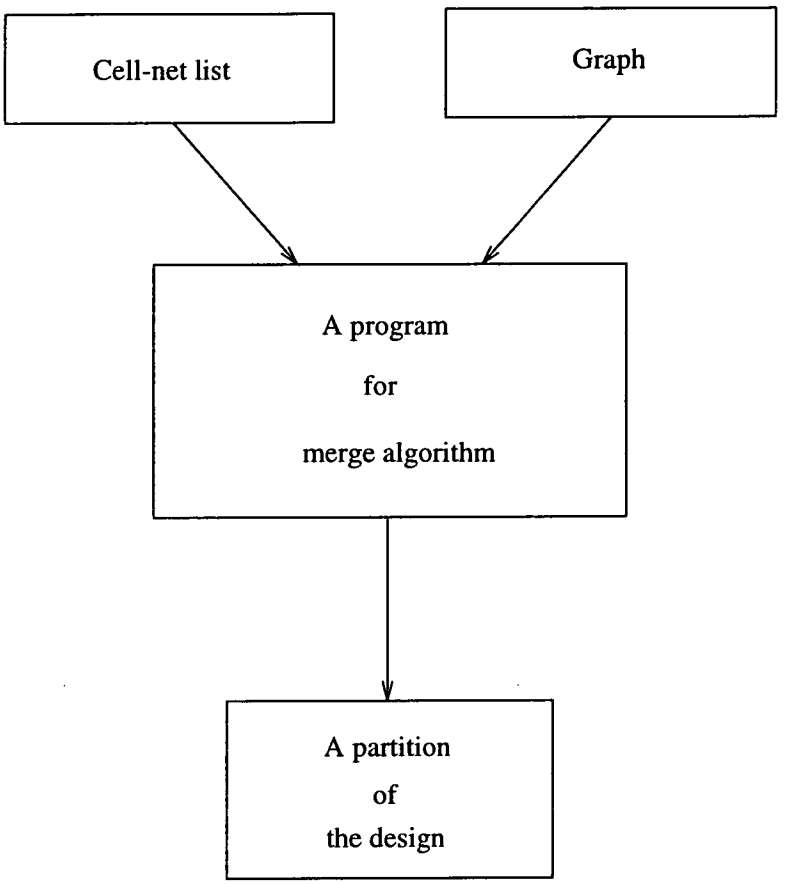


Figure 4.2 Flow chart showing the use of the merge algorithm to partition a design

4.1. The data structure for implementing the merge algorithm

A cell list is required which is generated from the original circuit "CIF" file. This list describes how the components in the circuit are interconnected. The cell list consists of a head of cell list which points to where the circuit is, the cell nodes which represent the cells themselves and point to a list that shows what nets are incident to the cells, and the net nodes which contain information related to the nets and the cells. A descriptive diagram to illustrate the structure of the cell list is shown in Figure 4.3. A complete cell list of Figure 3.9 is shown in Figure 4.4. Figure 3.9 is reproduced in Figure 4.5 for convenience. The cells are listed in the vertical and the related nets are listed in the horizontal direction.

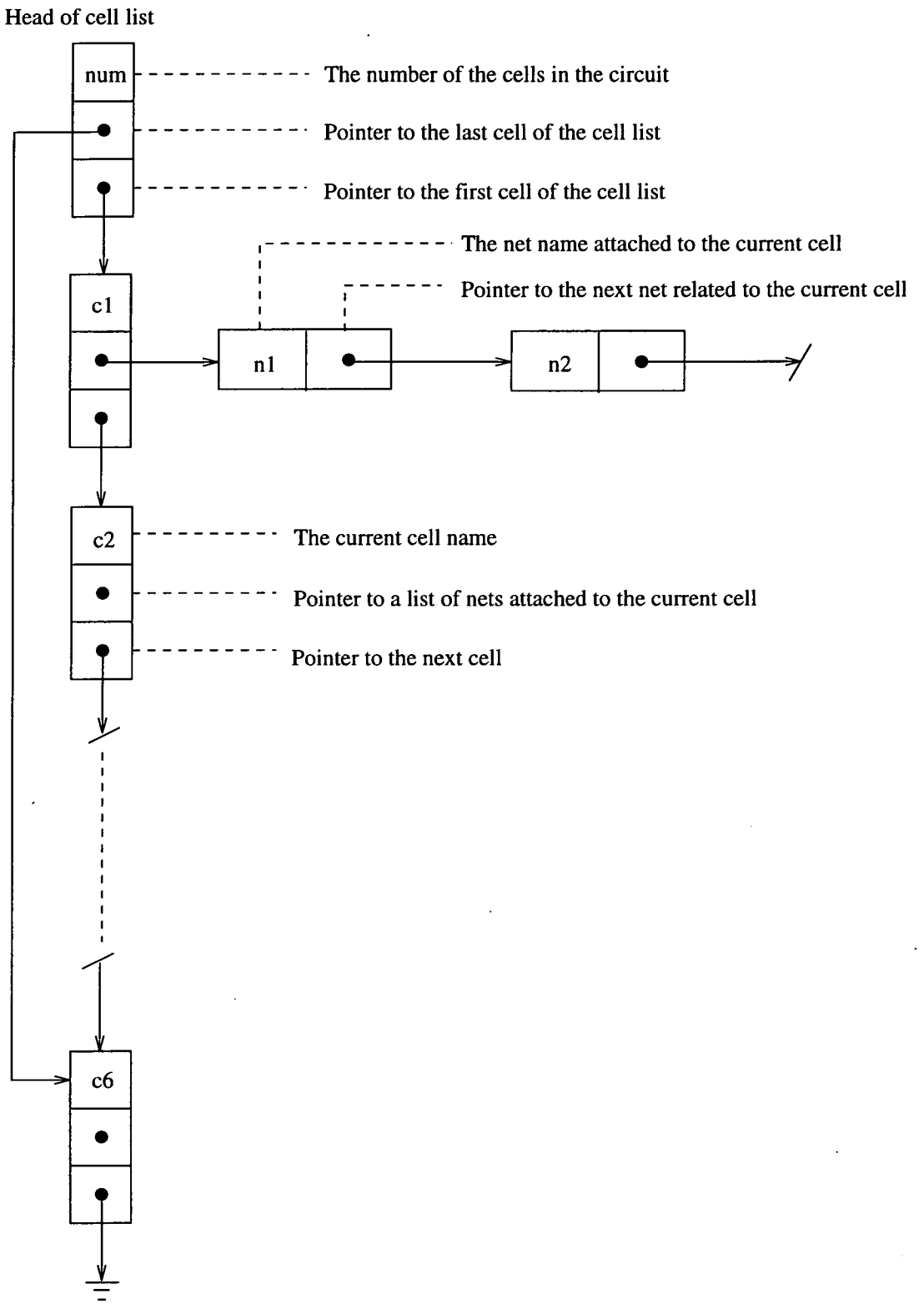


Figure 4.3 A diagram showing the structure of cell list

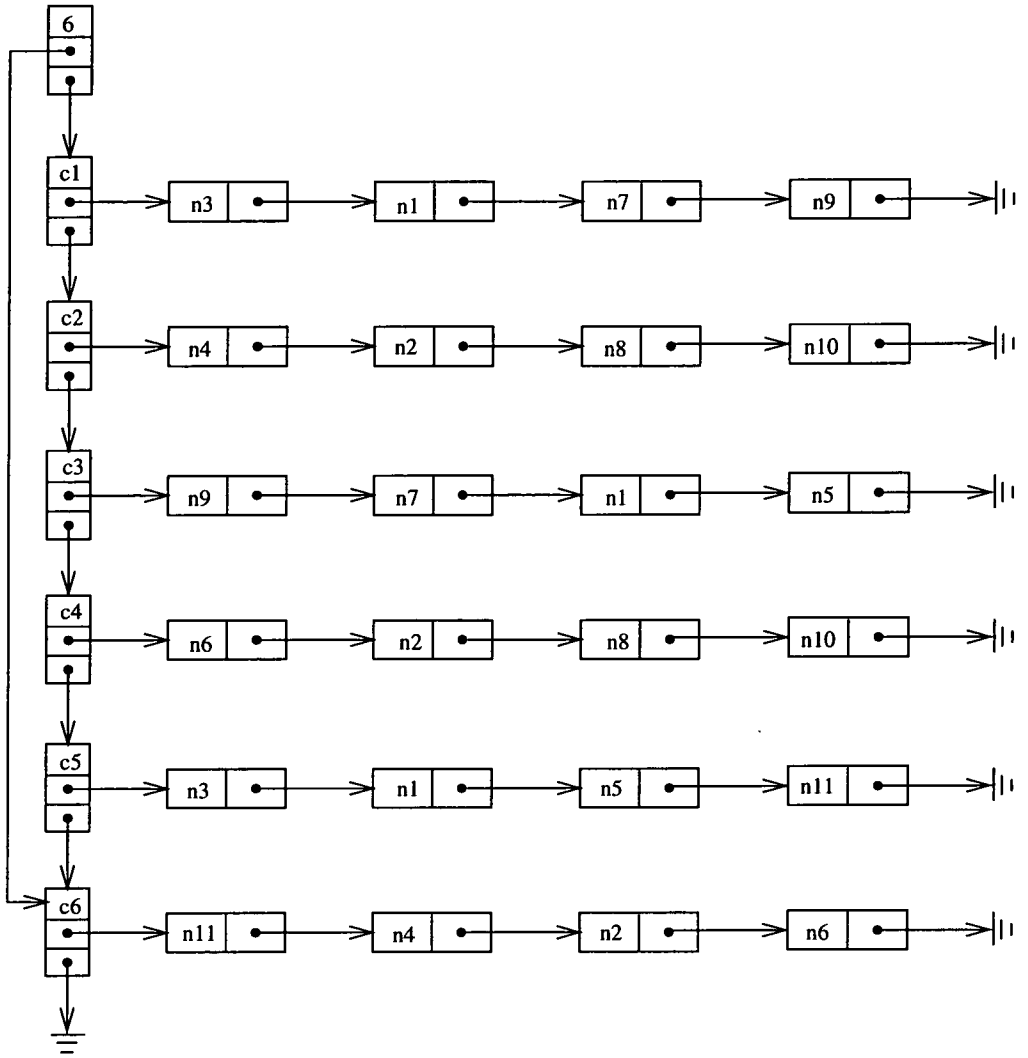


Figure 4.4 A complete cell list of Figure 3.9

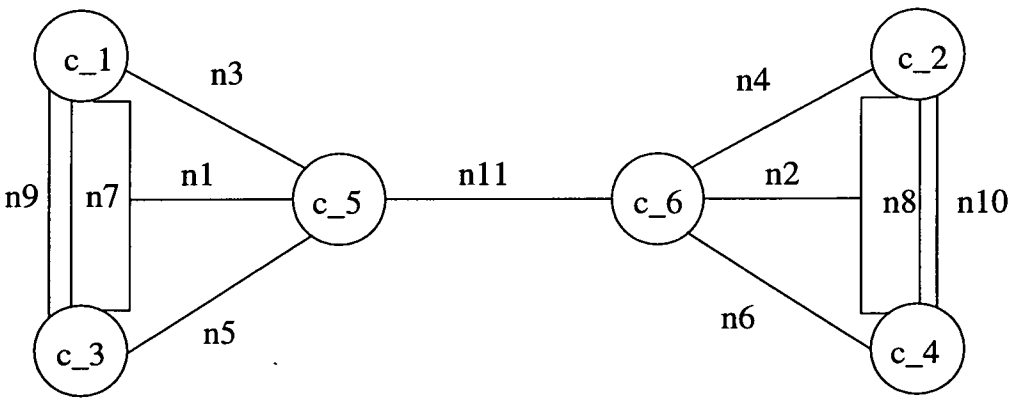


Figure 4.5 A copy of Figure 3.9

A net list is derived from the cell list. Basically it contains the same information as the cell list, the only difference is the nets are arranged in the vertical and the cells which are connected to the net are arranged in the corresponding horizontal list. Its main purpose is to facilitate generating another list called cell-net list which will be discussed later. The net list of Figure 3.9 is shown in Figure 4.6.

A cell-net list is created from both cell list and net list. In this new list the cell is called band-cell and the net is called band-net, this choice is made because the cell node includes information about what nets are completely inside this cell and the net node includes information about the number of cells outside this cell on a certain net and the number of cells inside this cell on the same net. A descriptive diagram of the cell-net is shown in Figure 4.7. A complete cell-net list of Figure 3.9 is shown in Figure 4.8.

A graph structure is needed to describe the number of connections between cells. This graph is generated from the cell-net list by counting the number of the same nets related to any pair of cells. After establishing this graph, a merge sequence list which arranges the nodes with the greatest number of connections at the front of the list can be set up by scanning through the whole graph. A descriptive diagram of these data structure is shown in Figure 4.9.

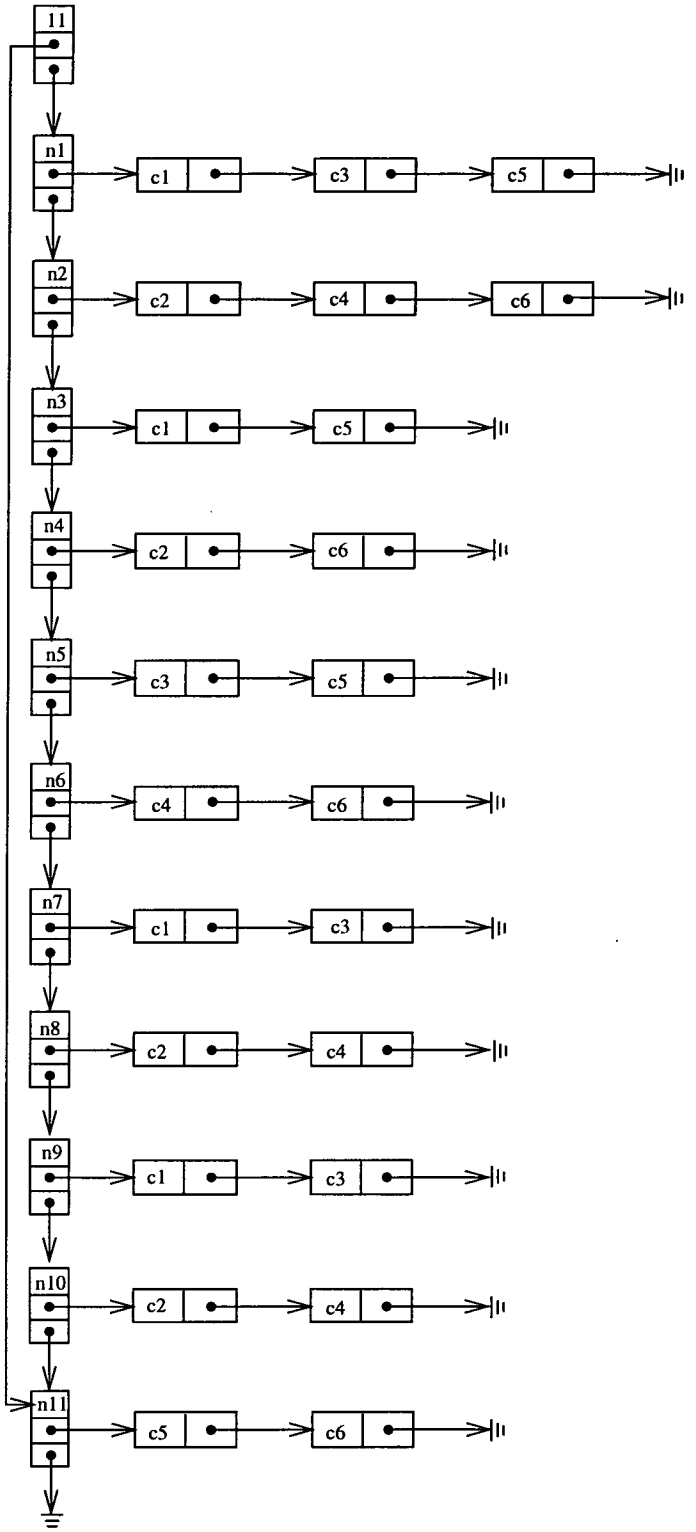


Figure 4.6 A complete net list of Figure 3.9

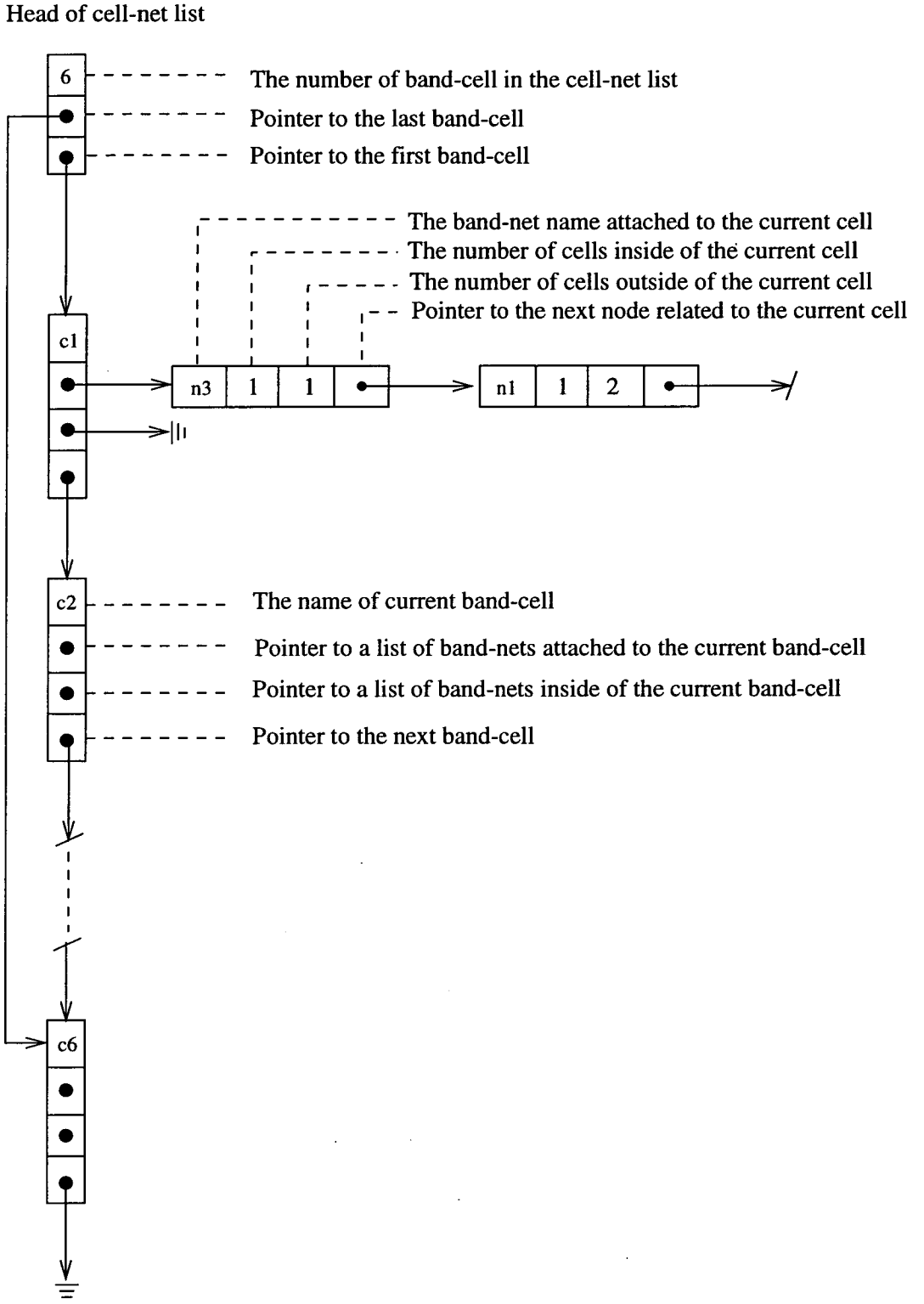


Figure 4.7 The structure of cell-net list

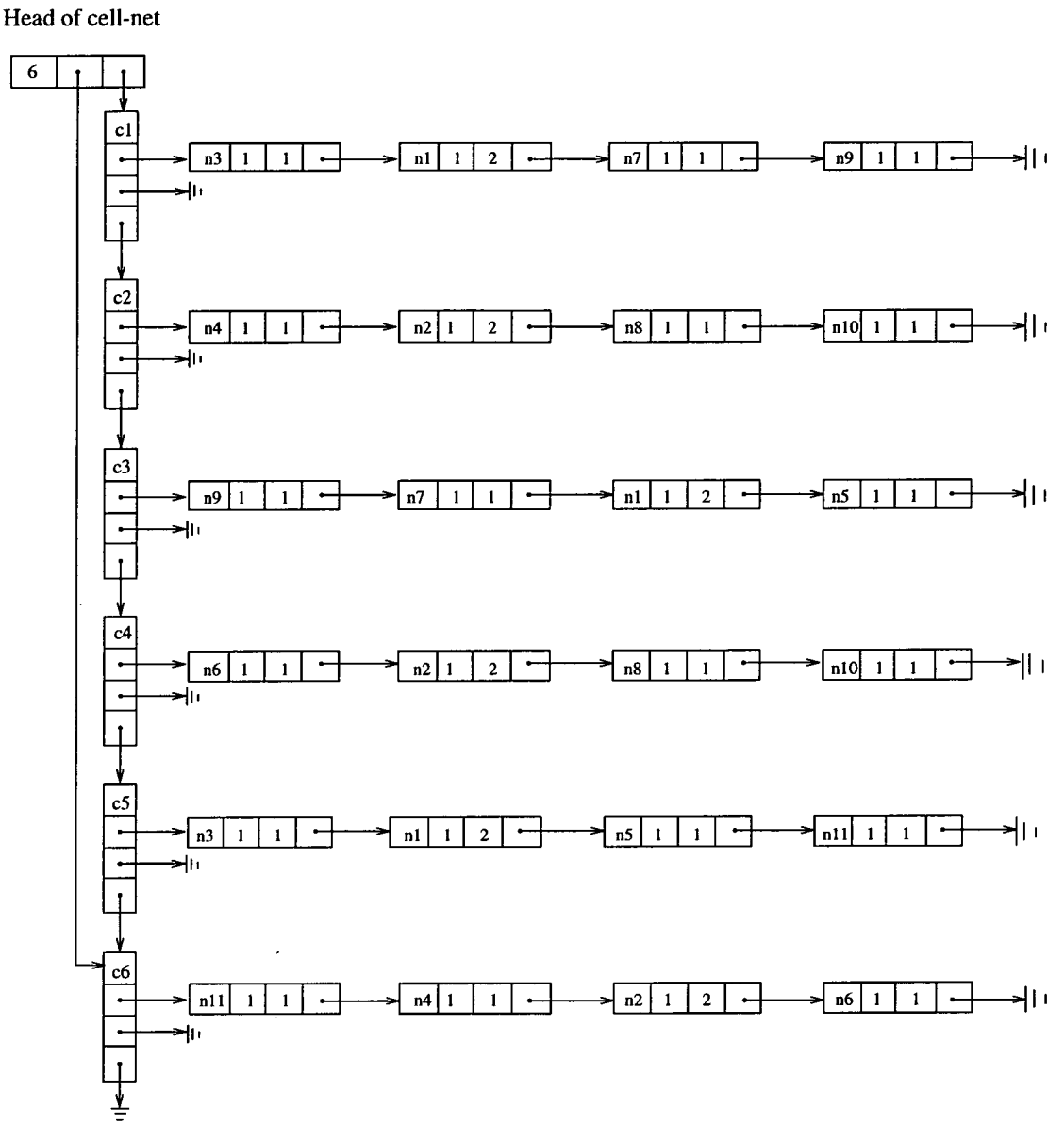


Figure 4.8 A complete cell-net list

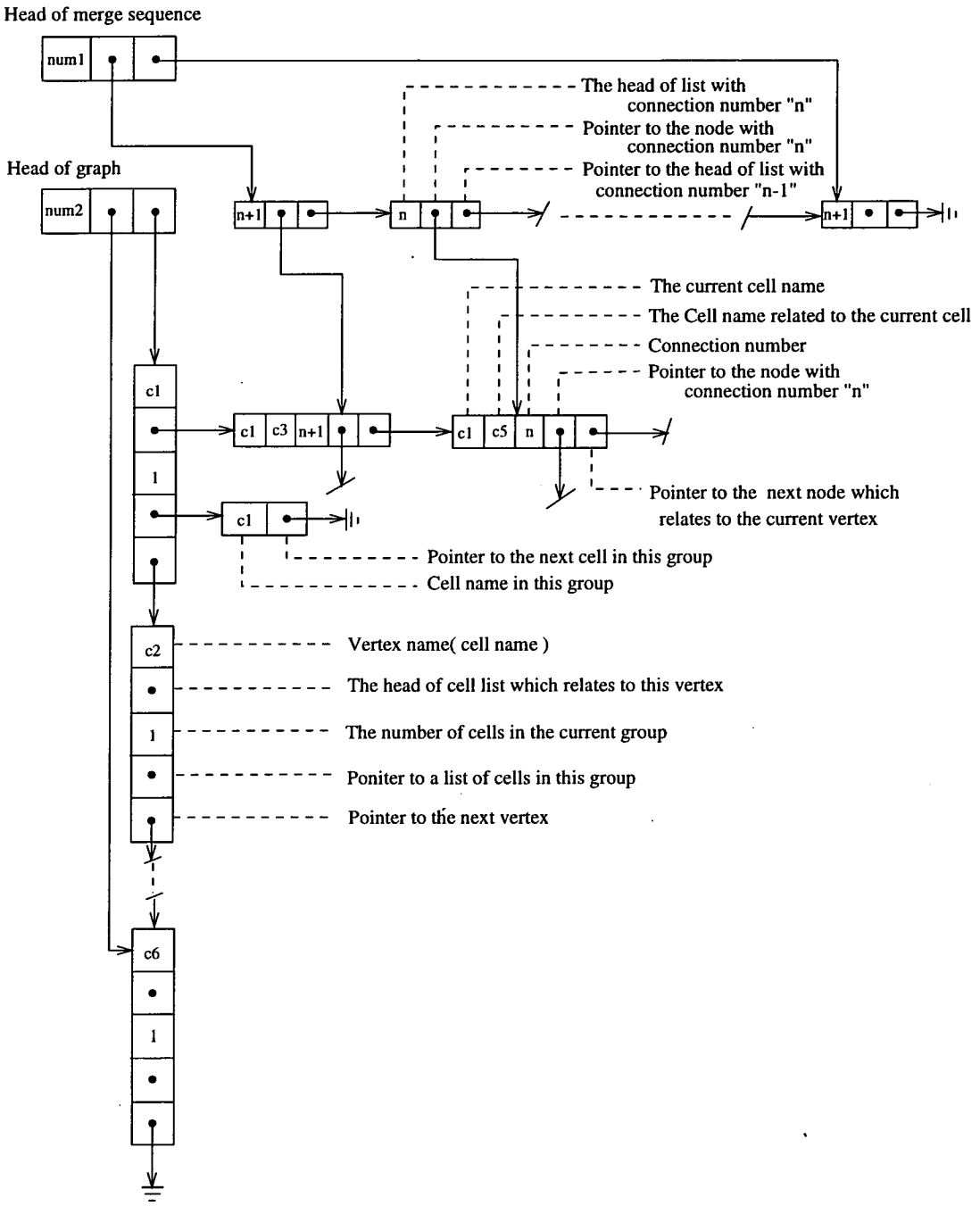


Figure 4.9 The data structure of the graph and merge sequence

4.2. The routines for merge

The objective of the merge operation is to combine a pair of cells as one cell, for instance, if the operation of merging cell c2 to cell c1 as one cell c1 is required, after

the merge the cell c2 is inside the cell c1, the number of connection between c1 and other cells must have been changed due to this merge and the merge sequence may also change, therefore the graph and the linked list for merge sequence must be modified at the same time. Because c2 is buried in c1 after merge, the nets incident to cell c2 become incident to cell c1, so the edges belonging to cell c2 are inserted into the list of edges belonging to cell c1 in the graph. In addition, all other cells related to c2 before merge must be changed to be related to c1.

There are two main procedures in the merge processes. One is called "merge_in_cell-net", another one is called "merge_in_graph". The procedure of merge_in_cell-net is to maintain an updated configuration of the circuit after the candidates for merging are merged. This will provide the precise number of connections between cells. The procedure of merge_in_graph is where the merge operations are realised.

4.2.1. The merge_in_cell-net

The graph in Figure 4.5 is used to demonstrate how the procedures of the merge_in_cell-net process. Assume that c3 will be merged to c1. A part of the cell-net list of Figure 4.8 is shown in Figure 4.10 before merge. To merge c3 to c1, there are three common nets n9, n7 and n1 between two cells, when the merging takes place, n9 and n7 become internal net of c1, and n1 is still external to c1, therefore n9 and n7 are removed from the list of band-net and inserted to the list of the internal band-net, the band-nets(n1 and n5) incident to band-cell c3 and still external to c1 are inserted to the list of band-net of c1. Finally,the band-cell c3 must be removed from the list of the band-cell.

The cell-net list after merge is shown in Figure 4.11. It is necessary to decide which of the common band-nets become internal nets or remain external. This is

implemented by subtracting the number of the field inside of the implicit cell (c3) from the number of the field outside of the explicit cell (c1), if the result of the field outside of the explicit cell (c1) is equal to zero, this means the corresponding band-net becomes internal net, otherwise the band-nets are still external to the band-cell.

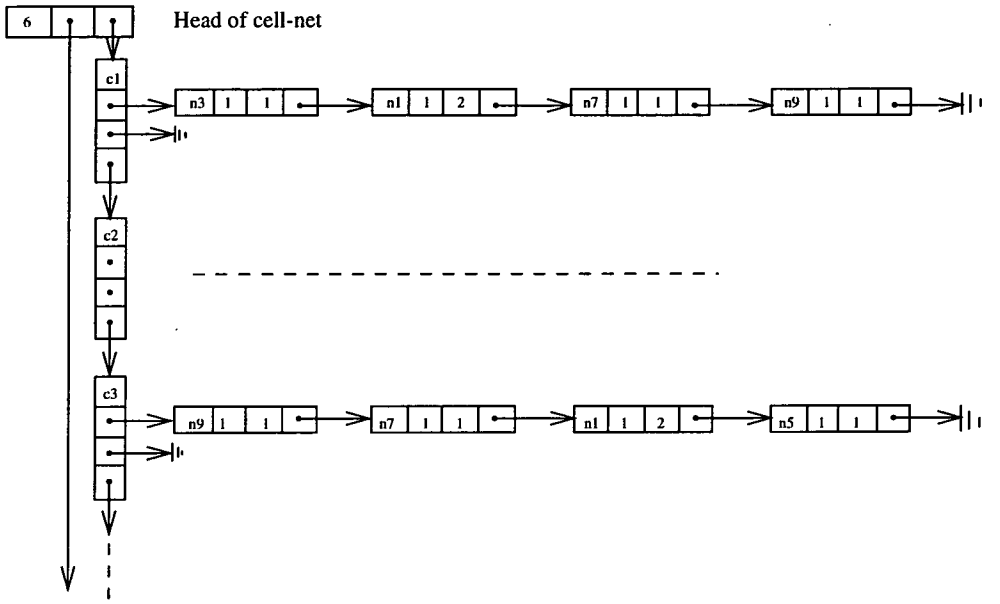


Figure 4.10 A part of the cell-net list of Figure 4.8

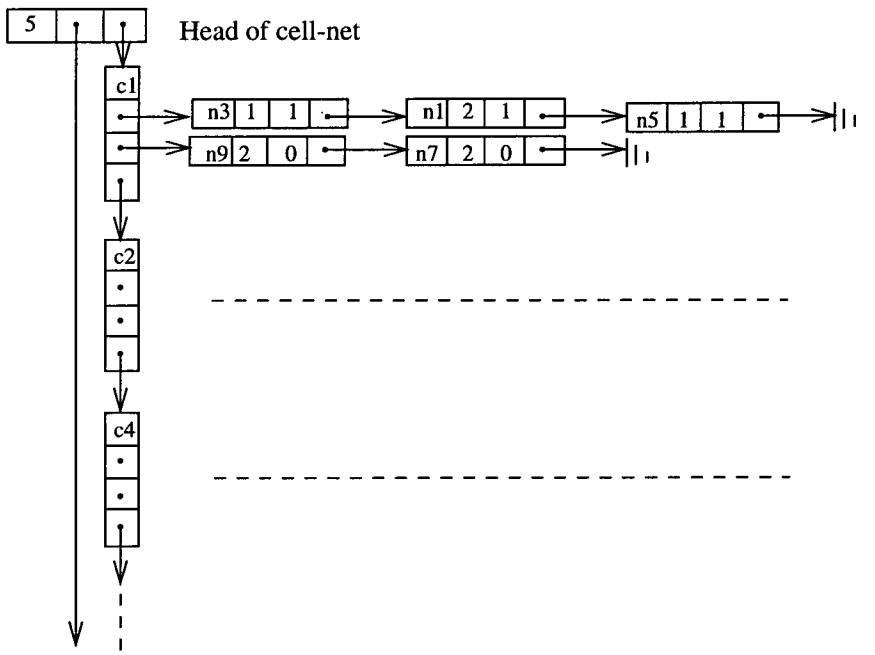


Figure 4.11 The cell-net list after merging c3 to c1

4.2.2. The merge_in_graph

Again the graph in Figure 4.5 is used to illustrate the merging procedures. According to the merging sequence list, the pair of cells c1 and c3 are the candidates to be merged, and c3 will be merged to c1, i.e. c3 is the implicit cell and c1 is the explicit cell. Before the processes in the merge_in_graph can be executed, the merging of c1 and c3 in the cell-net list has to be processed first.

The merge_in_graph procedure is comprised of four sub-procedures which are called "removing", "re-ordering", "attaching" and "replacing". The procedure of removing is to remove the implicit cell in the list of vertices and the cells in the edge list of the implicit cell. The procedure of re-ordering is to re-arrange the order of the merge sequence when the connection number has been changed or the edge node removed. The procedure of attaching is to put the cells which are related to the implicit cell to the explicit cell. The procedure of replacing is to replace all the implicit cell names to explicit cell names.

To merge c3 to c1, first of all, the edge node of c3 belonging to vertex c1 must be removed. Figure 4.12 shows the results of removing edge node of c3. Because the edge node of c3 is in the merge sequence list, the procedure of re-ordering must be applied, which is to re-arrange the priority of the merge sequence. While removing the edge node of c3, the merge sequence for the connection number "3" has to be modified. Figure 4.13 shows the results of the re-ordering due to removing a node.

Second, the cells related to the implicit cell (c3) must be attached to the edge list of the explicit cell (c1). Therefore, c5 which is connected c3 must be attached to the edge list of c1. In the edge list of c1, there has been an edge of c5 existing, a visit to the cell-net list is required to acquire the new number of connections. After merging c3 to c1 in the cell-net list, the number of connections between c1 and c5 is "3". So

the number of connections of edge node c5 should be changed to 3. On the other hand, this node is in the wrong merge sequence listing due to the connection number change. The merge sequence must be modified. The edge node c5 is moved from the list of connection number "2" to the list of connection number "3". The results of this changing are shown in Figure 4.14. If there were other cells which were related to the implicit cell, but not related to the explicit cells, they should be inserted to the edge list of c1 without visiting the cell-net list to obtain the new connection number.

Third, the cell c3 has become a member of group c1 and must be inserted to the group list of c1 and the number of cells inside of cell c1 is 2. Furthermore the implicit cell c3 should be removed from the vertex list and the cells in the list of edge node of c3 also have to be removed. The results of these actions are shown in Figure 4.15.

Finally, in the replacing procedure, every edge node in the graph must be visited to check if there are other cells relating to c3, once the edge nodes of c3 are found, the name of the edge node must be changed to c1, and a visit to cell-net list has to be performed to acquire the new connection number, if this number is different with the old connection number, then the procedure of re-ordering has to be applied to re-arrange the merge sequence. In this example, there is no c3 left in the graph, so the only actions that need to be carried out are visiting every edge node to check if c3 appears in any edge node and to change c3 to c1 when required.

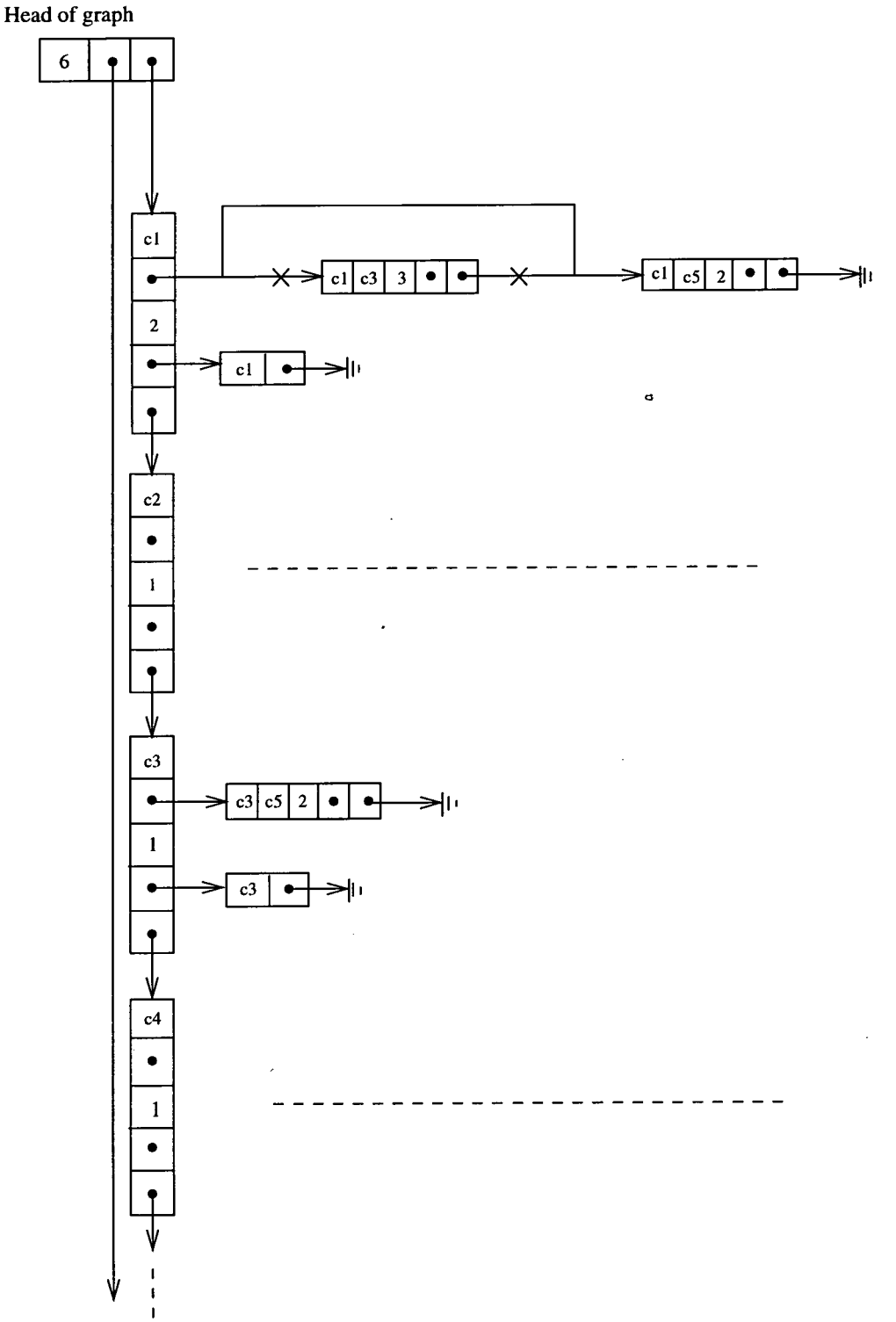


Figure 4.12 The result of removing edge node of c3

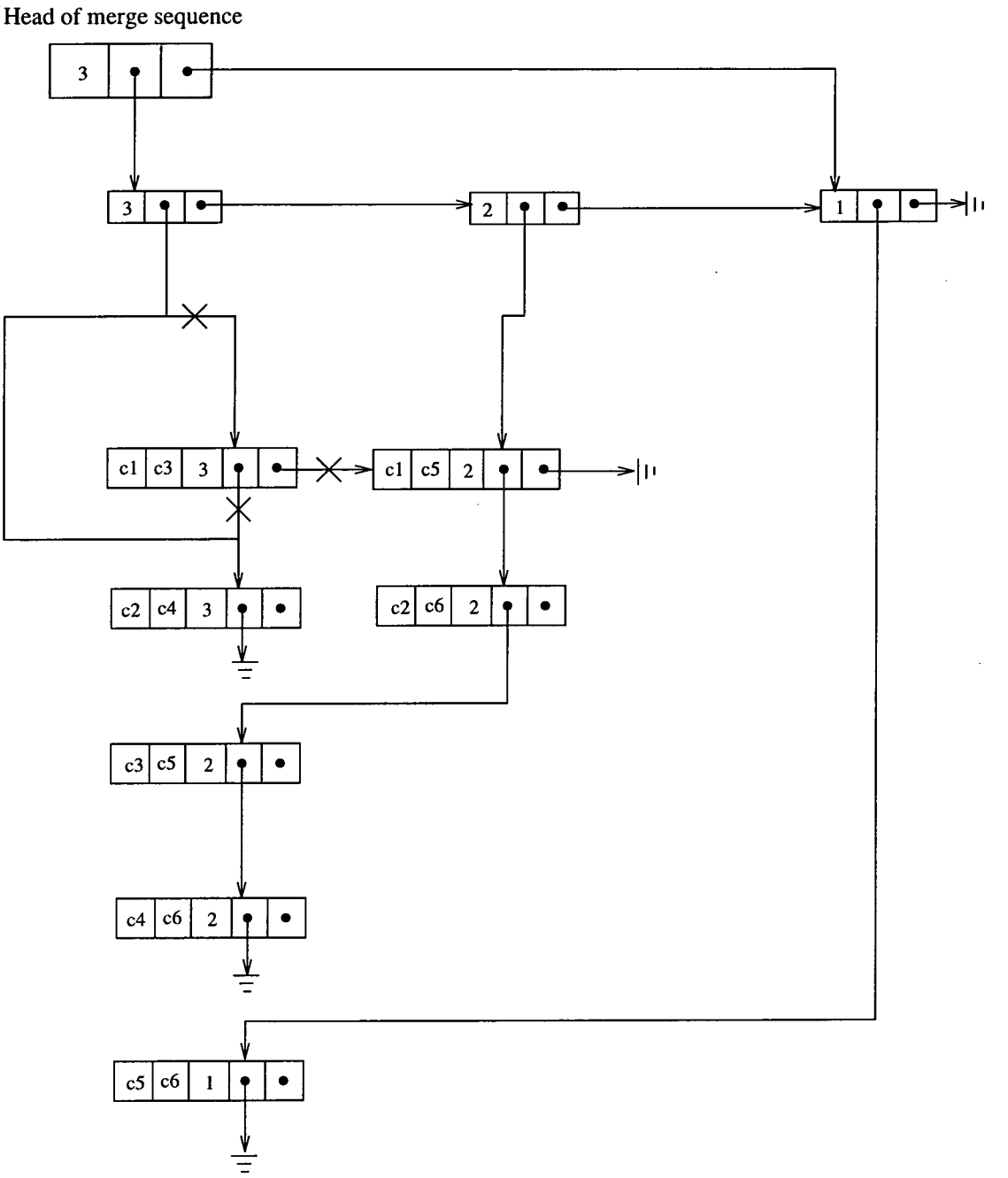


Figure 4.13 The result of re-ordering due to removing a node

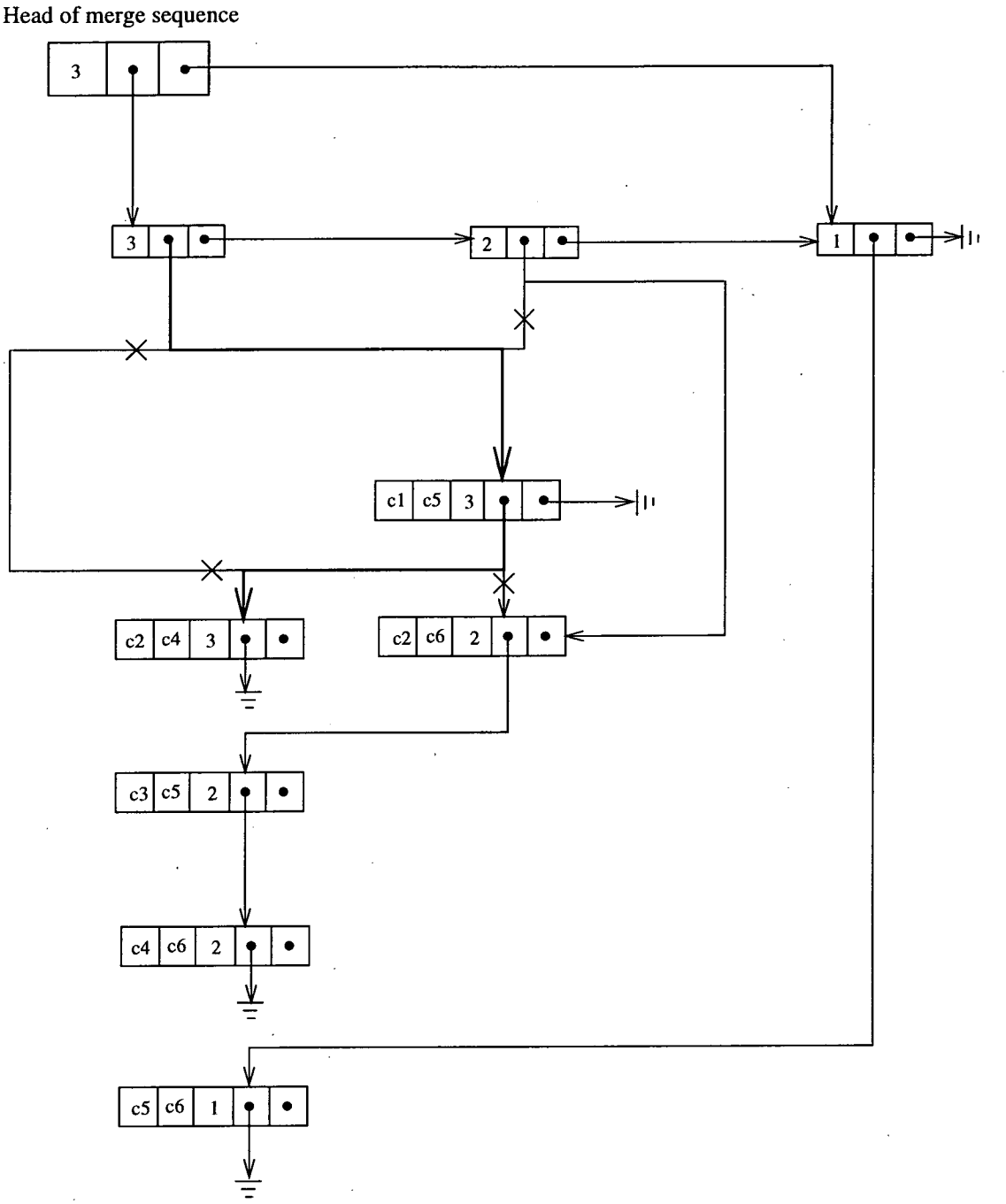


Figure 4.14 The result of re-ordering due to connection number change

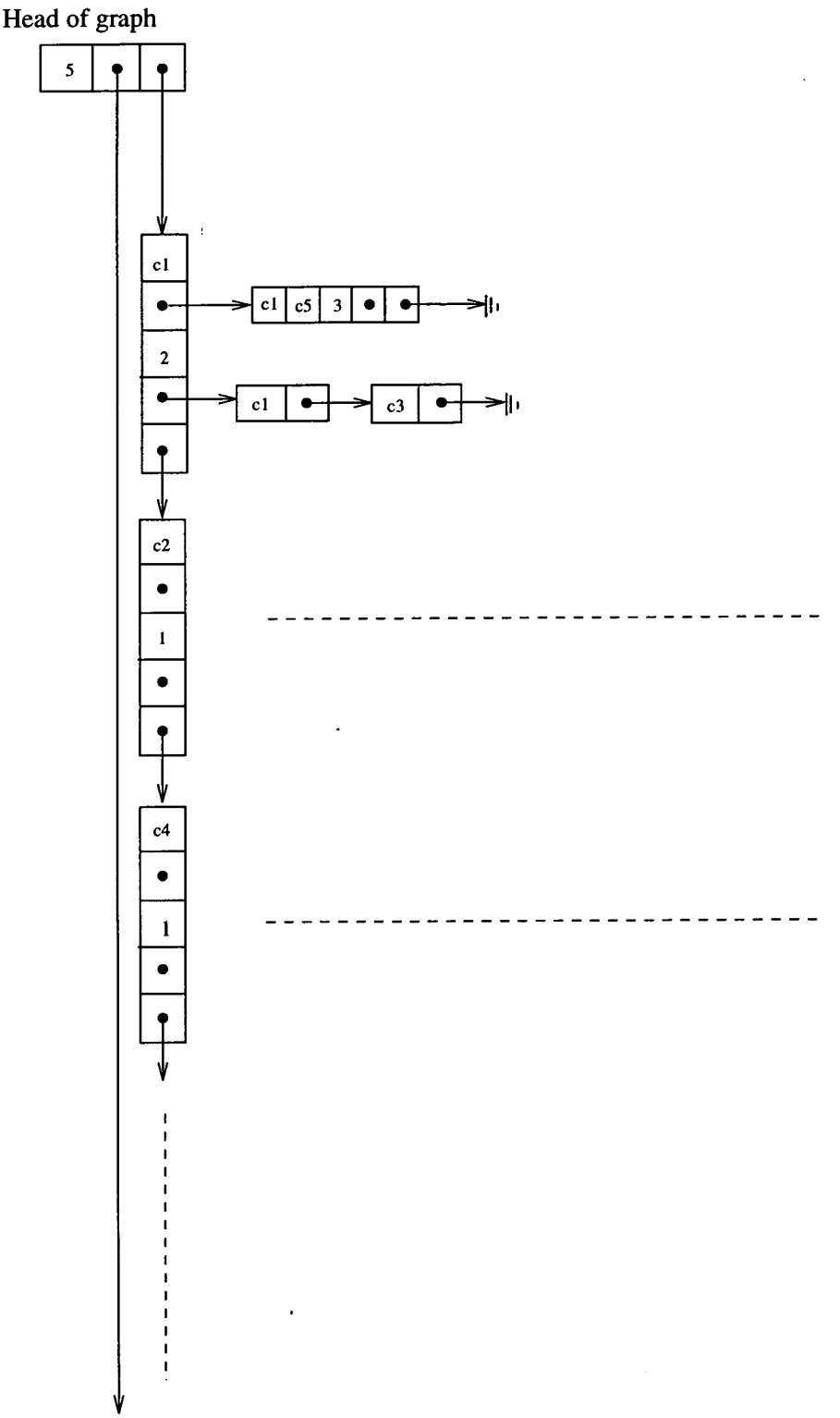


Figure 4.15 The result of merging c3 to c1

4.3. Summary

In this chapter the detailed data structure was developed which enabled the merge algorithm to operate effectively in creating "cell list" and "net list" from the source circuit netlist file. These two lists were used to create the cell-net list from which the graph of the circuit was derived.

The merge algorithm was operating on the cell-net list and graph. The cell-net list was used to obtain the new number of connections between cells after merging. The graph was used to form a new configuration of the circuit after each merge.

The next chapter exercises the merge algorithm on two test circuits, analyses the performance of the merge algorithm.

CHAPTER 5

The Performance of Merge Algorithm

This chapter uses two artificial circuits generated from the Random Circuits Generator (RCG) to test out the performance of the merge algorithm. Some representative results are presented.

5.1. Experimental system

5.1.1. Random circuits

Random circuits are created by specifying the number of cells, the number of input pins and the number of output pins to the RCG program (described in Appendix B). The output of RCG program is a text form describing the circuit in InCA net list format. The circuits produced by the RCG program are not realistic and do not possess meaningful functionality. They just provide information about how the components in circuits are interconnected. These circuits will be used to compare the performance of the different partitioning strategies.

5.1.2. Structural circuits

In the real world, circuits are usually constructed in a hierarchical manner which makes circuits more structural. To make random circuits more realistic and obtain more information about the connections inside circuits, a semi-automatic method to generate circuits was used. First, the RCG program is used to automatically create some smaller size of random subcircuits and then these subcircuits are manually

interconnected to form a complete circuit. Constructing a circuit in this way allows circuits with known optimal cutsize to be created and used to assess the performance of the various partitioning algorithms.

5.1.3. Test circuits

To demonstrate the performance of the merge algorithm two test circuits were obtained from the random circuit generator. Test Circuit1 (TC1) consisted of 100 cells connected by 132 nets and 20 input/output pins. Test Circuit 2 (TC2) was constructed by manually interconnecting six different circuits derived from the random circuit generator. An example TC2 circuit is shown in Figure 5.1. The main features of six units used in Figure 5.1 are shown in Table 5.1. The netlist of the six units were sequentially combined to form the TC2 netlist. In fact 720 different circuits were obtained to cover all possible permutations of the six separate netlists.

Unit	Number of Cells	I/O	Number of Nets
U1	10	5	13
U2	20	5	25
U3	30	5	43
U4	10	9	19
U5	20	6	28
U6	30	7	40

Table 5.1 The features of Test Circuits 2

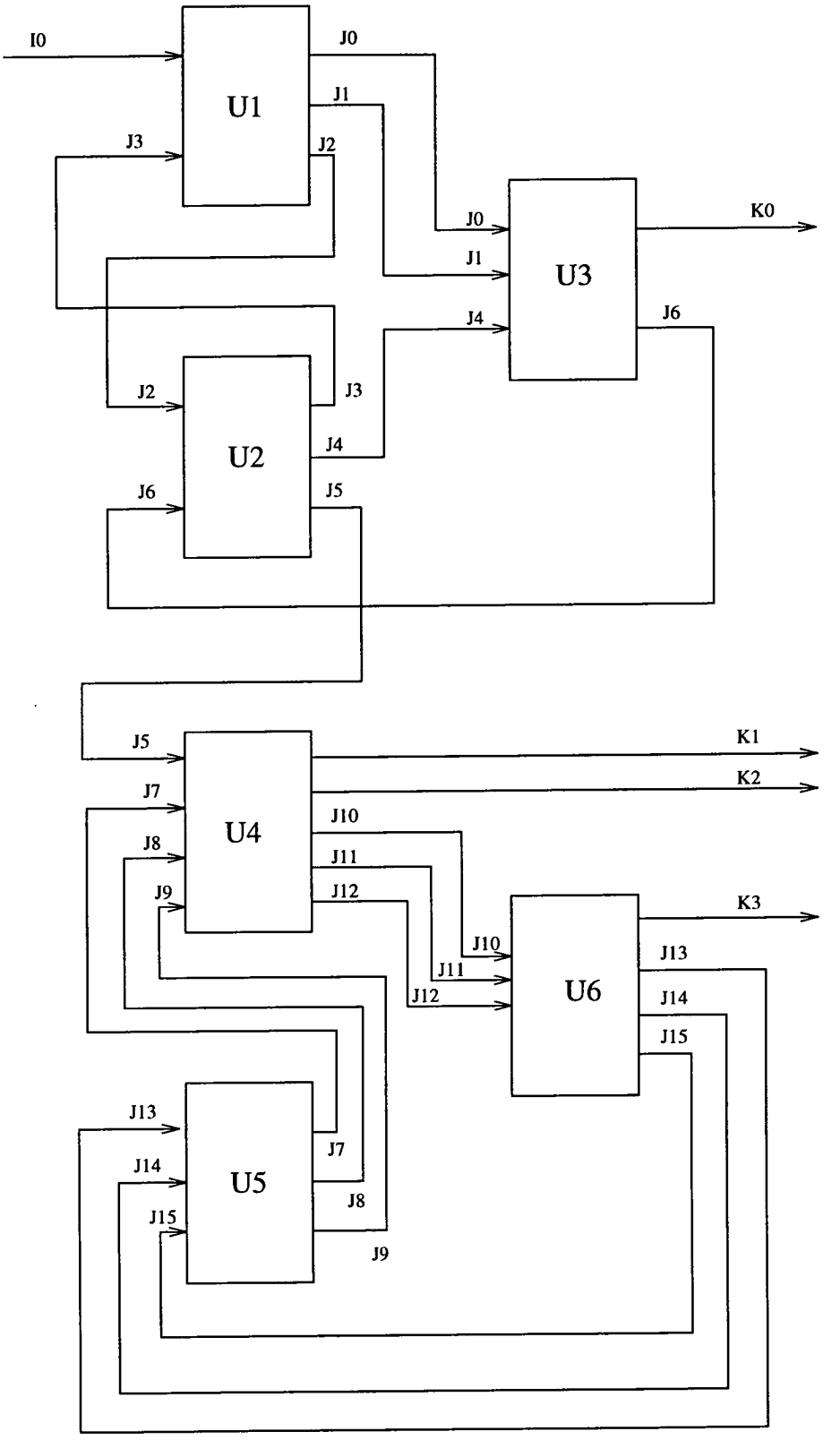


Figure 5.1 The block diagram of Test Circuit 2 (TC2)

5.2. Result

A balance requirement between groups is important. If this condition were neglected during the merge operation, all the cells might be merged to one group. To judge if a result is balanced a balance factor, denoted by B, is defined by:

$$B = \frac{\sum_{i=0}^{ng-1} \frac{|C_i - ave|}{ave}}{2(ng-1)} \tag{5.1}$$

where $ave = \frac{C}{ng}$, C_i is the number of cells in the group i, C is the total number of cells and ng is the required number of groups.

The range of B will fall between 0 and 1. If $B = 0$, the partition is perfectly balanced, while $B = 1$ indicates that all cells are merged to one group and the partition has totally lost balance.

Test circuit 1 was partitioned into two groups. Figures 5.2 to 5.6 show the results of varying the initial constraint S_0 and the number of stages L. The balance Factor B is also shown in each case to illustrate the way it can be used to monitor the balance of the results.

A suitable L value for TC1 is obtained from equation (3.1) and (3.2) to be 5. L values from 1 to 5 were used with S_0 values increasing from 2 by one for each partition until the value which makes all the cells in the circuit merge to one group. The cutsize and balance of the partition vary with initial constraint S_0 for fixed L. From a general view of these figures (i.e. Figure 5.2 to Figure 5.6), the cutsize decreases as the initial constraint increases and the balance factor increases as the initial constraint increases. It is easy to see that a bigger initial constraint will generate a smaller cutsize for partitioning. However, small cutsize leads to a poor

final balance. From these figures it can also be seen that the cutsize varies little when the balance factor falls in a certain range, for example between 0 and 0.5. This means that a substantially constant cutsize result is obtained when the result of partitioning is required to be in a reasonable balance.

A version of test circuit 2(TC2) having 720 possible implementations was designed to feature an optimal bi-partition cutsize equal to 1 in each case. The main purpose of this design was to test the ability of the merge algorithm to find this optimal solution. It was found that the merge algorithm found the optimal partition with 100% success. Performing the same partition with the Kernighan-Lin based approach 478 circuits were successfully partitioned with cutsize equal to 1. The results of the partition of the other 242 circuits demonstrated a much bigger cutsize. A sample of the intermediate result of partitioning a TC2 is shown in Figure 5.7.

5.3. Mapping a design to a target structure

The merge algorithm (abbreviated as MA) can be applied to map a design to the following hardware architectures:

- fixed hardware target structure
- flexible hardware target structure

The strategies and method used for partitioning a design for either of the above two possible hardware configurations may differ and depends on design architecture as well as the application purposes.

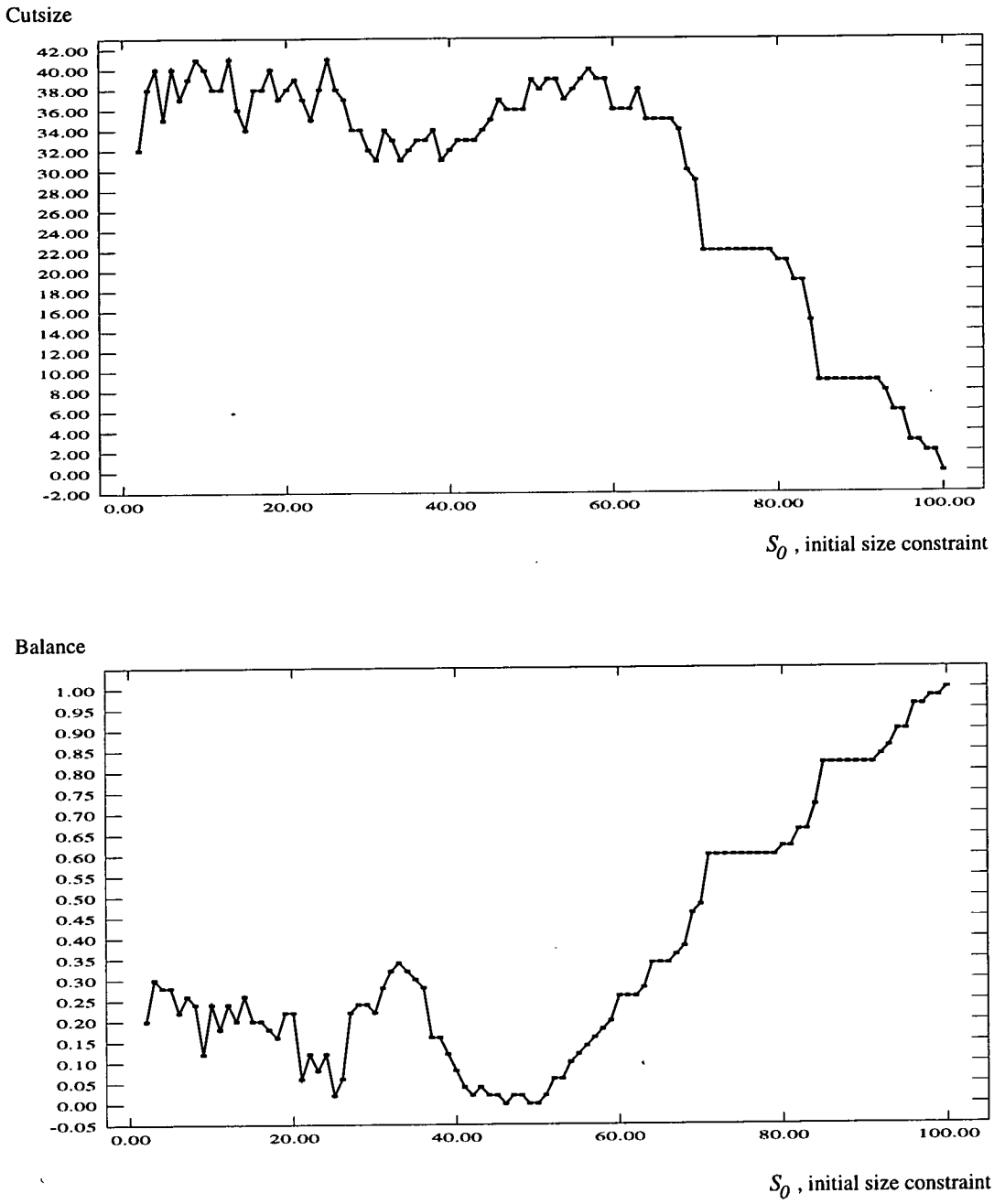


Figure 5.2 Cutsize and Balance versus initial size constraint, S_0, for $L = 1$

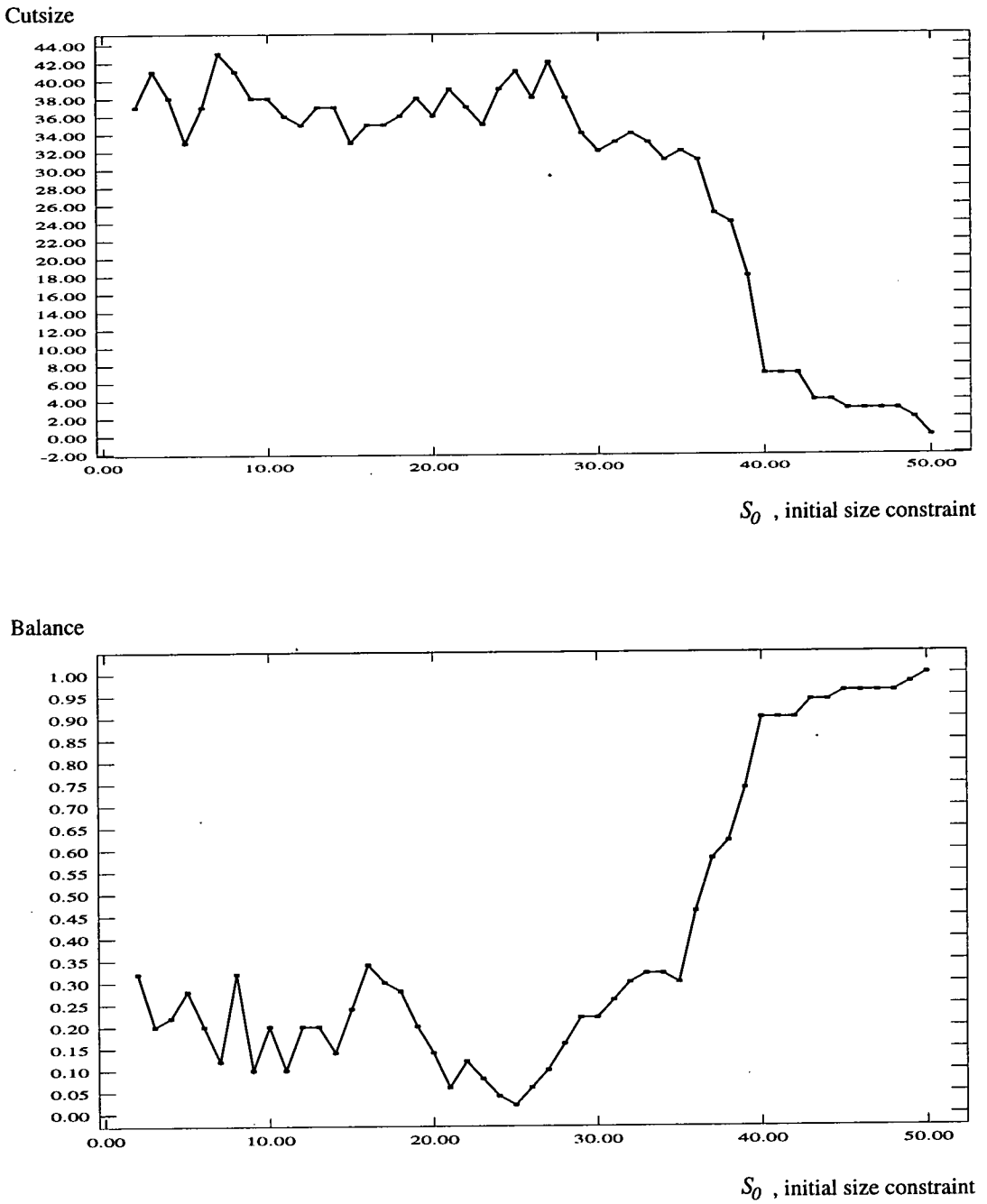
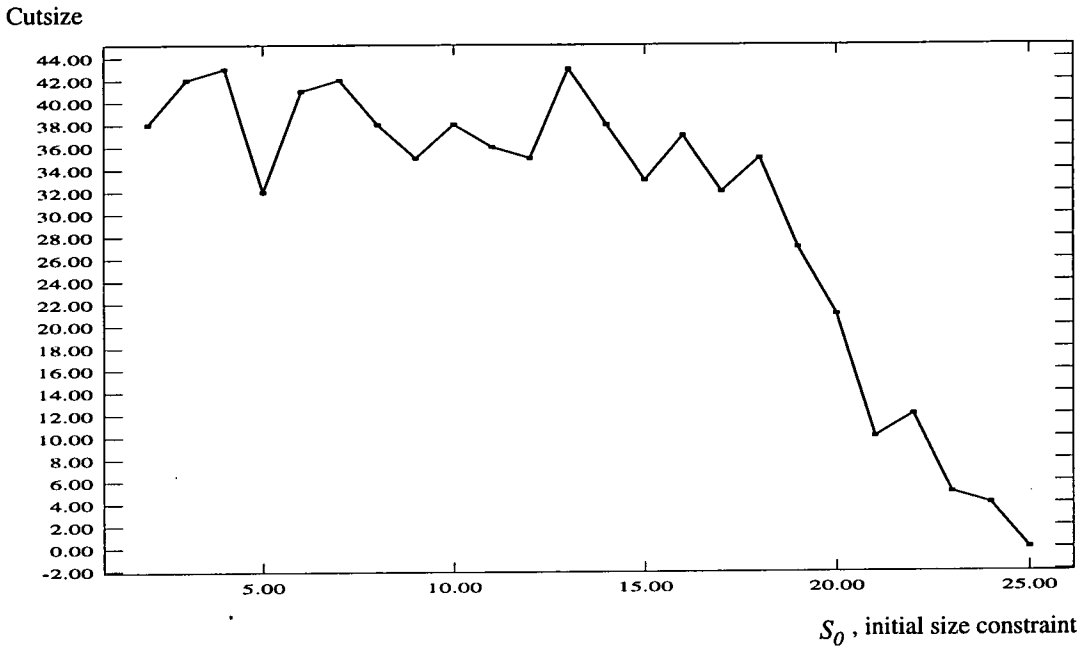


Figure 5.3 Cutsize and Balance versus initial size constraint, S_0, for $L = 2$



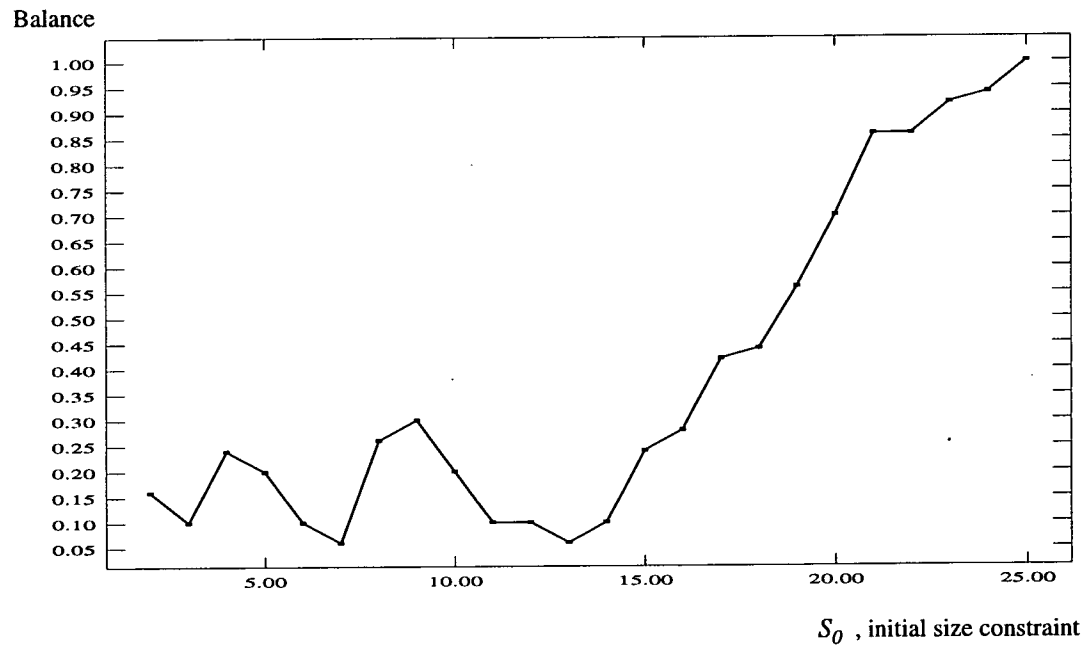
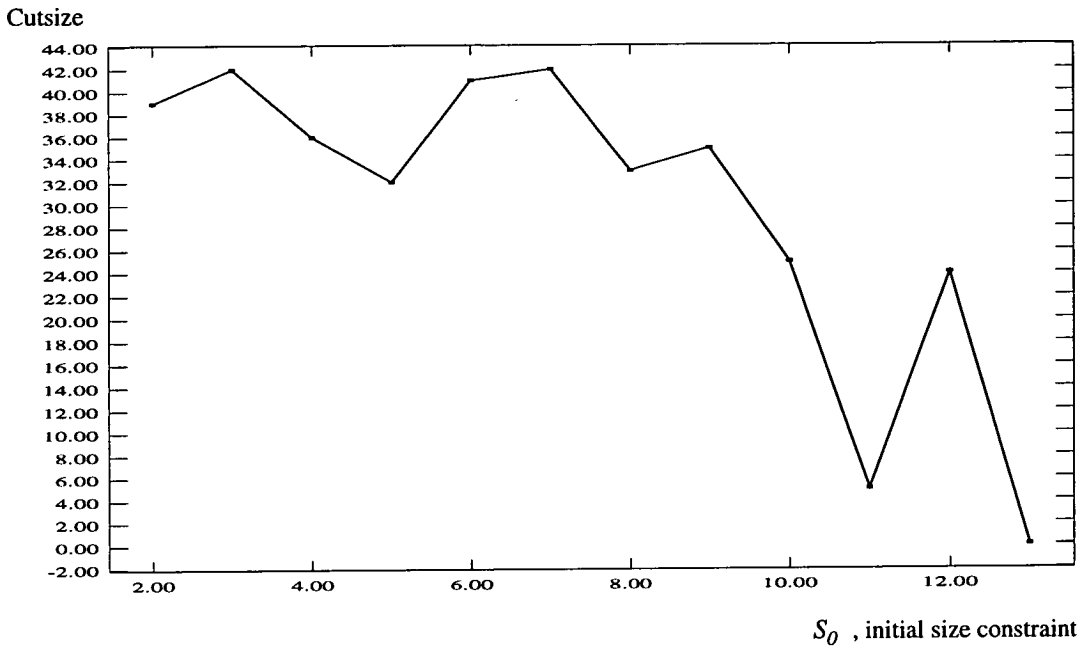


Figure 5.4 Cutsize and Balance versus initial size constraint, S_0, for $L = 3$



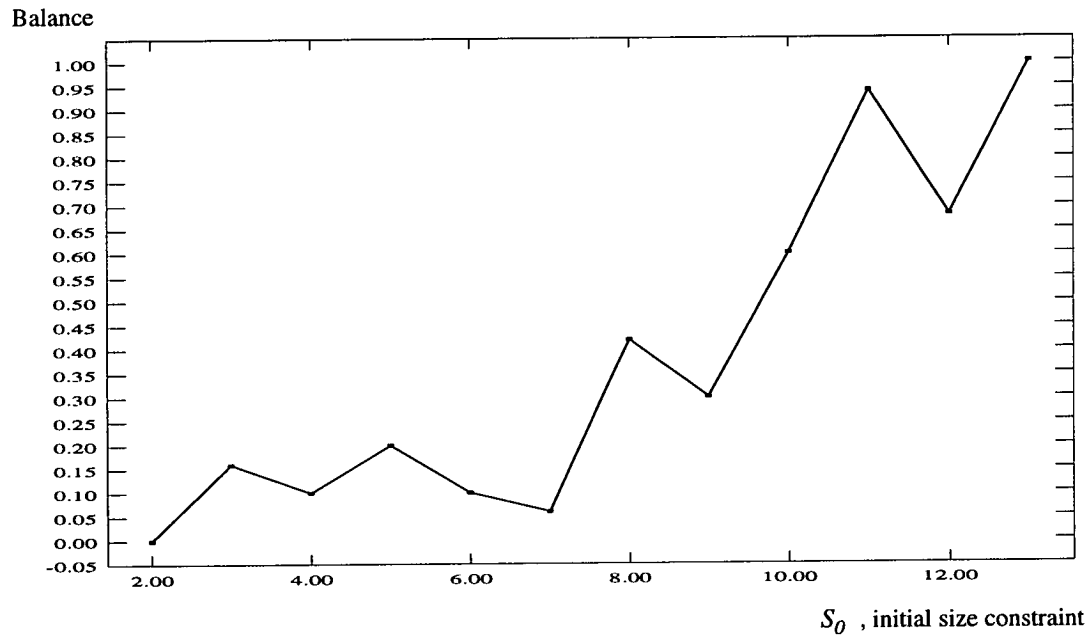
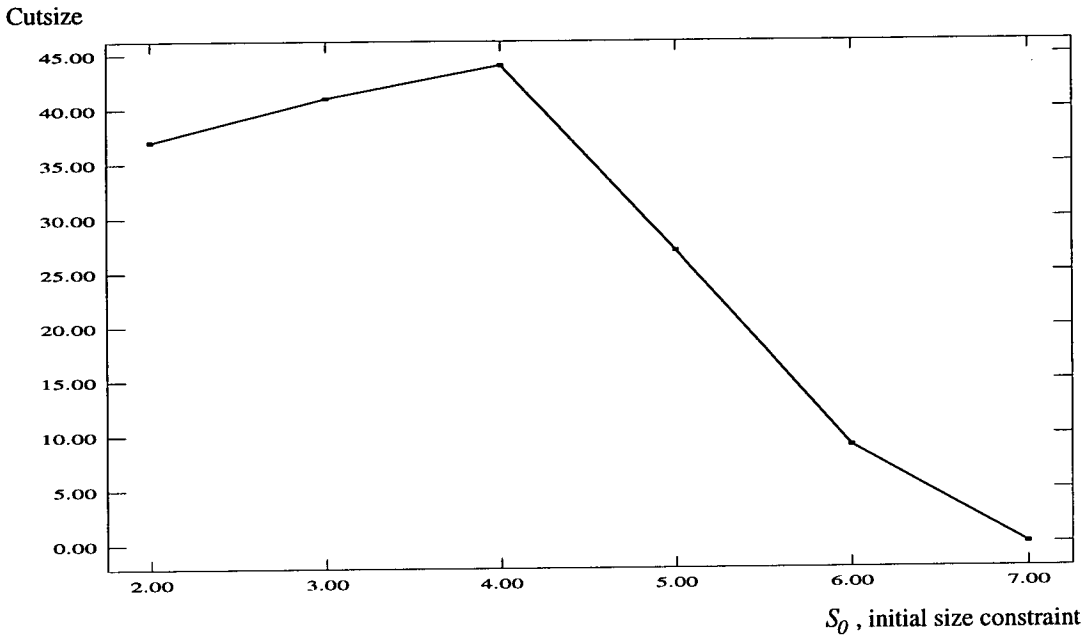


Figure 5.5 Cutsize and Balance versus initial size constraint, S_0, for $L = 4$



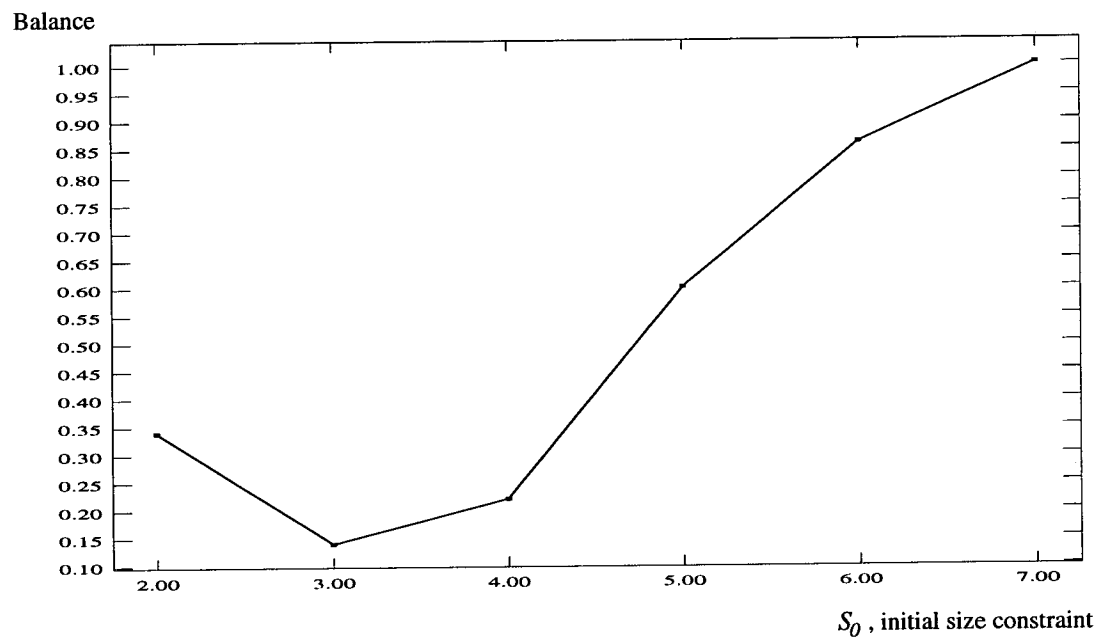


Figure 5.6 Cutsize and Balance versus initial size constraint, S_0, for $L = 5$

```
--I-- Flattening the system...
--I-- Creating cell-net list...
--I-- Creating graph...
--I-- Scanning and setting up list of descending weight...

The shrinked graph after L merging stages is defined by;

0     0 60 4   0 6 14
6     6 60 2   6 30 2
10   10 80 3   10 90 1   10 92 3
30
60   60 80 1
80   80 90 4
90   90 92 7
92

where the first column shows the group (or cell) identifier and each row
shows the connections with other groups e.g. 0 60 4 indicates that group
0 is connected to group 60 via 4 links.

The following list shows at its first column the number of cells in each
group.  Each row starts with the group identifier followed by the other
members of the group.

18   0 4 8 3 5 1 2 9 33 40 56 59 47 36 39 49 52 53
20   6 7 32 46 34 48 54 37 55 31 35 51 50 42 43 38 57 58 41
       45
20   10 21 20 14 24 13 15 25 11 22 12 19 17 18 27 16 26 23 29
       28
2    30 44
20   60 67 69 74 79 65 70 78 68 71 63 73 64 76 77 75 66 61 62
       72
11   80 83 85 89 86 81 84 82 87 88 114
20   90 91 96 116 105 95 103 104 113 115 93 98 108 109 94 99 101 111 102
       112
9    92 106 107 117 119 97 110 100 118

The following lists show the final partition of this special circuit with
groups 0 and 10 linked by 1 connection with each group containing 60 cells
as listed.

0     0 10 1
10

60   0 4 8 3 5 1 2 9 33 40 56 59 47 36 39 49 52 53 6
       7 32 46 34 48 54 37 55 31 35 51 50 42 43 38 57 58 41 45 60
       67 69 74 79 65 70 78 68 71 63 73 64 76 77 75 66 61 62 72 30
       44
60   10 21 20 14 24 13 15 25 11 22 12 19 17 18 27 16 26 23 29
       28 80 83 85 89 86 81 84 82 87 88 114 90 91 96 116 105 95 103 104
       113 115 93 98 108 109 94 99 101 111 102 112 92 106 107 117 119 97
       110 100 118
```

Figure 5.7 Listings demonstrating the bi-partition of a TC2 circuit.

5.3.1. Mapping a design to a fixed target structure

A fixed target structure consists of several nondisjoint FPGAs, which are systematically connected. Typically these types of structures are used in situations where many different designs are to be compiled onto one general purpose hardware structure. The VA and Anyboard mentioned in chapter 1 are in this category of hardware structure.

To fit the design into the fixed hardware target structure, the MA must be constrained by:

(1) Number of gates in a group not greater than G.

(2) Number of pins associated with a group not greater than P.

(3) Total interconnects between groups not greater than I.

where G is the number of gates that an FPGA can accommodate without the difficulty of placement and routing, P is the number of interface pins, and I is the number of interconnects between FPGAs in the fixed hardware target structure.

The partition which satisfies the gate counts and pin counts does guarantee enough internal resources to FPGAs, but does not guarantee there are enough external resource to FPGAs in the fixed hardware target structure. So that applying constraint on the interconnects between FPGAs is necessary.

The twelve FPGA fixed hardware structure shown in Figure 1.2 is used as an example to illustrate how to apply a constraint on the resources of interconnects between FPGAs. The twelve FPGA is redrawn in Figure 5.8 with the dots representing FPGAs which are arranged in a ring and eight of which are for functionality and connections , four of which are for connections only. Assume the

connections between groups only use pin to pin direct connections (28 wires in this case) or pass through the connection only FPGAs. To ensure a successful mapping in this particular hardware configuration , the following constraints must be observed.

(1) $\sum_{j=1, j\neq i}^{j=8} cz(i, j) \leq 92$, where $cz(i, j)$ is the cutsize of group i and group j, and 92 is composed of 28 pin to pin direct connection between adjacent FPGAs and 4 interconnect FPGAs each with 16 connections to each of 8 functional FPGAs.

(2) one pair of groups must be adjacent on the physical target structure.

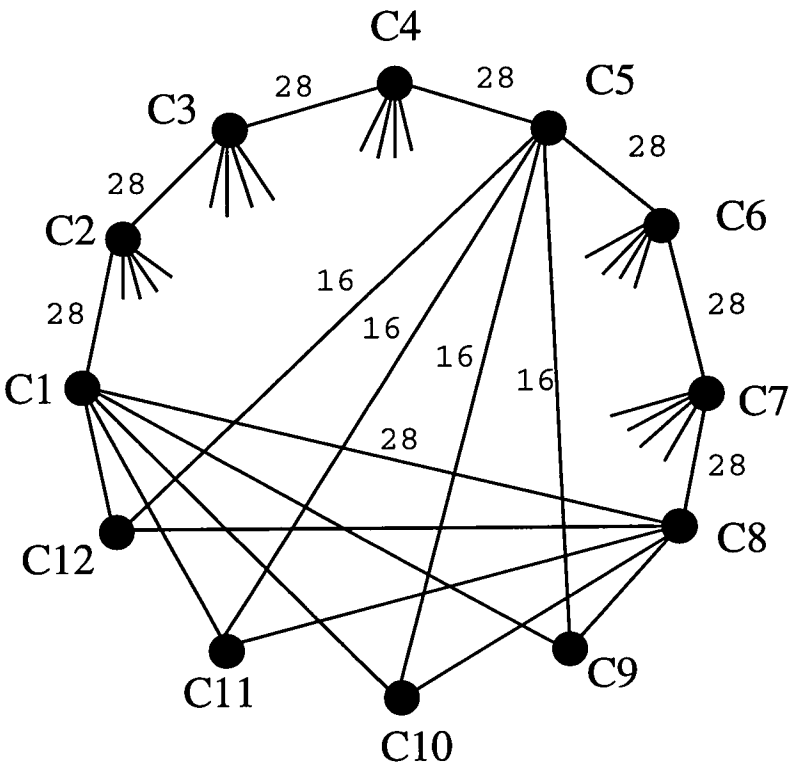


Figure 5.8 The ring structure

5.3.2. Mapping a design to a flexible hardware target structure

A large design may be partitioned into a range of FPGAs while minimising connectivity between FPGAs and achieving high gate utilisation and system performance. Once an optimum partition for a given design is found then the required hardware must be constructed using the wiring specification generated by partitioning software. Such hardware architecture is referred to as "flexible hardware target structure".

The main constraints for implementing the flexible hardware structure are the gate counts and pin counts in the variety of FPGAs. How to apply these constraints to the partitioning software depends on the user's situation. For example, if users have only a few types of FPGAs or limited number of FPGAs, then the above constraints must be applied to the partitioner which consequently will determine if the design is feasible or not in such situations. A better approach to achieving an optimum partition is finding a suitable partition from a solution space to suit user's requirements.

The MA can offer a range of solution space by applying different parameters to the merging processes. The following parameters can be changed to obtain the possible partitions for a design.

- the number of groups for the final partition
- the initial size constraints S_0
- the number of stages, L
- the size of subcircuit

By changing the individual parameter or combinational parameters, the MA will generate different results from which user may analyse them to find a feasible partition, if no promising results are found, go back to change the parameters again

that may produce a desirable partition. This process is repeated until users find a partition that suits their requirements.

5.4. Summary

Test circuit 1 (TC1), which is a random circuit with 100 cells, was thoroughly exercised by changing the initial size constraint S_0 and stage number L. This produced a large range of partitions. For practical applications, a designer may select a partition which suits his requirements.

Test circuit 2 (TC2) is a circuit with known optimal partition and was used to test the ability of the merge algorithm to locate this optimal partition. The merge algorithm demonstrated an excellent potential to find this optimal partition.

The merge algorithm has a flexible capability which facilitates the use of the software to suit various applications by meanings of modifying some minor procedures or changing some parameters. The important problems are identifying the requirements of the applications, then adjusting the merge algorithm to proceed under the specified constraints.

Generation of a range of possible designs in response to a product specification is an essential feature of successful product development [18]. A pseudo-parallel version of the merge algorithm which has better speed performance, is presented in the next chapter. This gives a brief summary of partitioning methods that will facilitate rapid exploration of possible partition solutions.

CHAPTER 6

The Pseudo Parallel Process for Merge Algorithm

This chapter presents a pseudo-parallel (or serial-parallel) method for implementing the merge algorithm to enhance the performance of the merge algorithm when the designs become large.

Two basic parallel processing methods are introduced which are pipeline processing and data parallelism. The latter method is used to implement the merge algorithm. The circuit is divided into several subcircuits which are sequentially passed to the merge algorithm, some synchronisation actions have to be carried out, then the size-reduced subcircuits are combined as a full circuit. These procedures are repeated until the desired partition is reached. The associated C programming code for the pseudo-parallel algorithm is listed in Appendix E.

Five test circuits were generated to exercise fully the pseudo-parallel merge algorithm. Some promising outcomes are also presented.

6.1. Parallel processing

A conventional computer uses one processor which executes a set of instructions in order to produce a result. At any one time there is only one operation being carried out by the processor.

Parallel processing is concerned with producing the same results using multiple processors. There are many approaches to parallel processing, two basic ways of

which are as follows:

(1) Divide the problem into a sequence of tasks, with each task operating on the results of the previous tasks. This approach is known as pipeline processing.

(2) Have many processors performing the same task simultaneously on different data sets. This approach is known as data parallelism.

In the pipeline method, a problem is separated into a cascade of tasks, each of which is executed by an individual processor. Data is passed through each processor performing a different operation on each of the data elements. Since the program is distributed over the processors in the pipeline and the data moves from one processor to the next, no processor can proceed until the previous processor in the pipeline has completed its task and passed the data to it. Figure 6.1 shows the data flow and task distribution for a pipeline of processors.

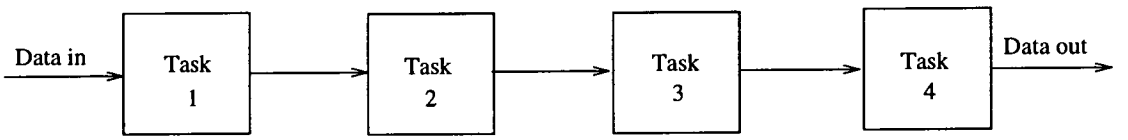


Figure 6.1 Task and data distribution for a pipeline of processors

Data parallelism generally distributes all of the data to be processed equally over all of the processors in the computer. Each processor contains the same program operating on a subset of the data. This is in contrast to a pipeline approach, in which the program is distributed rather than the data. Figure 6.2 illustrates the distribution of data and program task over processors.

Creating parallelism by distributing data is a popular approach because it largely avoids the difficulty of finding a way to decompose a problem into parallel pieces. Not only is program decomposition largely irrelevant, but distributing data by dividing it equally among the processors also provides automatic load balance.

Although the program decomposition becomes easier in data parallelism, the synchronisation between processes can become more difficult and complicated. Because each processor contains only a portion of the entire data set, any processor which requires other, nonlocal data, must obtain it by communicating with other processors. The difficulty of programming a processor network to communicate correctly and efficiently can more than compensate for the ease of program decomposition. Nevertheless, data parallelism is generally more flexible and easier to implement effectively than is pipeline parallelism [87-91].

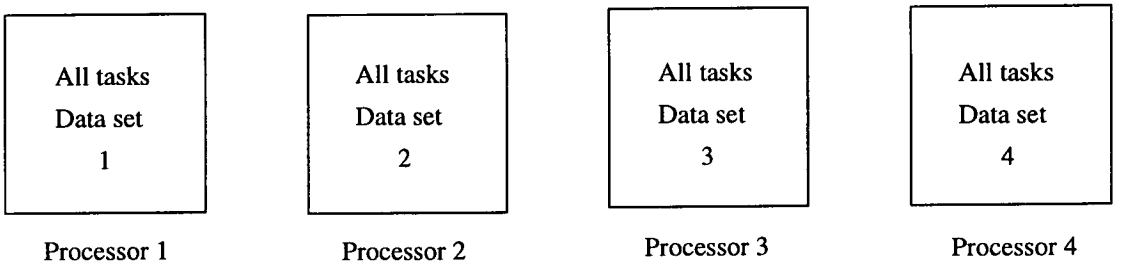


Figure 6.2 Task and data distribution for data parallelism

6.2. A pseudo-parallel processing method for the merge algorithm

The merge algorithm (abbreviated as MA) can be implemented as a data parallel algorithm. Such an implementation will lead to an increase in the speed of circuit partitioning.

The data, i.e. the graph, will be divided into several portions which will be distributed to many processors which execute the MA on the subset of the graph. Some synchronisation programs must be written to update the local data. Three functions are needed to facilitate the implementation of the data parallel approach. They are "divider", "coordinator" and "constructor". The divider is used to divide the data into a number of data subsets. The coordinator is used to synchronise the data distributed among processors and the constructor is used to re-construct a new full data set.

The parallelism of the MA was investigated by sequentially using a single processor to implement the parallel process. It is found that this pseudo-parallel processing method also reduces the time required for circuit partitioning.

6.2.1. The data parallel method

The first task to be carried out in data parallel system is to actually divide the data. If the total number of data elements is a multiple of the number of processors, the data can be divided evenly among the processors. But in most cases the number of data elements is not a multiple of the number of processors, the data can not be evenly divided. The policy of dividing data adopted in this work is to divide the data evenly to the preceding subsets and any data leftover is allocated to the last subset. The number of the subsets is decided by the number of processors available and the size of the target system.

The graph which represents the target system is divided into a number of sub-graphs which are passed to the MA which merges the pairs of cells in the same sub-graph. The candidates which are not in the same sub-graph are neglected. The merging is complete once all possible cells have been merged. This produces a number of reduced sub-graphs.

During the process of merging, some cells become implicit cells in some sub-graphs and the cell names of of the implicit cells have been changed to the cell names of the explicit cells. Because these events happen at the local sub-graph level, other sub-graphs are not informed that the names of those implicit cells having been changed if they exist in them. As a result, a coordinator is required to change the names of the implicit cells to the corresponding cell names of the explicit cells.

After these reduced sub-graphs have been modified for their proper cell names, they are fed to the constructor to obtain a new full, but reduced, graph.

The goal of merging is to acquire a desired partition, so the new full reduced graph must be investigated to check if it meets the requirements of the desired partition. If not, the new graph is passed to the divider again and the next iteration starts. Figure 6.3 shows the pseudo-parallel process for implementing the MA.

For actual parallel processing, the sub-graphs are distributed each to an individual processor on which the MA is executed. After all of the processors complete the merging operation, the same number of processors are used to execute the program of the coordinator function. Figure 6.4 shows the tasks needed for the data parallel implementation of the MA. The iterations continue until the final partition is reached. During each iteration, the number of sub-graphs may become less, hence the number of the required processors is reduced, therefore it may release some processors to other tasks in the system.

6.2.2. Functions required for the data parallel method

6.2.2.1. Divider

The responsibility of the divider is to prepare the well-divided subsets of data ready to be distributed to the available processors. A graph representing a circuit, the size of the sub-graph or the number of available processors will be the inputs of the divider. According to these inputs, the graph is divided into many sub-graphs with each one having the same number of cells, except the last-graph which contains the leftover in the graph. Figure 6.5 shows a simplified graph containing 2n cells. To divide this graph into two sub-graphs, which contain n cells each, two sub-graph heads are needed, the cells are sequentially allocated to a sub-graph until the required number of cells is reached, then the link between cell n and cell n+1 must be broken and the one in the front (i.e. cell n) is set to the end of one sub-graph, the one in the rear (i.e. cell n+1) is set to the start of another sub-graph. Figure 6.6

shows two sub-graphs obtained from the graph in Figure 6.5 by dividing it into two parts.

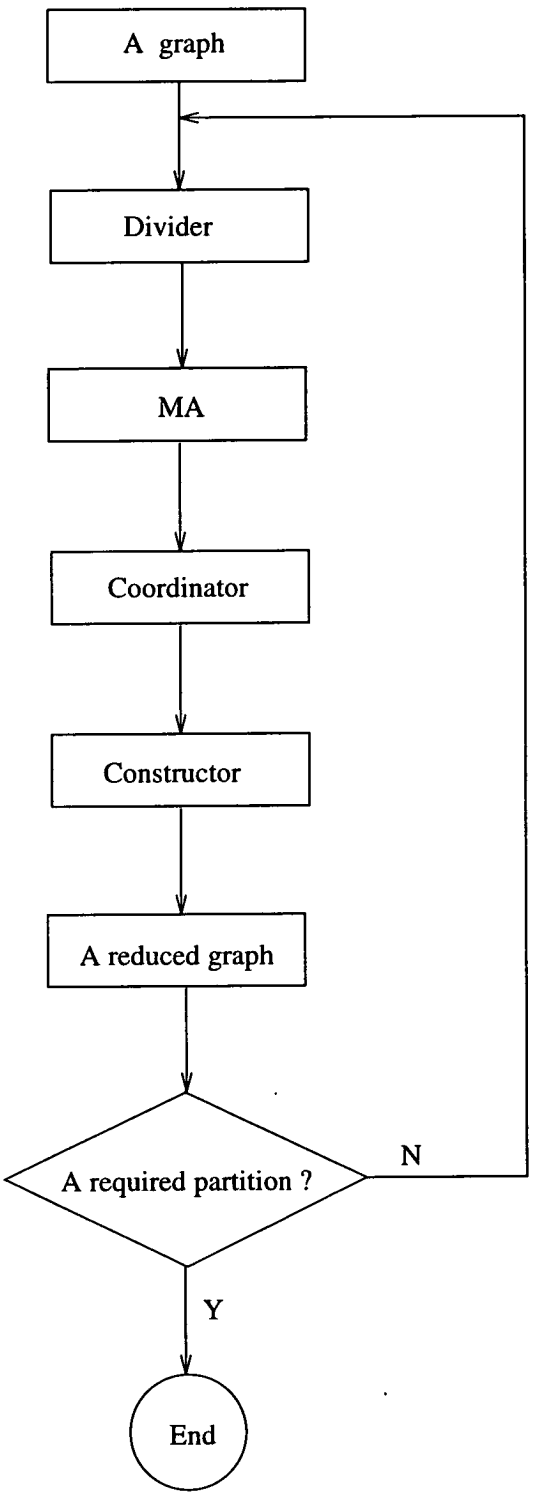


Figure 6.3 Flow chart of the pseudo-parallel merge algorithm

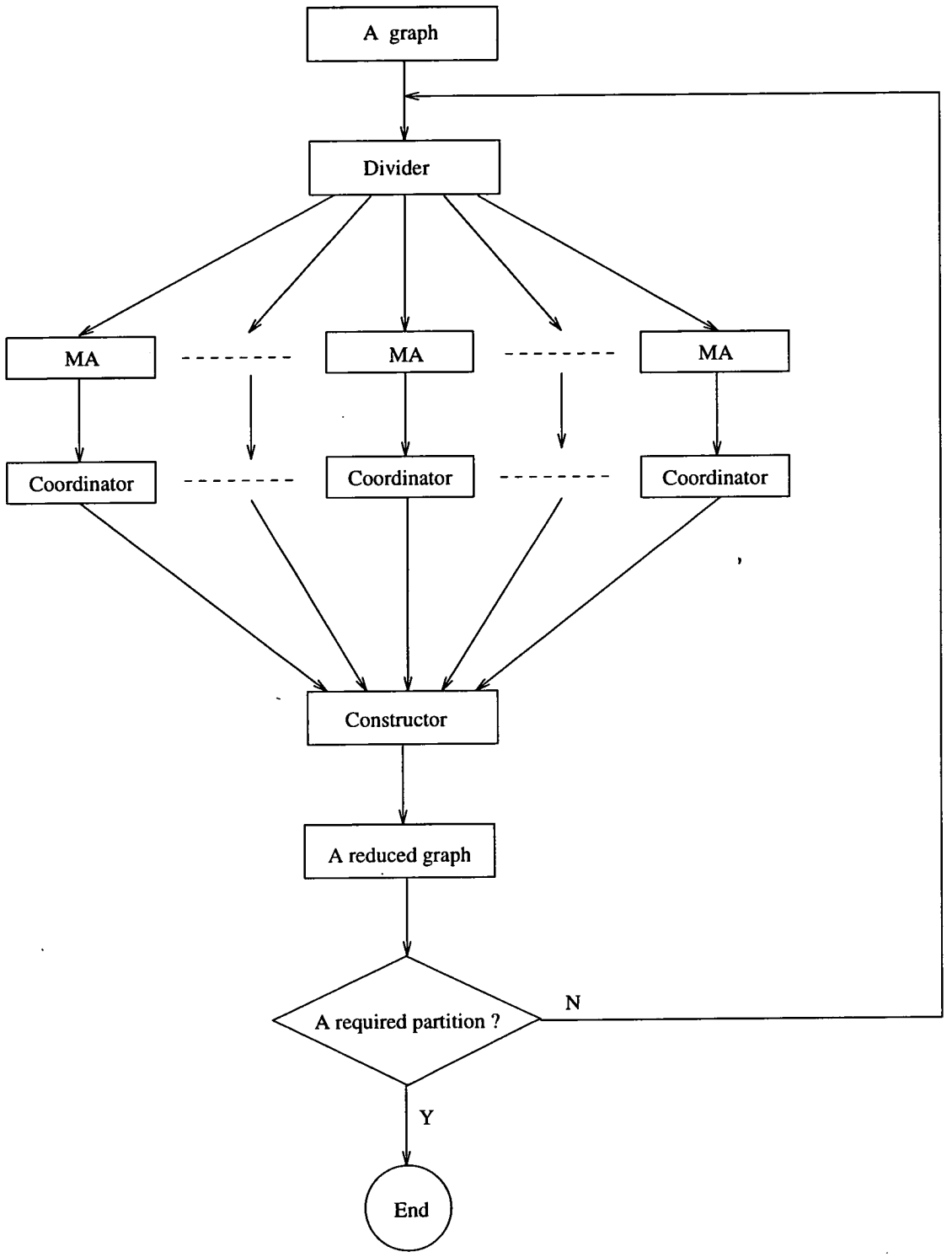


Figure 6.4 The flow chart of the parallel process implementation of the merge algorithm

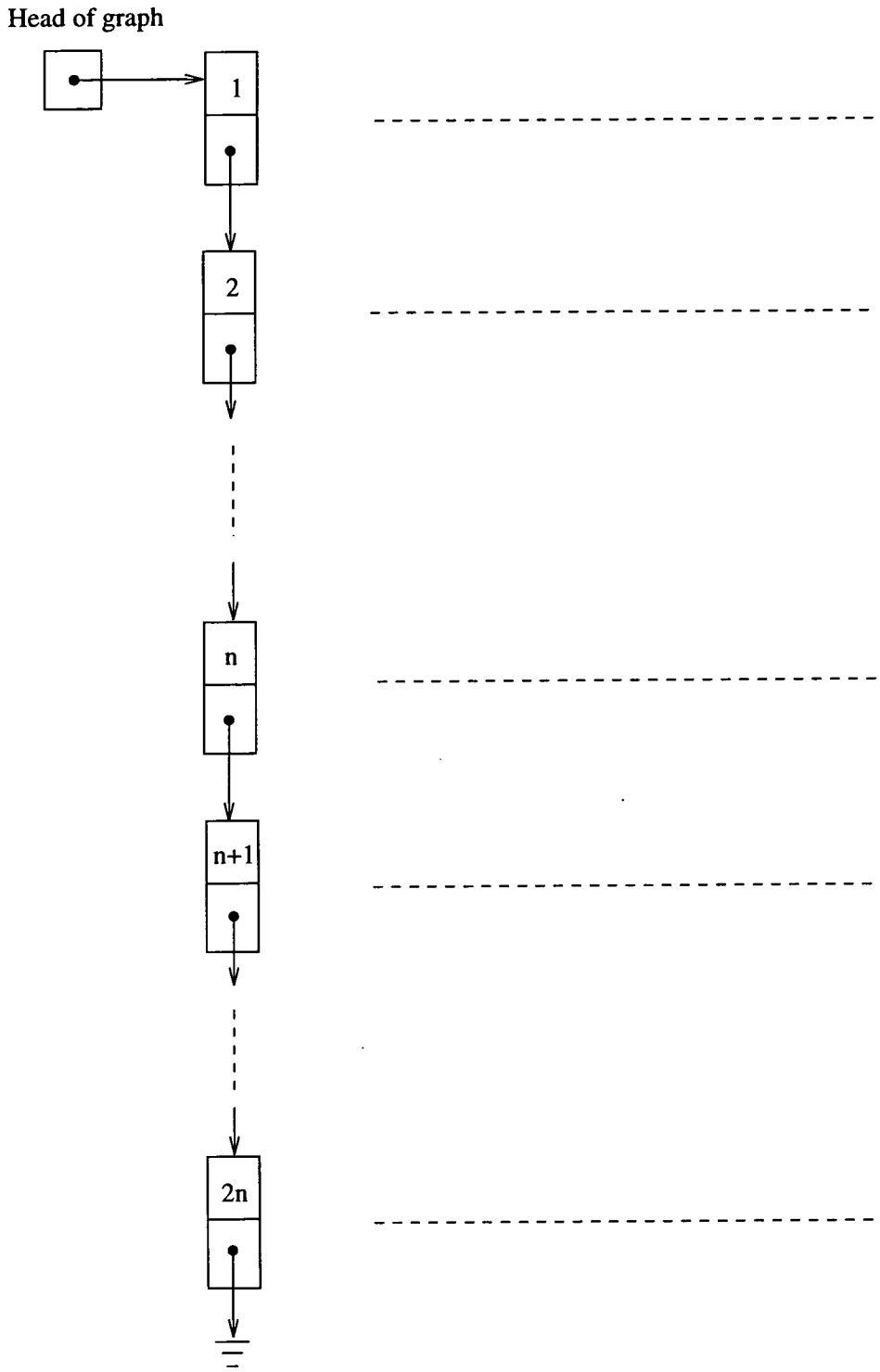


Figure 6.5 A simplified graph with 2n cells

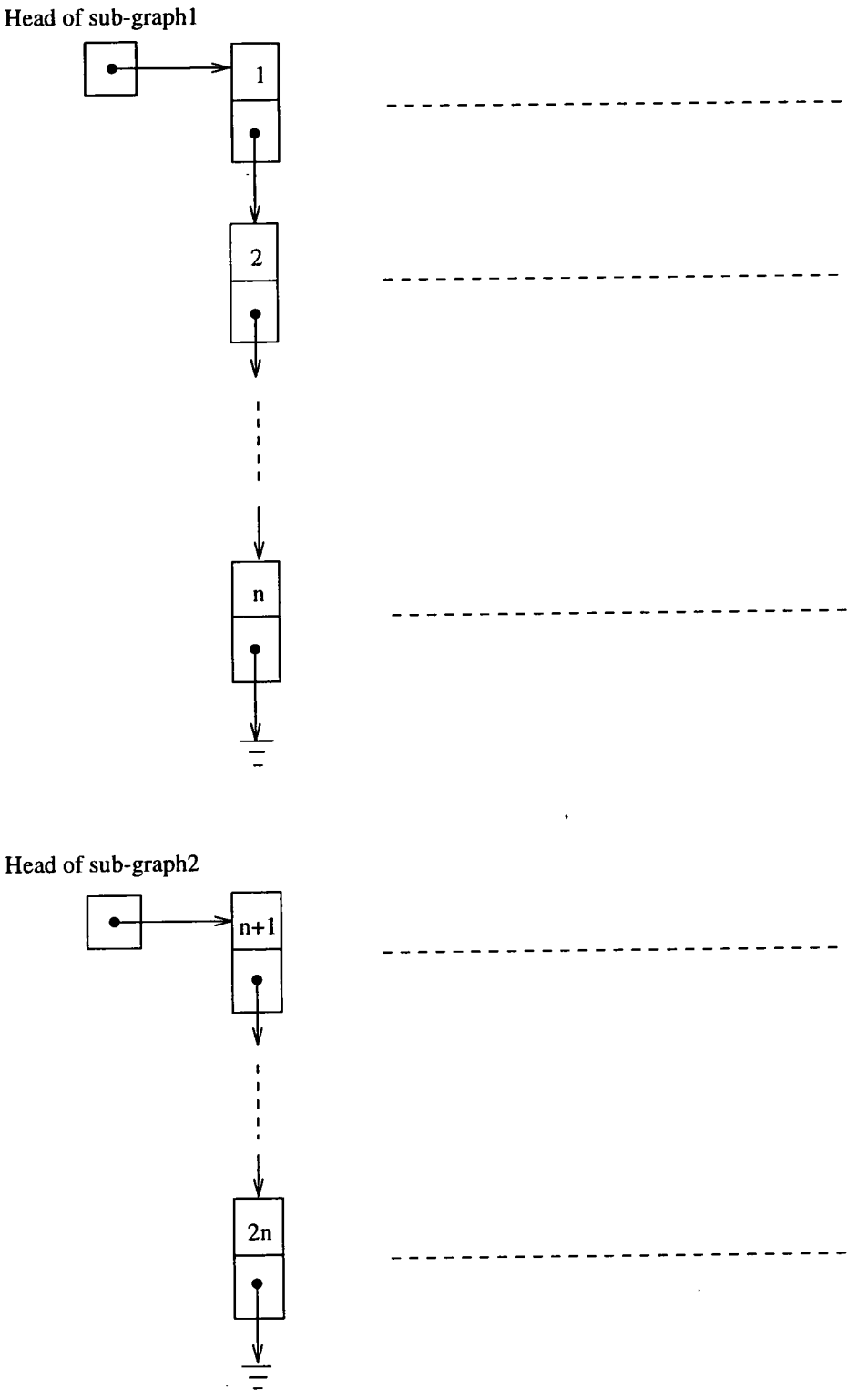


Figure 6.6 Two sub-graphs of Figure 6.5

6.2.2.2. A merge algorithm for parallel process method

The MA used previously operates on the full graph by selecting the cells with the maximum number of connections as the candidates to merge. For the parallel process, the MA will be working on the sub-graphs, only the cells which are all within the same sub-graph are merged. The MA proceeds until no cells in the sub-graph are available to merge.

Because merging occurs within a small portion of the full graph, a record of the corresponding implicit cells and explicit cells is kept to enable the following process to effect synchronisation among sub-graphs.

6.2.2.3. Coordinator

When the parallel version is applied to sub-graphs, the normal actions for changing the names of the implicit cells to the names of the explicit cells are executed only inside the sub-graphs, therefore a routine called the coordinator is required to process the cells which were implicit cells in certain sub-graphs and still are left in other sub-graphs. The main goal of the coordinator is to change the names of these cells to their proper cell names after completion of the merge operations.

The coordinator can be parallelized by loading this routine to a number of processor into which the sub-graph, the cell-net list and the record of the changed cell names are loaded. Every edge node in the sub-graph must be visited to investigate if the cell names have been changed. If they have been changed, then the cell-net list will be used to count the number of connections between the new pair of cells. Then the routine goes to the next edge node. This is continued until all the edge nodes in the sub-graph are exhausted. Figure 6.7 shows the flow chart of the coordinator function.

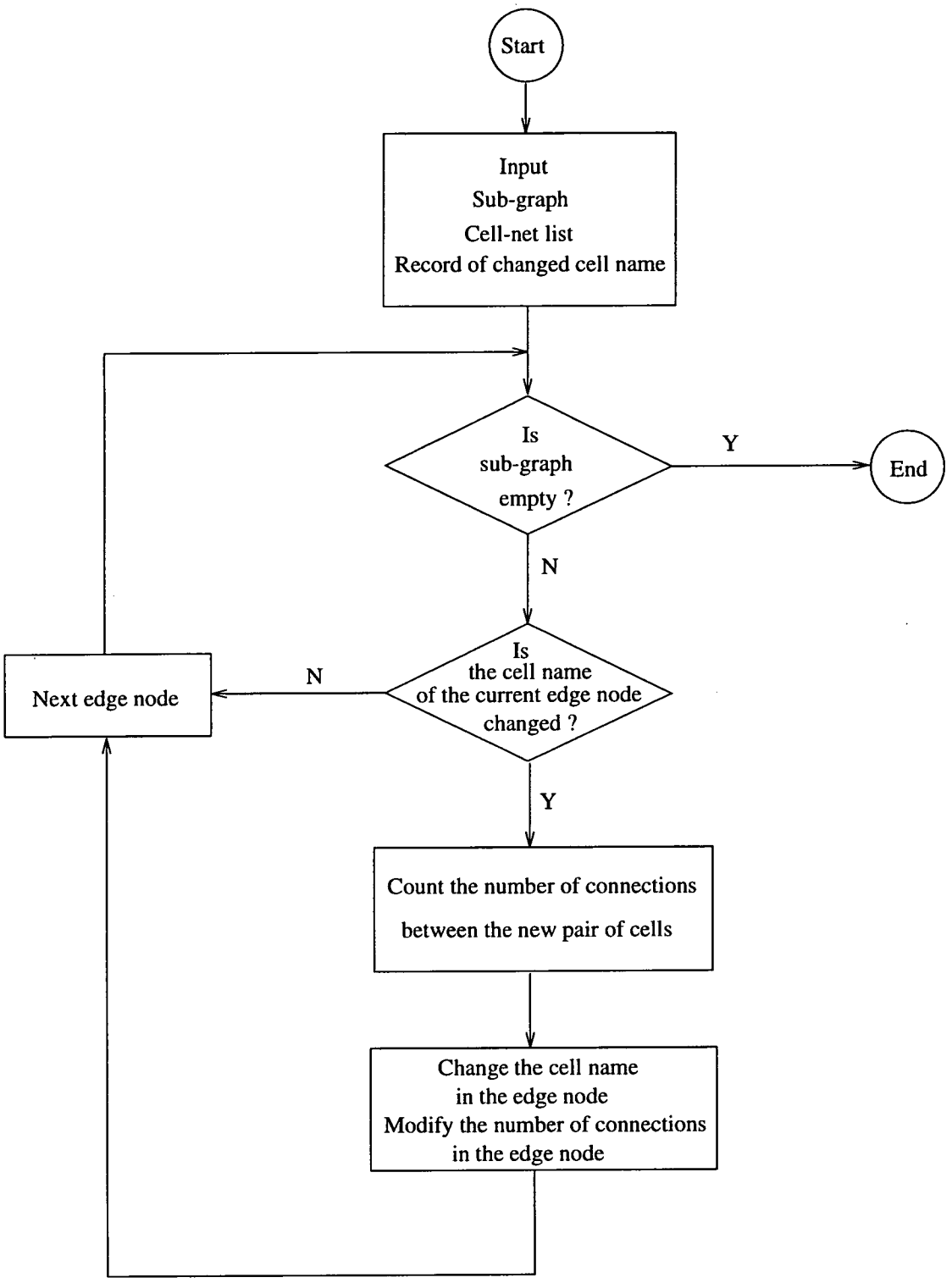


Figure 6.7 The flow chart of the coordinator

6.2.2.4. Constructor

The constructor is a routine that collects all the sub-graphs resulting from the previous processes and combines them into a new reduced full graph. The constructor will re-build the full graph by cascading the sub-graphs sequentially, it starts from the the first sub-graph and always points to the last cell in the sub-graph to the first cell of the next sub-graph. These steps are iterated until the last sub-graph is reached. Finally, the head of the first sub-graph is used as the head of the full graph. Figure 6.6 is used to illustrate the combining of the two sub-graphs which is shown in Figure 6.8.

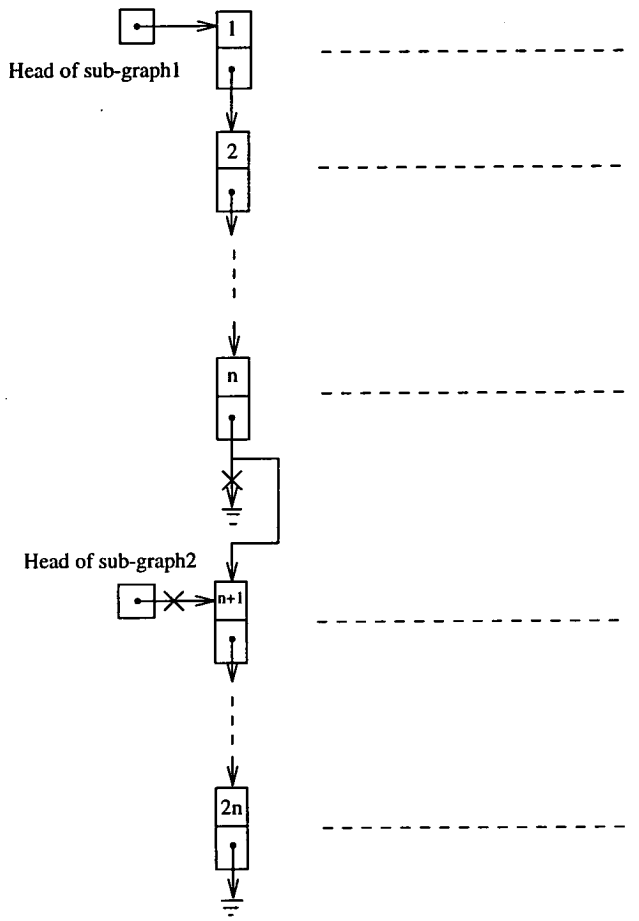


Figure 6.8 The re-construction of Figure 6.6

6.3. Performance of the pseudo-parallel MA

6.3.1. Test circuits

To demonstrate the performance of the pseudo-parallel process, the random circuit generator was used again to generate some test circuits which include one random circuit with 10,000 cells and four structural circuits with various numbers of cells which were constructed semi-automatically, i.e., every part in a circuit was generated automatically, but the connections between parts were defined manually. The main features of these test circuits are shown in Table 6.1.

Test circuits	#Cells	#Nets	#Modules	#I/O
ran01	10,000	12,882	1	20
stru01	1,000	1;292	5	2
stru02	2,000	2,578	5	2
stru03	10,000	12,850	5	2
stru04	12,000	15,471	6	8

Table 6.1 The main features of test circuits

The block diagrams of the four structural circuits (Figure 6.9 to 6.12) and the features of each part in the circuit (Table 6.2 to 6.5) are shown below.

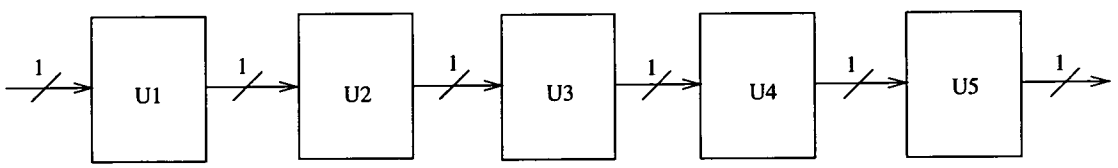


Figure 6.9 The block diagram of stru01

Units	#Cells	#Nets	#I/O
U1	200	255	2
U2	200	258	2
U3	200	263	2
U4	200	258	2
U5	200	258	2

Table 6.2 The main features of stru01

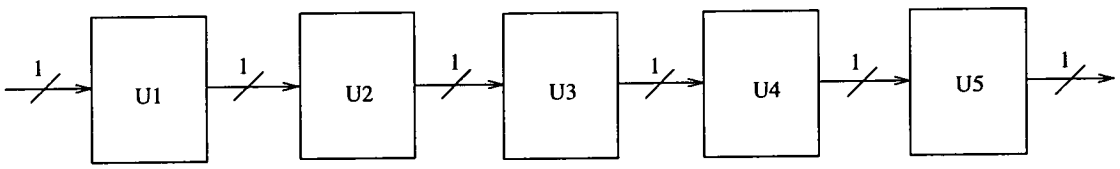


Figure 6.10 The block diagram of stru02

Units	#Cells	#Nets	#I/O
U1	400	510	2
U2	400	520	2
U3	400	518	2
U4	400	514	2
U5	400	516	2

Table 6.3 The main features of stru02

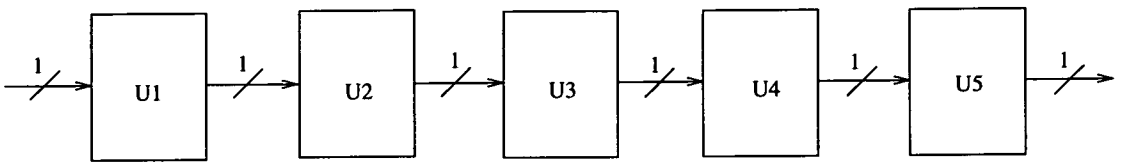


Figure 6.11 The block diagram of stru03

Units	#Cells	#Nets	#I/O
U1	2000	2571	2
U2	2000	2565	2
U3	2000	2564	2
U4	2000	2580	2
U5	2000	2561	2

Table 6.4 The main features of stru03

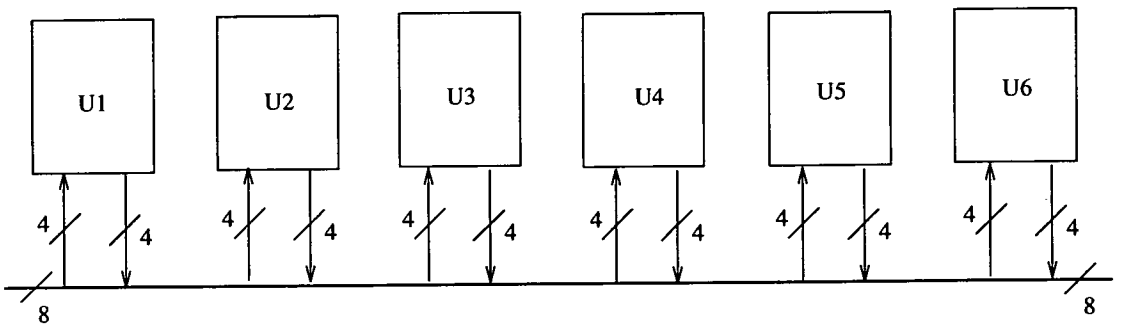


Figure 6.12 The block diagram of stru04

Units	#Cells	#Nets	#I/O
U1	1500	1937	8
U2	3000	3848	8
U3	1500	1925	8
U4	1500	1945	8
U5	3000	3849	8
U6	1500	1967	8

Table 6.5 The main features of stru04

6.3.2. Results

The five test circuits were processed by the pseudo-parallel merge algorithm which uses the "merge in stages" approach with a small initial size constraint. After each iteration the next size constraint is enlarged. The initial size constraint used here is formulated empirically as follows:

$$S_0 = C_{sc}\left(\frac{C}{100}\right)^{-1} \tag{6.1}$$

where S_0 is the initial size constraint, C_{sc} is the size of the sub-circuit, C is the total number of cells in the circuit and 100 is an empirical value. After each iteration the size constraint is doubled. This is repeated until no cells can be merged.

The pseudo-parallel algorithm consists of four sequentially implemented tasks, namely: divider, MA for parallel process, coordinator and constructor. To assess the performance of these four tasks, they are clocked by the system c.p.u. timer to enable the c.p.u. time to be monitored and recorded at each iteration.

The number of processors available in the system will decide how many divisions

the circuit can be divided into. All meaningful and possible number of processors were investigated when the pseudo-parallel merge algorithm was applied to these five circuits. A sample of result shown below is the result of the test circuit stru03 which was run under the assumption of a system with 4 processors available. In the beginning the number of cells in the full circuit is shown; at each iteration the number of cells in the sub-circuit and the number of divisions are displayed, then the c.p.u. time for each task and each sub-circuit are recorded, finally the total run time and the number of cells left in the circuit are shown. In the first iteration, the divider divides the test circuit into four subcircuits each one of which contains 2,500 cells, these four subcircuits are sequentially run through the MA and coordinator, then the constructor combines the four reduced subcircuits into one. The new reduced circuit consists of 2,177 cells after combination. The size of the new circuit is less than 2,500 cells, therefore there is only one division and only one processor is needed in the second iteration. Due to only one division left, there is no synchronisation needed, the coordinator and constructor are not executed and the final partition is obtained in this iteration.

```
The first iteration:
************************************************************
The size of the full circuit = 10,000 cells
The size of the subcircuit = 2,500 cells
The number of divisions = 4

time_divider=0.016667 secs

time_merge(1)=73.949997 secs
time_merge(2)=73.133331 secs
time_merge(3)=58.500000 secs
time_merge(4)=84.400002 secs

time_coordinator(1)=0.300000 secs
time_coordinator(2)=2.950000 secs
time_coordinator(3)=1.966667 secs
time_coordinator(4)=0.000000 secs

time_constructor=0.000000 secs
************************************************************
```

```
The second iteration:
************************************************************
The size of full circuit = 2,177 cells
the size of subcircuit = 2,500 cells
The number of division = 1

time_divider=0.000000 secs

time_merge(1)=265.866669 secs
************************************************************

The final:
************************************************************
Total run time (pseudo parallel) = 561.450012 secs
Predicted parallel run time = 353.21 secs
The number of cells in the size-reduced circuit = 2
************************************************************
```

The five test circuits are individually processed by changing the size of subcircuit and the c.p.u. time taken by each of the four procedures was recorded in the format of the above listing. In the last block the total run time (i.e. the pseudo-parallel run time) and the predicted parallel run time are calculated. The total run time is the time that only one processor is used by running the parallel MA in serial manner. The predicted parallel run time is assumed there are a number of processors running simultaneously, in each iteration the longest time a task takes is used as the net run time, because the other tasks which take shorter run time have to wait until the slowest task has been finished. The run time of the regular MA was also taken to compare with pseudo-parallel and predicted parallel. Figure 6.13 to figure 6.17 show the graph of the speed performance of these five test circuits with the horizontal axis as the size proportion in percentage and the vertical axis as the c.p.u. time the processes take.

The Figure 6.13 shows the speed performance of the test circuit ran01 with respect to the various subcircuit size in percentage to the full size circuit. If the subcircuit size falls between 50% to 100% of the full size circuit, then two processors are required. The less percentage the subcircuit size is, the more the processors are

required. The run time of the pseudo-parallel MA with multiple processors was dramatically reduced. It was about three times as fast as the regular MA with one processor. When the number of processors increased beyond four, the run time increased abruptly. This is due to the test circuit ran01 being randomly connected, the connections between cells may be far away from each other in the netlist, it is not easy to find a pair of cells to merge in the subcircuit itself when the full circuit is divided into too many subcircuits, therefore the size of the full circuit after re-construction does not reduce much, the program keeps looping until no further merge can be made, this makes the run time increase significantly.

Figure 6.14 to figure 6.17 show the speed performance of structural circuits with different structures and various sizes. There are two kinds of curves which are the curve of pseudo-parallel and the curve of predicted parallel algorithm. All these curves in these four graphs have a similar shape which demonstrates the run time of the pseudo-parallel with multi-processor is always much less than the regular merge algorithm, and the run time of predicted parallel is less than the run time of pseudo-parallel. There is a special situation when the size of the subcircuit matches the structure of the circuits and the run time is the shortest. For example, stru01 with 1000cells consisting of five blocks of circuits with 200 cells each has a shortest run time when the size of the subcircuit is 200.

To assess the pseudo-parallel MA, the structural circuits have known optimal cutsize. The test circuits stru01, stru02 and stru03 have optimal cutsize equal to 1 when doing bi-partition. The test circuit stru04 has a bus with 8 wires passing through every unit of the circuit. Obviously, the optimal cutsize for bi-partition should be 8. Applying pseudo-parallel MA to the first three test circuits, the optimal solutions for bi-partition were always found. Applying pseudo-parallel MA to the stru04, the optimal solution was not found, most of the case the cutsize were greater

than 8. This is due to some pairs of cells in the different units having bigger connection number through the bus which were selected to merge, and this draw more wires which do not belong to the bus, passing through the boundary. When changing the size constraints, the results of the cutsize varied. One point that can be sure is that the cutset of the partition always contains the bus.

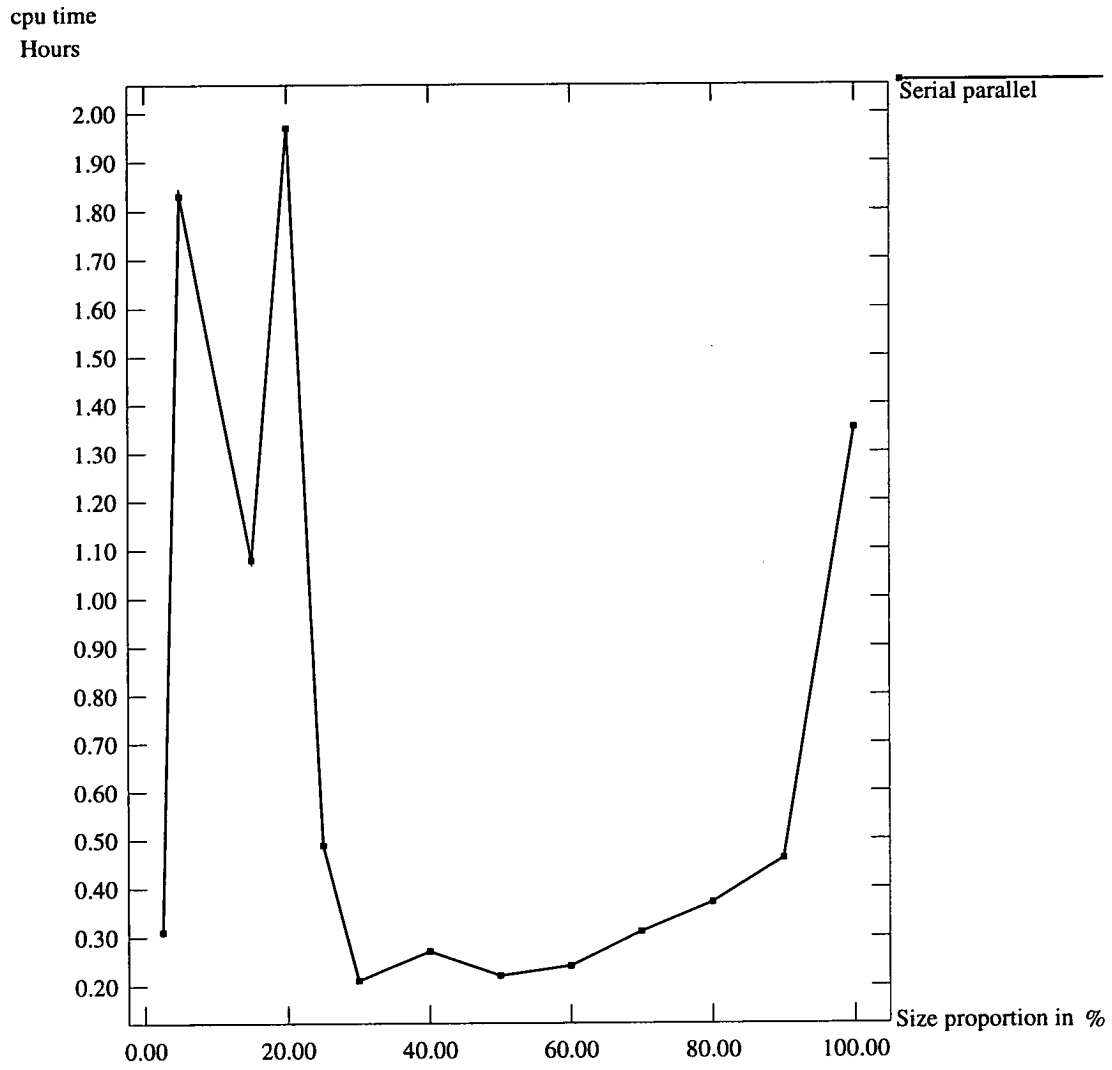


Figure 6.13 Speed performance of the pseudo-parallel merge algorithm when ran01 is partitioned

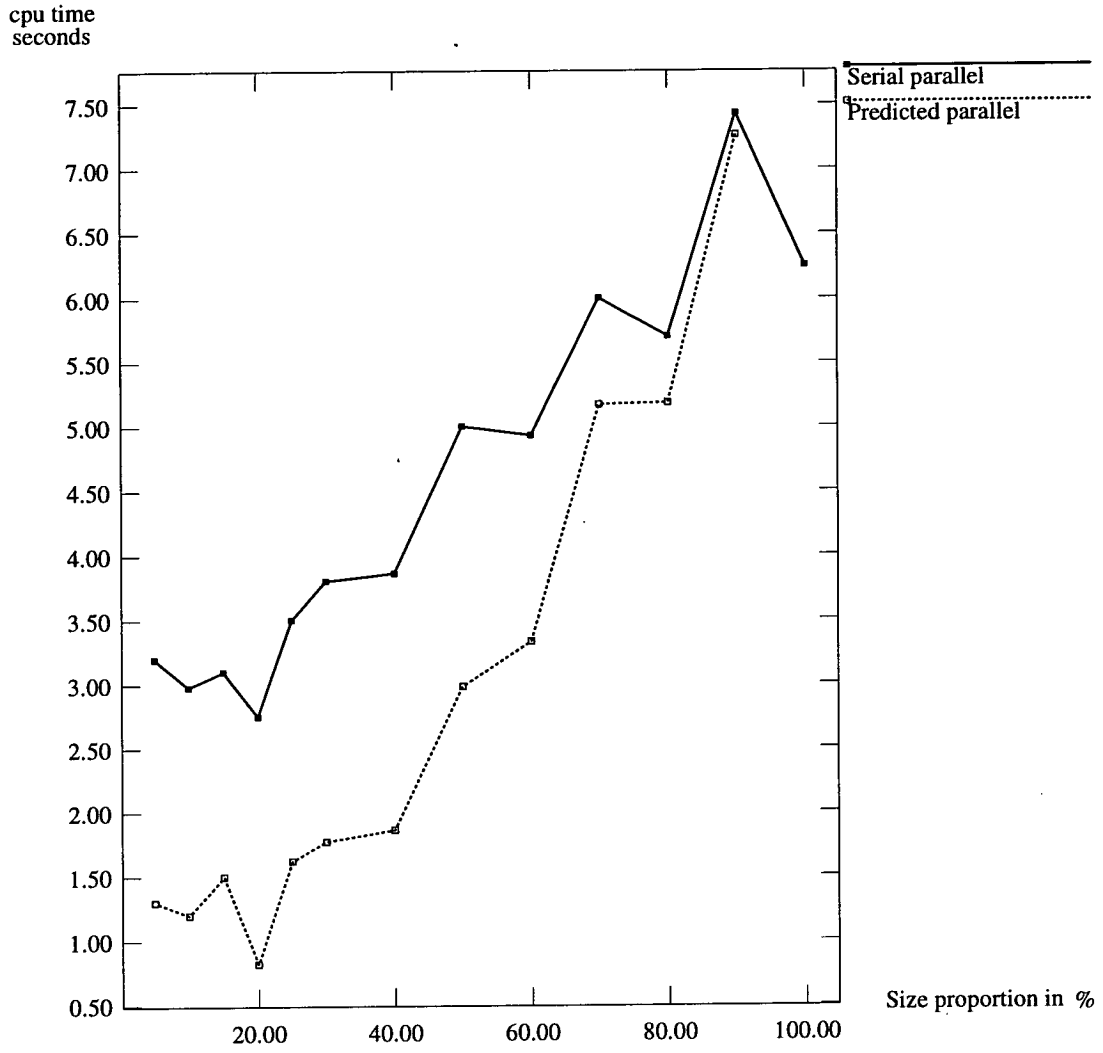


Figure 6.14 Speed performance of the pseudo-parallel merge algorithm when stru01 is partitioned

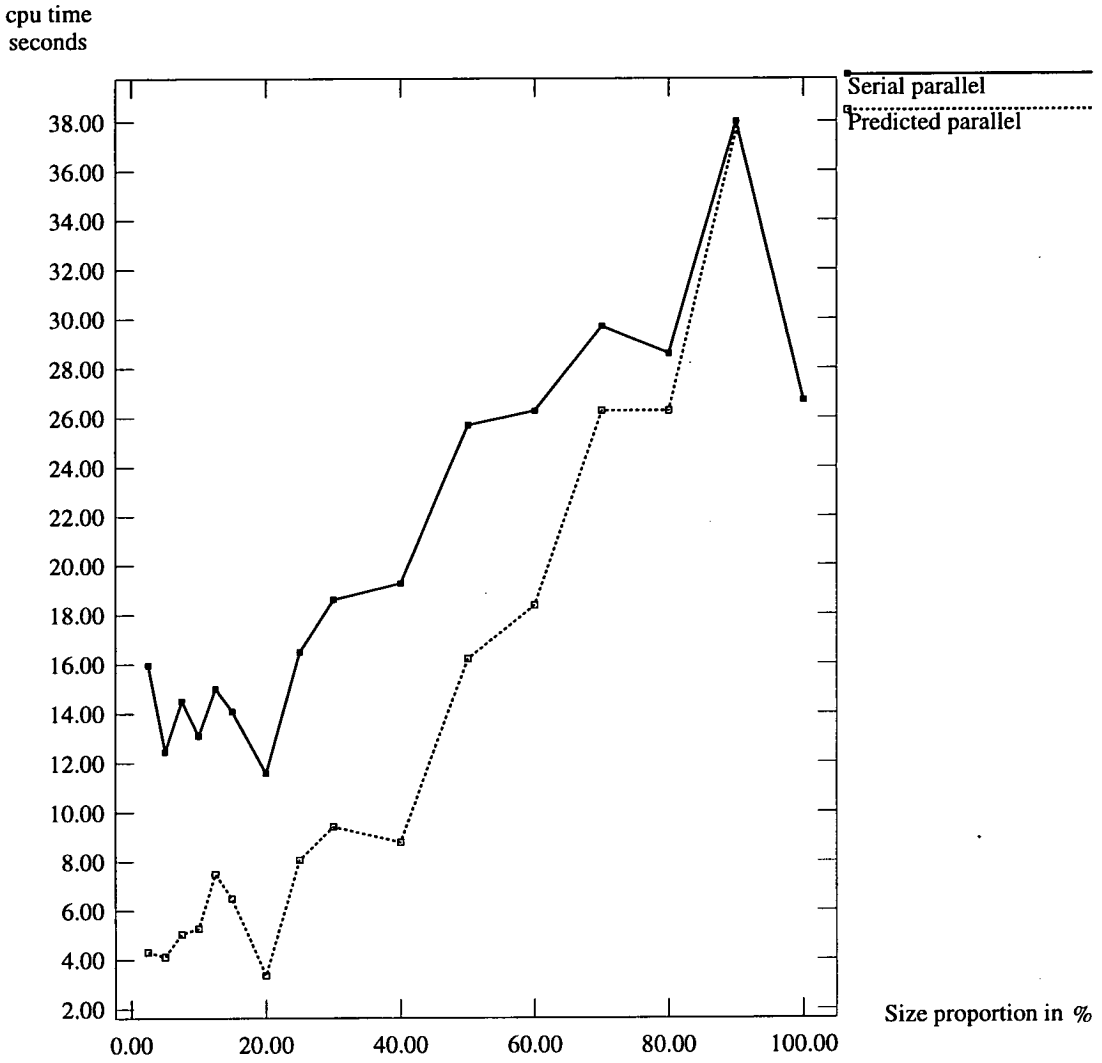


Figure 6.15 Speed performance of the pseudo-parallel merge algorithm when stru02 is partitioned

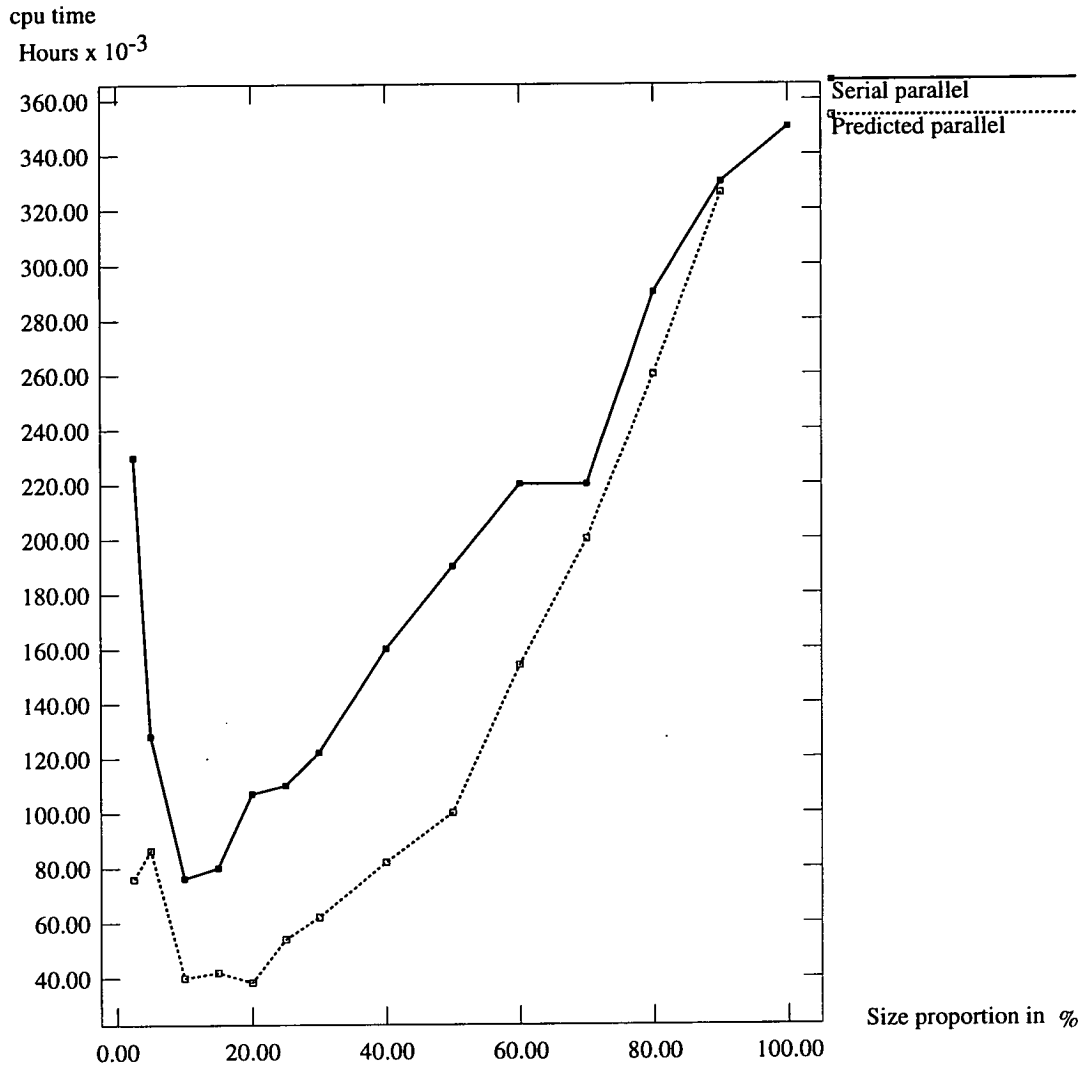


Figure 6.16 Speed performance of the pseudo-parallel merge algorithm when stru03 is partitioned

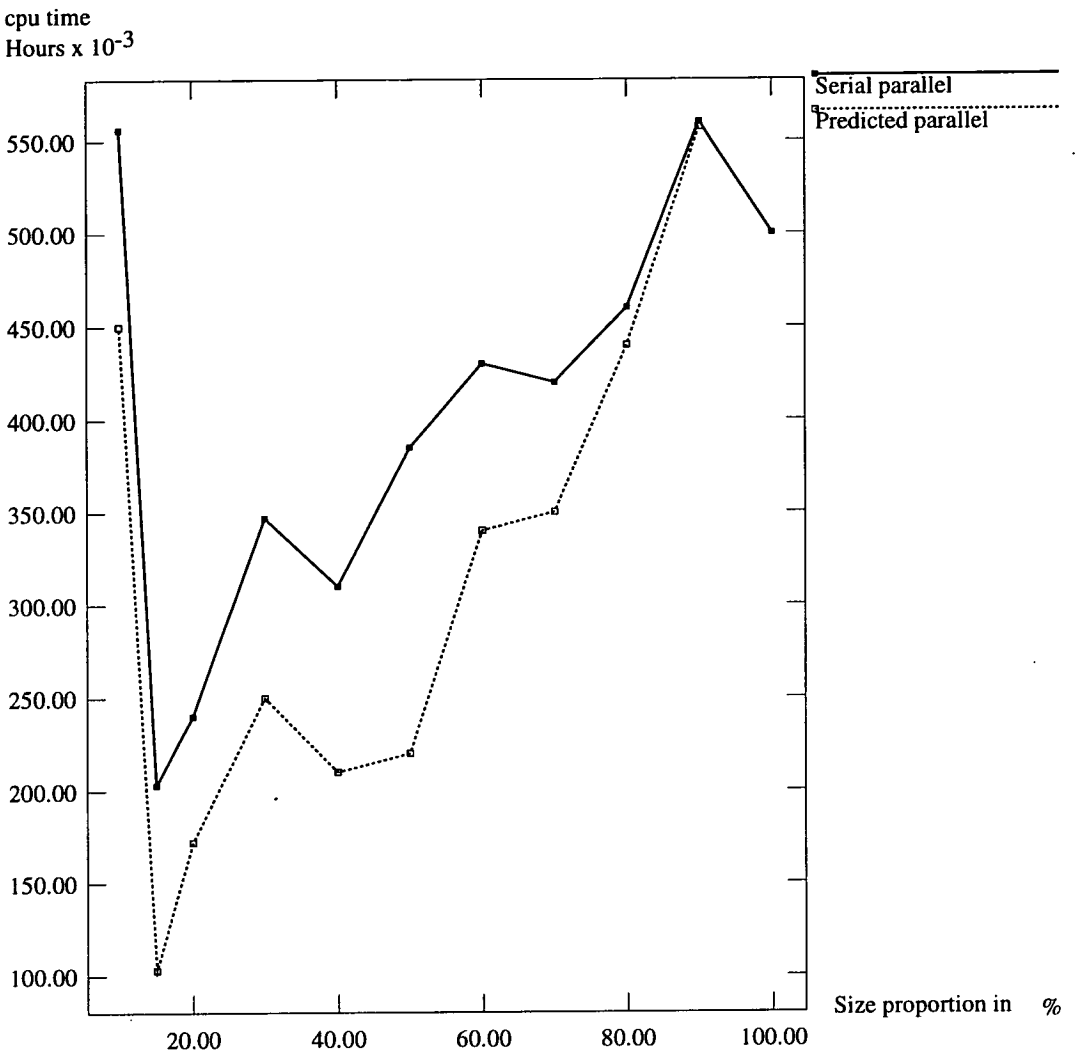


Figure 6.17 Speed performance of the pseudo-parallel merge algorithm when stru04 is partitioned

6.4. Summary

This pseudo-parallel MA presented in this chapter proved to have better performance than the regular MA while maintaining the same quality of the result. The performance of the predicted parallel MA is even better than the pseudo-parallel version. Both pseudo-parallel and real parallel can quickly let users explore a wide range of possible partitions.

CHAPTER 7

Summary and Conclusions

7.1. Summary

Integrated circuits continue to grow in size and complexity. Correspondingly, the design process associated with these circuits has grown lengthy and expensive. A rapid prototype of an ASIC design is strongly needed to reduce time-to-prototype and expenses of the overall design cycle [20], [92]. Typically designs cannot be implemented by a single device, hence how designs are partitioned into a set of PLDs that form a digital emulator so that the design's functionality can be evaluated, is an important problem. Therefore partitioning is one of the most important subtasks in preparing a design database for emulation.

In this thesis techniques for partitioning problems were reviewed. These are drawn from the branch of mathematics known as "graph algorithms". In many cases, however, the solution techniques belong to a class of computational problem for which no exact solution may be generated in polynomial time. This limitation is overcome by the introduction of heuristics to constrain the solution space. The use of heuristics may obtain a solution within a reasonable time by compromising the solution quality.

A merge algorithm for circuit partitioning was introduced. It is based on the simple concept that cells with the maximum number of connections should be the first to be merged. The merging operations are processed in several stages, having a predefined

initial size constraint on groups of cells specified at the first stage, the size constraint on groups is enlarged at the subsequent stages. By specifying the number of groups required and the size of each group in the final partition, a desired partition or suggested partition will be obtained.

The merge algorithm was implemented by means of the C programming language. A well-defined data structure was developed which is capable of dealing with the circuits having multi-terminal nets and can be extended easily to incorporate other information from the circuit itself or user's specification. The cell-net list was set up for coping with the multi-terminal net problem and was used to acquire the new number of connections between cells after merging. The graph was used to form a new configuration of the circuit after each merging operation was completed.

The merge algorithm was fully exercised on test circuits by giving various initial size constraint and associated number of stages which produced different results. A balance factor was introduced to judge if a result is in a reasonable balance. A poor balance usually leads to a small cutsize. When the balance factor falls in a certain range, the cutsize varies little. A structured circuit with known optimal partition was created to test the ability of merge algorithm to find this optimal partition. The results proved that the merge algorithm has a much better performance than the Kernighan-Lin based approach.

The merge algorithm presented in this thesis was extended to a pseudo-parallel version. A set of procedures were developed which are the divider, merge algorithm for parallel implementation, coordinator and constructor. The divider divides a circuit into several subcircuits which are sequentially processed by the parallel merge algorithm which produces size-reduced subcircuits. The coordinator modifies these smaller subcircuits which are recombined into a size-reduced full circuit. This cycle is repeated until a final partition is reached. Five test circuits were created to

exercise the pseudo-parallel merge algorithm. The results showed the pseudo-parallel version had better speed performance than the basic version of the MA.

7.2. Conclusions

A merge algorithm has been investigated and it has been shown to be an effective method for partitioning large circuits into a set of smaller circuits. A particular feature of the algorithm is its suitability for implementation by a parallel processor array. This parallel processing feature has been investigated by a pseudo-parallel technique which has itself been shown to offer practical advantages. The C programming language has been used to develop the experimental software implementation of the algorithm. So its development for use in future research should be straightforward. Further work is required to facilitate the practical applications of the MA. Some proposals follow.

7.3. Further work

This section suggests a number of potential research areas which extend and augment the work presented in this thesis.

7.3.1. Partitioning for improving place and route

The functionality and routing resources are limited in FPGA, and placement and routing is the most time-consuming part of the FPGA design process, therefore producing a design with higher routability is preferable than producing a design which is difficult to route [43]. Thus, making a good preparation in the partitioning stage for facilitating to execute "Place and Route" is important [11-12], [72-83].

In the partitioning stage, the partitioner will split a large network into several blocks to fit into the individual FPGAs. The next stage is the assignment of the network cells to physical cells on the Logical Cell Array in the FPGA [44] and the

configuration of routing structures to interconnect them as in the network (i.e. placement and routing).

A partitioner is typically only concerned with the gate and pin counts that an FPGA can accommodate, and leaves the most time-consuming tasks of performing the actual placement and routing to the vendor software for individual FPGAs [45-46]. This makes automatic placement start with a random placement which may be difficult to route, or in the worse case unroutable. A good initial placement would increase the routability.

Usually, the MA using the partitioning in stages strategy partitions a design into a number of large groups that fit the target devices. In the intermediate stage, the partition will consist of many small groups. The available of logic cells in an FPGA can be used as a constraint on the intermediate merging stage. This intermediate partition will provide a useful initial input to the automatic placement routine. In addition, the MA tends to merge more nets into a group and leave fewer nets between groups which will alleviate the problem of limited routing resources. Properly using the above two properties can give a good initial placement.

7.3.2. Improving timing performance

Many techniques can be used to improve the timing performance of a circuit. According to the design level applied, they can be divided into three categories [52]. At the structural level, the internal structures of gates and their interconnections in a circuit are modified to improve circuit performance. For example, the technique of converting a ripple-carry adder into a carry-lookahead adder belongs to this category [53]. This method will change the connections between the gates, even change the design. At the physical level, techniques of transistor sizing, buffering, and powering are used to improve gate speed [54-60]. These techniques result in

increased circuit size and circuit power consumption. It's improved performance carries a price. At the topological level, performance-driven placement of gates and performance-driven routing of wires are aimed at minimising the delay of the longest paths [52], [71], while the connections between components are retained.

Generally speaking, a longer path will result in a longer time delay and external connections will cause a longer delay than internal connections. If the longer paths have the higher priority to route and the cells on the timing sensitive paths have the precedence to place, the timing performance of the design can be improved by taking these considerations into account. A timing-driven partition algorithm with a capability to pre-allocate timing critical cells onto a group and supply the necessary information about the critical paths to the software for place and route to optimise the timing performance, possesses the ability to retain the connections among cells in the design [47-49].

The merge algorithm can be guided by timing constraints to partition circuits with timing-performance problem. Timing analysis software [50-51] is needed to produce a timing data base which sequentially lists the critical paths and the cells along these paths. According to the timing data base, the processes of merging cells on the critical path into a group are taking place in advance. After the merge operation on the timing data base is finished, the MA is applied to the new configuration of the circuit. This technique can prevent the critical timing path travelling through to external connections, which will result in a longer delay.

7.3.3. Equal weight

In the merge algorithm, when there are several pairs of cells having the same number of connections the choice is made to select the first pair of cells in the list to be merged. How the merge algorithm differentiates the pairs of cells having the same

weight and provide information that can guide the merge algorithm to make the best choice of cells to merge, should be investigated.

7.3.4. The methods for selecting candidates to merge

The simple concept that cells with the maximum number of connections should be merged first, is adopted in the merge algorithm. There should be some other metrics that can be used to make a good cells selection and improve the results generated from the algorithm using simple concept. This is worth investigation.

7.4. The benefits of the Merge Algorithm

Although the Kernighan-Lin based approaches are popular in practical use, there are still some disadvantages. The merge algorithm presented in this thesis possesses a number of advantages which are stated as follows:

- No initial partition required.
- An excellent potential to find the optimal partition.
- A well-defined data structure that can easily incorporate the required information to suit various circuit requirements.
- A multi-way partitioning algorithm.
- A parallelizable algorithm which can be executed on a multi-processor computer to reduce processing time as circuit size increases.

REFERENCES

[1] H. R. Charney and D. L. Plato, "Efficient partitioning of components," IEEE Design Automation, pp. 16.0-16.21, July 1968.

[2] T. Lengauer, Combinatorial Algorithms for Integrated Circuit Layout, Paderborn, Germany: University of Paderborn, 1990.

[3] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," Bell Syst. Tech. J., vol. 49, no. 2, pp. 291-307, Feb. 1970.

[4] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in Proc. 19th Design Automation Conf., 1982, pp. 175-181.

[5] B. Krishnamurthy, "An improved min-cut algorithm for partitioning VLSI networks," IEEE Trans. Comput., vol. C-33, pp. 438-446, May 1984.

[6] Y. C. Wei and C. K. Cheng, "Ratio cut partitioning for hierarchical designs," IEEE Trans. on Computer-Aided Design, vol. 10, no. 7, July 1991, pp. 911-921.

[7] InCA, Virtual ASIC Reference Manual, InCA Ltd., Gibbs House, Kennel Ride, Ascot, Berkshire, SL5 7NT, U. K., 1991.

[8] D. M. Schuler and E. G. Ulrich, "Clustering and linear placement," in Proc. 9th Design Automation Workshop, 1972, pp. 50-56.

[9] H. Shin and C. Kim, "A simple yet effective technique for partitioning," IEEE Trans. on VLSI Syst., vol. 1, no. 3, Sep. 1993, pp. 380-386.

[10] J. Cong and M. Smith, "A parallel bottom-up clustering algorithm with applications to circuit partition in VLSI design," in Proc. 30th ACM/IEEE Design Automation conf., 1993, pp. 755-760.

[11] M. A. Breuer, "Min-cut placement," J. Design and Fault-Tolerant Computing, vol. 1, no. 4, pp. 343-362, Oct. 1977.

[12] L. I. Corrigan, "A placement capability based on partitioning," in Proc. 16th Design Automation Conf., June 1979, pp. 406-413.

[13] M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, CA, 1979.

[14] R. Sedgewick, Algorithms in C, Princeton University, 1990.

[15] G. E. Moore, "Are We Really Ready for VLSI?," in Proc. Caltech Conf. on Very Large Scale Integration, 1979.

[16] R. B. Fair, "Challenges to manufacturing submicron, ultra-large scale integrated circuits," Proc. IEEE, 78(11), 1990, pp. 1687-1704.

[17] B. T. Egan, "Designers search for the secret to ease ASIC migration," Computer Design, December 1991, pp. 78-94.

[18] S. Pugh, Total Design, Addison-Wesley, 1990.

[19] K. Perry, "Eliminating barriers to FPGA use by timing driven partitioning," Electronic Engineering, January 1993, pp. 41-44.

[20] Quickturn Design Systems, "Rapid prototyping systems for early hardware verification," Electronic Product Design, October 1994, pp. 59-61.

[21] D.E. Van den Bout, J.N. Morris, D. Thomae, S. Labrozzi, S. Wingo and D. Hallman, "AnyBoard: An FPGA-based, Reconfigurable System," IEEE Design & Test of Computers, September 1992, pp. 21-29.

[22] D.A. Thomae, "Using the Anyboard Partitioner," Electrical and Computer Engineering Department, North Carolina State University, April 1992.

[23] D.A. Thomae and D.E. Van den Bout, "Automatic Circuit Partitioning in the Anyboard Rapid Prototyping System submitted to Microprocessors and Microsystems," Electrical and Computer Engineering Department, North Carolina State University, April 1992.

[24] D.A. Thomae, "Performance Directed Partitioning for the Anyboard," Electrical and Computer Engineering Department, North Carolina State University, March 1992.

[25] J. Frankle and R.M. Karp, "Circuit placement and cost bounds by eigenvector decomposition," in Proc. Int. Conf. on Computer-Aided Design, 1986, pp. 414-417.

[26] L.R. Ford and R.M. Fulkerson, Flows in Networks. Princeton, NJ: Princeton University, 1962.

[27] L.A. Sanchis,"Multi-way network partitioning," IEEE Trans. on Computers, vol. 38, January 1989, pp. 62-81.

[28] D.G. Schweikert and B.W. Kernighan, "A proper model for the partitioning of electrical circuits," in Proc. 9th Design Automation Workshop, 1972, pp.

57-62.

[29] C. Sechen and D. Chen, "An improved objective function for mincut circuit partitioning," in Proc. Int. Conf. on Computer-Aided Design, 1988, pp. 502-505.

[30] S. Kirkpatrick, C.D. Gellatt and M.P. Vecchi, "Optimization by simulated annealing," Science, vol. 220, 1983, pp. 671-680.

[31] D.S. Johnson, "The NP-completeness column, an ongoing guide," Journal of Algorithms, 1984, 10th edition, 5:147-160.

[32] D.S. Johnson, "The NP-completeness column, an ongoing guide," Journal of Algorithms, 1985, 16th edition, 6:434-451.

[33] J.W. Greene and K.J. Supowit, "Simulated annealing without rejected moves," in Proc. Int. Conf. on Computer Design: VLSI in Computers, IEEE, 1984, pp. 658-663.

[34] D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, "Optimization by simulated annealing: An experimental evaluation (part I)," Preprint, AT&T Bell Laboratories, Murray Hill, NJ, 1985.

[35] D.W. Matula and F. Shahrokhi, Graph partitioning by sparse cuts and maximum concurrent flow, Technical Report 86-CSE-6, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX, 1986.

[36] XILINX: The Programmable Gate Array Data Book. 2100 Logic Drive, San Jose, CA 95124, 1992.

[37] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," in ACM IEEE 27th Design Automation Conference Proceedings, June 1990, pp. 620-625.

[38] R. Murgai, N. Shenoy, R.K. Brayton and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithms for table look up architectures," in IEEE International Conference on Computer-Aided Design ICCAD-91, Nov. 1991, pp.564-567.

[39] N.-S. Woo, "A heuristic method for FPGA technology mapping based on the edge visibility," in ACM IEEE 28th Design Automation Conference Proceedings, June 1991, pp. 248-251.

[40] R.J. Franics, J. Rose and K. Chung, "Chortle: A technology mapping program for lookup table-based field programmable gate arrays," in ACM IEEE 27th Design Automation Conference Proceedings, June 1990, pp. 227-233.

[41] F. Dresig, O. Rettig and U.G. Baitinger, "Logic synthesis for universal logic cells," in Proceedings of International Workshop on Field Programmable Logic and Applications, Sep. 1991.

[42] D. Filo, J. Yang, F. Mailhot, and G. Micheli, "Technology Mapping for a Two-Output RAM-based Field-Programmable Gate Arrays," in European Design Automation Conference, Feb. 1991, pp. 534-538.

[43] M. Schlag, J. Kong and P.K. Chan, "Routability-Driven Technology Mapping for LookUp Table-Based FPGAs," IEEE ICCD, Oct. 1992, pp. 86-90.

[44] R.R. Munoz and C.E. Stroud, "Automatic partitioning of programmable logic devices," in VLSI Systems Design, Oct. 1987, pp. 74-86.

[45] Quickturn Systems Inc., 325 East Middlefield Road, Mountain View, CA 94043, 1991.

[46] P.K. Chan, M. Schlag and J.Y. Zien, "On Routability Prediction for Field-Programmable Gate Arrays," 30th ACM/IEEE Design Automation Conference, 1993, pp.326-330.

[47] I. Lin and D.H.C. Du, "Performance-Driven Constructive Placement," Proc. of 27th Design Automation Conference, 1990, pp. 103-106.

[48] M. Marek-Sadowska and S.P. Lin, "Timing Driven Placement," Proc. of ICCAD, 1989, pp. 94-97.

[49] W. Donath, "Timing Driven Placement Using Complete Path Delays," Proc. of 27th Design Automation Conference, 1990, pp. 84-89.

[50] K. Roy and J. Abraham, "The Use of RTL Descriptions in Accurate Timing Verification and Test Generation," IEEE Journal of Solid State Circuits, Sep. 1991.

[51] S.K. Nag and K. Roy, "Iterative Wirability and Performance Improvement for FPGAs," 30th ACM/IEEE Design Automation Conference, 1993, pp. 321-325.

[52] H.C. Chen, D.H.C. Du and L.R. Liu, "Critical Path Selection for Performance Optimization," IEEE Trans. on Computer-Aided Design, vol. 12, No. 2, Feb. 1993, pp. 185-195.

[53] J. Fishburn, "A depth-decreasing heuristic for combinational logic; or How to Convert a ripple-carry adder into a carry-lookahead adder or anything in-between," in Proc. 27th Design Automation Conf., 1990, pp. 361-364.

[54] A. Al-Khalili, Y. Zhu and D. Al-Khalili, "A module generator for optimized CMOS buffers," in Proc. 26th Design Automation Conf., 1989, pp. 245-250.

[55] M. Cirit, "Transistor sizing in CMOS circuits," in Proc. 24th Design Automation Conf., 1987, pp. 121-124.

[56] G. DeMicheli, "Performance-oriented synthesis in the Yorktown silicon compiler," in Proc. IEEE Int. Conf. Computer-Aided Design, 1986, pp. 138-141.

[57] J. Fishburn and A. Dunlop, "TILOS: A polynomial programming approach to transistor sizing," in Proc. IEEE Int. Conf. Computer-Aided Design, 1985, pp. 326-328.

[58] W. Kao, N. Fathi and C. Lee, "Algorithms for automatic transistor sizing in CMOS digital circuits," in Proc. 22nd Design Automation Conf., 1985, pp. 781-784.

[59] D. Maple, "Transistor size optimization in the tailor layout system," in Proc. 26th Design Automation Conf., 1989, pp. 43-48.

[60] F. Obermeier and R. Katz, "An electrical optimizer that considers physical layout," in Proc. 25th Design Automation Conf., 1988, pp. 453-459.

[61] M. Hanan and J. Kurtzberg, "Placement Techniques," in M. Breuer, ed., Design Automation of Digital System, Prentice-Hall, 1972.

[62] J.M. Kurtzberg, "Algorithms for Backplane Formation," in Microelectronics in Large Systems, Spartan Books, 1965, pp. 51-76.

[63] C. Berge, The Theory of Graphs and its Application. New York, John Wiley & Sons, Inc., 1962.

[64] F. Harary, Graph Theory, Reading, Mass., Addison-Wesley Publishing Company, Inc., 1969.

[65] O. Ore, Theory of Graphs, Providence, R.I., American Mathematical Society, 1962

[66] R.L. Gamblin, M.Q. Jacobs, and C.J. Tunis, "Automatic Packaging of Miniaturized Circuits," in G.A. Walker, ed., Advances in Electronic Circuit Packaging, Vol. 2, New York, Plenum Press, 1962, pp. 219-232.

[67] H.A. Nidecker and W.F. Simon, "Logic Partitioning-Component Assignment," Proc. 1968 ACM Nat. Conf., pp. 211-221.

[68] M.N. Weindling, "A Method for Best Placement of Units on a Plane," Proc. 1964 SHARE Design Automation Workshop, and Douglas Paper 3108, Douglas Aircraft Co., Santa Monica, California.

[69] G.Micheli, A. Sangiovanni-Vincentelli and P. Antognetti, Design Systems for VLSI Circuits:Logic Synthesis and Silicon Compilation, Martinus Nijhoff Publishers, 1987.

[70] D.F. Wong, H.W. Leong and C.L. Liu, Simulated Annealing for VLSI Design, Kluwer Academic Publishers, 1988.

[71] E.L. et al. Lawler, "Module Clustering to Minimize Delay in Digital Networks," IEEE Trans. on Computers, vol. C-18, Jan. 1969, pp. 47-57.

[72] L.I. Corrigan, "A placement capability based on partitioning," in Proc. 16th Design Automation Conf., June 1979, pp. 406-413.

[73] M. Edahiro and T. Yoshimura, "New placement and global routing algorithms for standard cell layout," in Proc. 27th Design Automation Conf., June 1990,

pp. 642-645.

[74] M. Igusa, M. Beardsiee, and A. Sangiovanni-Vincentelli, "ORCA: A sea-of-gates place and route system," in Proc. 26th Design Automatic Conf., June 1989, pp. 122-127.

[75] S. Murai, H. Tsuji, M. Kakinuma, K. Sakaguchi, and C. Tanaka, "A hierarchical placement procedure with a simple blocking scheme," in Proc. 16th Design Automation Conf., June 1979, pp. 18-23.

[76] C. Ng, S. Ashtaputre, E. Chambers, K. Do., S. Hui, R. Mody, and D. Wong, "A hierarchical floor-planning, placement, and routing tool for sea-of-gates design," in Proc. Custom Integrated Circuit Conf., May 1989, paper no. 3.3.

[77] T.-K. Ng, J. Oldfield, and V. Pitchumani, "Improvements of a min-cut partition algorithm," in Proc. IEEE Int. Conf. on Computer-Aided Design, Nov. 1987, pp. 470-473.

[78] T. Payne, R. Wells, and W. Gundel, "A study of automatic placement strategies for very large gate array designs," in Proc. IEEE Int. Conf. on Computer-Aided Design, Nov. 1987, pp. 194-197.

[79] C. Sechen and D. Chen, "An improved objective function for mincut partitioning," in Proc. IEEE Int. Conf. Computer-Aided Design, Nov. 1988, pp. 502-505.

[80] C. Sechen and A. Sangiovanni-Vincentelli, "Timber Wolf 3.2 : A new standard cell placement and global routing package," in Proc. 23rd Design Automation Conf., June 1986, pp. 432-439.

[81] M. Terai, "A method of improving the terminal assignment in the channel

routing for gate arrays," IEEE Trans. Computer-Aided Design, vol. 4, July 1989, pp. 329-336.

[82] K. Takahashi, K. Nakajima, M. Terai, and K. Sato, "Min-cut placement with global objective functions for large scale sea-of-gates arrays," IEEE Trans. on Computer-Aided Design of Integrated Circuits and systems, vol. 14, No. 4, April 1995, pp. 434-446.

[83] M. Hanan, Sr.P.K. Wolff, and B.J. Agule, "Some experimental results on placement techniques," Design Automation Conf., 1976, pp. 214-224.

[84] D.A. Thomsae, T.A. Peterson, and D.E. Van den Bout, "The anyboard rapid prototyping environment," Proceedings of the Advanced Research in VLSI Conf., 1990, pp. 356-370.

[85] M. McMahon, "Accelerators for faster logic simulation: The zycad approach," in Proc. 4th Int. IEEE VLSI Multilevel Interconnection Conf., 1987, pp. 981.

[86] H. Wolff,"How quickturn is filling a gap," in Electronics, April 1990, pp. 70.

[87] R. Cok, Parallel Programs for the Transputer, Prentice-Hall Inc., 1991.

[88] D.P. Bertsekas, and J.N. Tsitsiklis, Parallel and Distributed Computation: Numerical Methods, Prentice-Hall Inc., 1989.

[89] Y.C. Chen, W.T. Chen, and J.P. Sheu, "Designing efficient parallel algorithms on mesh-connected computers with multiple broadcasting," IEEE Trans, 1990, PDS-1, pp. 241-245.

[90] R.A. Duncan, "A survey of parallel computer architectures," IEEE Comput., 1990, 23, pp. 5-16.

[91] H. Li, and Q.F. Stout, Reconfigurable Massively Parallel Computers, Prentice-Hall Inc., 1991.

[92] R. Beresford, "An emulator for CMOS ASICs," VLSI System Design, May 1987, pp. 8.

[93] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, Network Flows: Theory, Algorithms, and applications. Englewood Cliffs, NJ: Prentice Hall, 1993.

[94] A.W. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," J. ACM, vol. 35, no. 4, Oct. 1988, pp. 921-940.

[95] C.K. Cheng, "The optimal circuit decompositions using network flow formulations," in Proc. IEEE Int. Symp. Circuits Syst., May 1990, pp. 2650-2653.

[96] R.E. Gomory and T.C. Hu, "Multi-terminal network flows," J. Soc. Indust. Appl. Math., vol. 9, 1961, pp. 551-570.

[97] T.C. Hu and K. Moerder, "Multiterminal flows in a hypergraph," in VLSI Circuit Layout: Theory and Design, T.C. Hu and E.S. Kuh, Ed. New York: IEEE, 1985.

[98] D.W. Matula and F. Shahrokhi, "The maximum concurrent flow problem and sparsest cuts," Southern Methodist Univ., Dallas, TX, Tech. Rep., 1986.

[99] M.R. Garey, D.S. Johnson, and L. Stockmeyer, Some Simplified NP-complete Graph Problems, Theoretical Computer Science, 1976, pp. 237-267.

[100]Y.C. Wei and C.K. Cheng, "Multiple-Level Partitioning: An application to the very large-scale hardware simulator," IEEE Journal of Solid-State Circuits, vol. 26, no. 5, May 1991, pp. 706-716.

[101]L.T. Liu, M.T. Kuo, C.K. Cheng, and T.C. Hu, "A replication cut for two-way Partitioning," IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, vol. 14, no. 5, May 1995, pp. 623-630.

Appendix A

The InCA CIF Netlist Format

In this appendix the InCA CIF netlist format is introduced. In order to form a netlist of a circuit, two kinds of netlist format are used, which are an "explicit" and "implicit" netlist format.

An explicit netlist format connects nets to ports of a cell by explicitly naming each port. EDIF 200 is an example of an explicit netlist format, e.g.

```
(net netname (joined (portRef A (instanceRef U100))
                     (portRef A (instanceRef U200))))
```

An implicit netlist has each net connected to a port of a cell by position in the cell port list. The InCA CIF which is an example of an implicit format, e.g.

INV(cell type) U300(instance) inputNet outputNet(associated nets)

Pin order

To write out an implicit netlist, the 'order' of the ports for each cell must be known. This information is usually obtained from an ASIC vendor's library databook and encoded into an ASCII file. The format of this file could be as follows :

```
line 1: Vendor_Pin_Order
line 2:
line 3: AND2 A B, Z ;
line 4: AND3 A B C , Z;
  :
  :
```

note1: the header 'Vendor_Pin_Order' is required.

note2: one entry per cell in the library, delimited by a semi-colon.

note3: each cell name and each port name per cell must be unique.

note4: cell type and ports are delimited by whitespace.

note5: input ports are followed by outports, delimited by a comma.

note6: bidirectional ports denoted as output ports.

note7: if a cell has an empty input (or output) list, then that list is denoted by a whitespace; the comma and semi-colon are still required.

The InCA CIF netlist format

An example of the CIF netlist format is as follows:

```
line1 : netnumber 0 ;
line2 : entry TOPLEVELCELL ;
line3 : library lib/libraryName ;
line4 : end
line5 :
line6 : part TOPLEVELCELL ;
line7 : Vendor_Independant_Format
line8 : inputs INPUT1 INPUT2 INPUT3 ;
line9 : outputs OUTPUT1 OUTPUT2 ;
line10: DFF U100 INPUT1 INPUT2 -- OUTPUT1 ;
line11: sheet BOTTOMLEVELCELL U200 INPUT3 OUTPUT2 ;
line12: primary INPUT1 INPUT2 ;
line13: primary INPUT2 INPUT2 ;
line14: primary INPUT3 INPUT3 ;
line15: primary OUTPUT1 OUTPUT1 ;
line16: primary OUTPUT2 OUTPUT2 ;
line17: end
line18:
line19:  part BOTTOMLEVELCELL ;
line20:  Vendor_Independant_Format
line21:  inputs INPUT1 ;
line22:  outputs OUTPUT1 ;
line23:  INV U100 INPUT1 OUTPUT1 ;
line24: primary INPUT1 INPUT1 ;
line25: primary OUTPUT1 OUTPUT1 ;
line26: end
```

The example given is a top level cell (TOPLEVELCELL) which contains two instances: U100 (library cell DFF) and U200 (hierarchical cell BOTTOMLEVELCELL), and the definition of the hierarchical cell. The following notes are valid for Concept Silicon v.1.1 :

note1: the keyword 'netnumber' is always set to 0 (zero).

note2: the keyword 'entry' denotes the name of the top level instance.

note3: the keyword 'library' denotes the name of the 'VPO' file used to generate this CIF (the 'lib' sub-directory must always be included).

note4: definitions of hierarchical cells are denoted by the keyword 'part' and instances of hierarchical cells are denoted by the keyword

note5: the header 'Vendor_Independant_Format' is required per cell.

note6: the keyword 'primary' is a repeated list of each input and output, one per line.

note7: the keyword 'end' (and the header 'Vendor_Independant_Format') are not delimited by a punctuation mark.

note8: the keyword '--' is used to denote a no-connection, eg. in the example, if the DFF entry in the VPO file was: DFF D CLK , Q QB ; then the Q output port is not connected to a net.

Appendix B

Random Circuit Generator

To obtain random circuits, routines are required capable of generating random number sequences. There are many methods available to generate random numbers. The best-known method which has been used extensively since it was introduced by Lehmer in 1951, is the so-called linear congruential method [14].

Different seeds will generate different random sequences. The range can be used to define the number of basic gates or the the number of possible nets in the circuit. A block diagram of this random number generator is shown in Figure B.1.

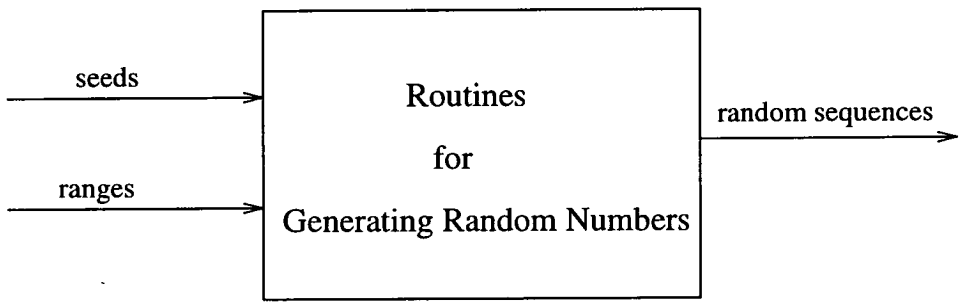


Figure B.1 The block diagram of Random Number Generator

Digital logic gates are used as basic cells in the random circuit to make it more realistic. The logic components which were used are as follows;

(1) NOT

(2) AND2 ---- with two inputs

(3) NAND3 ---- with three inputs

(4) OR2 ---- with two inputs

(5) NOR3 ---- with three inputs

(6) BDFF ---- D-type flip-flop

(7) BDFFSC ---- D-type flip-flop with "set" and "reset"

If more logic components are needed, their associated pin lay-outs are entered into the library which accommodates the logic components. Obviously the parameter for the number of basic gates has to be changed to the correct one.

Assumptions used to simplify the random circuit generation are as follows;

(1) Every pin on a basic cell must be connected to other cells, i.e. no empty pin left, except the pins of the components used as system inputs or outputs.

(2) No output pins connected together.

(3) The maximum number of fan-out is 10. This means the maximum number of basic cells on a net is 11. This number 10 is derived from the fact that the fan-out of a CMOS gate is typically 10.

The parameters which have to be input to the random circuit generator are seed1, seed2, the number of cells required in a circuit, the number of system inputs and the number of system outputs. The seed1 is used to generate the random part-type sequence. The seed2 is used to generate the random nets sequence. Figure B.2 shows the block diagram of a random circuit generator.

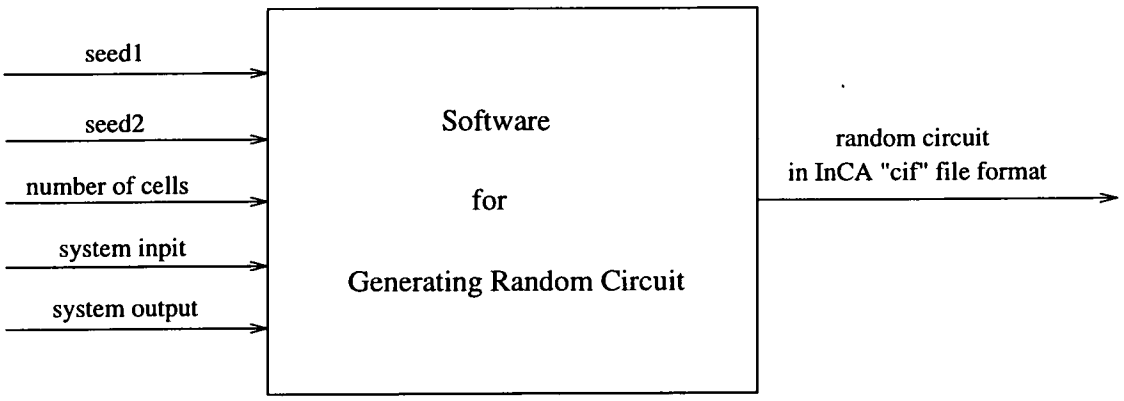


Figure B.2 The block diagram of Random Circuit Generator

The output is a text form describing the circuit in InCA "cif" file format. An example of random circuit which has been generated with seed1= 1, seed2= 2, number of cell= 10, system input= 5, system out= 5 is shown in the next page.

```
netnumber 0 ;
entry DESIGNS ;
library
lib/solo ;
end
part DESIGNSX0 ;
Vendor_Independant_Format
inputs  N0 N1 N2 N3 N4 ;
outputs N11 N12 N13 N14 N15 ;
         NAND3 cell_0 N7 N8 N9 N5 ;
         NAND3 cell_1 N10 N3 N4 N6 ;
         NOR3 cell_2 N11 N12 N13 N7 ;
         AND2 cell_3 N14 N15 N8 ;
         BDFFSCT cell_4 N5 N2 N8 N0 N9 N10 ;
         NOR3 cell_5 N1 N4 N2 N11 ;
         NOT cell_6 N3 N12 ;
         NOT cell_7 N4 N13 ;
         NAND3 cell_8 N1 N13 N5 N14 ;
         OR2 cell_9 N6 N0 N15 ;
primary N0 N0 ;
primary N1 N1 ;
primary N2 N2 ;
primary N3 N3 ;
primary N4 N4 ;
primary N11 N11 ;
primary N12 N12 ;
primary N13 N13 ;
primary N14 N14 ;
primary N15 N15 ;
end
part DESIGNS ;
Vendor_Independant_Format
inputs  N0 N1 N2 N3 N4 ;
outputs N11 N12 N13 N14 N15 ;
sheet DESIGNSX0 cell_10 N0 N1 N2 N3 N4 N11 N12 N13 N14 N15 ;
primary N0 N0 ;
primary N1 N1 ;
primary N2 N2 ;
primary N3 N3 ;
primary N4 N4 ;
primary N11 N11 ;
primary N12 N12 ;
primary N13 N13 ;
primary N14 N14 ;
primary N15 N15 ;
end
```

The following is the C programming code for the random circuit generator which consists of a header file and a main program which contains four other functions: namely "ranx()" which generates random numbers, "io()" which returns the number of input and output pins on a component, "no_output_pins()" which returns the total number of output pins, and "cifout()" which creates the netlist in InCA CIF netlist format. The header file and program code are as follows:

```
/*
   The Header File
*/
#define NUM_BASIC_PART 7
#define FOR_NO_CELL 50000
#define FOR_TOTAL_PIN 50000
#define MAX_PIN_CELL 6     /* BDFFSCT has  6 pins  */
#define FOR_SYSIN 500
#define FOR_SYSOUT 500
#define FOR_INTERNAL_OUT 2000
#define TOTAL_NET 50000
#define MAX_BIND_NUM 11
```

```
/*
    The Main Program
*/
#include <stdio.h>
#include "cnst.h"

int nocell, si, so ; /* si: no. of system input
                        so: no. of system output */
float p ;

main()
{
  int part_type[FOR_NO_CELL] ;
  int ran_net[FOR_TOTAL_PIN] ;
  int bind[TOTAL_NET] ;
  int M[FOR_NO_CELL][MAX_PIN_CELL] ;
  int single[FOR_NO_CELL] ;
  int num_single = 0 ;
  int flag ;
  int tol_nets ;
  int son, sof ; /* son: system out only, sof: system out with feed */
  int ip, op, i, j, k, m, temp ;
  int cntr = 0 ;
  int net_no ;
```

```
int part = 0 ;
int out_net ;
int sel = 0, b = 0 ;
int tol_inputs ;

readin() ;
ranx( part_type, NUM_BASIC_PART, nocell ) ;
dsp_array( part_type, nocell ) ;
tol_nets = si + no_output_pins( part_type, nocell ) ;
printf("tol_nets=%d\n", tol_nets ) ;
stop();
if( tol_nets > TOTAL_NET )
  printf("TOTAL_NET is too small\n") ;
printf("-I- Generating random circuit array\n") ;

out_net = si ; /* initialise the first output net */
son = so * p ;
sof = so - son ;
tol_inputs = (tol_nets - son)/2 ;

ranx( ran_net, tol_nets - son, FOR_TOTAL_PIN ) ;

for( i = 0 ; i < tol_nets ; i++ )
  bind[i] = 0 ;

for( i = 0 ; i < nocell ; i++ ){
  printf("row %d\n", i ) ;
  io( part_type[i], &ip, &op );
  for( j = 0  ; j < MAX_PIN_CELL ; j++ ){
    if( j < ip ){
    if( sel % 2 != 0 ){ /* sel is odd */
      for(;;){
        if( bind[b] < MAX_BIND_NUM ){
          M[i][j] = b ;
          sel++ ;
          b++ ;
          bind[b]++ ;
          break ;
          } /* if( bind[b] < MAX_BIND_NUM ) */
          b++ ;
        if( b == tol_inputs )
          b = 0 ;
        } /* for(;;) */
      } /* if( sel % 2 != 0 ) */
    else{
        net_no = ran_net[ cntr++ ] ;
        for(;;){
        if( net_no < tol_inputs ){
            if( bind[net_no] < MAX_BIND_NUM ){
              M[i][j] = net_no ;
            bind[net_no]++ ;
            break ;
            }
          else{
```

```
      for(;;){
        if( bind[b] < MAX_BIND_NUM )
          break ;
            b++ ;
        if( b == tol_inputs )
          b = 0 ;
          }
          M[i][j] = b ;
        bind[b]++ ;
          b++ ;
        if( b == tol_inputs )
          b = 0 ;
        break ;
        }
      } /* if( net_no < tol_inputs ) */
      for(;;){
        if( bind[b] < MAX_BIND_NUM )
          break ;
            b++ ;
        if( b == tol_inputs )
          b = 0 ;
        }
          M[i][j] = b ;
        bind[b]++ ;
          b++ ;
        if( b == tol_inputs )
          b = 0 ;
        break ;
    } /* for(;;) */
  } /* */
} /* if( j < ip ) */
else if( j >= ip && j < ip+op ){
     net_no = out_net++ ;
     M[i][j] = net_no ;
     bind[net_no]++ ;
/* TO prevent looping */
     for( k = 0 ; k < ip ; k++ ){
       if( M[i][k] == M[i][j] ){
       bind[M[i][j]]-- ;
       for(;;){
         net_no = ran_net[ cntr++ ] ;
         if( net_no != M[i][j] ){
           M[i][k] = net_no ;
           bind[net_no]++ ;
           break ;
             }
           } /* for(;;) */
         } /* if( M[i][k] == M[i][j] ) */
       } /* for( k = 0 ; k < ip ; k++ ) */
     } /* if( j >= ip && j < ip+op ) */
  else{
    M[i][j] = -1 ;
  }
} /* 2nd for */
```

```
  } /* 1st for */

  for( i = si+son ; i < tol_nets ; i++ )
    if( bind[i] == 1 )
      single[num_single++] = i ;

    k = 0 ;
    for( i = 0 ; i < nocell ; i++ ){
      io( part_type[i], &ip, &op ) ;
      for( j = 0 ; j < ip ; j++ ){
      if( bind[M[i][j]] > 2 ){
        bind[M[i][j]]-- ;
        M[i][j] = single[k++] ;
        bind[M[i][j]]++ ;
        if( k == num_single ){
          flag = 1 ;
          break ;
          }
        }
      }
      if( flag == 1 )
      break ;
    }

/* ***************************************************************************
        To prevent pins on the same component have the same net
*************************************************************************** */
  for( i = 0 ; i < nocell ; i++ ){
   for( j = 0 ; j < MAX_PIN_CELL ; j++ ){
    temp = M[i][j] ;
    for( k = 0 ; k < MAX_PIN_CELL ; k++ ){
      if( M[i][k] == -1 )
        break ;
      if( k == j )
        continue ;
      if( temp == M[i][k] ){
        for(;;){
          net_no = ran_net[cntr++] ;
          if( bind[net_no] < MAX_BIND_NUM ){
            for( m = 0 ; m < MAX_PIN_CELL ; m++ ){
              if(  M[i][m] == net_no )
                break ;
            }
            if( m == MAX_PIN_CELL ){
              bind[M[i][k]]-- ;
              M[i][k] = net_no ;
              bind[net_no]++ ;
              break ;
            }
          }
        } /* for(;;) */
      } /* if( temp == M[i][k] ) */
    } /* k */
   } /* j */
```

```
 } /* i */
  cifout( M, part_type, bind, si, tol_nets, nocell, so, son ) ;
}


readin()
{
  printf("How many cells in a circuit:") ;
  scanf("%d",&nocell) ;
  printf("system input:") ;
  scanf("%d", &si ) ;
  printf("system output:") ;
  scanf("%d", &so) ;
  printf("percentage of output_only( 0.0 -> 1.0 ):") ;
  scanf("%f", &p ) ;
}
```

```
/*
    This is the function to generate random numbers
*/
#include <stdio.h>
#define MM 100000000
#define M1 10000
#define B 31415821
#define MAXTERM 1000
#define MAXRAN_NUM 5000

static int a ;
extern int maxterm ;
int N[MAXRAN_NUM] ;
int n ;

void ranx( M , r, nocell )

int  r, nocell ;
int  M[] ;

{
  int i, j ;
  int N[MAXRAN_NUM] ;

  printf("input seed a:") ;
  scanf("%d", &a ) ;

  for ( i = 0 ; i < nocell ; i++ )
      M[i] = random( r ) ;
}

int random( r )
```

```
int r ;

    {
      a = ( mult( a, B  ) + 1 ) % MM ;
      return (( a/M1 ) * r ) / M1 ;
    }                          /* To obtain different range of random */
                                      /* numbers, change "10" which means the */


int mult ( p, q )
    int p, q ;
    {
      int p1, p0, q1, q0 ;

      p1 = p/M1 ;
      p0 = p%M1 ;
      q1 = q/M1 ;
      q0 = q%M1 ;

      return ((( p0*q1+p1*q0) % M1 )* M1 + p0*q0 ) % MM ;
    }




/*
    This function is used to return the number of input and output pins
*/
io( type, nin, nout )
  int type ;
  int *nin, *nout ;
{
  switch( type ){
    case 0 : *nin = 1 ;
             *nout = 1 ; break ;
             /* NOT */
    case 1 : *nin = 2 ;
             *nout = 1 ; break ;
             /* AND2 */
    case 2 : *nin = 3 ;
             *nout = 1 ; break ;
             /* NAND */
    case 3 : *nin = 2 ;
             *nout = 1 ; break ;
             /* OR2 */
    case 4 : *nin = 3 ;
             *nout = 1 ; break ;
             /* NOR3 */
    case 5 : *nin = 2 ;
             *nout = 2 ; break ;
               /* BDFF */
    case 6 : *nin = 4 ;
```

```
            *nout = 2 ; break ;
            /* BDFFSCT */
  }
}
```

```
/*
    This function is used to calculate the total number of output pins
*/
no_output_pins( part_type, size )
  int part_type[] ;
  int size ;
{
  int nin, nout ;
  int tol, i ;

  tol = 0 ;
  for( i = 0 ; i < size ; i++ ){
    io( part_type[i], &nin, &nout ) ;
    tol = tol + nout ;
  }
  return( tol ) ;
}
```

```
/*
    This function is used to create the netlist in InCA "CIF" format
*/
#include <stdio.h>
#include "cnst.h"
#define MAX_LENGTH 30
char buf[MAX_LENGTH] ;

cifout( M, part_type, bind, si, tol_nets, nocell, so, son )
  int M[][MAX_PIN_CELL] ;
  int part_type[] ;
  int bind[] ;
  int si, so, son, nocell, tol_nets ;
{
  FILE *fp ;
  int i, j ;

  fp = fopen("designs","w") ;
  fprintf( fp,"netnumber 0 ;\n" ) ;
  fprintf( fp,"entry DESIGNS ;\n" ) ;
  fprintf( fp,"library\n" ) ;
```

```
fprintf( fp,"lib/solo ;\n" ) ;
fprintf( fp,"end\n" ) ;
fprintf( fp,"part DESIGNSX0 ;\n" ) ;
fprintf( fp,"Vendor_Independant_Format\n" ) ;
fprintf( fp,"inputs  " ) ;

for( i = 0 ; i < si ; i++ ){
  fprintf( fp,"NET%d ", i ) ;
}
fprintf( fp,";\n" ) ;

fprintf( fp,"outputs " ) ;
for( i = tol_nets-so ; i < tol_nets ; i++ )
  fprintf( fp, "NET%d ", i ) ;
for( i = si ; i < tol_nets ; i++ ){
  if( i >= si+son && i < si+so )
    continue ;
  if( bind[i] == 1)
    fprintf( fp,"NET%d ", i ) ;
}
fprintf( fp,";\n" ) ;

for( i = 0 ; i < nocell ; i++ ){
  part_namef( part_type[i], buf ) ;
  fprintf( fp,"         %s cell_%d ", buf, i ) ;
  for( j = 0 ; j < MAX_PIN_CELL ; j++ ){
    if( M[i][j] == -1 )
    break ;
    fprintf( fp,"NET%d ", M[i][j] ) ;
  }
  fprintf( fp,";\n" ) ;
}


for( i = 0 ; i < si ; i++ ){
  fprintf( fp,"primary NET%d NET%d ;\n", i, i ) ;
}
for( i = tol_nets-so ; i < tol_nets ; i++ )
  fprintf( fp,"primary NET%d NET%d ;\n", i, i ) ;
for( i = si ; i < tol_nets ; i++ ){
  if( i >= si+son && i < si+so )
    continue ;
  if( bind[i] == 1)
  fprintf( fp,"primary NET%d NET%d ;\n", i, i ) ;
}
fprintf( fp,"end\n" ) ;
fprintf( fp,"part DESIGNS ;\n" ) ;
 fprintf( fp,"Vendor_Independant_Format\n" ) ;
fprintf( fp,"inputs  " ) ;

for( i = 0 ; i < si ; i++ ){
  fprintf( fp,"NET%d ", i ) ;
}
fprintf( fp,";\n" ) ;
```

```
fprintf( fp,"outputs " ) ;
for( i = tol_nets-so ; i < tol_nets ; i++ )
    fprintf( fp, "NET%d ", i ) ;
  fprintf( fp,";\n" ) ;

  fprintf( fp,"sheet DESIGNSX0 cell_%d ", nocell ) ;
  for( i = 0 ; i < si ; i++ ){
    fprintf( fp,"NET%d ", i ) ;
  }
   for( i = tol_nets-so ; i < tol_nets ; i++ ){
     fprintf( fp,"NET%d ", i ) ;
   }
   fprintf( fp,";\n" ) ;

for( i = 0 ; i < si ; i++ ){
    fprintf( fp,"primary NET%d NET%d ;\n", i, i ) ;
  }
  for( i = tol_nets-so ; i < tol_nets ; i++ )
    fprintf( fp,"primary NET%d NET%d ;\n", i, i ) ;
  fprintf( fp,"end\n" ) ;

  fclose( fp ) ;
}



part_namef( type, buf )
  char buf[] ;
  int type ;
{
  switch( type ){

      case 0 : strcpy( buf,"NOT") ; break ;

      case 1 : strcpy( buf,"AND2") ; break ;

      case 2 : strcpy( buf,"NAND3") ; break ;

      case 3 : strcpy( buf,"OR2") ; break ;

      case 4 : strcpy( buf,"NOR3") ; break ;

      case 5 : strcpy( buf,"BDFF") ; break ;

      case 6 : strcpy( buf,"BDFFSCT") ; break ;

  }
}
```

Appendix C

Data Preparation

C.1 A simple one-level netlist

There are many ways to describe a circuit, a schematic circuit diagram is a graphical method for depicting the connections among cells, but it is difficult for a computer to read it. A text file which describes the connections among cells in a circuit is needed to let the computer scan through and acquire the necessary information associated with this circuit according to which the remaining analysis can be proceeded to achieve a successful design. Usually a text file representing a circuit is called a netlist file. The netlist format used is InCA CIF netlist format which is described in Appendix A.

A simple example circuit is a two to four line decoder the schematic of which is shown in Figure C.1. The inputs are two address lines, in0 and in1, and an active low enable. The decoded outputs, again active low, are the lines out0 to out3. The InCA CIF netlist of this decoder is shown in Listing C.1 and gives all the information. The schematic shows, such as, what are the inputs and outputs, what components are used and how they are connected. It is presented to illustrate the InCA CIF netlist format which is used as the netlist format of the test circuits in the "Merge Algorithm".

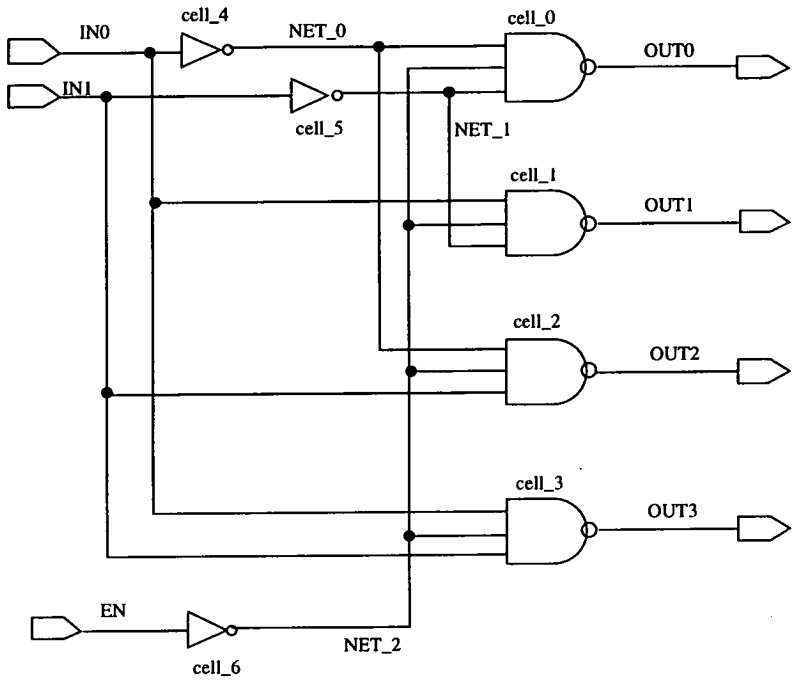


Figure C.1 Two to four line decoder schematic.

```
netnumber 0 ;
entry DEC2TO4 ;
library
lib/solo ;
end
part DEC2TO4 ;
Vendor_Independant_Format
inputs  IN1 IN0 EN ;
outputs OUT3 OUT2 OUT1 OUT0 ;
   NAND3 CELL_0 NET_0 NET_2 NET_1 OUT0 ;
   NAND3 CELL_1 IN0 NET_2 NET_1 OUT1 ;
   NAND3 CELL_2 NET_0 NET_2 IN1 OUT2 ;
   NAND3 CELL_3 IN0 NET_2 IN1 OUT3 ;
   NOT CELL_4 IN0 NET_0 ;
   NOT CELL_5 IN1 NET_1 ;
   NOT CELL_6 EN NET_2 ;
primary IN1 IN1 ;
primary IN0 IN0 ;
primary EN EN ;
primary OUT3 OUT3 ;
primary OUT2 OUT2 ;
primary OUT1 OUT1 ;
primary OUT0 OUT0 ;
end
```

Listing C.1 The netlist of two to four decoder in InCA CIF format.

C.2 Hierarchical structural netlist

It would be difficult to create a large complex design as a one-level network full of basic cells. A hierarchical structure design is usually used to describe a large system. At the top level of the hierarchy, the whole design can be viewed as a single functional block. The next lower level would decompose the single block into several functional blocks each of which can be further decomposed into more functional blocks. The functional blocks in the lower level would specify more detailed operation than the higher level. At the very lowest level of hierarchy, the functional blocks are decomposed into basic cells. As a system is built up as a hierarchy in this way, it is easy to inspect its operation at any level required, from the top-level overall function down to the working of basic cells.

A simple and trivial design hierarchy is illustrated in Figure C.2. This design (named DSGN_SAMPLE) consists of two identical parts (called SPEC_UNIT) which are made up of basic gates such as NAND and INVERTER, and shown in Figure C.3. In this hierarchical design, there are three levels, at the top level the whole design is a single functional block with four inputs and two outputs the schematic diagram of which is shown in Figure C.4. At the level below that, this single block is split into two functional blocks and the schematic diagram for this level is shown in Figure C.5. At the lowest level, the functional blocks of SPEC_UNIT are further decomposed into basic logic gates.

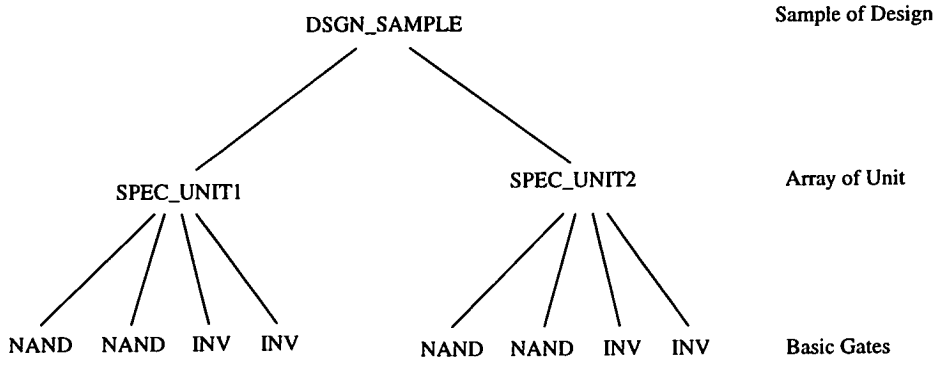


Figure C.2 The hierarchy for a design

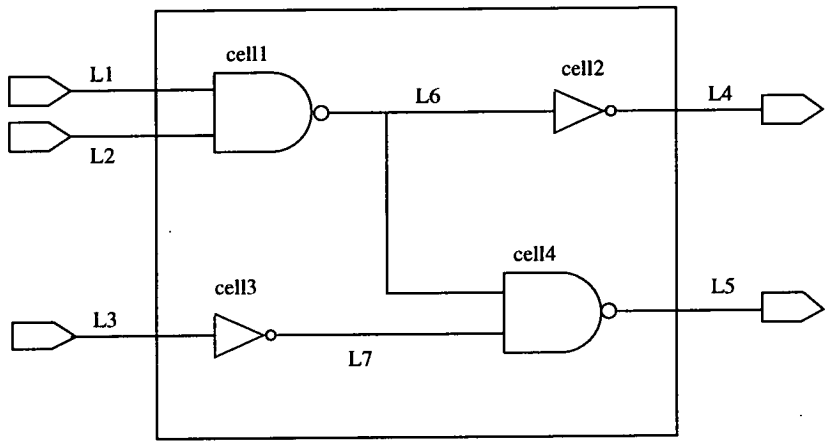


Figure C.3 The part of SPEC_UNIT

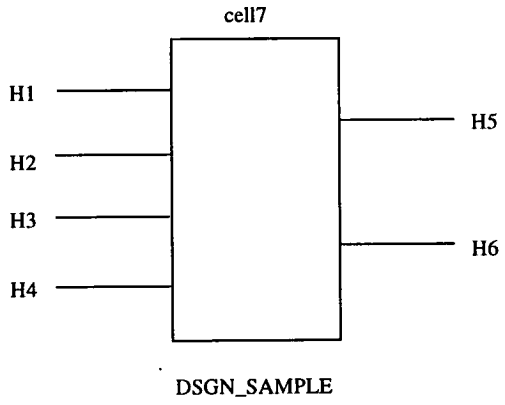


Figure C.4 The design sample

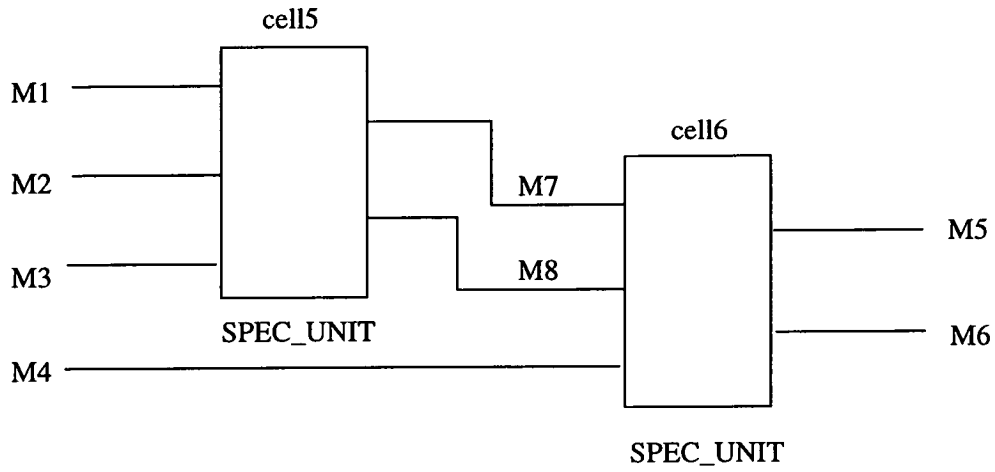


Figure C.5 The schematic diagram for the array of SPEC_UNITs

C.3 Flattening a system

To implement a design, the hierarchy has to be flattened into a series of basic cells. When flattening a system, it is needed to start from the top most level (i.e. system level) and the following procedures are repeated.

(1) Substitute the input and output signal names of the next lower level units with the corresponding unit signal names in the current level, and make sure every different net of units with the same type has a unique net name in the next lower level.

(2) Repeat step(1), until lowest level is reached.

The hierarchical CIF netlist of Figure C.2 is shown in the Listing C.2. This netlist is flattened to a series of basic logic gates according to the above flattening procedures. The Listing C.3 and C.4 show the intermediate stage and final flattened netlist whose schematic diagram consisting of basic gates is shown in Figure C.6.

```
part SPEC_UNIT1
inputs  L1 L2 L3 ;
outputs L4 L5 ;
        NAND CELL1 L1 L2 L6 ;
      INV CELL2 L6 L4 ;
      INV CELL3 L3 L7 ;
      NAND  CELL4 L6 L7 L5 ;
end
part SPEC_UNIT2
inputs  L1 L2 L3 ;
outputs L4 L5 ;
      NAND CELL1 L1 L2 L6 ;
      INV CELL2 L6 L4 ;
      INV CELL3 L3 L7 ;
      NAND  CELL4 L6 L7 L5 ;
end
part UNIT_ARRAY
inputs  M1 M2 M3 M4 ;
outputs M5 M6 ;
      SPEC_UNIT1 CELL5 M1 M2 M3 M7 M8 ;
      SPEC_UNIT2 CELL6 M7 M8 M4 M5 M6 ;
end
part DSGN_SAMPLE
inputs  H1 H2 H3 H4 ;
outputs H5 H6 ;
      DSGN_SAMPLE CELL7 H1 H2 H3 H4 H5 H6 ;
end
```

Listing C.2 The hierarchical netlist of DSGN_SAMPLE

```
part SPEC_UNIT1
inputs  L1 L2 L3 ;
outputs L4 L5 ;
        NAND CELL1 L1 L2 L6 ;
        INV CELL2 L6 L4 ;
        INV CELL3 L3 L7 ;
        NAND  CELL4 L6 L7 L5 ;
end
part SPEC_UNIT2
inputs  L1 L2 L3 ;
outputs L4 L5 ;
        NAND CELL1 L1 L2 L6 ;
        INV CELL2 L6 L4 ;
        INV CELL3 L3 L7 ;
        NAND  CELL4 L6 L7 L5 ;
end
part UNIT_ARRAY
inputs  H1 H2 H3 H4 ;
outputs H5 H6 ;
        SPEC_UNIT CELL5 H1 H2 H3 M7X1 M8X1 ;
        SPEC_UNIT CELL6 M7X1 M8X1 H4 H5 H6 ;
end
```

Listing C.3 The intermediate stage flattening from the top to the next lower level

```
part DSGN_SAMPLE
inputs  H1 H2 H3 H4 ;
outputs H5 H6 ;
        NAND CELL1 H1 H2 L6X1 ;
        INV CELL2 L6X1 M7X1 ;
        INV CELL3 H3 L7X1 ;
        NAND  CELL4 L6X1 L7X1 M8X1 ;
        NAND CELL5 M7X1 M8X1 L6X2 ;
        INV CELL6 L6X2 H5 ;
        INV CELL7 H4 L7X2 ;
        NAND  CELL8 L6X2 L7X2 H6 ;
```

Listing C.4 The final flattened netlist of DSGN_SAMPLE

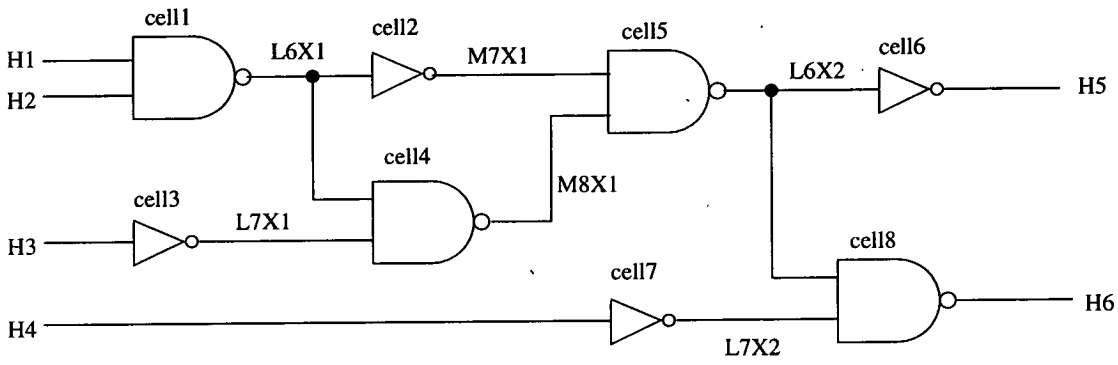


Figure C.6 The flattened design consisting of basic gates

C.4 A parser for CIF format netlist

There are two procedures to deal with a netlist file which describes a real circuit. They are "processing the characters" and "building the results into a required data structure". The former is commonly called syntax analysis and the latter semantic analysis. A parser is the tool that processes characters in an input stream and emits function calls to a module that implements the semantic analysis and translates it to a form or a data structure suitable for further processing. The data structures needed

to be built will be discussed in the next section.

C.5 The data structure for implementing the merge algorithm

A cell list is required which is generated from the original circuit "CIF" file. This list describes how the components in the circuit are interconnected. The cell list consists of a head of cell list which points to where the circuit is, the cell nodes which represent the cells themselves and point to a list that shows what nets are incident to the cells, and the net nodes which contain information related to the nets and the cells. The exact structure and declaration in C language notation are shown below. Cells and nets are numbered sequentially and individually.

```
struct net
{
  int  name ;           /* net name                              */
  struct net  *next ;   /* a pointer points to the next net      */
} ;
typedef struct net net_node ;


struct cell
{
  int  name ;           /* cell name                             */
  netnode  *link ;      /* a pointer points to a list of nets
                           which is incident to this cell       */
  struct cell  *next ;  /* a pointer to link the next cell       */
} ;
typedef struct cell cell_node ;


struct cell_list        /* the head of the cell_list             */
{
  int  number ;         /* the number of cells in the circuit    */
  cell_node  *first ;   /* points to the first cell of the
                                                 list of cells    */
  cell_node  *last ;    /* points to the last cell of the
                                                 list of cells    */
} ;
typedef struct cell_list head_cell_list ;
```

A net list is acquired from cell list. Basically it contains the same information as the

cell list, the only difference is the nets are arranged in the vertical and the cells which are connected to the net are arranged in the horizontal list corresponding to it. Its main purpose is to facilitate generating another list called cell-net list.

A cell-net list is created from both cell list and net list. The data structure and the declaration are shown below.

```
struct band-net
{
  int  name ;              /* the band-net name              */
  int  inside ;            /* the number of the cells inside  */
  int  outside ;           /* the number of the cells outside */
  struct band_net  *next ; /* points to the next band-net     */
} ;
typedef struct band-net  band_net ;


struct band-cell
{
  int  name ;               /* the band-cell name             */
  band_net  *link ;         /* points to the list of band-net */
  band_net  *inlink ;       /* points to a list of band-nets
                               which are completely inside
                               this band-cell, it initially
                                              points to null */
  struct band-cell  *next ; /* points to the next band-cell   */
} ;
typedef struct band-cell band_cell ;


struct band_list
{
  int  number ;             /* the number of the band_cell in
                                                    the list */
  band_cell  *first ;       /* points to the first band-cell  */
  band_cell  *last ;        /* points to the last band-cell   */
} ;
typedef struct band_list head_band_list ;
```

A graph structure is needed to describe the number of connections between cells. This graph is generated from the cell-net list by counting the number of the same nets related to any pair of cells. After establishing this graph, a merge sequence list which arranges the nodes with the greatest number of connections at the front of list

is set up by scanning through the whole graph. The data structure of the graph and merge sequence list and their declarations are shown below.

```
struct edge
{
  int  vname ;          /* the current cell name              */
  int  ename ;          /* the cell name relating to the
                                                  current cell */
  int  cntn ;           /* the number of connections between
                              cell "vname" and cell "ename" */
  struct edge  *slink ; /* pointer to link the merging
                                                    sequence */
  struct edge  *next ;  /* points to the next cell relating
                                          to the current cell */
} ;
typedef struct  edge  edge_node ;


struct grp_node
{
  int  name ;              /* the cell name           */
  struct grp_node  *next ; /* points to the next cell */
} ;
typedef struct  grp_node  group_node ;


struct vertex
{
  int  name ;             /* the current cell name             */
  edge_node  *link ;      /* points to a list of cells which
                                    relate to the current cell */
  group_node  *glink ;    /* points to a list of cells which
                             are implicit cells of the current
                                                        cell */
  int  noc ;              /* the number of implicit cells in
                                        the current cell */
  struct vertex  *next ; /* points to the next vertex         */
} ;
typedef struct  vertex  ver_node ;


struct graph
{
  int  number ;           /* the number of vertices in the
                                                      graph */
  ver_node  *first ;      /* points to the first vertex       */
  ver_node  *last ;       /* points to the last vertex        */
} ;
typedef struct  graph  head_graph ;


struct big_node
```

```
{
  int  noc ;                /* the number of connections       */
  edge_node  *slink ;       /* points to a list of edge with
                                                the same "noc" */
  struct big_node  *next ; /* points to the next big_node      */
} ;
typedef struct  big_node  bignode ;


struct big
{
  int  number ;             /* the number of big_node          */
  bignode  *first ;         /* points to the first big_node    */
  bignode *last ;           /* points to the last big_node     */
} ;
typedef struct  big  head_big ;
```

C.6 Creating cell and net list

The circuit can be represented in two ways, one of which is through a list of cells for each net (i.e. net list), another of which is through a list of nets for each cell (i.e. cell list). The following input routine will deal with circuits described in "CIF" format to generate cell list and net list. The input of this routine is the circuit and the outputs are the cell list and net list.

```
/* the routine for creating cell and net list  */

clist_head() ; /* create the head of cell list */
nlist_head() ; /* create the head of net list  */
FOR each cell DO
    insert_vertex(clist,cell) ;
    FOR each net incident to the current cell DO
        insert_edge(clist,cell,net) ;
        IF the current net is a new net
           THEN insert_vertex(nlist, cell) ;
                insert_edge(nlist,net,cell) ;
        ELSE
           insert_edge(nlist,net,cell) ;
    END FOR
END FOR
```

A sample of functions written in C language for creating the head of a list, insert a vertex and insert an edge is shown below.

```
/*
 *  Creating a head of a list
 */
cellist *crt_head()
{
  cellist *new;

  if((new = (cellist *) malloc(sizeof(cellist)) != NULL)
    {
      new -> number = 0 ;
      new -> first  = new -> last = NULL ;
    }
  return ( new ) ;
}


/*
 *  Insert a vertex to a list
 */
int insert_vertex ( list, name )

  cellist *list ;
  int   name ; /* the name of a cell or a net */

{
  cellnode *new ;

  if ((new = (cellnode *) malloc(sizeof(cellnode))) == NULL)
    return ( NOMEM ) ;
  else
    {
      new -> name = name ;
      new -> link = NULL ;
      new -> next = NULL ;
      list -> number++ ;
      if ( list -> number == 1)
        list -> first = list -> last = new ;
      else
        {
           list -> last -> next = new ;
           list -> last = new ;
        }
      return ( NOERR );
    }
}


/*
 *  Insert an edge to a list
 */
 int insert_edge( list, vertex, edge )

 cellist  *list;
 int  vertex ;
```

```
int  edge ;

{
  cellnode *r_ver ;
  netnode *new ;

  r_ver = cell_exist ( list, vertex ) ;
  if ( r_ver == NULL )
    return (CELL_NEXIST) ;
  if((new = (netnode *) malloc(sizeof(netnode))) == NULL)
    return(NOMEM);

  new -> name = edge ;
  new -> next = NULL ;
  if ( r_ver -> link == NULL )
    r_ver -> link = new ;
  else
  {
    new -> next = r_ver -> link ;
    r_ver -> link = new;
  }
  return ( NOERR ) ;
}
```

C.7 The routine for creating cell-net list

The cell and net list representing the circuit are used to produce the cell-net list. The information contained in this list is the number of cells inside the current cell and the number of cells outside the current cell with respect to the net the cells are on. Initially, the cell can be viewed as if it is inside itself, so the number of cells inside the current cell for a certain net is one, the number of cells outside the current cell for the same net is one less than the total number of cells on this net. The following is the routine for generating cell-net list.

```
/* routine for generating cell-net list */

crt_head() ; /* create the head of cell-net list */
FOR each cell in cell list DO
    insert_bcell() ;
    FOR each net incident to the cell DO
        noc = num_of_cell(nlist, net) ;
        insert_band(head, cell, net, 1, noc) ;
    END FOR
END FOR
```

C.8 The routine for creating graph and merge sequence

The cell-net list above is used to generate the graph of the circuit which contains information about the number of connection between any pair of cells in the circuit. Cells are sequentially numbered and listed as vertices, each cell in the vertices list points to another list of cells whose cell number are greater than the cell number of each cell in the vertices list and related to it. A routine scans through the whole graph to set up the merge sequence by arranging the edge nodes according to the number of connection between cells in descending order. The procedures for generating graph and merge sequence are as follows:

/* routine for generating graph */

```
crt_graph() ;
FOR each band-cell "c1" in the band-cell list DO
insert_vertex() ;
    FOR each band-cell "c2" behind the band-cell "c1"
          in the outer loop in the band-cell list DO
        num = nofnet_betwbc(c1,c2) ;
        IF num ≠ 0 THEN
           insert_edge() ;
    END FOR
END FOR


/* routine for generating merge sequence */

crt_sequence() ;
FOR each edge node in the graph DO
    IF the number of connection is new THEN
       insert a node for the new connection number
       in descending order ;
       insert the edge node to the list pointed
       by the new node ;
    ELSE
       find the old node ;
       insert the edge node to the list pointed
       by the old node ;
END FOR
```

Appendix D

The C Programming Code for the Merge Algorithm

This appendix contains the C programming code for the merge-in-stages algorithm. There are three header files named "graph.h", "band.h" and "list.h" which contain the data structure declarations for graph, cell-net list and cell list and net list (cell and net list using the same data structure), respectively. These declarations can be found in Appendix C.

The following procedure is a function named merge_in_stages which executes the merge algorithm in four phases. It carries out the free merge operation in "L" stages under the stage size constraint in the first phase, selects the leading groups in phase 2, merges cells with leading groups in the phase 3 and finally merges the remaining cells. This function includes three functions which are "check_gr_num()", "merge_bc()" and "merge()".

```
#include <stdio.h>
#include "graph.h"
#include "band.h"
#include "list.h"
#define MAX_NO_SEED 10


void merge_in_stages( graf, bighead, blist )

graph *graf ;
big *bighead ;
bandlist *blist ;

{
    int i, j, k, cnt = 0 ;
    int s0, s1, l, L ;
```

```
vernode *vtemp ;
edgenode *etemp, *ept ;
bignode *pt ;
int seed_i, flag, flag1, flag2 ;
int seeds[MAX_NO_SEED] ;
int si_con[10] ;  /* si_con : size constraint */
int temp, temp1 ;
int nocell[MAX_NO_SEED] ;



for( i = 0 ; i < MAX_NO_SEED ; i++ )
  seeds[i] = -1 ;

seed_i = 0 ;

printf ("Input the number of group:") ;
scanf ("%d", &ng ) ;
printf(" Input L :") ;
scanf("%d", &L) ;
printf(" Input s0 :") ;
scanf("%d", &s0) ;
s1 = s0 ;

for( l = 0 ; l < L  ; l++ ){
  flag = 0 ;
  do{
   pt = bighead -> first ;
   if( pt == NULL ){
       printf("List of bighead is null\n") ;
       break ;    /* Execution of program should be stopped */
     }
     ept = pt -> slink ;
     if( ept != NULL )
     do{
            v1 = ept -> vname ;
            v2 = ept -> ename ;
            if( (check_gr_num( graf, v1 )
                 + check_gr_num( graf, v2 )) <= s1 ){
            merge_bc( blist, v1, v2 ) ;
            merge( graf, blist, bighead, v1, v2 ) ;
              break;
            }
            ept = ept -> slink ;
            if( ept == NULL ){
              pt = pt -> next ;
              if( pt == NULL )
                break;
              ept = pt -> slink ;
            }
        } while( ept != NULL ) ;
   } while( pt != NULL ) ;
  s1 = s1 * 2 ;
} /* for( l = 0 ; l < L ; l++ ) */
```

```
printf("finished level merging\n") ;


  i = 0 ;
  for(;;){
    pt = bighead -> first ;
    if( pt != NULL )
    do{
      ept = pt -> slink ;
      if( ept != NULL )
      do{
        flag = 0 ;
        v1 = ept -> vname ;
        v2 = ept -> ename ;
        if( (check_gr_num( graf, v1 )
             + check_gr_num( graf, v2 )) <= fsc ){
          for( j = 0 ; j < ng ; j++ ){
            if( seeds[j] == v1 || seeds[j] == v2 ){
              flag = 1 ;
              break ;
            }
          }
          if( flag == 0 ){
            merge_bc( blist, v1, v2 ) ;
          merge( graf, blist, bighead, v1, v2 ) ;
            seeds[seed_i++] = v1 ;
            flag = 2 ;
            i++ ;
            break ;
          }
        }
        ept = ept -> slink ;
      } while( ept != NULL ) ;
    if( flag == 2 )
      break ;
    pt = pt -> next ;
  } while( pt != NULL ) ;
  if( i == ng || pt == NULL )
    break ;
}

printf("leading groups have been selected\n") ;


 for(;;){
    pt = bighead -> first ;
    if( pt != NULL )
    do{
      ept = pt -> slink ;
      if( ept != NULL )
      do{
        flag = 0 ; flag1 = 0 ; flag2 = 0 ;
        v1 = ept -> vname ;
```

```
        v2 = ept -> ename ;
        for( j = 0 ; j < ng ; j++ ){
          if( v1 == seeds[j] )
            flag1 = 1 ;
          if( v2 == seeds[j] ){
            flag2 = 1 ;
            k = j ; /* take the index of seed for later use */
          }
        }
      if( ( flag1 + flag2 ) == 1 ){
        if( (check_gr_num( graf, v1 )
             + check_gr_num( graf, v2 )) <= fsc ){
            if( flag2 == 1 )
              seeds[k] = v1 ;
          merge_bc( blist, v1, v2 ) ;
          merge( graf, blist, bighead, v1, v2 ) ;
            flag++ ;
            break ;
        }
      }
        ept = ept -> slink ;
      } while( ept != NULL ) ;
      if( flag == 1 )
        break ;
      pt = pt -> next ;
    } while( pt != NULL ) ;
    if( flag != 1 )
      break ;
  }
printf(" finish merging with the leading groups\n") ;



for( i = 0 ; i < MAX_NO_SEED ; i++ )
  seeds[i] = -1 ;
for( i = 0 ; i < MAX_NO_SEED ; i++ )
  nocell[i] = 0 ;

if( graf -> number > ng ){
  vtemp = graf -> first ;
  do{
      if( vtemp -> noc > nocell[0] ){
        seeds[0] = vtemp -> name ;
        nocell[0] = vtemp -> noc ;
        for( j = 0 ; j < ng ; j++ ){
          for( k = 1 ; k < ng ; k++ ){
            if( seeds[j] > seeds[k] ){
              temp = seeds[k] ;      temp1 = nocell[k] ;
              seeds[k] = seeds[j] ;  nocell[k] = nocell[j] ;
              seeds[j] = temp ;      nocell[j] = temp1 ;
            }
          }
        }
      }
```

```
      vtemp = vtemp -> next ;
   } while( vtemp != NULL ) ;
}

 for(;;){
    if( graf -> number <= ng )
      break ;
    pt = bighead -> first ;
    if( pt != NULL )
    do{
      ept = pt -> slink ;
      if( ept != NULL )
      do{
        flag = 0 ;
        flag1 = 0 ;
        v1 = ept -> vname ;
        v2 = ept -> ename ;
        for( i = 0 ; i < ng ; i++ ){
          if( v1 == seeds[i] || v2 == seeds[i] )
            flag1++ ;
        }
        if( flag1 < 2){
        merge_bc( blist, v1, v2 ) ;
        merge( graf, blist, bighead, v1, v2 ) ;
          flag++ ;
          break ;
        }
        ept = ept -> slink ;
      } while( ept != NULL ) ;
      if( flag == 1 )
        break ;
      pt = pt -> next ;
    } while( pt != NULL ) ;
    if( flag != 1 )
      break ;
  }


if( graf -> number > ng ){
  cnt = 0 ;
  vtemp = graf -> first ;
  do{
    flag = 0 ;
    for( i = 0 ; i < ng ; i++ )
      if( vtemp -> name == seeds[i] )
        flag++ ;
    if( flag == 0 ){
      v1 = vtemp -> name ;
      v2 = seeds[cnt++] ;
      merge_bc( blist, v1, v2 ) ;
      merge( graf, blist, bighead, v1, v2 ) ;
      if( cnt == ng )
        cnt = 0 ;
    }
```

```
    vtemp = vtemp -> next ;
  } while( vtemp != NULL ) ;
}
}
```

The following is the function "check_gr_num()" which is a short routine to return the number of cells in the group specified by the argument of the function.

```
#include <stdio.h>
#include "graph.h"

int check_gr_num( graf, v )

graph *graf ;
int v ;

{
  vernode *vpt ;

  vpt = ver_exist( graf, v ) ;

  if( vpt == NULL )
    return( VER_NEXIST ) ;

  return( vpt -> noc ) ;
}
```

The following function is "merge_bc()" which carries out the merge operation in the cell-net list. The cells to be merged are specified in the function arguments.

```
#include <stdio.h>
#include "band.h"

int merge_bc( list, v1, v2 )

bandlist *list ;
int v1, v2 ;

{
  band *bpt1, *bpt2, *bpt3 ;
  bandcell *bcpt1, *bcpt2 ;
```

```
  bcpt1 = bcell_exist( list, v1 ) ;
  bcpt2 = bcell_exist( list, v2 ) ;

  if ( bcpt1 == NULL || bcpt2 == NULL )
    return( BANDCELL_NEXIST ) ;

  bpt1 = bcpt1 -> link ;
  bpt2 = bcpt2 -> link ;
  bpt3 = bcpt2 -> inlink ;

  if( bpt3 != NULL )
  do{
    insert_inlink(list,v1,bpt3->name,bpt3->inside,0,bpt3->io) ;
    del_band( list, v2, bpt3 -> name ) ;
    bpt3 = bpt3 -> next ;
    } while ( bpt3 != NULL ) ;

  if( bpt2 != NULL )
  for(;;)
   {
    if( bpt1 != NULL )
    do{
       if( bpt2 -> name == bpt1 -> name )
      {
        bpt1 -> inside += bpt2 -> inside ;
        bpt1 -> outside -= bpt2 -> inside ;
        bpt1 -> io += bpt2 -> io ;
        if( bpt1 -> outside == 0 )
        {
          insert_inlink(list,v1,bpt1->name,bpt1->inside,0,0) ;
          del_band( list, v1, bpt1 -> name ) ;
          }
          break;
        }
       bpt1 = bpt1 -> next ;
      } while ( bpt1 != NULL ) ;
    if ( bpt1 == NULL )
      insert_band(list,v1,bpt2->name,bpt2->inside,bpt2->outside,
                  bpt2 -> io ) ;
      bpt2 = bpt2 -> next ;
      if( bpt2 == NULL )
        break ;
      bpt1 = bcpt1 -> link ;
    }
  del_bandcell( list, v2 ) ;
}
```

The following function is "merge()" which merges cells in the graph. The cells to be merged are again specified in the function arguments.

```
#include <stdio.h>
#include "graph.h"
#include "band.h"

int merge( graf, list, bighead, c1, c2 )

graph *graf ;
bandlist *list ;
big *bighead ;
int   c1, c2 ;

{
 gnode *gtemp ;
 vernode *ver1, *ver2, *vtemp1, *vtemp2 ;
 edgenode *etemp, *etemp1, *etemp2 ;
 int t ;
 int cntn ;

 ver1 = ver_exist( graf, c1 ) ;
 ver2 = ver_exist( graf, c2 ) ;

 if ( ver1 == NULL || ver2 == NULL )
   return ( VER_NEXIST ) ;

 del_edg ( graf, bighead, c1, c2 ) ;
/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */


       /* Add the same edges in ver2 to ver1 */
       /* if different edges, insert edges */


/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
 etemp2 = ver2 -> link ;
 etemp1 = ver1 -> link ;

if( etemp2 != NULL )
for(;;)
  {
    if ( etemp1 != NULL )
     do{
      if ( etemp1 -> ename == etemp2 -> ename )
              {
         cntn = nofnet_betwbc( list, c1, etemp2 -> ename ) ;
         if( etemp1 -> cntn != cntn )
          {
            del_bigedge( bighead, etemp1, etemp1 -> cntn ) ;
            insert_bigedge( bighead, etemp1, cntn ) ;
            etemp1 -> cntn = cntn ;
           }
           break ;
         }
         else
```

```
          etemp1 = etemp1 -> next ;
        } while ( etemp1 != NULL ) ;
      if ( etemp1 == NULL )
        insert_edg ( graf, bighead, c1, etemp2 ) ;
        etemp2 = etemp2 -> next ;
      if ( etemp2 == NULL )
        break ;
    etemp1 = ver1 -> link ;
  }



/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */

                    /* Change all c2 to c1 */


/* %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% */
 vtemp1 = graf -> first ;
 etemp1 = etemp = vtemp1 -> link ;
 for(;;)
 {
   if ( etemp1 != NULL )
     do{
       if ( etemp1 -> ename == c2 )
         do{
         if(  etemp -> ename == c1 )
         {
          cntn = nofnet_betwbc(list,vtemp1->name,etemp->ename) ;
          if( etemp -> cntn != cntn )
              {
                del_bigedge( bighead, etemp, etemp -> cntn ) ;
                etemp -> cntn = cntn ;
                insert_bigedge( bighead, etemp, cntn ) ;
              }
            del_edg( graf, bighead, vtemp1 -> name, etemp1 -> ename ) ;
            break ;
          }
         etemp = etemp -> next ;
         } while ( etemp != NULL ) ;
         if( etemp == NULL )
           {
            if( etemp1 -> vname > c1 )
             {
               re_order( graf, list, bighead, etemp1, c1 ) ;
               del_edg(graf,bighead,vtemp1->name,etemp1->ename) ;
             }
             else
              {
              cntn = nofnet_betwbc( list, c1, etemp1 -> vname ) ;
              if( etemp1 -> cntn != cntn )
                 {
                   del_bigedge( bighead, etemp1, etemp1 -> cntn ) ;
```

```
                    insert_bigedge( bighead, etemp1, cntn ) ;
                    etemp1 -> cntn = cntn ;
                  }
                 etemp1 -> ename = c1 ;
               }
               break ;
             }
          etemp1 = etemp1 -> next ;
       } while ( etemp1 != NULL ) ;
     vtemp1 = vtemp1 -> next ;
     if ( vtemp1 == NULL )
       break ;
    etemp1 = etemp = vtemp1 -> link ;
   }
  insert_grp ( graf, c1, c2 ) ;
  del_ver ( graf, bighead, c2 );
}
```

Appendix E

The C Programming Code for Pseudo Parallel Merge Algorithm

This appendix contains C programming code for the pseudo parallel merge algorithm. It has the same header file as the merge algorithm in the Appendix D. This software contains a main function named as "pseudo_parallel()", which further contains five other functions: namely "divider()", "coordinator()", "constructor()", "parallel_merge()", "changecellname()". The function "divider()" is used to divide the circuit into several subcircuits. The "coordinator()" is used to change the implicit cell name to the corresponding explicit cell name. The "constructor()" is used to re-combine the size-reduced subcircuits to a size-reduced full circuit. The function "parallel_merge()" is used to merge cells in the graph. The "changecellname()" is used to record the cells being needed to change the names.

The following is the function "pseudo_parallel()".

```
#include <stdio.h>
#include "graph.h"
#include "band.h"
#include "list.h"

int M[100][100];
int cellname[90001] ;
int v1, v2 ;
int cnt = 0 ;
int ng ;
int nameofsub ;
int sizeconstraint ;
```

```
    int fsc ; /* final size constraint */
    int factor ;

extern char net[][50] ;

void pseudo_parallel( graf, blist, arr1, arr2 )

graph *graf ;
bandlist *blist ;
bandcell *arr1[] ;
vernode *arr2[] ;

{
    int i, i1, i2, k, j, n, cnt = 0 ;
    int s0, s1, l, L, bottom ;
    graph *subcircuit[501] ;
    vernode *vtemp;
    edgenode *etemp, *ept ;
    gnode *gpt ;
    big *subseqhead[501], *bighead ;
    bignode *pt ;
    int lim,lim1, ave ;
    int seed_i, flag ;
    int div=0, div1, divcnt=0, sizeofsub ;
    int oldNumber ;

     for( i=0 ; i<90000 ; i++ )
        cellname[i] = i ;

    printf ("Input the number of group:") ;
    scanf ("%d", &ng ) ;

    for( i =1 ; i <= 500 ; i++ ){
      subcircuit[i] = crt_graph() ;
      subseqhead[i] = crt_big() ;
     }

    printf ("Input the size of subcircuit:") ;
    scanf ("%d", &sizeofsub ) ;

    printf ("Input final size constraint:") ;
    scanf ("%d", &fsc ) ;

    printf ("Input factor:") ;
    scanf ("%d", &factor ) ;

    sizeconstraint = sizeofsub * 1 / factor ;

    oldNumber = 0 ;

pati_again:
    if( graf->number == oldNumber ){
      printf("No further merge\n") ;
     div = divider( graf, subcircuit, graf->number ) ;
```

```
     sub_seq( subcircuit[1], subseqhead[1] ) ;
p_merge(subcircuit[1],subseqhead[1],1,blist,ng,cnt,fsc,arr1,arr2) ;
      goto finish ;
    }
    oldNumber = graf->number ;

    div = divider( graf, subcircuit, sizeofsub ) ;
    div1 = div ;

   for( i=1 ; i<=div ; i++ )
     sub_seq( subcircuit[i], subseqhead[i] ) ;

cnt++ ;

merge_rest :
for( i=1 ; i<=div ; i++ ){
p_merge(subcircuit[i],subseqhead[i],i,blist,ng,cnt,sizeconstraint,arr1,arr2);
  if( div == 1){
    if( subcircuit[1]->number > ng )
p_merge(subcircuit[1],subseqhead[1],1,blist,ng,cnt,fsc,arr1,arr2) ;
    goto finish ;
  }
  else
}

for( i=1 ; i<=div ; i++ ){
  change_cellname(subcircuit[i], cellname) ;
}

for( i=1 ; i<=div ; i++ ){
   coordinator( subcircuit[i], cellname, blist, arr1, arr2 ) ;
}

combine( graf, subcircuit, div ) ;

   for( i=1 ; i<=div ; i++ ){
     subcircuit[i]->number = 0 ;
     subcircuit[i]->first = NULL ;
     subcircuit[i]->last = NULL ;
     subseqhead[i]->number = 0 ;
     subseqhead[i]->first = NULL ;
     subseqhead[i]->last = NULL ;
   }
sizeconstraint = sizeconstraint * 2 ;
goto pati_again ;

finish:

  return ;
}
```

The following is the function "divider()".

```
#include <stdio.h>
#include "graph.h"

int divider( graf, subcircuit, sizeofsub )

  graph *graf, *subcircuit[] ;
  int sizeofsub ;

{
  int i, cnt, j ;
  vernode *vpt ;

  i = 0 ;
  cnt = 0 ;
  vpt = graf->first ;


  if( vpt != NULL ){
  do{
    cnt++ ;
    if( cnt == 1 ){
      i++ ;
      subcircuit[i]->first = vpt ;
    }
    if( cnt == sizeofsub && vpt->next != NULL ){
      subcircuit[i]->last = vpt ;
      subcircuit[i]->number  = cnt ;
      cnt = 0 ;
    }
      if( vpt->next == NULL ){
        subcircuit[i]->last = vpt ;
        subcircuit[i]->number = cnt ;
        break ;
      }
    vpt = vpt->next ;
   } while( vpt != NULL ) ;
   }
   for( j=1 ; j<=i ; j++ ){
     subcircuit[j]->last->next = NULL ;
}
  return(i) ;
}
```

The following is the function "coordinator()".

```
#include <stdio.h>
```

```
#include "graph.h"
#include "band.h"

void coordinator( graf, cellname, blist, arr1, arr2 )

  int cellname[] ;
  graph *graf ;
  bandlist *blist ;
  bandcell *arr1[] ;
  vernode *arr2[] ;

{
  vernode *vpt ;
  edgenode *ept, *ept1 ;
  int last ;

  last = graf ->last -> name ;
  vpt = graf->first ;
  if( vpt != NULL ){
    do{
      ept = vpt->link ;
      if( ept != NULL ){
        do{
         if( ept -> ename >  last ){
          if( ept->ename != cellname[ept->ename] ){
            ept->ename = cellname[ept->ename] ;
            ept->cntn = nofnet_betwbc(blist,vpt->name,ept->ename,arr1) ;
            ept1 = vpt->link ;
            do{
              if( ept1->ename == ept->ename ){
                if( ept1 != ept ){
                  del_edg_para( graf, vpt->name, ept1->ename, arr2 ) ;
                  break ;
                }
              }
              ept1 = ept1->next ;
            } while( ept1 != NULL ) ;
          }/* if( ept->ename != cell[ept->ename] ) */
         }
          ept = ept->next ;
        } while( ept != NULL ) ;
       } /* if( ept != NULL ) */
       vpt = vpt->next ;
     } while( vpt != NULL ) ;
  } /* if( vpt != NULL ) */
}
```

The following is the function "constructor()".

```
#include <stdio.h>
#include "graph.h"

constructor( graf, subcircuit, div )

  graph *graf, *subcircuit[] ;
  int div ;

{
  int i, j ;
  int num=0 ;

  for( i=1 ; i<=div-1  ; i++ ){
    num = num + subcircuit[i]->number ;
    j = i + 1 ;
    subcircuit[i]->last->next = subcircuit[j]->first ;
  }
  graf->first = subcircuit[1]->first ;
  graf->number = num + subcircuit[div]->number ;
  graf->last = subcircuit[div]->last ;
}
```

The following is the function "parallel_merge()".

```
#include <stdio.h>
#include "graph.h"
#include "band.h"


parallel_merge(graf,subseqhead,i,blist,ng,cnt,sizeconstraint,arr1,arr2)

  graph *graf ;
  big *subseqhead ;
  bandlist *blist ;
  int ng, i ; /* i: the name of subcircuit */
  int cnt ;
  int sizeconstraint ;
  bandcell *arr1[] ;
  vernode *arr2[] ;

{

  int v1, v2, bottom ;
  bignode *pt ;
  edgenode *ept ;
  int flag ;
  int lim ; /* size constraint */

  bottom = graf->last->name ;
```

```
/*
  lim = (graf->number/3)*cnt ;
*/
  while( graf -> number  >  ng ){
    pt = subseqhead -> first ;
    if( pt != NULL )
    do{
      flag = 0 ;
      ept = pt -> slink ;
      if( ept != NULL ){
        do{
          v1 = ept -> vname ;
          v2 = ept -> ename ;
          if( (check_gr_num( graf, v1, arr2 )
               + check_gr_num(graf,v2,arr2)) <= sizeconstraint ){
            merge_bc( blist, v1, v2, arr1 ) ;
            merge(graf,blist,subseqhead,v1,v2,bottom,arr1,arr2);
            flag = 1 ;
            break ;
           }
          ept = ept -> slink ;
         } while( ept != NULL ) ;
       }
      if( flag == 1)
        break ;
      pt = pt -> next ;
    } while( pt != NULL ) ;
    if( pt == NULL )
      break ;
  }
}
```

The following is the function "changecellname()".

```
#include <stdio.h>
#include "graph.h"

void change_cellname(graf, cellname)

int cellname[] ;
graph *graf ;

{
  vernode *vpt ;
  gnode *gpt ;
  int i;

  vpt = graf -> first ;
  if( vpt != NULL ){
```

```
    do{
      gpt = vpt -> glink ;
      if( gpt != NULL ){
        do{
          cellname[gpt->name]= vpt->name ;
          gpt = gpt->next ;
        } while( gpt != NULL ) ;
      }
      vpt = vpt->next ;
    } while ( vpt != NULL ) ;
  }

}
```