AUTOVLSI: A Layout System for General-Cell VLSI Design

by

Hazem Muhebbadin Ahmad Naji Abu-Saleh

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

February, 1995

AUTOVLSI SYSTEM: A LAYOUT SYSTEM FOR GENERAL-CELL VLSI DESIGN

BY

Hazem Muhebbadin Ahmad Naji Abu-Saleh

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

FEBRUARY 1995

AUTOVLSI SYSTEM: A LAYOUT SYSTEM FOR GENERAL-CELL VLSI DESIGN

By

Hazem Muhebbaddin Ahmad Naji Abu-Saleh

February 1995

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA
COLLEGE OF GRADUATE STUDIES

This thesis, written by

Hazem Muhebbaddin Ahmad Naji Abu-Saleh

under the direction of his thesis advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

Thesis Committee:

Sadiq Sait · M.
Dr. Sadiq M. Sait (Chairman)

Dr. Habib Youssef (Co-Chairman)

Dr. Mohammad S.T. Benten (Member)

Department Chairman

Dean, College of Graduate Studies

July 8, 1995
Date

Dedicated to

My Parents
Wife, and Brother

whose support and prayers
led to this achievement.

إهـــداء إلـى

والـــدَيَّ

زوجـــتـي ، و أخـــي

الـذيـن بدعـائـهـم و دعـمـهـم
تـم إنـجـاز هـذا الـبـحـث

Acknowledgments

All praise be to Allah for his limitless help and guidance. Peace and blessings of Allah be upon his prophet Muhammad.

Acknowledgment is due to King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, for the generous help and support for this research.

I would like to express my profound and appreciation to my advisor, Dr. Sadiq M. Sait, Associate Professor of Computer Engineering, for his guidance and patience throughout this thesis.

I would also like to thank co-chairman, Dr. Habib Youssef Associate Professor of Computer Engineering, whose continuous encouragement can never be forgotten, and Dr. Mohammad S.T. Benten Associate Professor of Computer Engineering for his consistent support and valuable suggestions.

Thanks also to the faculty of the Computer Engineering Department; Department Chairman, Dr. Sameer Abdul-Jauwad for their consistent encouragement and support.

Help in printing, encouragement and good wishes of the following friends is also worthy of acknowledgment. They are: Ammar Al-Akel, Ayman Altounji, Samer Al-Sayed, and Rashad Bahsaly.

Finally, special thanks must be given to my family for their encouragement and moral support.

Contents

List of Figures

Thesis Abstract

Name: Hazem Muhebbaddin Ahmad Naji Abu-Saleh.

Thesis Title: AutoVLSI SYSTEM: A Layout System for General-Cell VLSI Designs.

Major Field: Computer Engineering.

Date of Degree: February 1995.

Designing complex digital systems from hardware description languages (HDL) models has been in vogue in the last decade.

The task of translating an HDL description to silicon can be divided into two stages. The first stage consists of the translation of the language model to an intermediate logic level description (hardware compilation), and the second stage is the translation of logic level description to VLSI layouts (physical design).

Physical design consists of tasks such as logic assignment, placement of layout cells, global routing, and detailed routing, etc. In this work, we address the physical design stage. Efficient algorithms were selected, modified where needed, and implemented for the different tasks of the physical design stage.

The proposed work will enable a complete automation of VLSI layouts from hardware description models.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

February 1995

خلاصة الرسالة

الإســـــــــــم: حازم محب الدين أحمد ناجي أبوصالح.
عنوان الرسالة: نظام *AutoVLSI* : نظام تصميم المخططات التفصيلية -ذات العدد المرتفع جدا من الدوائر المتكاملة- باستخدام خلايا عامة.
التخصص: هندسة الحاسب الآلي.
تاريخ الشهادة: فبراير ١٩٩٥.

إن تصميم أنظمة رقمية معقدة باستخدام لغات وصفية لتصميم الحاسب الآلي أصبح موضوعا دارجا منذ عشر سنوات مضت.

مهمة تحويل نظام رقمي موصوف بإحدى هذه اللغات إلى مخطط تفصيلي -ذي عدد مرتفع جدا من الدوائر المتكاملة- على شريحة سليكونية تنقسم إلى مرحلتين؛ في المرحلة الأولى: يتم تحويل الوصف إلى وصف منطقي متوسط يعرف باسم القائمة الشبكية. و المرحلة الثانية: هي تحويل هذه القائمة الشبكية إلى مخطط تفصيلي -ذي عدد مرتفع جدا من الدوائر المتكاملة- و تعرف هذه المرحلة بالتصميم الفعلي.

مرحلة التصميم الفعلي تتكون من عدة مهام منها تحديد الخلايا، و وضع الخلايا على المخطط التفصيلي، و تحديد قنوات التوصيل بين هذه الخلايا (التوصيل العالمي)، و التوصيل الفعلي فيما بين الخلايا الموضوعة (التوصيل المفصل). نقوم في هذا العمل بدراسة الحلول الأوتوماتيكية لمرحلة التصميم الفعلي. و قد تم إختيار مخططات البرامج الفعالة، و تعديلها عند الحاجة، و أخيرا برمجتها لتنفيذ مهام مرحلة التصميم الفعلي.

إن العمل المقدم سوف يمكن توليد المخططات التفصيلية -ذات العدد المرتفع جدا من الدوائر المتكاملة- أوتوماتيكيا بدءا من الوصف المنطقي بأسلوب القائمة الشبكية لنظام رقمي.

درجة الماجستير في العلوم
جامعة الملك فهد للبترول و المعادن
الظهران ، المملكة العربية السعودية
فبراير ١٩٩٥

Chapter 1

Introduction

With its high speed, high reliability and compact implementation, digital technology has infiltrated every field of modern electronics, and used in assembling computers, Hi-Fi systems, camcorders, audio-visual devices, security systems, washing machines, microwave ovens, and even video toys.

The design of modern digital systems requires contributions from several engineering specialists. First, a *system designer*, or *system architect*, determines the described characteristics for the final system and prepares a detailed specification that should define all inputs, outputs, environmental conditions, operation speeds, etc. A *logic design engineer* translates the system specification into a logic design that can meet the functional requirements. The task of the *circuit engineer* is to design circuits that provide the required logic functions. He has the option of designing the complete circuit using available *off-the-shelf* digital components (integrated circuits), or designing some parts of the circuit using VLSI (very large scale integration) technology and implementing his own IC (integrated circuit) [22].

Digital circuits have attracted international attention. Different nations using

different languages had to deal with them, so instead of using a bulky schematic description of a digital circuit, hardware description languages (HDL's) were invented to ease understanding of digital circuits specifications by different language speakers. HDL's were used in different fields of digital research and teaching.

Before 1970, digital circuits were analyzed and designed almost exclusively by hand. However, designing a digital system by hand is not easy. It requires three different engineers. Each of them has to make a lot of decisions and spend enough time that is practical in terms of days or months –depending on the complexity of the system– to accomplish his task.

As digital circuits increased in complexity and variety of use, there was a need for inventing fast ways of developing these circuits. This was the motive for finding what is known as a hardware programming language (AHPL), which is an integrated environment of an HDL and a hardware compiler that translates the description of the circuit into a logic design that meets the functional requirements. This compiler was to replace the task of the *logic design engineer*, because the compiler works much faster, more accurate, and produces more optimized designs. Designing complex digital systems using hardware programming languages has been in vogue in the last decade.

The task of translating HDL descriptions to silicon can be divided into two stages. The *first stage* consists of the translation of the language model to an intermediate logic level description (*hardware compilation*), and the *second stage* is the translation of logic level description to VLSI layouts (*physical design*).

The project implemented in this thesis (called *AutoVLSI*) addresses the physical design stage. Universal AHPL (UAHPL), an extension of A Hardware Programming

Language (AHPL), with its built-in hardware compiler is used for generating the logic level description of the digital circuit. *AutoVLSI* receives the logic description of a circuit generated by UAHPL and generates a VLSI layout for this description. The logic level description is characterized by two lists. The first shows what digital gates are used in the design, and the second gives details about the interconnection between inputs and outputs of these gates. This is called a netlist description of the circuit.

AutoVLSI system is not restricted to UAHPL as the generator of the circuit netlist description. It accepts any netlist description whose gates are all available in the system library of VLSI cells. However, *AutoVLSI* supports logic assignment for the types of gates used by UAHPL and does not have VLSI cells representing them in the system library. Suppose, for example, that UAHPL uses in its generated netlist an *8-input AND* gate which is not available in the system library, then *AutoVLSI* assigns four *2-input AND* gates and one *4-input AND* gate –assuming of course that both types of gates exist in the system library– to replace the *8-input AND* gate used by UAHPL.

Physical design problem can be broadly divided into two stages. *Placement stage* which gives the exact place of each VLSI cell on the layout plane, and *detailed routing stage* which wires the interconnection of these placed cells. Achieving complete routing for all nets is a must. One net which is not completely routed results in a malfunction in the circuit. Usually an intermediate stage called *global routing* is inserted in between the above two stages to predict the paths of wires and facilitate the *detailed routing stage.* Each stage of the physical design process consists of many tasks and will be described later in this chapter.

The *detailed routing* stage, was studied earlier at King Fahd University of Petroleum and Minerals (KFUPM) in 1987. A maze router that follows Lee algorithm was implemented in FORTRAN [23].

In this thesis work, we address the other two stages of physical design, namely *placement* and *global routing* plus integrating the whole work of *logic assignment*, *placement*, *global routing* and *detailed routing* into a complete digital automation system that automatically produces VLSI design layouts for digital circuits described in UAHPL. The proposed system will also produce additional data for simulation and verification of the generated layouts.

Many heuristics and algorithms related to these above mentioned topics have been carefully studied, analyzed, and examined. Finally, a subset of them were adopted, modified as needed, and implemented to generate *Auto VLSI* system. All programs except those for the *detailed routing stage*, were implemented in the Turbo *C* programming language from Borland company using an 80486-microprocessor IBM personal computer. Many other supporting utility programs were also implemented. Some of them find an estimate for the chip area and compute space utilization. Others, plot a symbolic graph of the layout showing the relative placement and the paths of routes. A brief description of every software used in *Auto VLSI* system is given in Appendix A. Moreover, the system flow-chart is illustrated in Appendix B.

1.1 Logic Design

A *digital system* is any system for transmission or processing information that is represented by physical quantities (signals) that can take only one of the two discrete

values: 0 or 1.

A *logic designer* translates the system specification into a logic design that can meet functional requirements. The task of the *circuit engineer* is to design circuits that provide the required logic functions. Digital circuits differ in terms of difficulty. They might consist of a few gates, or they can have Arithmetic/Logic units, multiplexers, buffers, and even microprocessors. The first choice when implementing a digital circuit is to use some ready-made integrated circuits (ICs). A definition of an IC is quoted from [22]:

> A circuit consisting of active and passive elements fabricated on a single semiconductor chip and mounted in an individual package.

Digital circuits can be implemented using only SSI (small scale integration) ICs if they are very simple, MSI (medium scale integration) and LSI (large scale integration) ICs or a combination of them, if they are more complicated. Once the designer decides upon what ICs the circuit needs, he lays them out on a PCB (printed circuit board) and makes the external interconnections between different pins of each IC to complete the design.

Regardless of the kind of ICs by which the circuit is implemented, there could be some ICs that are not 100% utilized while taking space on the PCB. Moreover, the external interconnections decrease the reliability of the circuit and might introduce some timing problems. And, finally, every digital circuit of adequate size will have components such as arithmetic/logic units, shift registers, multiplexers, buffers, and so on. Some of these components might not be available as *off-the-shelf* components. Therefore, for one or more of the above reasons, the designer might have to choose implementing the special part of his digital circuit on a VLSI chip. In such a case,

he might prefer to integrate all of the digital circuit he is designing into one chip.

In summary, there are many reasons that push a designer to use VLSI technology and fabricate his own IC instead of completely depending on *off-the-shelf* components to design his circuit (or part of it).

1.2 Hardware Programming Languages (HPLs)

To aid communication between digital circuit designers, computer description languages(HDLs) evolved from the research field. HDLs were initially designed to describe digital circuits at the behavioral level. Later on, other applications for HDLs emerged. They have been used as input to a simulator at the register transfer level, and as input to a hardware compiler that automatically translates the high level description to logic design.

As digital circuits increased in complexity and the variety of uses, the benefits of HPLs increased more and they started to be a universal tool for designing digital circuits. An example of a hardware programming language that supports HDL, a hardware compiler and a register transfer level simulator is UAHPL (Universally A Hardware Programming Language).

UAHPL, an extension of A Hardware Programming Language (AHPL), is used as the register transfer language for specifying a digital design. It is a simple language yet sufficient to model highly complex digital systems such as parallel processors and data flow machines. It has a compiler that converts the circuit specifications into an intermediate representation in form of tables after performing syntax and semantic analysis. This intermediate form can be used by the supplied simulator to verify whether the specification exhibits the desired behavior described in the

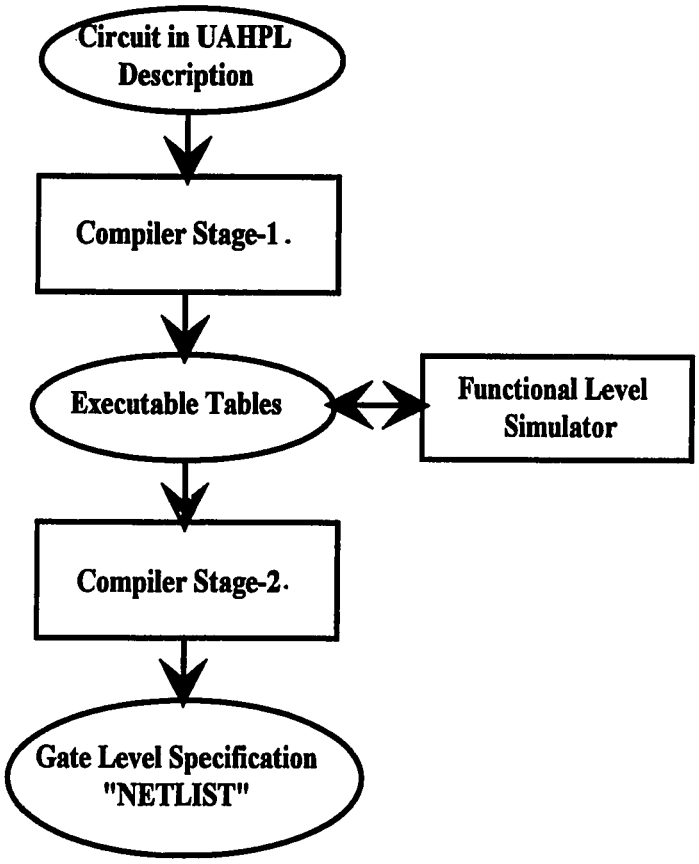


Figure 1.1: Diagram of UAHPL system.

specification. Another compiler translates the intermediate tabular form to logical netlist that gives details about the gates needed for the design and their relative interconnections. A diagram of UAHPL system can be seen in Figure 1.1.

1.3 VLSI Technology

VLSI or Very Large Scale Integration, refers to a technology through which it is possible to implement large circuits –circuits with up to a million transistors– in silicon. VLSI has been successfully used to build microprocessors, memory controllers, I/O

controllers, and interconnection networks. It can also be used to implement complete digital systems in silicon as opposed to using MSI and LSI *off-the-shelf* components [2].

There exist many advantages for implementing digital systems in silicon such as:

- *Achieving additional complexity in silicon.* VLSI allows complexity to be included on a single silicon substrate. This means that complex digital functions, requiring tens of ICs, can fit onto a single silicon die and be integrated in one IC. This will enormously reduce board space requirements.

- *Increasing reliability.* Board design reliability increases as fewer parts are placed on a circuit board. This reliability continues to improve as a lower parts count contributes to a reduction in connections and traces on the circuit board.

- *Maximizing performance.* Timing problems will be reduced by using as less ICs as possible on the circuit board. The same goes for power requirements.

- *Providing security for new designs.* Using *off-the-shelf* ICs provide the designers with a very little security for their designs. The only way of securing their designs is by clearing the IC title from the top side of the case. On the other side, no one can predict the circuit implemented by a private IC. Thus, designs that uses at least one private IC will be protected from illegal copies.

Designing a VLSI circuit involves a trade-off between cost, performance, ease of design, time, and many other factors. A good layout is one which occupies minimum area, uses short wires for interconnection, and uses as few *vias* as possible [2].

Since the complexity of VLSI circuits is in the order of millions of transistors, designing a VLSI circuit is a complex task. Clearly it is not possible to sit down with paper and pencil to design a million-transistor circuit. Instead, there evolved many styles for designing a VLSI layout that extensively reduce the amount of effort and time exerted by the engineer. Also many computer software were implemented to aid the design of a VLSI layout.

1.3.1 Layout Styles

There are several styles for a VLSI layout. Mainly, the style of the layout is determined by the way it is designed, and by the type of VLSI cells used. Some of these layout styles are:

- Full custom,
- Gate array,
- Standard-cells,
- General-cells,
- PLA (Programmable Logic Arrays),
- FPGAs (Field Programmable Gate Arrays).

The complexity of implementing a VLSI layout decreases from top to bottom for these styles. *Full custom layouts* are completely designed by hand using a layout editor. The concept of VLSI cells does not exist in such style. The designer has to implement all VLSI circuits that represent the digital components of the circuit.

In *standard-* and *general-cell styles*, a set of pre-designed VLSI circuits are used to represent different digital components. These VLSI circuits are called macro-cells (cells for short) or modules[1]. The difference between these two styles is the type of cells they use. In *standard-cell style*, usually all different cells have fixed height, as opposed to cells of the *general-cell style* which can be of varying heights.

A detailed description, merits, and shortcomings of each of these styles can be found in [2], [5].

1.3.2 Generating VLSI Layouts

Physical design of a VLSI circuit is the phase that precedes the fabrication of the circuit. The performance, area, and reliability of the circuit depends critically on the way the circuit is physically laid out. As a conclusion, physical design is a complex optimization problem. It is therefore customary to subdivide the problem into more manageable subproblems. A common subdivision is as follows

- Logic assignment
- Cells Placement
- Nets Routing

Logic Assignment

The compiler of the hardware description language (HDL) used to generate the design of a digital system represented in netlist format will have its own set of gates which are used to specify the logic design. For example, it could have a *20-input*

[1]The terms cell and module will be used interchangeably throughout the chapters of this thesis.

AND gate, *4-input* **OR** gate, *3-input* **NOR** gate, *5-input* **AND** gate, etc. It is not practical to implement all VLSI cells to represent all different gates used by the hardware compiler. Moreover, some of these gates might not have a feasible VLSI design, while others could need special technology to be implemented.

Logic assignment is the process of unifying these parameters. A pre-specified set of universal gates is prepared for the system and during the assignment stage, all gates of the digital system's netlist are replaced by an equivalent design of gates in the system. A new netlist that has only gates available in the system library will be generated in the assignment stage. Each gate of the system library has a VLSI design that represents it. When creating the final VLSI layout, each gate or cell is replaced by its VLSI design that is already available and previously simulated and checked [19].

Cells Placement

Placement is the task of finding the exact positions of each cell in the new netlist that are to be placed in the VLSI layout. Added to the two main objectives which are minimizing total wire length and layout area, placement has one major objective to satisfy and that is facilitating routing. A router uses the space area between placed cells to connect different pins of the cells with each other. In the placement stage, enough spacing must be afforded to allow routing of all nets.

Many techniques exist for the placement task. They can be broadly divided into two categories: *constructive* and *iterative*. A *constructive technique* will usually begin by placing one cell in the layout, then it starts picking other cells one at a time and tries to find the most suitable place for them. It is fast, but usually does

not generate optimistic layouts [2], [5].

Iterative techniques begin by a complete initial layout that could have been acquired randomly or from a constructive algorithm, and improve upon this initial solution to reach a better one. They consume a lot of execution time, yet sometimes might not markedly improve the initial solution.

Nets Routing

Routing is usually the most difficult and time consuming stage. Detailed routers connect all physical pins of a net on different cells to each other randomly using some connecting material like silicon or metal. From an electrical point of view, it means to keep them at the same potential.

A detailed router for the *general-cell style* is restricted to put these routes only in a space area between the cells of the layout. Since routing all nets of a VLSI design is a must, then insuring enough space ahead of the routing stage will save a lot of time. Thus, a step called global routing is usually done ahead of the detailed routing.

Global Router This task is usually inserted exactly after the placement stage to help achieve complete routing of the design. The space area between two cells that is parallel to the vertical borders of the cells is called *vertical channel*. If this space area is parallel to the horizontal borders of cells, then it is called a *horizontal channel*.

The placement stage can be modified so that it produces the data of the exact location and dimensions of every channel in the layout after the initial placement task. Global routing uses these data and the data of available nets according to the

netlist and determines the exact channels used to connect different pins of different cells. Moreover, it can tell how many wire segments are to pass in each channel assuming that the router will always select the shortest path to connect pins of a net. This assumption is safe, and in fact supports the objective of minimizing the total wire length.

According to the information gained from the global routing task, spacing between cells can be exactly determined, and in most cases will be enough to route all nets. The initial layout acquired from the placement stage can be modified to support this amount of free space.

Detailed Router The detailed router receives data from the placement stage that has the exact locations of cells in the layout. According to the netlist, it connects every two pins at a time using the shortest available path. Global routing will certainly improve the routing process but there still could be some connections not accomplished due to the fact that the detailed router may not use the same channels that were predicted in the global routing stage. The detailed router will of course tell if any of these non-accomplished connections exist. A designer can finish these connections by hand using a layout editor.

1.3.3 VLSI Layout Programs

Many computer programs that aid the design of VLSI layouts were implemented by different universities, research institutes, and commercial companies. They support the process of designing a layout by tools that check the design correctness against VLSI rules and simulate the functionality of the final design [1], [3], [6].

One of these famous programs is MAGIC. It is a VLSI layout editor with a built-in simulator. It supports also a circuit extractor that saves the layout in many different formats used by other tools for electrical verification, simulation, or printing for example [3], [17].

1.4 *Auto VLSI*

Auto VLSI is an integrated system that automatically generates VLSI mask layout for any digital circuit described by a netlist. This netlist description can be obtained from the hardware compiler of any hardware programming language (HPL). For several years in KFUPM, a hardware programming language called UAHPL has been used to describe digital circuits, simulate them, and generate their corresponding netlists. The system proposed in this thesis was tuned to UAHPL type of netlists.

The system main objective is to minimize turn-around time. Other objectives are to minimize total wire length and total layout area. *Auto VLSI* system has a library of general VLSI cells. All these cells were designed and simulated to make sure that generated layouts using these cells will be correct by construction.

Auto VLSI system can be broadly divided into four basic stages. The first stage assigns VLSI cells from the cell library of the system to gates of the layout netlist description. The second stage is placement. It finds the best position of each cell according to its size and the connectivity relation between it and all other cells in the layout. The third stage which is global routing, predicts where the detailed router is expected to run wires on the layout and adjusts the layout such that enough space is supplied between cells of the layout. Finally, the fourth stage is the detailed routing. A software that uses Lee algorithm is used to connect different pins of cells

which belongs to the same net by wire segments of different materials. Segments are placed in a way that each different net is electrically isolated from any other net.

Each one of these stages has several tasks (see Figure 1.2). Many heuristics and algorithms were studied, modified as needed, and then implemented to achieve a system that best accomplishes its objectives.

1.5 Description of Contents

Chapter 2 addresses logic assignment. It explains the methodology of general cell library and the rules of designing general cells to be used by *AutoVLSI* system. A sample set of VLSI cells used by *AutoVLSI* are described in this chapter, and a procedural algorithm which assigns these cells to gates used by UAHPL is given.

Chapter 3 is about the placement stage. It starts by giving an introduction about placement problem, how it is defined, the complexity of the problem, and how a layout quality is found. Then different techniques of placement are discussed, and finally how placement is done in *AutoVLSI* is explained.

Chapter 4 explains how global routing is done to predict wires' paths on the layout plane ahead to the detailed routing stage. The global routing algorithm adopted by *AutoVLSI* is explained, and the complete operation is illustrated by a small layout example.

Chapter 5 starts by giving an overview about the function of detailed routers. It focuses on maze routing, and finally explains how the router program that was earlier implemented is integrated with other software to form the *AutoVLSI* system.

Chapter 6 is the conclusion of this thesis. It summarizes the work, and shows some results and statistical data. It explains how compaction can be applied and

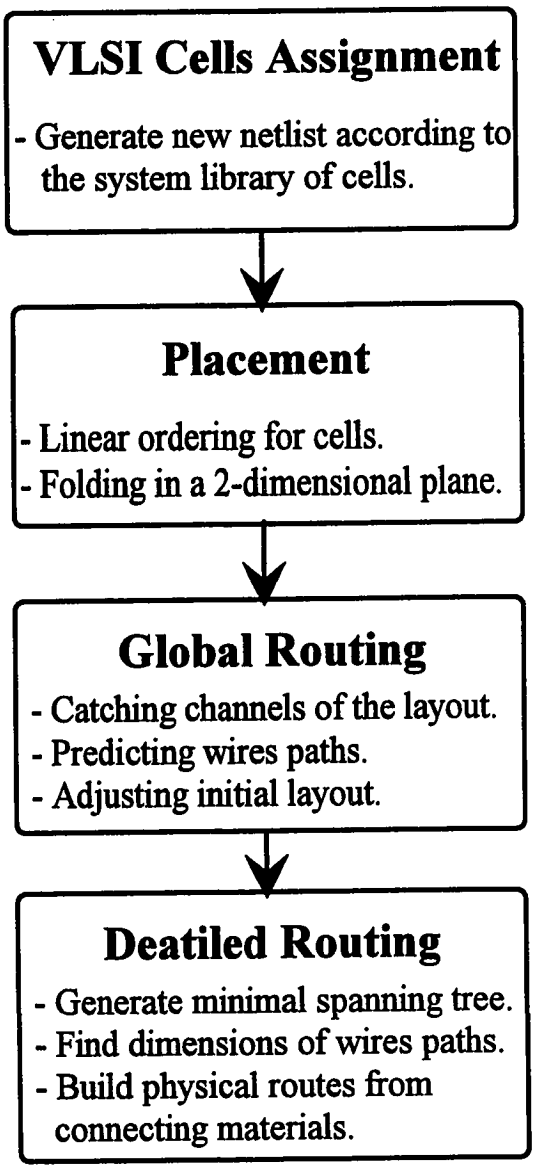


Figure 1.2: Stages of *Auto VLSI* system including their tasks.

finally gives some ideas about future work.

Chapter 2

Logic Assignment

The problem of designing general-cell style layout, is very much similar to that of designing a PCB (Printed Circuit Board). However, instead of selecting *off-the-shelf* SSI or MSI components, we have to select components from a pre-designed VLSI cell library. And, instead of placing the components on a PCB, we place the cells in silicon. A common advantage is that designs can be completed quickly [2].

If we assume that there exist for every gate in every possible netlist description – generated by the HPL used– a VLSI circuit (or cell) that matches the gate's function, inputs, and outputs, then there is no need for any logic assignment. The designer must only be concerned about where to place each VLSI cell on the layout floor and how to interconnect different pins of the cells according to the netlist description.

However, the above assumption does not apply for UAHPL and many other HDLs. A stage called *logic assignment* is needed to transform the input netlist into a netlist of cells available in the system library.

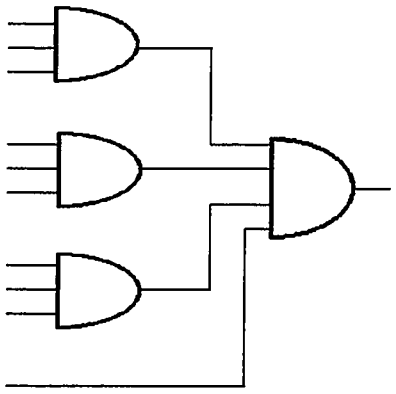
2.1 Logic Assignment for UAHPL Gates

Not all gates used by UAHPL have a corresponding cell in the library. For example, UAHPL could use a *10-input AND* gate in its generated netlist. The solution depends absolutely on available VLSI cells in the system's library of cells. If there exist *3-input* and *4-input AND* VLSI cells, then the *10-input AND* gate will be replaced by three *3-input AND* cells and one *4-input AND* cell (see Figure 2.1 (a)).

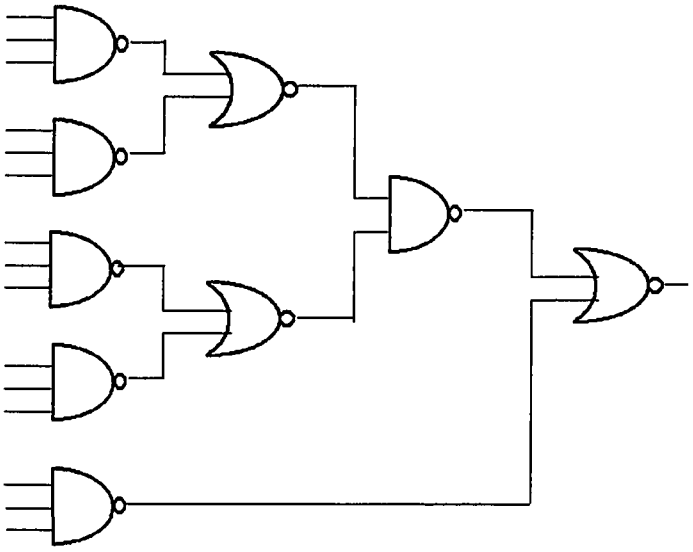
In general, it is not possible to try matching all possible gates used by an HPL. The alternative is to have in the library of VLSI cells the modules for the universal logic gates which are *2-input NAND* cell or *2-input NOR* cell. If we want to replace the *10-input AND* gate, then we would need six *2-input NAND* cells and three *2-input NOR* cell as the circuit shows in Figure 2.1 (b).

AutoVLSI system first reads the types of VLSI cells available in the system library. Then it checks for every gate if an equivalent cell exists with the same number of inputs. If so, it does not take any action for such a gate. On the other hand, if there does not exist an equivalent cell for one of the gates, then it substitutes that gate by universal cells (NAND/NOR) as needed.

The software that does the logic assignment is called **COOKNET**. It receives three input files. The first is *GATES.LIB* which is prepared by the user and contains the data for the VLSI cells available in the system library. The other two input files are the ones generated by the hardware compiler of UAHPL and represent the netlist description of the design. They should have been extracted from a file that has the extension *"ERD"*. These two files are to be named *G1.DAT & IO1.DAT*. **COOKNET** produces two output files, namely *G2.DAT & IO2.DAT*. All gates



(a) ***10-input AND*** **gate using** ***3-input*** **and** ***4-input AND*** **cells.**



(b) ***10-input AND*** **gate using universal** ***2-input NAND*** **and** ***2-input NOR*** **cells.**

Figure 2.1: Logic assignment for a *10-input AND* gate.

represented in *G2.DAT* file have equivalent VLSI cells in the system library as was imposed by the user.

Beside performing logic assignment, **COOKNET** will also eliminate buffers used in the design, redundant inverters, and any isolated gates which neither their inputs nor output are connected to other gates.

2.1.1 *AutoVLSI* Library of VLSI Cells

A very small library of primitive VLSI cells was prepared while developing *AutoVLSI* system. The current library has ten different cells that are needed by the logic assignment stage to replace different gates of netlist descriptions generated by UAHPL for any digital design. These are:

1. *Ex-input* (External Input) cell.
2. *Ex-output* (External Output) cell.
3. *1-input NAND* cell, to work as an *INVERTER.*
4. *1-input NOR* cell, to work as an *INVERTER* (same as the cell in 1).
5. *2-input NOR* cell.
6. *3-input NOR* cell.
7. *2-input XOR* cell.
8. *D Flip-Flop* cell with **SET** control.
9. *D Flip-Flop* cell with **RESET** control.

10. *D Flip-Flop* cell with **ENABLE** control.

Each VLSI cell is carefully hand designed using the **MAGIC** layout editor, and simulated using both its built-in **RSIM** simulator and **RNL** [1],[17]. The cells layouts are given in Figure 2.2.

Simulation outputs of the *INVERTER (1-input NAND gate)*, *2-input NOR,* and *D Flip-Flop* with **SET** control cells using **MAGIC** built-in **RSIM** simulator are given in Figures 2.3 ,2.4, and 2.5 respectively.

The data file *GATES.LIB* for the above library of VLSI cells have five records only. The first two cells (*Ex-input/output*) and the last three cells (*D Flip-Flop*'s) do not need to be specified since their existence is mandatory to UAHPL for all designs. Each record representing a cell would consist of three pieces of information: cell name, type number, and number of inputs of the cell. Cell name can be any combination of letters or numbers, while type numbers are predefined according to the gate type and they are given in Figure 2.6. These type numbers are the ones used by UAHPL to identify the gates used in the netlist description. They can be seen in *G1.DAT* file. *GATES.LIB* file for the above list of cells is given in Figure 2.7.

COOKNET software modifies the type numbers of all cells in its output files. It replaces the second digit from the left of the type number by the number of inputs of the cell as found in *GATES.LIB* file. For example, the *1-input NAND* cell would have a type number 4102, while the *2-input NOR* cell would have a type number 4205. These modified type numbers are very important to be understood since actual cells files will be named after them.

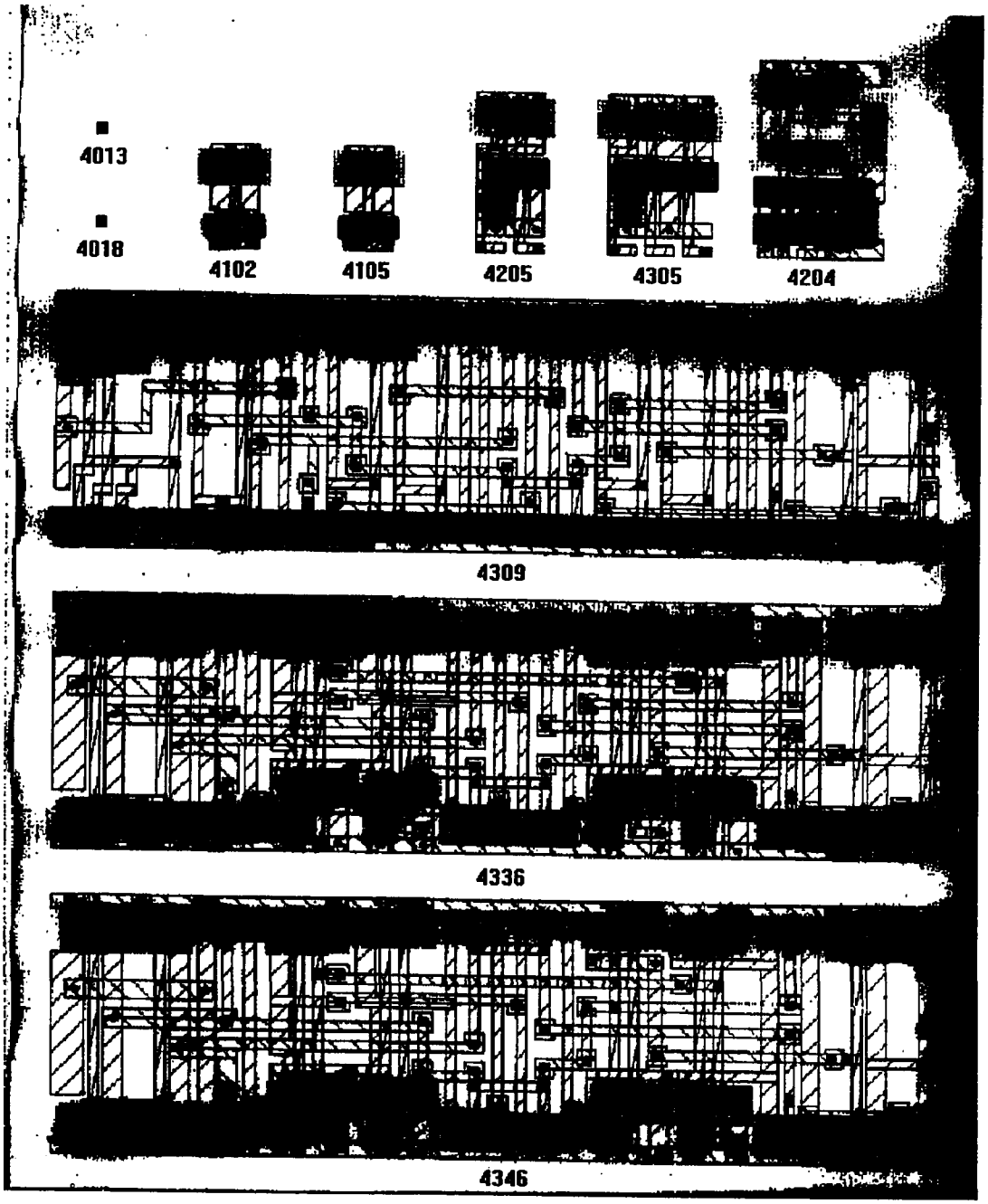


Figure 2.2: VLSI circuits of the cells in *AutoVLSI* library.

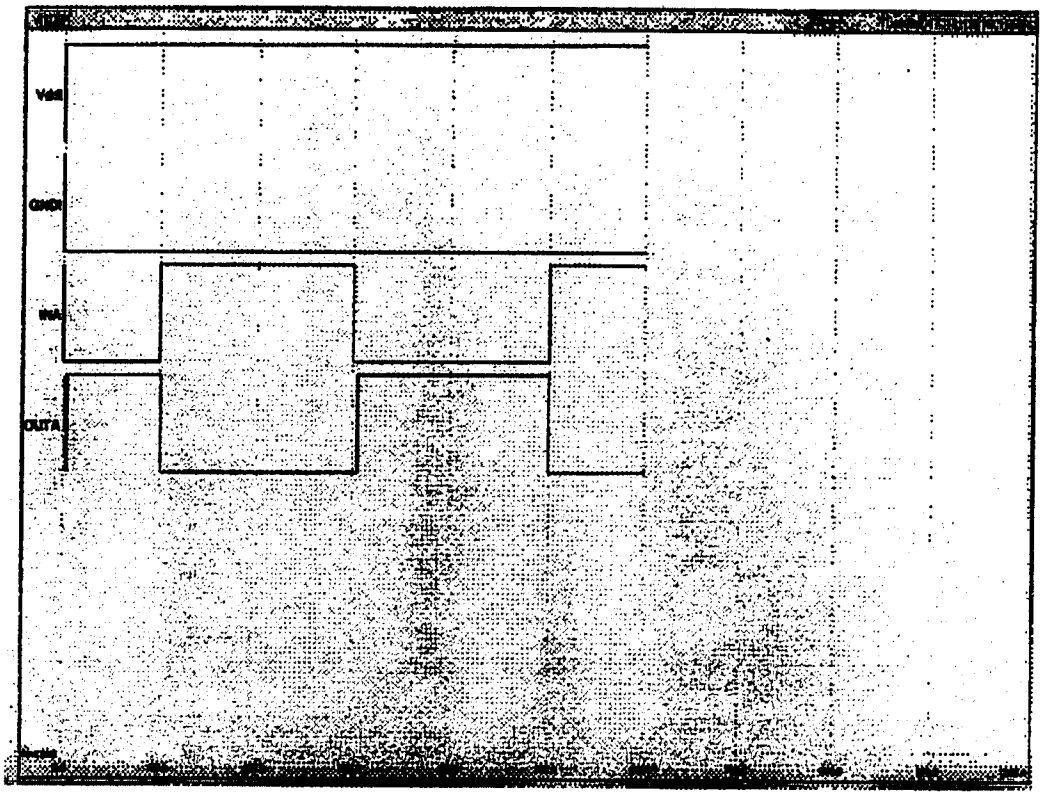


Figure 2.3: RSIM simulation output for cell 4102.MAG (i.e., *INVERTER* gate).

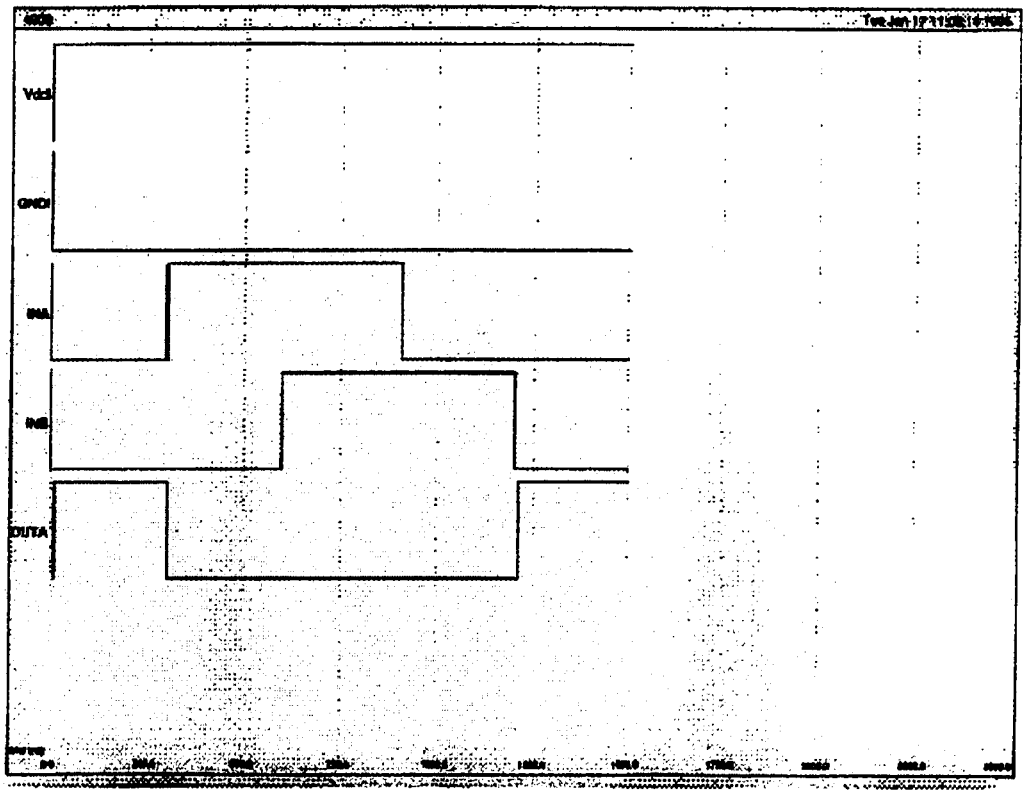


Figure 2.4: RSIM simulation output for cell 4205.MAG (i.e., *2-input NOR* gate).

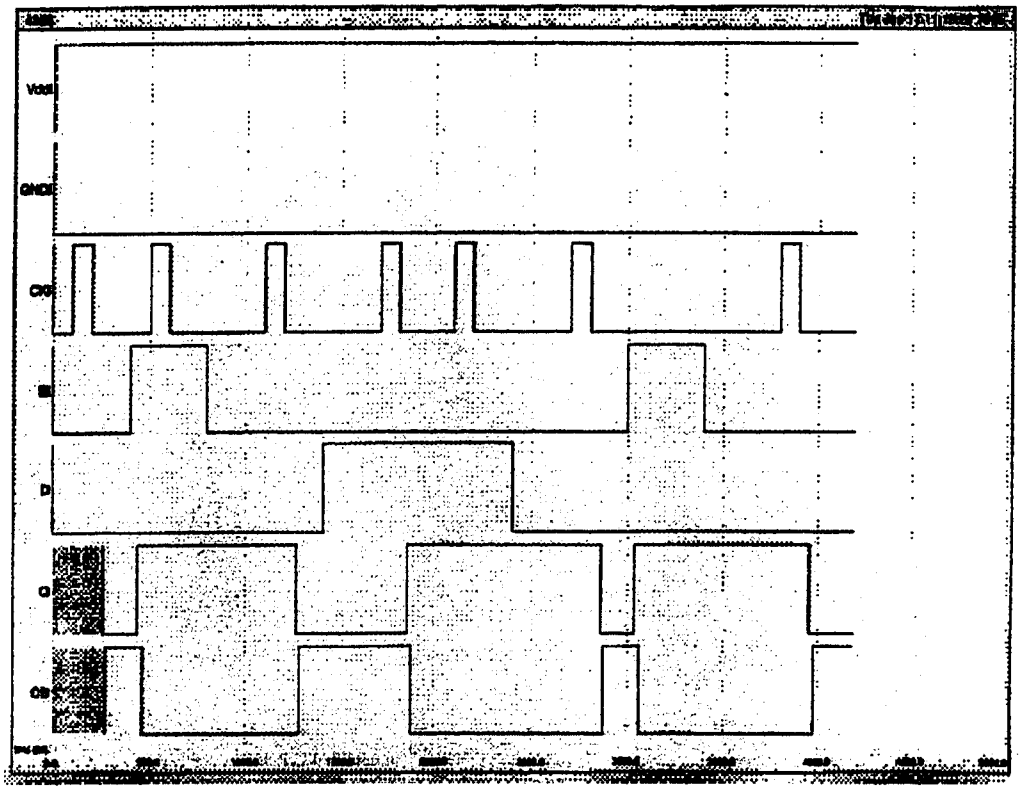


Figure 2.5: RSIM simulation output for cell 4336.MAG (i.e., *D Flip-Flop* with **SET** control).

TYPE NUMBER	GATE TYPE	TYPE NUMBER	GATE TYPE
3998	PASS	4017	BUSEST
3999	WIRE	4018	EXOUTPUTS
4001	AND	4019	EXOUTPUTS
4002	NAND	4020	EXBUSES
4003	OR	4021	EXBUSESO
4004	XOR	4022	EXBUSESA
4005	NOR	4023	EXBUSEST
4006	DFCS	4024	CLUNIT
4007	DFF	4025	CLUI
4008	AUX	4026	CLUO
4009	MEMCK	4027	CTERMS
4010	MEMEN	4028	FNREG
4011	MEMJK	4029	CND
4012	INPUTS	4030	ORCS
4013	OUTPUTS	4031	CLAND
4014	BUSES	4032	CLNAND
4015	BUSESO	4033	CLOR
4016	BUSESA	4034	CLXOR

Figure 2.6: Table of predefined gates' type numbers.

```
NAND1  4002  1
NOR1   4005  1
NOR2   4005  2
NOR3   4005  3
XOR2   4004  2
```

Figure 2.7: List of *GATES.LIB* file.

2.2 *AutoVLSI* General-cell Library

The *general-cell* layout style relaxes a lot of the restrictions imposed by *standard-cell* layout style. Cells can have any rectangular shape resulting in a better arrangement and a more compact floorplan. The main advantage of the *general-cell* style is the ability of storing larger blocks such as *arithmetic/logic units*, *registers*, and *memories* in the cell library. Such blocks can be designed to have efficient layout characteristics [2].

However, there are some rules when designing a *general-cell* used by *AutoVLSI* system. These rules are explained in the following subsection.

2.2.1 Rules of Designing a VLSI Cell for *AutoVLSI* System

As mentioned above, *AutoVLSI* imposes some rules on the VLSI cells used. Some of these rules are mandatory while others are optional. However, they are very easy to apply and do not violate the relaxation of *general-cell* style. These rules are listed below and illustrated in Figure 2.8.

- Cells must be of rectangular shape.
- For the purpose of satisfying VLSI design rules explained in Section 4.3, *AutoVLSI* assumes the layout to be a 2-dimensional grid plane where a grid size is $(8 \times 8)\lambda$, and the minimum allowed spacing between any two cells is 4λ. Every cell must be centered in between $h \times w$ grid units such that space strips, considered as part of the cell, of width equal to 2λ are left empty all around the cell (see shaded area in Figure 2.8). By doing so, *AutoVLSI* satisfies the limitation of leaving at least 4λ of space between any two cells placed on the

floor even if they were placed next to each other. Each cell's height or width will be a multiple of 8 plus 4λ. For example, if a cell occupies one grid unit such as *Ex-input* or *Ex-output* cells, then actual cell height and width will be 4λ. A cell occupying 5-vertical grid units will have actual height equals to $[(5-1)\times 8]+4 = 36\lambda$. Also a cell occupying 3-horizontal grid units will have actual width equals to $[(3-1)\times 8]+4 = 20\lambda$.

- Pins of a cell [inputs, outputs, clock (Clk), power (Vdd), or ground (Gnd)] must lie on the borders of the cell. If on a horizontal border then they must be of *polysilicon* material, else if they are on a vertical border then they must be of *metal1* material. If a port resides on a corner then it must be a contact point between *polysilicon* and *metal1* materials.

- The material of a pin should at least cover the middle 4λ of the grid they are laid on.

- It is preferable that every pin has two or more occurrences on different borders of the cell since this will offer the router several choices to reach that pin in order to connect it to other pins.

- Pins on a cell can be named individually for the purpose of clearance and simulation. However, we suggest to name the clock pins as CK!, the power pins as Vdd!, and the ground pins as GND!. Putting an exclamation mark after these names will indicate them as global pins (i.e., all pins named Vdd! are supposed to have the same value at all times).

- Once a VLSI cell design is completed, it should be simulated individually to make sure that it is functioning correctly as expected.

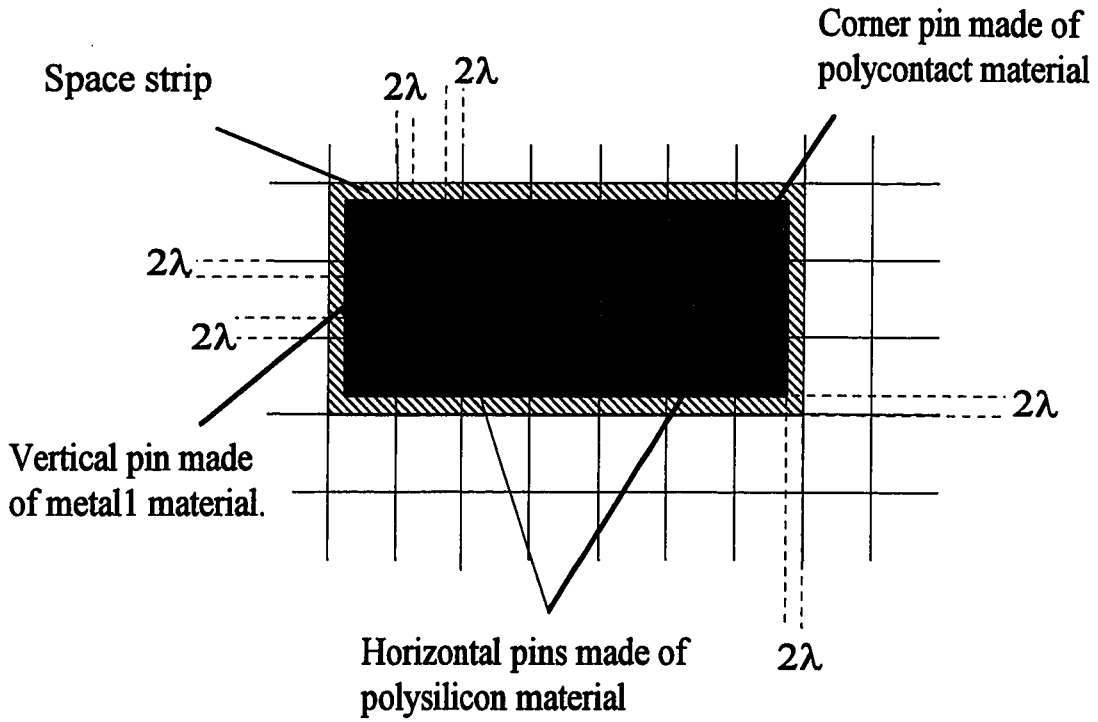


Figure 2.8: Illustration of the rules of designing a VLSI cell for *Auto VLSI* system.

- A cell must be saved in **MAGIC** format and named according to the type numbers which **COOKNET** software adopts. The file names for the above listed library of cells are given in Figure 2.9. Cells files should be available in a sub–directory called "mag" which branches from the design main directory.

Saving the Data of VLSI cells

After designing the VLSI cells to be used by *Auto VLSI* system, the user has to create two data files which hold the information about the cells. The first file should be named *AGHW.DAT* which stands for "Actual Gates Heights & Widths", and will be used in the placement stage. While the second file should be named *POC.DAT* which stands for "Pins on Cells", and will be used

CELL DESCRIPTION	FILE NAME
EX-INPUT	4018.MAG
EX-OUTPUT	4013.MAG
1-input NAND	4102.MAG
1-input OR	4105.MAG
2-input NOR	4205.MAG
3-input NOR	4305.MAG
2-input XOR	4204.MAG
D Flip-Flop with SET control	4336.MAG
D Flip-Flop with RESET control	4346.MAG
D Flip-Flop with ENABLE control	4309.MAG

Figure 2.9: Files names of VLSI cells in *AutoVLSI* system library.

in both global and detailed routing stages.

The Format of *AGHW.DAT* File

Every cell in the library should have a corresponding record in the file *AGHW.DAT*. A record would consist of three pieces of information: cell type, cell height, and cell width. Cells' height and width values should be in terms of grid units rather than λ.

AGHW.DAT file created for the above mentioned library of cells is given in Figure 2.10.

Format of *POC.DAT* File

This file should include all cells and define the coordinates in grid units for their pins with respect to the cell origin $(0, 0)$. There is a standard format for

```
4018   1   1
4013   1   1
4102   6   3
4105   6   3
4205   9   4
4305   9   6
4204   11  7
4309   14  47
4336   14  48
4346   14  48
```

Figure 2.10: List of *AGHW.DAT* file.

defining the coordinates of pins for every cell. The format is as follows:

- **Cell Type #**

  NUMBER OF PINS FOR THE VDD

  Coordinates for the first Vdd pin

  .

  .

  .

  Coordinate for the last Vdd pin

  NUMBER OF PINS FOR THE GND

  Coordinates for the first Gnd pin

  Coordinates for the second Gnd pin

  .

.

.

Coordinate for the last Gnd pin

Number of Input pins

NUMBER OF PINS FOR THE FIRST INPUT

Coordinates for the first pin of the first Input

Coordinates for the second pin of the first Input

.

.

.

Coordinates for the last pin of the first Input

NUMBER OF PINS FOR THE THIRD INPUT

.

.

.

NUMBER OF PINS FOR THE LAST INPUT

Coordinates for the first pin of the last Input

Coordinates for the second pin of the last Input

.

.

.

Coordinates for the last pin of the last Input

Number of Output pins

NUMBER OF PINS FOR THE FIRST OUTPUT

Coordinates for the first pin of the first Output

Coordinates for the second pin of the first Output

.

.

.

Coordinates for the last pin of the first Output

NUMBER OF PINS FOR THE SECOND OUTPUT

Coordinates for the second pin of the second Output

Coordinates for the second pin of the second Output

.

.

.

Coordinates for the last pin of the second Output

NUMBER OF PINS FOR THE THIRD OUTPUT

.

.

.

NUMBER OF PINS FOR THE LAST OUTPUT

Coordinates for the last pin of the last Output

Coordinates for the last pin of the last Output

.

.

Coordinates for the last pin of the last Output

For the *Ex-Input* and *Ex-Output* cells, there is no Vdd or Gnd pins. Thus, the first two lines of their records in *POC.DAT* file have 0. For the *D Flip-Flops*, there are five inputs as imposed by UAHPL language. The first input stands for **D**, the second stands for **CLK!**, and the third input is either **ENABLE**, **SET**, or **RESET** control pins. The last two inputs are each assigned to one pin whose coordinates are (−1,−1) to indicate that it is not used. Each of these Flip-Flops has one output which stands for **Q**. The records for both, *2-input NOR* cell and *D Flip-Flop* with **ENABLE** control cell are given in Figure 2.11.

```
4204
 2
  1 10
  6 10
 2
  1 0
  6 0
 2
  2
   1 0
   1 10
  2
   3 0
   3 10
 1
  2
   6 3
   6 4
```

```
4309
 2
  1 13
  3 13
 2
  0 0
  10 0
 5
  2
   0 3
   0 9
  2
   1 0
   0 13
  2
   2 0
   2 13
  1
   -1 -1
  1
   -1 -1
 1
  2
   46 0
   46 1
```

Figure 2.11: Records of *2-input NOR* cell and *D Flip-Flop* with **ENABLE** control cell in *POC.DAT* file.

Chapter 3

Placement

3.1 Introduction

3.1.1 Definition

In most general terms, placement consists of assigning rectangular modules to locations on a two-dimensional surface while satisfying given constraints and optimizing a given objective. Examples of constraints are to have all modules of standard shape and size. An example of an objective would be to minimize wire length.

Constraints on modules affect to a certain extent the complexity of placement. If the modules are of standard shape and size, then it will be easy to place them in fixed row/column fashion as in the standard-cell layout style (Figure 3.1).

However, when the size of modules is allowed to be different while conserving the same shape as in the general cell layout style, the problem becomes more difficult. Modules in the last case can no more fit in row/column fashion without wasting a lot of space. Obviously, the worst case is when both size and shape of modules can

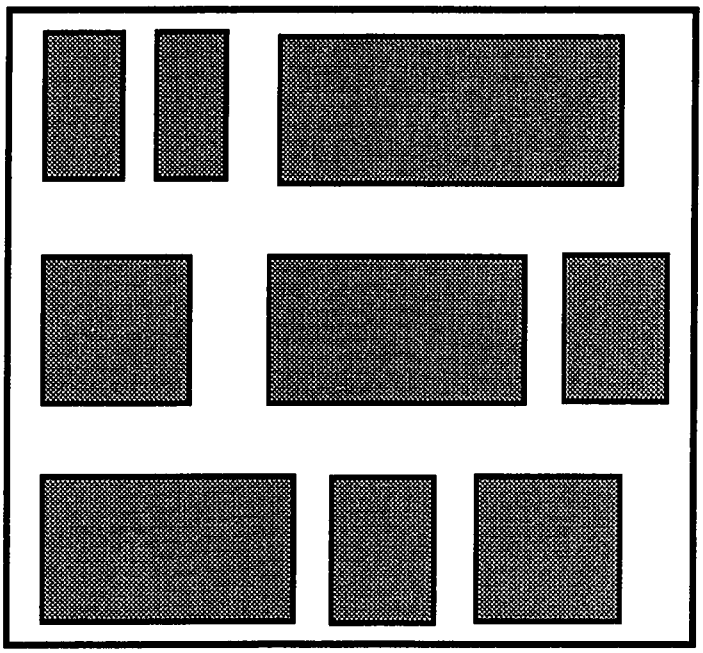


Figure 3.1: An example of a standard cell style layout.

vary.

3.1.2 Cost Function

The overall objective of placement is to facilitate routing. Other objectives are minimizing turn-around time (i.e., the time needed to finish generating the layout), minimizing overall area of the layout, and finally minimizing the total estimated wire length to be used later for routing nets. Facilitating routing objective is given the first degree of interest since placing blocks without being able to route them will result in a useless design. Chip area is important to be optimized in order to be able to put more functionality into a given chip. Minimizing the wire length will reduce the capacitive delays associated with longer nets and speed up the operation of the chip.

Turn-around Time

Turn-around time is the execution time needed for *AutoVLSI* system to generate the required layout. In many occasions, it is more important to the designer to accomplish the layout as fast as possible rather than producing a highly optimized layout nor a very short wired one. In *AutoVLSI* system, turn-around time is given a higher priority than both minimizing layout area and total wire length.

Estimating Total Chip Area

The layout is a grided two-dimensional plane that has a positive X-Y coordinate system (i.e., the lower left corner of the layout has $0, 0$ coordinate). In the general cell layout style, modules are defined to be rectangular boxes of different sizes. Each

Algorithm for finding an estimate for the chip area

1 - Find the Module m which has the maximum $h(m)+y(m)$

2 - Find the Module m' which has the maximum $w(m)'+x(m)'$

3 - Estimate area to be = $[\ h(m)+y(m)\ ] * [w(m)'+x(m)']$

Figure 3.2: Algorithm for estimating layout area.

module m would have four characteristics: w_m module width, h_m module height, x_m the X coordinate of the module when placed, and y_m the Y coordinate of the module when placed. The chip area is the sum of modules area and the spacing left between placed modules for the purpose of routing. In *AutoVLSI*, a conservative assumption that no wiring is to be done above the highest module placed on top of the layout nor wiring is done to the right of the widest module placed in the most right boarder, will allow us to find an estimate for the chip area according to the algorithm in Figure 3.2.

Figure 3.3 shows an example of m and m' in a general cell layout.

Estimation of Total Wire Length

In estimating total wire length, we assume Manhattan geometry. The estimation procedure must be as quick as possible since circuits usually contain hundreds of multi-point nets. Next we briefly describe several techniques used for wire length estimation.

The shortest route for connecting a set of pins together is *Steiner tree* (Figure 3.4 (a)). In this method, a wire can branch at any point along its length. This method is usually not used by routers because of the complexity of computing both the optimum branching points (the Steiner points). *Minimal spanning tree* connections

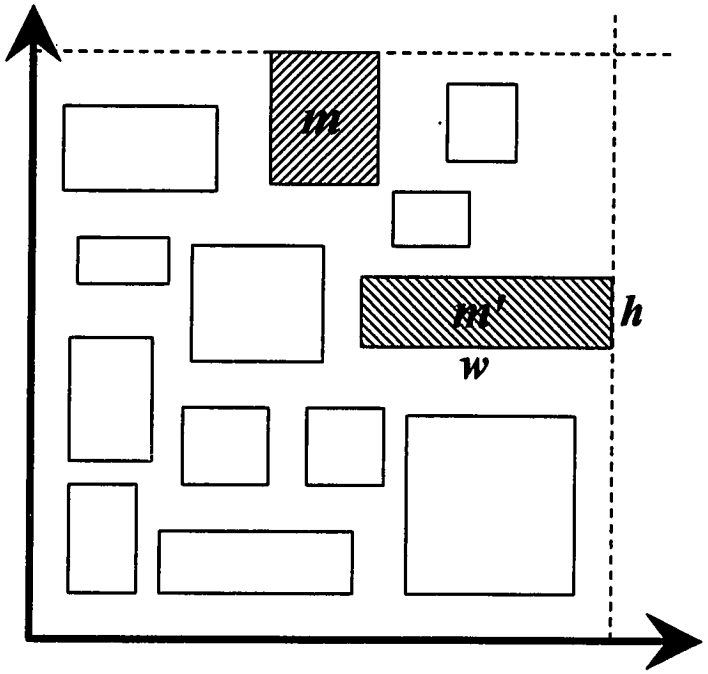


Figure 3.3: An example to illustrate m and m' in the algorithm for estimating layout area.

(Figure 3.4 (b)), allow branching only at the pin locations. Algorithms exist for generating a minimal spanning tree given the netlist and modules' coordinates. An example of a minimal spanning tree algorithm is Kruskal. Such algorithm has a polynomial time complexity and requires long time to execute. *Chain connections* technique (Figure 3.4 (c)) does not allow any branching. Each pin is connected to the next one in the form of a chain. Algorithms for accomplishing such connections are simpler to implement and faster but result in slightly longer interconnections. The most widely used technique for its simplicity, ease of implementation and speed of execution is the *semi-perimeter* approximation (Figure 3.4 (d)). The technique is to find the smallest bounding rectangle for all points to be connected by the wire, and estimate the wire length by half the length of the perimeter of this bounding rectangle.

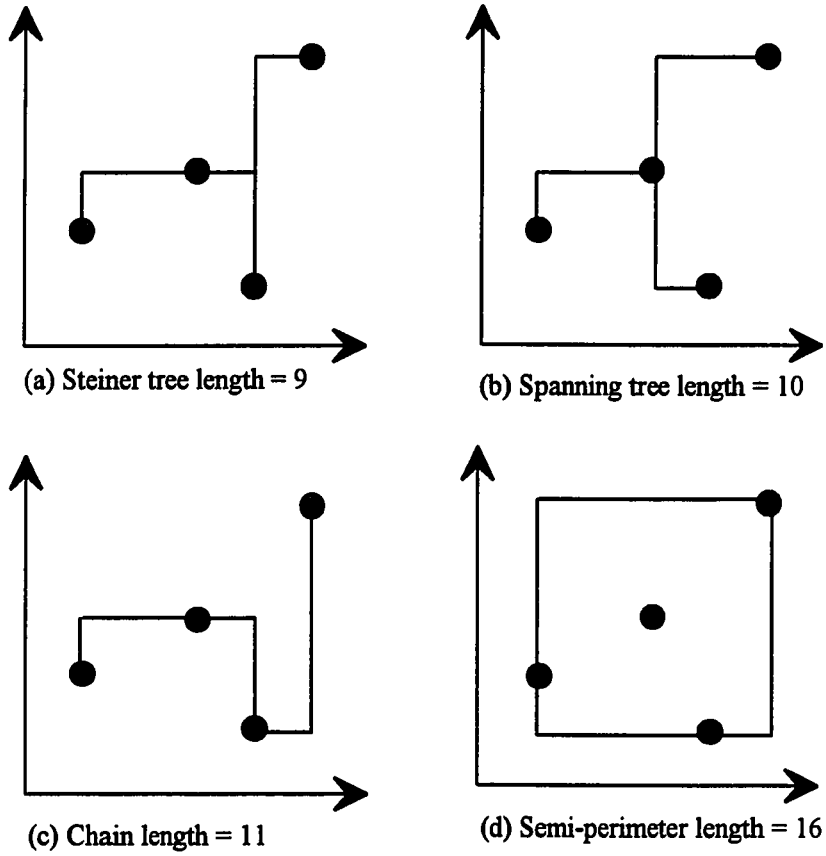


Figure 3.4: Different techniques of estimating the wire length.

For Manhattan wiring, this method gives the exact wire length for all two-pin and three-pin nets. For four-pin nets, the semi-perimeter estimate will predict a wire length 33% less than both the chain connection and spanning tree techniques. However, in practical circuits, two- and three-pin nets are most common. This technique was adopted by *AutoVLSI*.

The Cost Function

It is common practice to define a cost function or an objective function, which consists of the sum of turn-around time, total chip area, and total estimated wire length. It has been stated that all these three objectives need to be minimized in order to achieve high quality layouts in a reasonable amount of time. The cost function used by *AutoVLSI* to evaluate the generated layout quality is as follows:

$$Cost = \alpha T + \beta A + \gamma W$$

where α, β and γ are factors for turn-around time, chip area, and wire length relatively. These factors constitute the weight for each of the function objectives. They are set to 3, 2, and 1 relatively. These values indicate that turn-around time is the most expensive objective while area of the layout comes next in order. Finally, wire length is the cheapest and the objective to minimize it is given the lowest priority.

3.1.3 Complexity of the Placement Problem

The placement problem is an NP-complete problem and, therefore, can not be solved in polynomial time.

Even for the simplest placement style, namely standard-cell placement where modules are of fixed size and are to be placed in row/column fashion, there exist as many as $n!$ different layouts for placing n modules [2]. Therefore, it is not practical to try all possible layouts to find the best solution. Instead, many heuristic techniques were developed to give good solutions for the problem, not necessarily the best solution but they reduce the time of generating the layout a lot. Some of these heuristics are discussed in the following section [8].

3.2 Techniques of Placement

The placement problem involves the assignment of building blocks of the layout to specific locations. This includes the assignment of logic gates within a gate array, the placement of cells in a standard cell layout, or the placement of macro cells in a general cell layout [2], [5].

Placement algorithms may be subdivided into two basic categories: constructive placement and iterative improvement. *Constructive placement* algorithms build a placement from initial data such as sizes of the cells to be placed and the netlist (i.e., the interconnections between cells). *Iterative improvement* algorithms start with a given initial placement (which can be given by the user or obtained by a constructive placement algorithm) and modify the layout to improve its quality [5].

3.2.1 Constructive Placement Algorithms

A constructive placement algorithm constructs a solution by placing one module at a time. Two decisions have to be made and these are:

1. which module to pick from the unplaced list of modules, and

2. where to place the selected module.

Most constructive placement algorithms are based on primitive connectivity rules. Typically, a seed module is selected and placed in the chip layout area. Then another module is selected according to the connectivity between it and the placed modules (most connected first) and is placed at a vacant location close to the placed modules, such that the wire length is minimized. These algorithms are generally very fast, but usually results in poor layouts. They take a very small amount of computation time compared to iterative algorithms.

Some constructive placement algorithms are: linear placement & folding, numerical optimization, placement by partitioning, and force-directed technique. A full description of each can be found in [5], [3].

3.2.2 Iterative Improvement Algorithms

Iterative improvement algorithms produce good placement but require enormous amount of computation time. They start by a given placement solution that could be user defined, randomly built, or an output from a constructive placement algorithm, and try to improve layout cost by running several iterations.

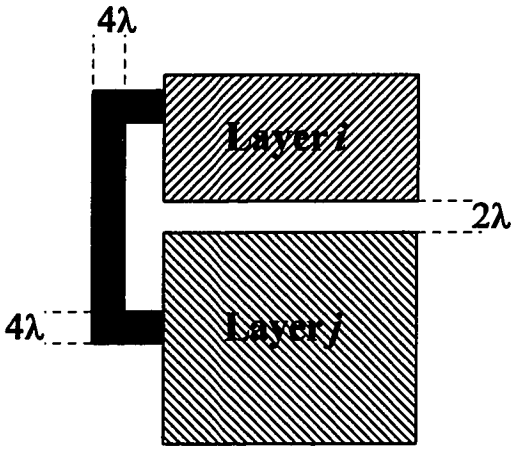
The simplest iterative improvement strategy interchanges randomly selected pairs of modules and accepts the interchange if it results in a better cost. The algorithm terminates when no improvement is observed after a given large number of trials, or after running the algorithm for some maximum given amount of execution time.

Currently popular iterative improvement techniques include simulated annealing, genetic algorithm, and some force-directed techniques.

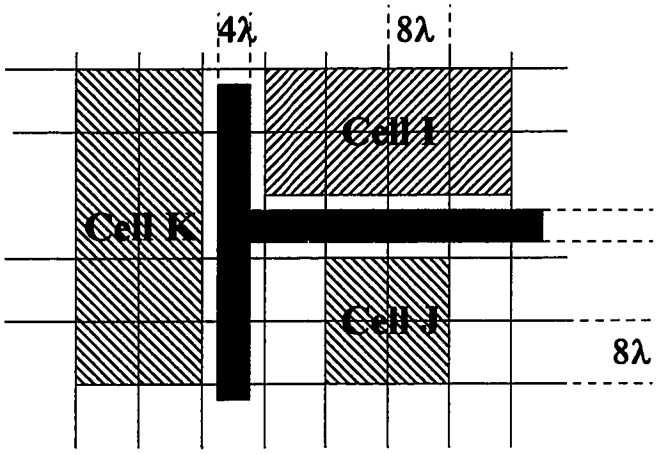
3.3 Placement in *AutoVLSI*

The placement procedure used in *AutoVLSI* system to place general cells is constructive and consists of two phases. Its objective is to compromise between the speed of generating the layout, layout area, and total wire length; plus of course assuring full routibality. The first phase is to linearly order the cells in terms of their interconnection, in order to minimize wire length. Linear ordering of cells produces a one-dimensional layout as if a matrix consisting of one row or one column. The second phase is to fold these ordered cells on the two-dimensional layout in a manner that minimizes the layout area. It does so by leaving the minimum allowed spacing between each cell and its neighbors.

How much space to leave between a module and its neighboring modules is determined by the VLSI design rules. *AutoVLSI* deals with the layout as a rectangular array of grid cells. Every module when placed will occupy one or more grid cells. Routing material is supposed to run in the space between modules to connect different pins of different modules. In *SCMOS* technology, the common factor of the minimum allowed distance between any two layers is 4λ, and the common factor of the minimum width of a routing material (metal or polysilicon) is also 4λ (Figure 3.5 (a)). Thus, we restrict our grid unit size to $(8 \times 8)\,\lambda$. This means that leaving one grid unit between any two modules will be sufficient to run one wire (whose width is 4λ) in between them while leaving a space of 4λ between the wire and its neighboring cells (Figure 3.5 (b)). From now on, the dimensions of a module are



(a) VLSI rules.



(b) Wires running between cells.

Figure 3.5: Illustration of a grid cell and VLSI design rules.

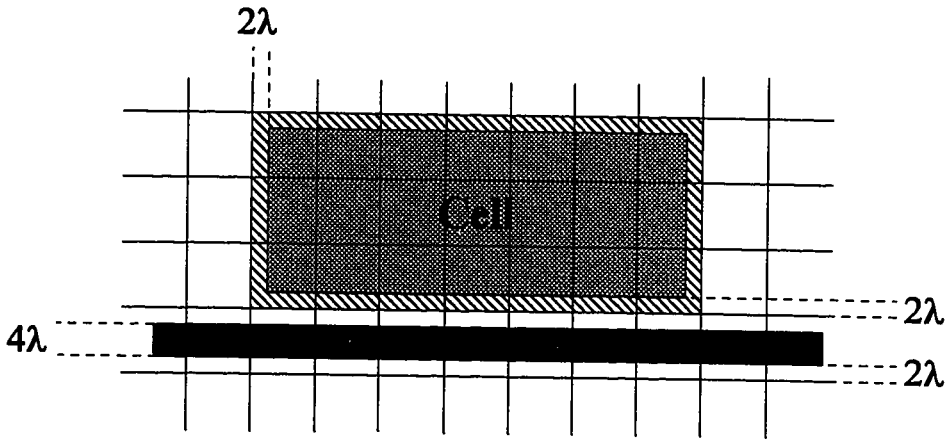


Figure 3.6: Actual size of a cell.

relative to grid cells rather than λ. If a module's width equals to 3, and height equal to 2 then it is assumed to occupy a box whose width equals 3 grid cells, and whose height equals 2 grid cells, but the actual size of the module in terms of lambda is $(28 \times 12)\,\lambda^2$, since we leave 2λ from each side of the cell to afford the 4λ spacing constraint if a wire is to pass close to it (Figure 3.6).

3.3.1 *Phase I:* Linear Ordering of Cells

Linear ordering can be defined as follows: given a circuit consisting of modules interconnected with nets, put the modules in a linear sequence such that the number of nets cut by a plane separating two adjacent modules is minimized [2], [16].

The circuit will be described in this section by a set of modules and a set of nets. A net is a set of modules which are interconnected by one wire. The algorithm starts by selecting a seed module which will be the first module in the order, then

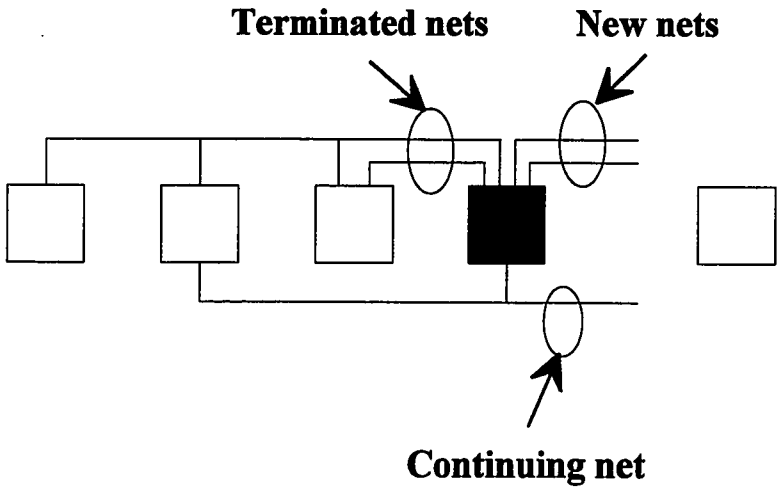


Figure 3.7: Illustration of *terminated*, *continuing*, and *new* nets.

enters a **Repeat** loop. In each iteration of this loop a gain function is computed for every module in the set of unordered modules. The module with maximum gain is selected, removed from the set of the unordered modules, and added to the sequence of ordered modules. Before explaining how a seed module is selected, and how the gain is computed, let us define the terms *terminated net*, *continuing net*, and *new net*.

A net n is said to be terminated by module m and called *terminating net* if module m was the only remaining unordered module included in the set of modules of n. On the contrary, if none of the modules in the set of modules of net n is ordered, then n is a *new net* for all its modules. If some of the modules of a net n are ordered and **more than one** unordered modules belong to the set of modules of n, then n is a *continuing net* for those remaining unordered modules. Figure 3.7 illustrates graphically these three types of nets.

Now we can define the gain function for module m to be:

$gain_m$ =number of nets terminated by m−number of new nets started by m.

When computing the gain for the unordered modules in order to select the next one in sequence, a tie (i.e., more than one module having the same gain) could exist. A tie is solved by selecting the module which terminates the largest number of continuing nets. If another tie exist, the module that is connected to a larger number of continuing nets is preferred. If we have another tie, the most lightly connected module is selected. In case of another tie, the module that has the lowest index (was encountered first in the set of unordered modules) is selected. A Pascal like description of the linear ordering algorithm is given in Figure 3.8.

The following example (taken from [2]) is used to illustrate the linear ordering algorithm.

Example 3.1 *Given the following netlist with* 6 *modules* [*M1*, *M2*, *M3*, *M4*, *M5*, *M6*] *and* 6 *nets N1*= {*M1*, *M3*, *M4*, *M6*}, *N2*= {*M1*, *M3*, *M5*}, *N3*= {*M1*, *M2*, *M5*}, *N4*= {*M1*, *M2*, *M4*, *M5*}, *N5*= {*M2*, *M5*, *M6*}, *N6*= {*M3*, *M6*}, *linearly order the cells so that the number of cut nets by a plane between any two cells is minimized. Assume M1 to be the seed cell.*

- *Solution:*

The linear ordering heuristic of Figure 3.8 will produce the following sequence [*M1*, *M4*, *M5*, *M2*, *M3*, *M6*]. At the first step, *M1* is placed in the first location of the order since it is the seed cell. Then every module gain is computed and *M4* comes to have the maximum gain, so it is placed in the second location. At the second step, we have a tie between modules *M2*, *M3*, and *M5* having all a gain equal to −1. All three cells do not terminate any net. However, module

Linear_Ordering Algorithm

```
S:=Set of all modules;
Order:=Sequence of ordered modules; (* initially empty *)
Begin
 Seed:=Select seed module;
 Order:=[seed];
 S:=S-{seed};
 Repeat
  ForEach module m in S Do
   Compute m gain as follows:
   gain = number of terminated nets by m - number of new nets starting by m;
  End ForEach
  Select the module m* with maximum gain;
  If there is a tie Then
   Select the module that terminates the largest nymber of nets;
   If there is a tie Then
    Select the module that has the largest number of continuing nets;
    If there is a tie Then
     Select the module with the least number of connections;
     If there is a tie Then
      Select the module that satisfies the previous " If" and has lowest index;
     End If;
    End If;
   End If;
  End If;
  Order:=[Order,m*];
  S:=S-{m*};
 Until S is empty;
End.
```

Figure 3.8: Linear Ordering algorithm.

M5 has the largest number of continuing nets and therefore it is selected in the second **If** statement of the algorithm and placed in the third location of the order. At the third step, *M2* has a maximum gain and placed in position 4 of the order. At the fourth step we have a tie between the remaining modules *M3* and *M6* since they both have a gain equal to 0. The first **If** statement in the algorithm fails because both terminate one net. The second **If** statement also fails because they have the same number of continuing nets, and the third **If** statement fails also because they are both connected to the same number of nets. The **Else** statement of the algorithm directs to select *M3* since it has the lower index. Finally, at the fifth step only *M6* is left and so it has the maximum gain and selected to occupy position 6 in the order. A step-by-step execution of the algorithm showing how cells were selected in order is given in Figure 3.9.

A graphical representation of the ordered cells with their interconnection is shown in Figure 3.10.

In *AutoVLSI* system, *N9.DAT* is a file that has the data about each cell and the nets it participates in. In order to achieve linear ordering for cells, we need this data in a format of nets and their participated cells. The **GNETS** software, reads in *N9.DAT* file and produces *NETS.GNE* file that follows such format. Every net, followed by the cells participating in, reserves one entry in this file. The **LOC** software reads *NETS.GNE* file, and *GP2.DAT* file which shows the type of each cell. The **LOC** software starts by sorting all cells according to their index number. Then it sequentially moves input cells of the type "4018" from the main list of cells to put them at the beginning of the sequence of ordered cells since they are to be

Step #	Module	Terminated Nets	New Nets	Gain	Continuing Nets
0	*M1**	=	*N1, N2, N3, N4*	-4	*N1, N2, N3, N4*
1	*M2*	=	*N5*	-1	*N3, N4*
	M3	=	*N6*	-1	*N1, N2*
	*M4**	=	=	0	*N1, N4*
	M5	=	*N5*	-1	*M2, N3, N4*
	M6	=	*N5, N6*	-2	*N1*
2	*M2*	=	*N5*	-1	*N3, N4*
	M3	=	*N6*	-1	*N1, N2*
	*M5**	=	*N5*	-1	*N2, N3, N4*
	M6	=	*N5, N6*	-2	*N1*
3	*M2**	*N4*	=	1	*N3, N5*
	M3	*N2*	*N6*	0	*N1*
	M6	=	*N6*	-1	*N1, N5*
4	*M3**	*N2*	*N6*	0	*N1*
	M6	*N5*	*N6*	0	*N1*
6	*M6**	*N6*	=	1	=

Figure 3.9: Execution table of Example 3.1 [2].

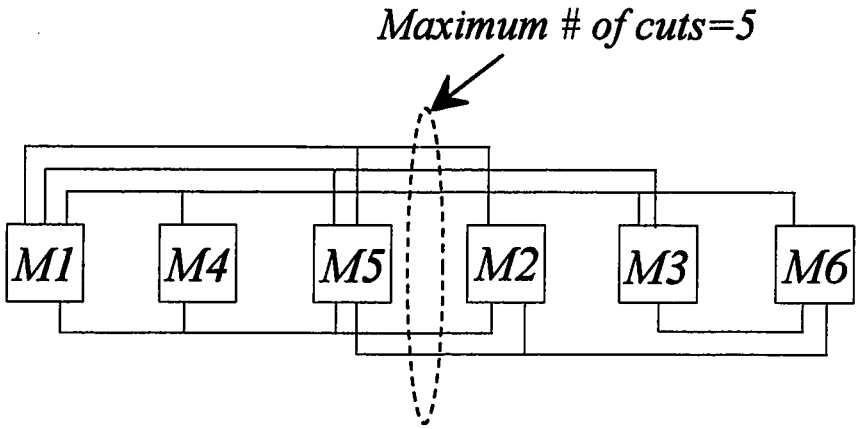


Figure 3.10: A graphical representation of the ordered cells of Example 3.1.

placed in a separate row in the layout. At the same time, it removes any output cell of the type "4013" encountered in the main list of cells to a temporarily list because they are known to be placed at the final top row of the layout. Secondly, it linearly orders remaining cells which are neither input nor output cells according to their interconnection relationship, and moves them to the sequence of ordered cells. Finally, it moves output cells sequentially from the temporarily list to the sequence of ordered cells. The ordered sequence of cells is saved in *CELLS.LOC* file.

3.3.2 *Phase II:* Folding Cells in Two-dimensional Plane

In *phase I*, cells of the circuit have been ordered in one-dimensional sequence according to their interconnection details. This will minimize the wire length needed to connect different pins of the cells. If the layout can be one dimensional, then we can directly place cells in one row or one column next to each other as they appear in the sequence. This is absolutely impractical since layouts of very small or very large aspect ratio (i.e., $\frac{length}{width}$) would result. Instead, many techniques exist for

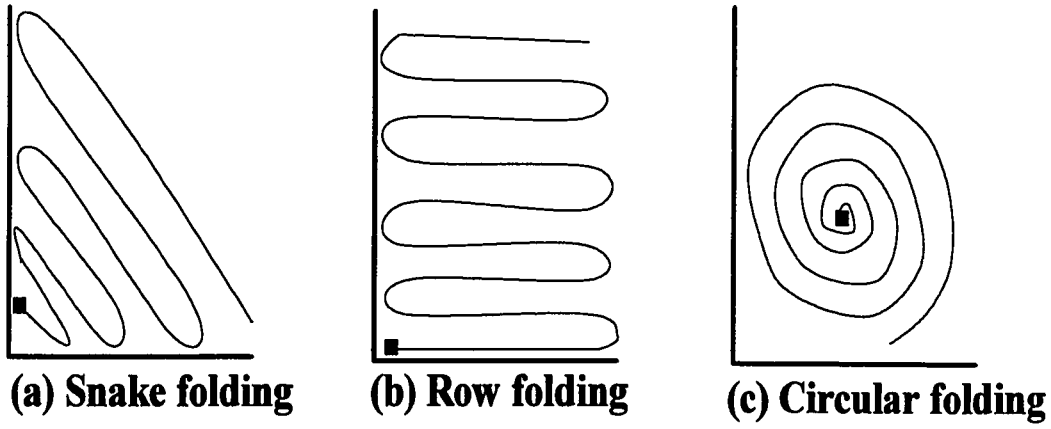


Figure 3.11: Different techniques of folding a sequence of cells.

folding this one-dimensional sequence of cells into a two-dimensional plane to form the chip layout. Some of these techniques are graphically presented in Figure 3.11.

The easiest way of folding is called row-folding and is shown in Figure 3.11 (b). This technique places cells in a *left-to-right* direction next to each other starting from the bottom of the layout until the layout maximum width (MW) is reached. Then it searches for the highest cell in that row and reconstructs another row whose bottom is the top of the highest cell. It continues placing cells in order after changing direction *(right-to-left).* Row-folding is a very fast technique and has been used successfully for standard-cell layouts (Figure 3.12). If the same technique is to be used for general-cell layouts, an enormous amount of dead space would result reflecting an increase in both chip area and wire length. Figure 3.13 shows an example of general-cell folding into a two dimensional plane using row-folding technique. The hashed boxes represent dead space. A modified version of row-folding technique does the same style of folding but takes into consideration the shape of the top face of the layout (to be called *front-face*). This *front-face* will have some gaps -according

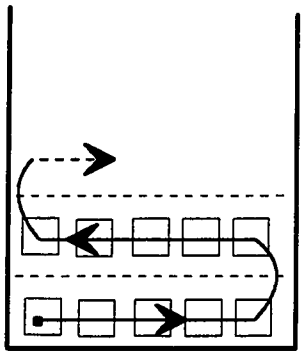


Figure 3.12: Row-folding example for standard-cell layout.

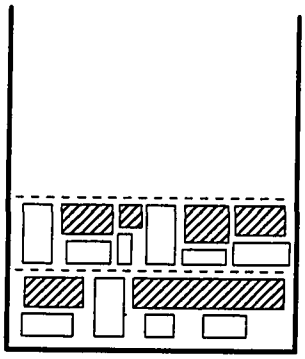


Figure 3.13: Row-folding example for general-cell layout.

to differences in cells' height- that can be filled by the coming cell in order. Such a gap could be neighboring the last placed cell and then the next cell will be placed directly without disturbing the order of cells. In other cases, the gap could be away from the neighbor of the last placed cell. Such position could increase or decrease wire length and layout area. The decision of where to place the next cell is taken according to a cost function computed for each proposed position. First, the cost of placing the cell next to the previously placed one is computed. Then, if there are any gaps where the cell can fit, the cost for placing the cell in any of these gaps is computed. The cell is placed in the position where the cost is lowest. The *front-face* of the layout is saved after every new cell is placed. The cost function is the sum of an estimate for the chip area and an estimate for the wire length.

A description of the modified Row-folding algorithm is given in Figure 3.14.

Example 3.2 *Given the following cells along with their dimensions (width, height): M1* (3,2), *M2* (1,1), *M3* (2,2), *M4* (2,3), *M5* (2,4), *M6* (2,3), *M7* (1,1), *M8* (1,1), *M9* (4,2) *and the following linear order: [M6, M8, M1, M4, M3, M7, M5, M2, M9]. Use row-folding technique to place these cells in a two-dimensional layout, then use the modified version (algorithm in Figure 3.14). Consider MW (maximum width of the layout)*= 13, *and the cost function equals the estimate of the layout area.*

- *Solution:*

1. The Row-folding algorithm is applied without any modification. The placement starts by *M6* and assigns it to location (1,1) since there should be a minimum spacing of one grid unit between cells and the left/bottom boundaries of the layout. Originally, the algorithm starts by a *left-to-right* direction

Algorithm for the Modified Row-folding technique

1 - Define MW to equal maximum width of the layout.
2 - Start placing cells in order next to each other, in the direction left to right in the most bottom row of the layout; Until MW is reached.
3 - Flip direction of placement (Right to Left ; Left to Right).
4 - Pick a position for new cell:
  4.1 - Save the Front-face of the layout.
  4.2 - Pick next cell m* in order.
  4.3 - Compute the cost of placing m* next to the last placed cell.
  4.4 - If there exist any gaps in the Front-face where m* can fit
    4.4.1 - Compute the cost of placing m* in every one of these gaps.
    4.4.2 - Place m* in the position that yields the lowest cost.
  4.5 - Else
    4.5.1 - Place m* next to the last placed cell.
5 - If there exist more cells and MW is Not reached
  5.1 - Go to 4.
6 - If there exist more cells
  6.1 - Go to 3.
7 - Quit.

Figure 3.14: Algorithm for the modified version of Row-folding technique.

of placement, so *M8* is assigned to location $(4,1)$ (i.e., to the right of *M6*) leaving one grid of space, *M1* is assigned to location $(6,1)$, and *M4* is assigned to location $(10,1)$. When *M3* is checked for placement, it appears that placing it adjacent to *M4* will exceed *MW*, so the row is locked and the direction of placement is switched to *right-to-left*. In the locked row, *M6* has the maximal height which is 3 and its *Y-coordinate* is 1. The new started row *Y-coordinate* is set to the sum of the *Y-coordinate* and the height of the highest cell in the previous row, and increased by one spacing grid. Thus the next module in order, *M3* is assigned to location $(11,5)$. Note that there is no need for spacing between a cell and the right boundary of the layout. *M7* comes in sequence and is placed to the left of *M3* leaving a space of one grid, at location $(9,5)$. Proceeding in the same manner, *M7* is assigned to location $(6,5)$ and *M2* is assigned to location $(4,5)$. *M9* has a width of 4 grid units, and does not fit in the current row, so the direction of placement is switched and a scan to the current row reveals that the highest cell is *M5* having a *Y-coordinate* equals to 5 and width equals to 4. *M9* is assigned to location $(1,10)$ leaving one grid unit of space between it and the left boundary of the layout, and one grid unit of space between it and the highest cell in the last row. An estimate for the area of the final layout can be computed according to the previous algorithm in Figure 3.8 and it is found equal to 156 grid units. The layout is shown in Figure 3.15.

2. If we apply the modified version of the Row-folding algorithm, the first row will not change. However, after placing modules *M6*, *M8*, *M1*, and *M4*, a *front-face* is saved and we encounter one gap between *M6* and the right boundary

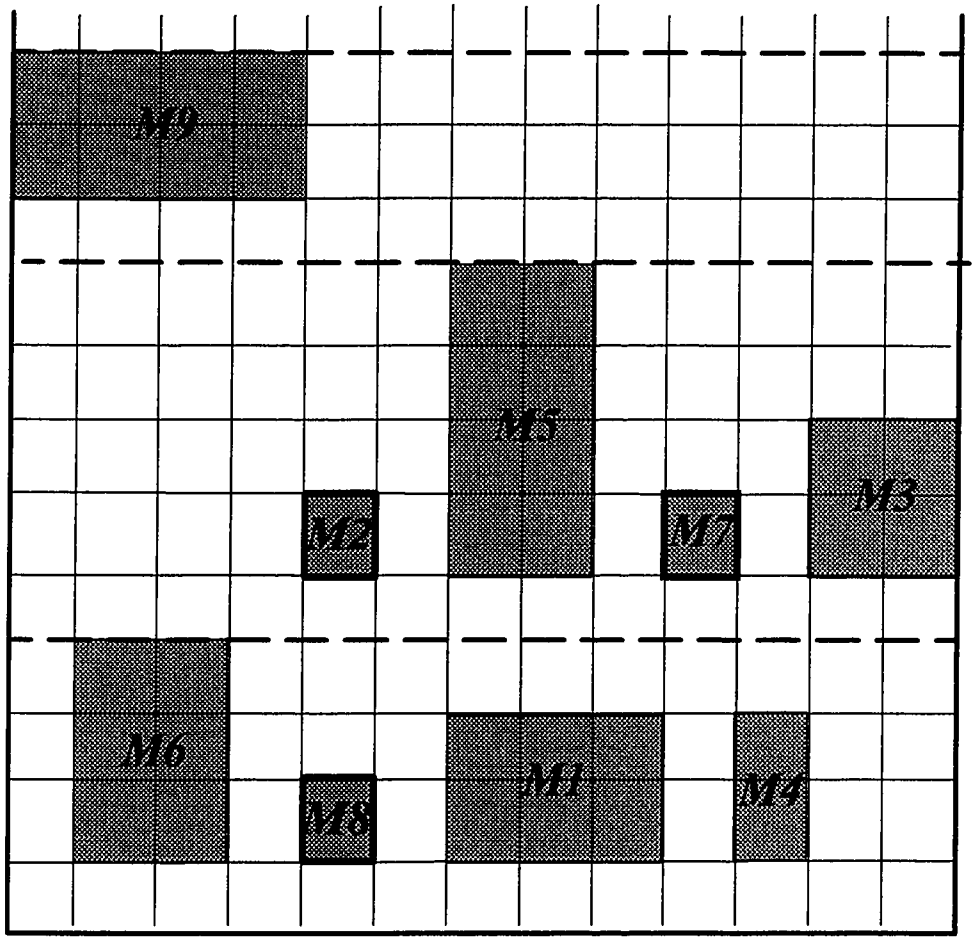


Figure 3.15: The layout of Example 3.2 generated by Row-folding algorithm.

of the layout. It appears that placing module *M3* in the gap will result in smaller chip area rather than placing it in the sequenced location $(11, 5)$. So *M3* is assigned to location $(11, 4)$ and a new *front-face* is saved. *M7* does fit in the gap between *M6* and *M3*; so it is placed next to *M3* at location $(8, 4)$. *M5* fits in the gap between *M6* and *M7* and is assigned to location $(6, 4)$. *M2* is placed in location $(4, 4)$ and a new *front-face* is saved. In this last *front-face* there is a gap between *M5* and the left boundary of the layout that is enough to hold *M9*. So, instead of placing *M9* in the sequenced location $(1, 10)$, it is inserted into the gap and assigned to location $(1, 6)$. The resulting layout is shown in Figure 3.16, and has an estimated area equals to 104 grid units.

3. For this very small example, the modified version of the Row-folding algorithm was able to save 52 grid units of the layout area which is about 33% of the layout area found by applying the original algorithm. Moreover, the space utilization $= \left(\frac{156-37}{156} \times 100\right) = 76.3\%$ of the layout generated by Row-folding algorithm, is larger than the space utilization $= \left(\frac{104-37}{104} \times 100\right) = 64.4\%$ of the layout generated by the modified version of the algorithm, which means that less space was wasted in the second layout.

It should be very clear that this modified version of the algorithm will not necessarily produce smaller layout. In many examples it may result exactly in the same layout structure and area. This is absolutely dependent on the dimensions of modules, the sequence they are encountered when being placed, and the maximum width of the layout. However, it is guaranteed not to produce a larger layout than what the original Row-folding algorithm produces.

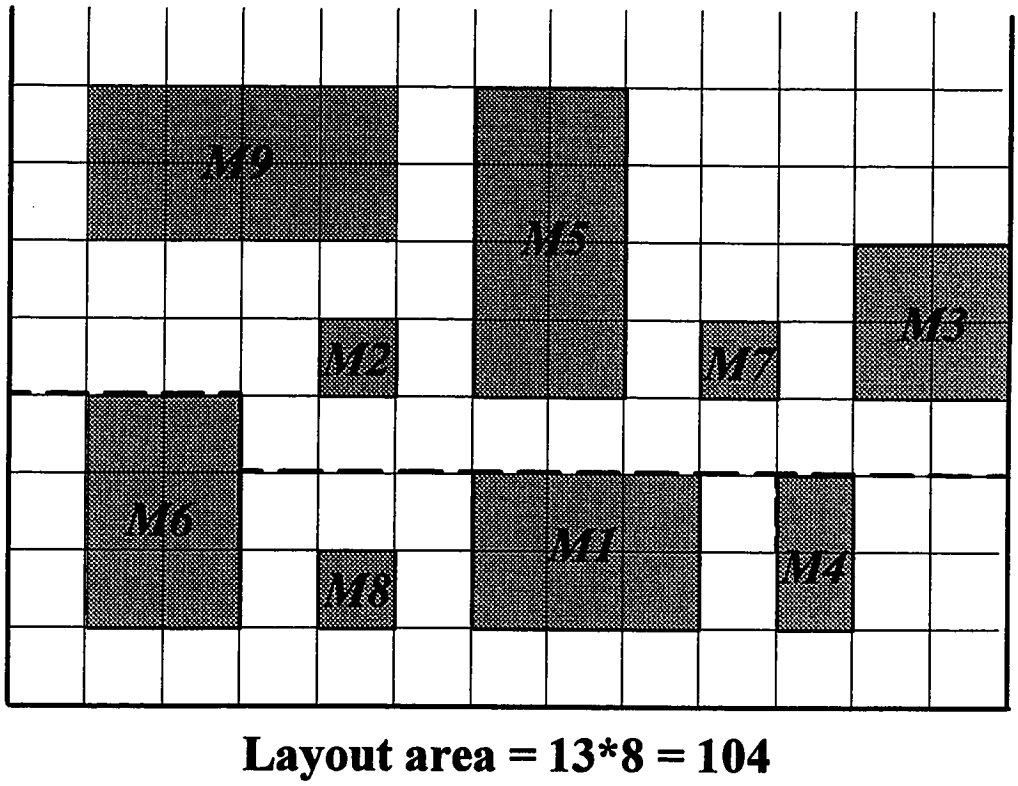


Figure 3.16: The layout of Example 3.2 generated by the modified version of Row-folding algorithm.

Chapter 4

Global Routing

4.1 Introduction

Physical design of a VLSI circuit consists of two tasks. Placing cells of the VLSI circuit in a two-dimensional plane to form the circuit layout, and connecting different sets of pins on these cells by routes of wires made of some connecting material that is usually metal or polysilicon [2]. The main objectives that must be achieved are minimizing both total wire length used for connections and total layout area.

The first objective was approached by linearly ordering the cells according to the interconnections between them such that highly interconnected cells are put in sequence close to each other. This ordering results the minimum length of wires representing these interconnections if the cells are placed in one row or one column.

Since cells of the circuit are pre-determined in the cell assignment stage and each of these cells have a fixed size and area, then minimizing the layout area is concerned by the amount of space in the layout to be used by the router to place the wires between these cells. The necessary routing space is iteratively determined

by starting with minimum space in the layout without disturbing the linear order, and then increase this space as needed to accomplish all wiring.

The layout first was created by folding the ordered cells in a row fashion using *front-face* technique while keeping the minimum spacing that VLSI design rules allow. An assumption that only one wire can pass between a cell and its neighbors yielded a minimum space of one grid unit. This assumption is absolutely impractical but yields a minimum total layout area. If we directly submit such layout to the detailed router, it will not be able to find enough space for routing all wires.

Global routing is a pre-routing stage done to predict where the detailed router is going to run wires on the layout floor. Once it predicts such information, it adds enough spacing between the circuit cells to allow the detailed router to place these wires [2], [12].

4.1.1 Definitions and New Terms

A *net* is defined to be a set of points (also called pins) on the boundary of some cells of the layout that must be electrically connected with each other [2]. These points could be inputs, outputs, ground, or power ports for a cell. When cells of *AutoVLSI* were designed, there was a restriction to put such points of the cell on the boundary of the cell and to assign one grid unit to each point. The same point could exist more than once on the cell boundary, so for each existence it is assigned one grid unit. An example of a VLSI cell (*2-input NOR*) showing how points like IN1, IN2, OUT, GND, and VCC are laid on the boundary of the cell and each is assigned one grid unit is given in Figure 4.1.

When two cells are **vertically** neighboring each other there will exist rows of

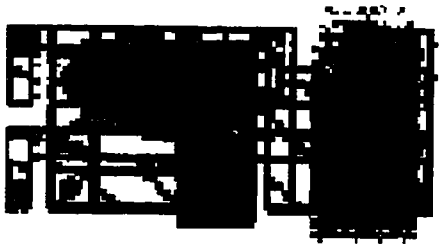


Figure 4.1: Illustration of the ports of a VLSI cell. The cell represents a *2-input NOR* gate.

space grids between the cells. These rows are defined as the *horizontal channel* between these two cells. If two cells are **horizontally** neighboring each other, then the columns of space grids between them are defined as a *vertical channel* (see Figure 4.2) [11], [13]. For layouts generated by *AutoVLSI* system, the concept of *horizontal* and *vertical* channels is eliminated. Simply, a rectangular area consisting of some empty grid units is defined to be a *channel*. A channel has both vertical and horizontal *capacity* which equals the number of grids vertically or horizontally that are on the borders of the channel. For example, the horizontal channel of Figure 4.2 has a horizontal capacity of 2 grids, and a vertical capacity of 8 grids. Wires routed on the layout are to run only in these channels and according to VLSI design rules, each wire unit is assigned to one grid. A wire segment will usually occupy some grid units that are either horizontally or vertically sequenced. Thus, the capacity of a channel is equal to the number of wire segments it can have. The horizontal channel of Figure 4.2 can have 2 horizontal wire segments, and 8 vertical wire segments. A *congested channel* is defined to be any channel whose capacity is less than the number of wire segments which are to pass through it.

If the designer knows prior to the detailed routing stage the information about

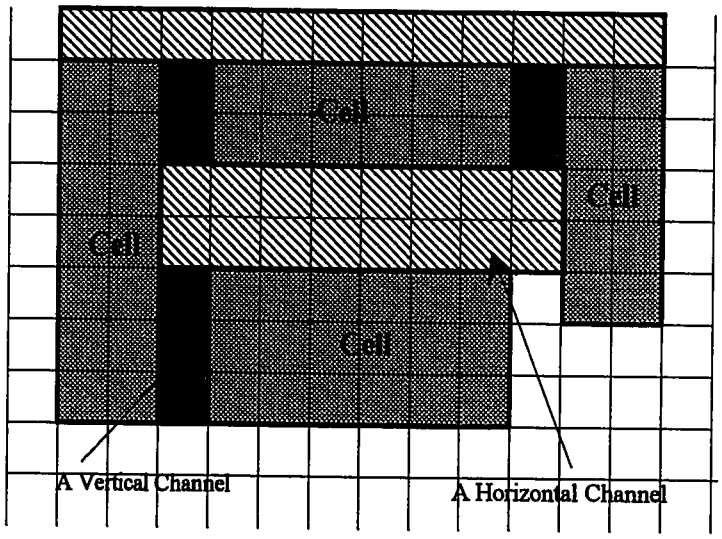


Figure 4.2: Illustration of Horizontal and Vertical channels.

the places of wire segments on the layout (i.e., which channels they are going to pass), then channels capacities can be increased as needed to eliminate any congested channels.

A channel's *vertical field* (VF) is the area bounded by the channel $Xmin$ and $Xmax$, and on top of the channel $Ymax$, while a channel's *horizontal field* (HF) is the area bounded by the channel $Ymin$ and $Ymax$, and to the right of the channel $Xmax$.

4.1.2 Tasks of the Global Routing Stage

Global routing stage can be divided into three tasks. The first task, is to determine the channels of the layout based on the information given for the cells. Each cell origin X,Y-coordinates, width, and height are known. A procedure is implemented

to identify channels according to this information. The task is called "channel identification".

The second task is to predict for each net, the channels it is going to pass by (i.e., to determine a routing plan for each net). This prediction will determine if there are any congested channels, in order to resolve them before the detailed routing stage. A channel connectivity graph for the final set of channels found in the first task is constructed. This graph gives details about how channels are connected to each other, and will be used in the "determination of wires paths" procedure.

The third task consists of resolving any congested channels. This is done by increasing channels horizontal and vertical capacities to make them able to accommodate more wires. This task is called "routing region adjustment".

4.2 *Task I:* Channel Identification

A channel is determined by its lowest left corner (llp) and its most upper right corner (urp) X- and Y-coordinates. These coordinates are formulated between brackets as follows: (llp-X, llp-Y, urp-X, urp-Y). The coordinates of these two points are used to compute both, the channel horizontal capacity (CHC) and the channel vertical capacity (CVC) as follows:

$$CHC = urp\_Y - llp\_Y \tag{4.1}$$

$$CVC = urp\_X - llp\_X \tag{4.2}$$

4.2.1 Procedure for Channel Identification

The procedure of identifying channels on a layout consists of 6 steps. It assumes cells data (i.e., X-coordinate, Y-coordinate, width, and height) are saved in a list [11], [13].

- *First step:* From each cell, two vertical lines are created and inserted in a list of vertical lines. The first vertical line X-coordinate equals the X-coordinate of the cell, while the second line X-coordinate equals the cell X-coordinate + the cell width. Each vertical line minimum Y-coordinate (Ymin) is found by scanning all cells to determine which cell the line hits first when extended **downward** from the border of the current cell. If such a cell exists, then the line Ymin will equal the sum of that cell Y-coordinate + that cell height. If an extended vertical line does not hit any cell, then its Ymin is taken to be 0. Similarly, the maximum Y-coordinate (Ymax) is found. But this time, the vertical line is extended **upward** to find the first cell it hits. In case it hits a cell, its Ymax will equal the cell Y-coordinate. Otherwise, if the V-line does not hit any cell when extended upward, its Ymax is set to ∞ (infinity). Finally, one extra vertical line whose X-coordinate equals the maximum X-coordinate of all V-lines in the list + 1, and whose Ymin is 0, and Ymax is ∞, is inserted in the list of V-lines to simulate the right border of the layout.

- *Second step:* The list of vertical lines created in the first step is scanned to eliminate any duplicated lines having the same X-coordinate, Ymin, and Ymax. Each one of the remaining V-lines will have a unique combination of X-coordinate, Ymin, and Ymax. The list of vertical lines is sorted in ascending order according to the lines X-coordinate values.

- *Third step:* Two horizontal lines (H-lines) are created for each cell in the list. The first H-line has a Y-coordinate equals to the Y-coordinate of the cell, while the second H-line has a Y-coordinate equals to the sum of the cell Y-coordinate and the cell height. Another list is created for horizontal lines coinciding with the cells top or bottom horizontal borders. A horizontal line is extended to the left and checked if it will hit any cell. If it does, then its Xmin is set to the cell X-coordinate + the cell width, otherwise its X-coordinate is set to 0. The Xmax of a H-line is found by extending it to the right to detect the first cell it hits [if any]. Its Xmax is set to the cell X-coordinate if it was hit first by the H-line, or otherwise it is set to ∞. Once all cells are scanned, one more H-line whose Y-coordinate equals the maximum Y-coordinate of all H-lines in the list + 1, and whose Xmin and Xmax are equal to 0 and ∞ is added to the list of horizontal lines to simulate the upper boundary of the layout.

- *Fourth step:* Duplicated H-lines which have the same X-coordinate and the same Ymin and Ymax are deleted from the list leaving one line to represent them. Then the list is sorted in ascending order according to the lines Y-coordinate values.

- *Fifth step:* The identification of channels starts by picking the first horizontal line in the list and scanning the vertical lines list. On the first intersection between the horizontal line and a vertical line, a channel lower left point(llp) (X,Y) coordinate is determined, while on the second intersection in order, the same channel upper right point(urp) X-coordinate is determined, and a new channel llp (X,Y) coordinate is determined. When the current horizontal line intersects with the last vertical line in the list, the last channel urp X-coordinate is determined and a new horizontal line is picked in order from the list of horizontal lines. The same operation goes for the new selected horizontal line, but this time once an intersection with a V-line is encountered, then a previous channel urp Y-coordinate is determined in addition to determining a new channel llp (X,Y) coordinate or a previous channel urp X-coordinate.

- *Sixth step:* Channels identified in the previous step are boxes resulting from intersections between H-lines and V-lines. Some of these channels will overlay cells' boxes. These channels can not be used for routing since they are already occupied by the cells, so they are eliminated from the list of channels. Any channel whose Xmin and Ymin are equal to any of the cells X- and Y-coordinates is eliminated from the list.

The pseudo-code of this algorithm is given in Figure 4.3, and is illustrated with the following example.

Example 4.1 *A layout consists of the following cells: C1* (10, 6, 3,2), *C2* (11, 1, 2, 2), *C3* (14, 1, 2, 2), *C4*(8, 1, 2, 2), *C5* (1, 6, 5, 2), *C6* (7, 6, 2, 3), *C7* (1, 1, 6, 4). *Each cell C is given along with its* ($X-coordinate$, $Y-coordinate$, *width, and*

```
Procedure Identify_Channels
 structure vetrical_line   { x, y1, y2 :integer; } vli;
 structure horizontal_line { y, x1, x2 :integer; } hli, prev_hli;
 structure channel         { x1, y1, x2, y2 :integer; } cli;

 hli=First H-line in the list;
 prev_hli=NULL;

Begin
 While (hli is not the last H-line) Do
  begin
   vli=First V-line in the list;
   While (vli exists) Do
    begin
     If ( hli intersects with vli ) Then
      begin
       If ( hli and vli are one of the 2 patterns [redacted] ) Then
        begin
         Start New Channel with Coordinates (vli->x, hli->y, , );
         If (there are any previous channels and prev_hli=hli) Then
          begin
           Update the last channel before New Channel coordinates (*, *, vli->x, *);
           Update_prev_Channels(cll->prev);
          end;
         Mark New Channel as not Complete;
        end
       Else If (there are any previous channels and prev_hli!=hli) Then
        begin
         Update previous Channels (*,*,vli->x,*);
         Update_prev_Channels(cll->prev);
        end;
       prev_hli=hli;
      end;
     vli=next V-line in the list;
    end;
   hli=next H-line in the list;
  end;
 Remove last New Channel;
End.
```

Figure 4.3: Algorithm for identifying channels.

height). Apply the channel identification algorithm given in Figure 4.3 and show graphically the resulting channels.

- *Solution:*

A V-line will have three coordinates and will be formulated between brackets as follows (X-coordinate, Ymin, Ymax). Similarly for H-lines and their coordinates will be formulated as follows (Y-coordinate, Xmin, Xmax).

1. **Creating V-lines list:** C1 will generate the first two V-lines, V1, and V2. The first (i.e., V1) will have X-coordinate equals to C1 X-coordinate = 10. When V1 is extended downward it does not hit any cell, thus its Ymin equals 0. When it is extended upward then again it does not hit any cell so its Ymax is set to ∞. V1 coordinates are $(10, 0, \infty)$. V2 X-coordinate equals the sum of C1 X-coordinate and C1 width value ($sum = 13$). When extending V2 in both directions upward or downward, it does not hit any cell; so V2 coordinates will be $(13, 0, \infty)$. C2 generates V3 whose X-coordinate equals C2 X-coordinate. Extending V3 downward does not hit any cell; so its Ymin is set to 0. However, when V3 is extended upward, then the first cell it hits is C1. Thus, V3 Ymax is set to C1 Y-coordinate. V3 has the coordinates $(11, 0, 6)$. V4 is generated by the right border of C2 and have coordinates $(13, 0, \infty)$. Each other cell will generate another two V-lines making a total of 18 V-lines. After all cells are scanned, two vertical lines V19 and V20 whose coordinates are $(0, 0, \infty)$ and $(17, 0, \infty)$ are added to the list to model the layout left and right borders. A list of all generated V-lines in sequence is given in Figure 4.4 (a).

2. **Removing duplicated V-lines and sorting:** Duplicated V-lines are V4, V8, V13, and V14. They are deleted from the list of V-lines and the list is ordered according to the X-coordinate value. The final list of vertical lines is given in Figure 4.4 (b). A graph of the layout with the last list of vertical lines is given in Figure 4.5.

3. **Creating H-lines list:** C1 will generate two H-lines. The first (H1) has a Y-coordinate equals to the Y-coordinate of C1. When H1 is extended in either direction, to the left or to the right, it does not hit any cell, so its Ymin is set to 0, while its Xmax is set to ∞. H1 coordinates are $(6, 0, \infty)$. The second H-line (H2) generated from C1 has a Y-coordinate equals to C1 Y-coordinate + C1 height. When it is extended to the left, it hits C6 and thus its Xmin is set to C6 X-coordinate + C6 width. However, when it is extended to the right it does not hit any cell and its Xmax is set to ∞. H2 has the coordinates $(8, 12, \infty)$. After all cells are scanned to generate the corresponding horizontal lines, two horizontal lines that represent the layout top and bottom borders are added to the list. The one representing the bottom border has coordinates $(0, 0, \infty)$, while the one representing the top border has X-coordinate equals to the maximum X-coordinate of all H-lines +1. Its coordinates are $(13, 0, \infty)$. A complete list of all generated horizontal lines is given in Figure 4.6 (a).

4. **Removing duplicate H-lines and sorting:** The list of horizontal lines after removing duplicated ones and sorting is given in Figure 4.6 (b), while a graph of the layout with the final list of horizontal lines is given in Figure 4.7.

V-line #	X-coordinate	min. Y-coordinate	max. Y-coordinate
1	10	0	∞
2	13	0	∞
3	11	0	6
4	13	0	∞
5	14	0	∞
6	16	0	∞
7	8	0	6
8	10	0	∞
9	1	0	∞
10	6	5	∞
11	7	0	∞
12	9	3	∞
13	1	0	∞
14	7	0	∞
15	10	0	∞
16	12	8	∞
17	4	8	∞
18	6	5	∞
19	0	0	∞
20	17	0	∞

(a) The Vertical-lines list for Channels Identification Example. List generated at Step 1.

V-line #	X-coordinate	min. Y-coordinate	max. Y-coordinate
1	0	0	∞
2	1	0	∞
3	4	8	∞
4	6	5	∞
5	7	0	∞
6	8	0	6
7	9	3	∞
8	10	0	∞
9	11	0	6
10	12	8	∞
11	13	0	∞
12	14	0	∞
13	16	0	∞
14	17	0	∞

(b) Resulting list from (a) after Step 2.

Figure 4.4: Vertical lines lists for Example 4.1.

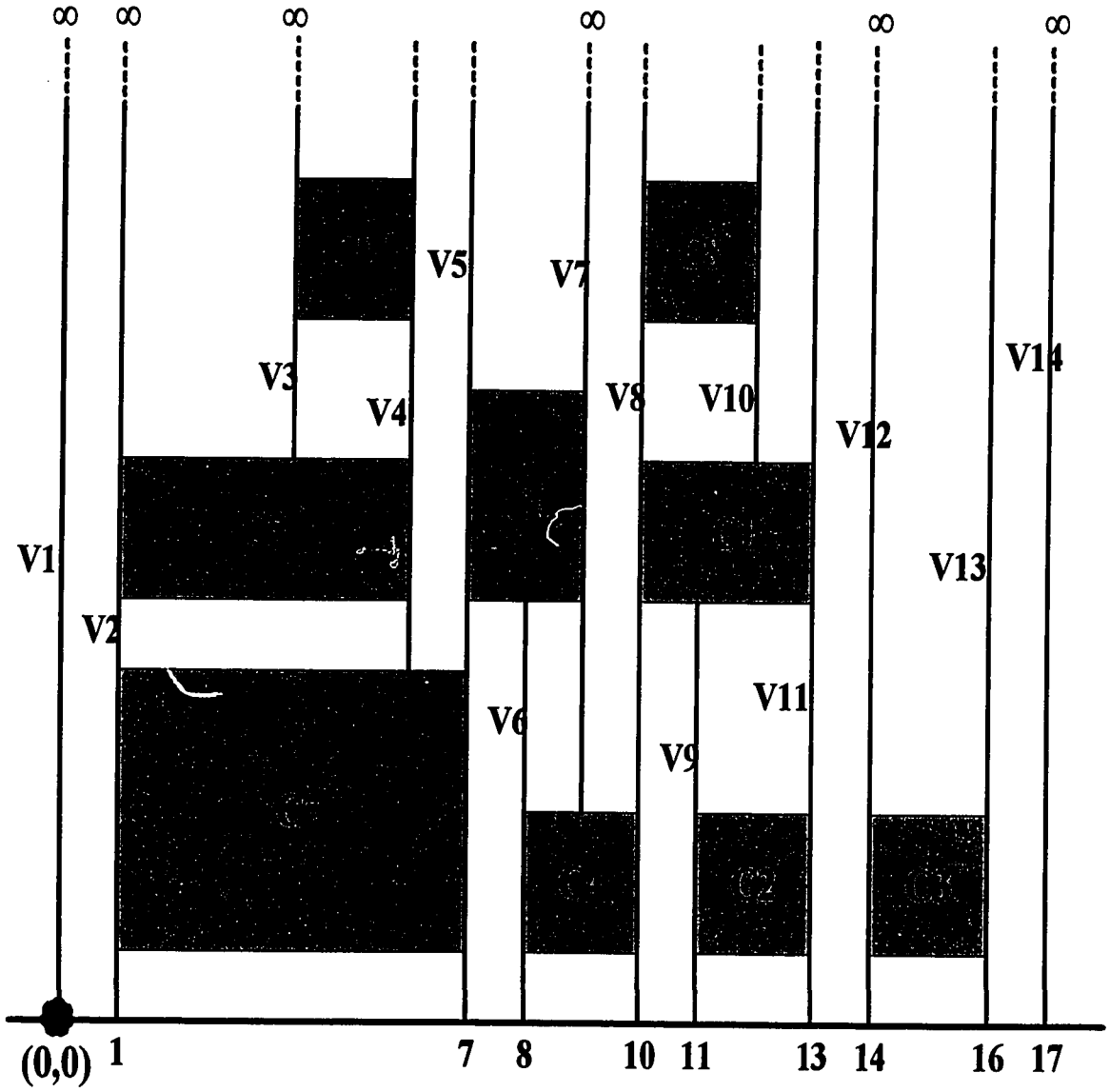


Figure 4.5: Illustration of vertical lines for the layout of Example 4.1.

H-line #	Y-coordinate	min. X-coordinate	max. X-coordinate
1	6	0	∞
2	8	9	∞
3	1	0	7
4	3	7	∞
5	1	0	∞
6	3	7	∞
7	1	0	∞
8	3	7	∞
9	6	0	∞
10	8	0	7
11	6	0	∞
12	9	0	∞
13	1	0	∞
14	5	0	∞
15	10	0	∞
16	12	0	∞
17	10	0	∞
18	12	0	∞
19	0	0	∞
20	13	0	∞

(a) The Horizontal-lines list for Channels Identification Example. Generated at Step 3.

H-line #	Y-coordinate	min. X-coordinate	max. X-coordinate
1	0	0	∞
2	1	0	∞
3	3	7	∞
4	5	0	∞
5	6	0	∞
6	8	9	∞
7	8	0	7
8	9	0	∞
9	10	0	∞
10	12	0	∞
11	13	0	∞

(b) Resulting list from (a) after Step 4.

Figure 4.6: Horizontal lines list for Example 4.1.

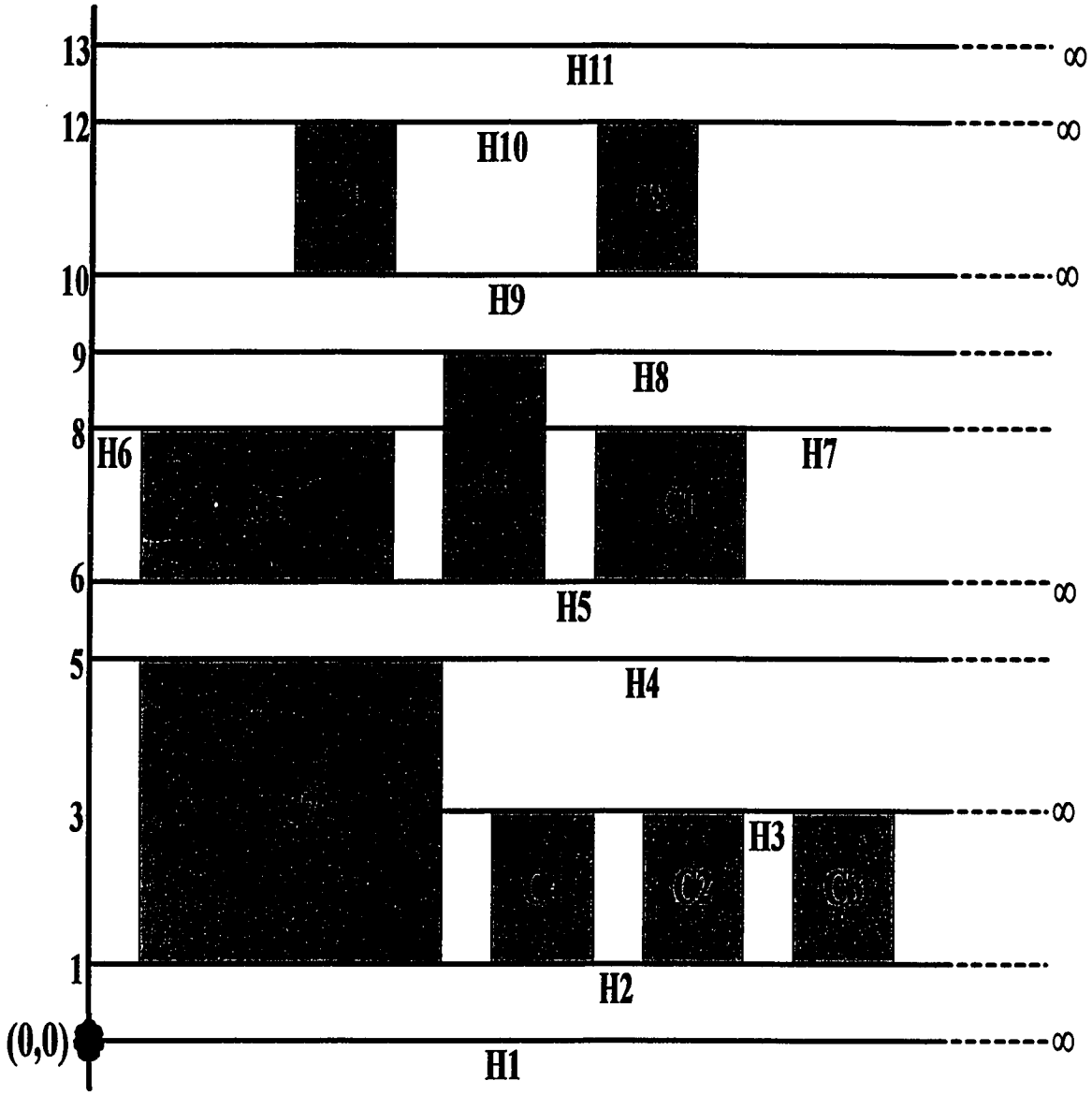


Figure 4.7: Illustration of horizontal lines of the layout of Example 4.1.

5. **Channel Identification:** The first horizontal line H1 in the final list of horizontal lines (Figure 4.6 (b)) is picked. A scan of the vertical lines in the final list of vertical lines (Figure 4.4 (b)) will reveal that H1 intersects with V1. A new channel CH1 is started with lower left point (llp) X-coordinate equals to V1 X-coordinate, and llp Y-coordinate equals to H1 Y-coordinates. So far, its set of coordinates will be $(0, 0, -, -)$. Then H1 is found to intersect with V2 in order. Since V2 is not the last V-line in the list, two actions are taken. First, CH1 Xmax is determined and is equal to V2 X-coordinate. The coordinates of CH1 are updated to be $(0, 0, 1, -)$. The second action taken is to start a new channel CH2 with coordinates $(1, 0, -, -)$. The same two actions are done when V5 is found to intersect H1. CH2 coordinates are updated to be $(1, 0, 7, -)$, and a new channel CH3 is started with the coordinates $(7, 0, -, -)$. When the final vertical line V14 is encountered, one action only is taken which consists of updating the last channel CH9 coordinates to $(16, 0, 17, -)$. H2 is now picked and the whole operation is repeated while adding some more actions as follows. V1 is the first V-line that intersect with H2. As before, a new channel (CH10) is created with the following coordinates $(0, 1, -, -)$. Now a new action is taken. All previous channels are checked. If any channel's coordinates are all determined except the last coordinate, and the third coordinate equals V1 X-coordinate, then the last coordinate of that channel is updated to H2 Y-coordinate. For the instance, for V2, there does not exist any previous channel whose third coordinate equals to 0. However, when V2 is scanned in order since it intersects with H2, CH1 third coordinate is 1 and its last coordinate is not yet determined, so it is set to H2 Y-coordinate which is 1. While

H2 is scanned with V5, V6, V8, V9, V11, V12, V13, and V14, the coordinates of all previous channels CH2, CH3, CH4, ..., etc., until CH9 are completed by V2 Y-coordinate. A new H-line is picked now and the whole operation is repeated. A list of all created channels along with their coordinates is given in Figure 4.8. The reader can notice that these cells can be obtained graphically by overlapping the layout with the vertical lines of Figure 4.5 over the layout with horizontal lines of Figure 4.7. The resulting graph is given in Figure 4.9.

6. **Eliminating occupied channels:** Finally, the list of identified channels in step 5 is scanned to eliminate those channels which overlay cells of the layout. CH11 has coordinates $(1, 1, 7, 5)$, and C7 has coordinates $(1, 1)$, so CH11 is eliminated. Similarly for CH13, CH15, and CH17 since they overlay cells C4, C2, and C3 relatively. The final set of channels is shown in Figure 4.10.

4.2.2 Merging Channels

Practical experience revealed that the number of identified channels according to the given procedure can be very large. The previous small example, the number of identified channels is equal to 80 (see Figure 4.10). For a layout with one cell only, there exists as many as 9 channels no matter what the cell coordinates or dimensions are. Since the system needs to save channels data in the memory, then it tries to reduce the number of channels as much as possible. This is done by merging adjacent channels together.

The algorithm for merging channels is shown in Figure 4.11 and is illustrated in the following text.

Channel data is assumed to be stored in a linked list structure where each item

CH #	min. X	min. Y	max. X	max. Y	CH #	min. X	min. Y	max. X	max. Y
1	0	0	1	1	46	16	6	17	8
2	1	0	7	1	47	0	8	1	9
3	7	0	8	1	48	1	8	4	9
4	8	0	10	1	49	4	8	6	9
5	10	0	11	1	50	6	8	7	9
6	11	0	13	1	51	9	8	10	9
7	13	0	14	1	52	10	8	12	9
8	14	0	16	1	53	12	8	13	9
9	16	0	17	1	54	13	8	14	9
10	0	1	1	5	55	14	8	16	9
11	1	1	7	5	56	16	8	17	9
12	7	1	8	3	57	0	9	1	10
13	8	1	10	3	58	1	9	4	10
14	10	1	11	3	59	4	9	6	10
15	11	1	13	3	60	6	9	7	10
16	13	1	14	3	61	7	9	9	10
17	14	1	16	3	62	9	9	10	10
18	16	1	17	3	63	10	9	12	10
19	7	3	8	5	64	12	9	13	10
20	8	3	9	5	65	13	9	14	10
21	9	3	10	5	66	14	9	16	10
22	10	3	11	5	67	16	9	17	10
23	11	3	13	5	68	0	10	1	12
24	13	3	14	5	69	1	10	4	12
25	14	3	16	5	70	4	10	6	12
26	16	3	17	5	71	6	10	7	12
27	0	5	1	6	72	7	10	9	12
28	1	5	6	6	73	9	10	10	12
29	6	5	7	6	74	10	10	12	12
30	7	5	8	6	75	12	10	13	12
31	8	5	9	6	76	13	10	14	12
32	9	5	10	6	77	14	10	16	12
33	10	5	11	6	78	16	10	17	12
34	11	5	13	6	79	0	12	1	13
35	13	5	14	6	80	1	12	4	13
36	14	5	16	6	81	4	12	6	13
37	16	5	17	6	82	6	12	7	13
38	0	6	1	8	83	7	12	9	13
39	1	6	6	8	84	9	12	10	13
40	6	6	7	8	85	10	12	12	13
41	7	6	9	9	86	12	12	13	13
42	9	6	10	8	87	13	12	14	13
43	10	6	13	8	88	14	12	16	13
44	13	6	14	8	89	16	12	17	13
45	14	6	16	8					

Figure 4.8: Complete list of channels identified from the layout in Example 4.1.

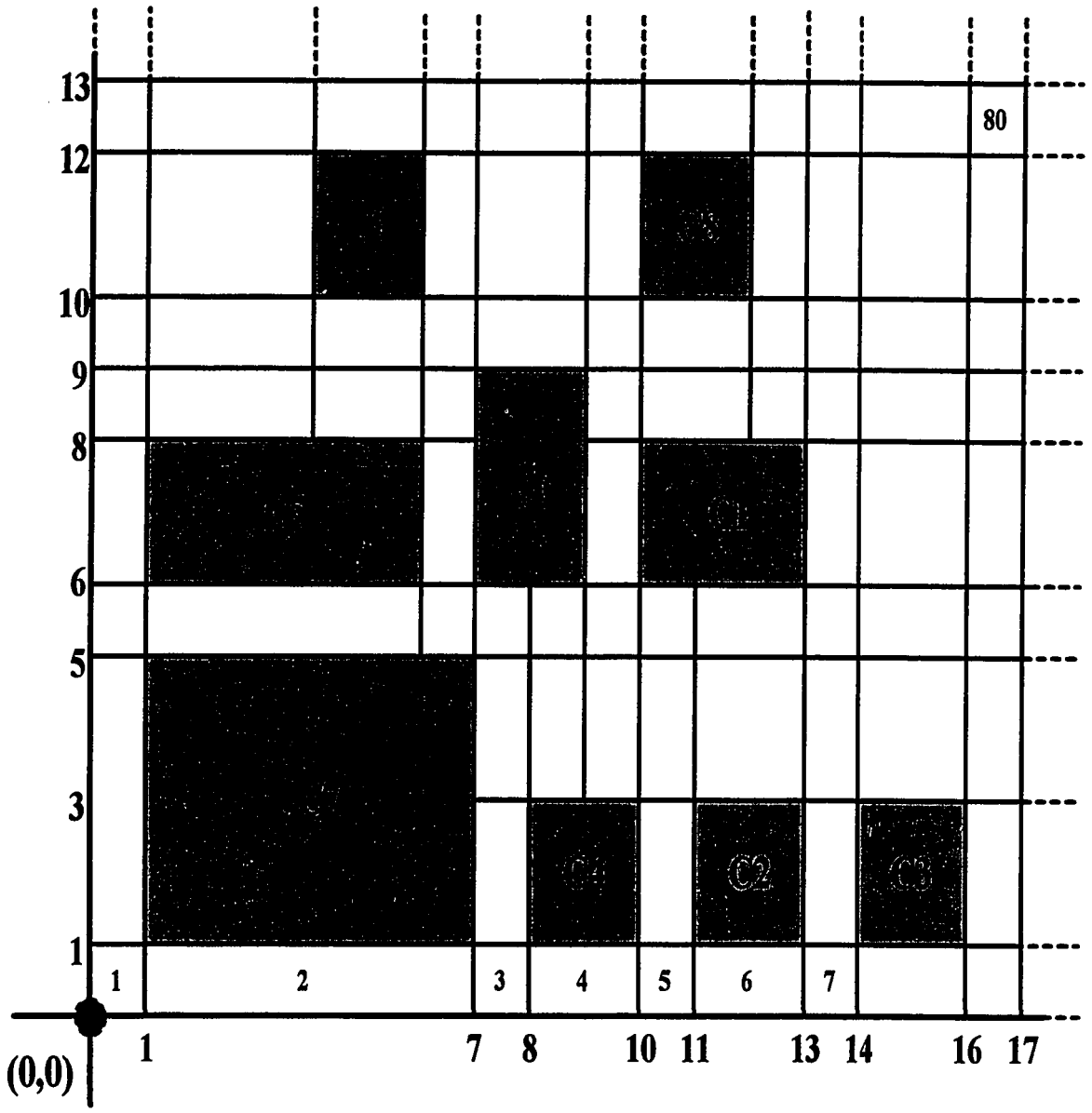


Figure 4.9: Graph of the layout in Example 4.1 showing final set of channels.

CH #	min. X	min. Y	max. X	max. Y	CH #	min. X	min. Y	max. X	max. Y
1	0	0	1	1	41	1	8	4	9
2	1	0	7	1	42	4	8	6	9
3	7	0	8	1	43	6	8	7	9
4	8	0	10	1	44	9	8	10	9
5	10	0	11	1	45	10	8	12	9
6	11	0	13	1	46	12	8	13	9
7	13	0	14	1	47	13	8	14	9
8	14	0	16	1	48	14	8	16	9
9	16	0	17	1	49	16	8	17	9
10	0	1	1	5	50	0	9	1	10
11	7	1	8	3	51	1	9	4	10
12	10	1	11	3	52	4	9	6	10
13	13	1	14	3	53	6	9	7	10
14	16	1	17	3	54	7	9	9	10
15	7	3	8	5	55	9	9	10	10
16	8	3	9	5	56	10	9	12	10
17	9	3	10	5	57	12	9	13	10
18	10	3	11	5	58	13	9	14	10
19	11	3	13	5	59	14	9	16	10
20	13	3	14	5	60	16	9	17	10
21	14	3	16	5	61	0	10	1	12
22	16	3	17	5	62	1	10	4	12
23	0	5	1	6	63	6	10	7	12
24	1	5	6	6	64	7	10	9	12
25	6	5	7	6	65	9	10	10	12
26	7	5	8	6	66	12	10	13	12
27	8	5	9	6	67	13	10	14	12
28	9	5	10	6	68	14	10	16	12
29	10	5	11	6	69	16	10	17	12
30	11	5	13	6	70	0	12	1	13
31	13	5	14	6	71	1	12	4	13
32	14	5	16	6	72	4	12	6	13
33	16	5	17	6	73	6	12	7	13
34	0	6	1	8	74	7	12	9	13
35	6	6	7	8	75	9	12	10	13
36	9	6	10	8	76	10	12	12	13
37	13	6	14	8	77	12	12	13	13
38	14	6	16	8	78	13	12	14	13
39	16	6	17	8	79	14	12	16	13
40	0	8	1	9	80	16	12	17	13

Figure 4.10: Channels list of the layout in Exampl 4.1 showing final set of channels.

Merging-Channels Algorithm

```
Structure channel { x1, y1, x2, y2 : integers
                    next : pointer to channel } CH1, Ch2;

Load channels in a linked list of the structure channel (LOC);
Repeat
 Flag=False;
 Assign CH1 to first channel in LOC;
 While (CH1 is not the last channel in LOC)  Do
  {
   CH2 = CH1 -> next;
   While ( CH2 exists )  Do
    {
     If ( CH2->x1 == CH1->x2 .and. CH2->y1 == CH1->y1
        .and. CH2->y2 == CH2->y2 )  Then
      {
       Flag = True;
       CH1->x2 = CH2->x2;
       Delete CH2 from LOC;
       CH2 = CH1 -> next;
      }
     Else If ( CH2->y1 == CH1->y2 .and. CH2->x1 == CH1->x1
              .and. CH2->x2 == CH1->x2 )  Then
      {
       Flag=True;
       CH1->y2=CH2->y2;
       Delete CH2 from the list of channels;
       CH2 = CH1 -> next;
      }
     Else
      CH2 = CH2 -> next;
    }
   CH1 = CH1 -> next;
  }
Until Flag=False;
```

Figure 4.11: Merging-Channels algorithm.

in the list has the four coordinates of a channel, in addition to a pointer that points to the next channel in the list. First channel *c1* is picked from the list of all channels and checked if it horizontally or vertically neighbors any other channel in sequence. Two channels (*ch1* and *ch2*) are considered horizontal neighbors if and only if *ch1* min.Y=*ch2* min.Y, and *ch1* max.Y=*ch2* max.Y, and *ch1* max.X=*ch2* min.X. If *ch1* min.X=*ch2* min.X, and *ch1* max.X=*ch2* max.X, and *ch1* max.Y=*ch2* min.Y, then *ch1* and *ch2* are vertical neighbors.

If the picked channel *c1* whose coordinates are ($c1x1$, $c1y1$, $c1x2$, $c1y2$) is found to neighbor any other channel (*c2*), then the two channels are merged into one as follows:

- If *c1* horizontally neighbors *c2*, then *c1* coordinates are changed to ($c1x1$, $c1y1$, $c2x2$, $c1y2$).
- If *c1* vertically neighbors *c2*, then *c1* coordinates are changed to ($c1x1$, $c1x2$, $c1y1$, $c2y2$).

Once *c1* and *c2* are merged, channel *c2* is deleted from the linked list of all channels.

After checking *c1* with all other channels, *c1* is assigned to the next channel of the linked list and the whole operation is repeated again. After scanning all channels, if there is any merging in that iteration, then the operation is repeated again and c1 is assigned to the first channel of the list. This is so because in case of any merging, the new merged channel might be capable to unify with previous channels. If no merging resulted from an iteration, then the process terminates with the new channels saved in the linked list.

Example 4.2 *For the list of* 80 *channels found in the previous example, apply the Channels Merging algorithm given in Figure 4.11 and show the list of resulting channels and a graph for the layout with the channels merged.*

- *Solution:*

1. In the first iteration of the **Repeat** loop, channel number 1 is horizontally merged(HM) with channels 2, 3, 4, 5, 6, 7, 8, and 9 consequently. Channel number 10 is vertically merged(VM) with channels 23, 34, 40, 50, 61, and 70. Channel 11 is VM with channels 15, and 26. As a matter of fact, there exist 22 merging processes in the first **Repeat** loop. They are summarized in the table of Figure 4.12 (a).

2. In the second iteration of the **Repeat** loop, only 3 merging operations exist. Namely, channels number 7 and 10, 14 and 16, and 15 and 18. The number of channels is reduced in this loop from 22 to 19 channels (see Figure 4.12 (b)).

3. In the third iteration of the **Repeat** loop, no merging exist, thus Flag keeps its False value, and the algorithm terminates. The final list of channels is shown in Figure 4.13.

The layout with the new channels is shown in Figure 4.14.

4.3 *Task II:* Constructing CCG and Determining Wires Paths

After channel identifying task, the detailed router is supposed to pick each net (a collection of wires) and try to determine a routing plan (path) for each [2], [12].

Repeat Loop #	New Channel #	Unified Channels Numbers
1	1	1 2 3 4 5 6 7 8 9
	2	10 23 34 40 50 61 70
	3	11 15 26
	4	12 18 29
	5	13 20 31 37 47 58 67 78
	6	14 22 33 39 49 60 69 80
	7	16 17
	8	19 30
	9	24 25
	10	27 28
	11	35 13 53 63 73
	12	36 44 55 65 75
	13	41 42
	14	45 46
	15	51 52
	16	54 64 74
	17	56 57
	18	62 71
	19	66 77
	20	72
	21	76

(a)

Repeat Loop #	New Channel #	Unified Channels Numbers
2	1	1
	2	2
	3	3
	4	4
	5	5
	6	6
	7	7 10
	8	8
	9	9
	10	11
	11	12
	12	13
	13	14 16
	14	15 18
	15	17
	16	19
	17	20
	18	21
	19	22

(b)

Figure 4.12: Execution table for Example 4.2.

CH #	min. X	min. Y	max. X	max. Y
1	0	0	0	1
2	0	1	1	13
3	7	1	8	6
4	10	1	11	6
5	13	1	14	13
6	16	1	17	13
7	8	3	10	6
8	11	3	13	6
9	14	3	16	13
10	1	5	7	6
11	6	6	7	13
12	9	6	10	13
13	1	8	6	10
14	10	8	13	10
15	7	9	9	13
16	1	10	4	13
17	12	10	13	13
18	4	12	6	13
19	10	12	12	13

Figure 4.13: List of channels for the layout in Example 4.2.

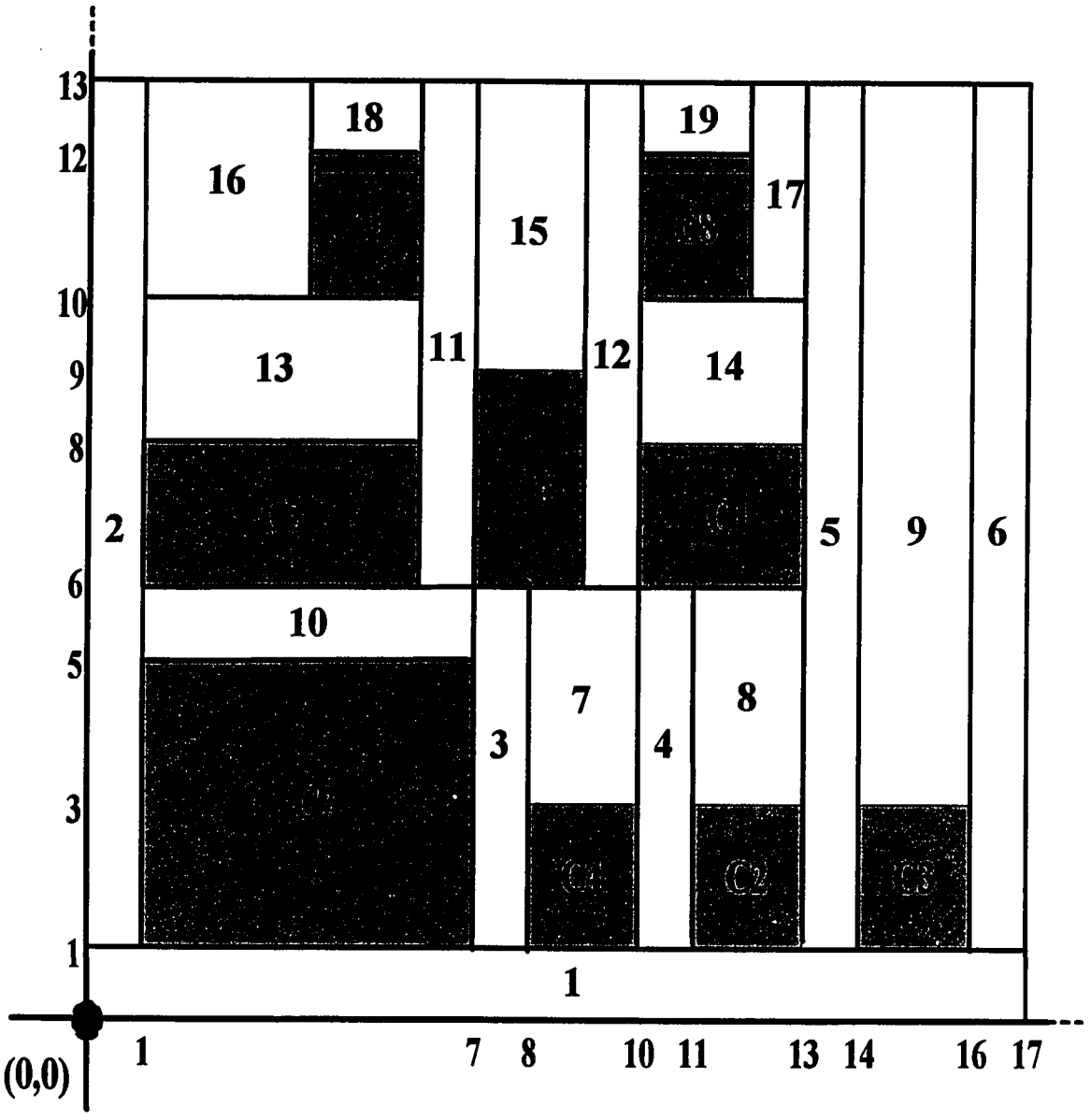


Figure 4.14: Layout of Example 4.2 showing the resulting unified channels.

A main objective is to minimize the total length of all wires of all nets. A wire connects two ports of a cell or two different cells. Each port is one grid unit layer, so the wire must connect between two different grids. In order to minimize wire length, the router usually starts by finding the shortest path between the two grids. If it fails, then it tries to use longer paths.

4.3.1 Construction of the Channel Connectivity Graph (CCG)

The final set of channels generated in *task I* is supposed to be covering all routing area of the layout. This space is between different cells and around the boundary cells of the layout. Two channels are said to be connected if they are neighboring each other, or in other words, if they share a border. If the shared border is vertical then the two channels are horizontally connected, and if it is horizontal then they are vertically connected.

The CCG is a nondirected graph that reveals the connectivity information between channels of a layout. In CCG, channels are represented by nodes, and an arc between two nodes indicates that corresponding channels are connected. Each arc is labeled either by "V" or "H" to tell if the corresponding channels are vertically or horizontally connected [2]. In our implementation, the CCG is represented as an *adjacency list structure* in which all the *"connections"* are explicitly recorded as pointers in a linked list [24]. Vertices or nodes of CCG are represented as records of a linked list of the format:

$$\boxed{Number\,|llp\_X|\,llp\_Y\,|urp\_X|\,urp\_Y\,|CHC|\,CVC\,|NextVertex|\,EdgeList}$$

Number is the channel number, followed by four fields representing the four coordinates of the channel. CHC and CVC are channel horizontal and vertical

capacities. NextVertex is a pointer to the next channel in the set of channels. EdgeList is a pointer to another linked list whose records are of the format:

$$\boxed{Label\,|Successor|\,NextEdge}$$

Label tells if the two channels are vertically or horizontally connected. It is one of the two characters "V" or "H". Successor is a pointer to a vertex that represents an arc between its own vertex and the one it is pointing to. NextEdge is a pointer to next successor in the EdgeList. To better understand this adjacency structure, an example is given in Figure 4.15 showing the adjacency structure for the given connectivity graph.

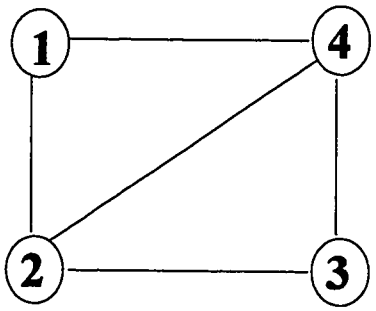
The algorithm that creates a CCG for a set of channels is given in Figure 4.16, and the following example illustrates the algorithm.

Example 4.3 *Apply the algorithm given in Figure 4.16 to the following set of channels:* {(1, 0, 5, 1), (0, 1, 5, 2), (5, 0, 6, 3), (1, 2, 5, 4)}. *Show resulting adjacency graph and CCG.*

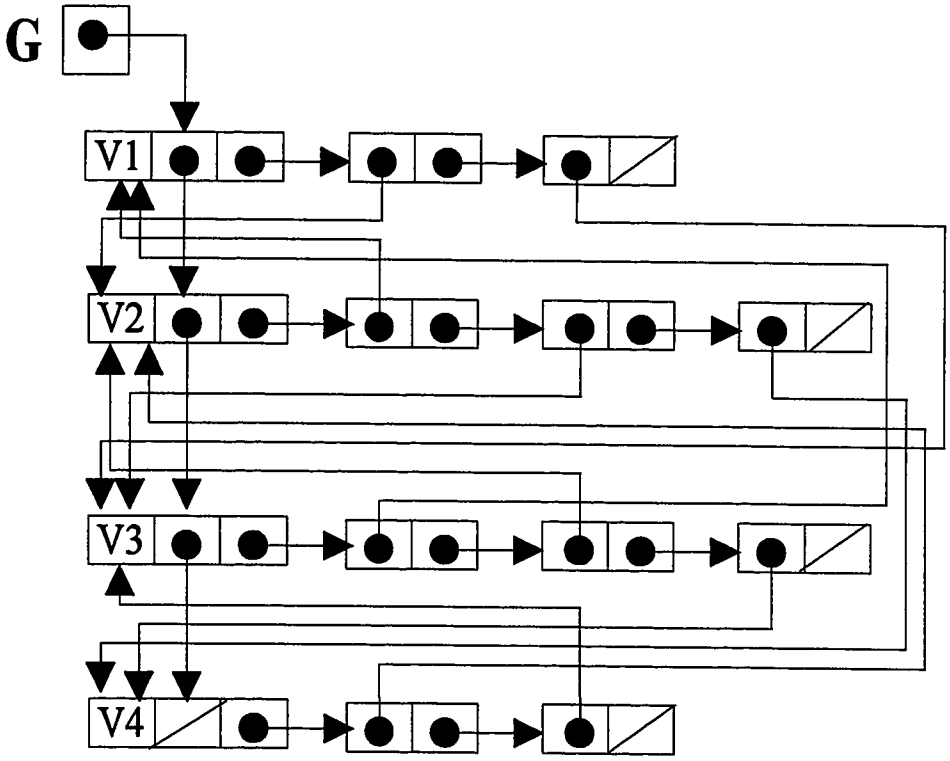
- *Solution:*

1. The channels are graphically presented in Figure 4.17 (a). The set of channels is loaded in an adjacency structure G and each channel CHC & CVC are computed and entered in each vertex record. Also each vertex EdgeList is initialized to NULL and represented graphically by a crossed box.

2. In the first iteration of the outer while loop, CH1 is assigned to channel 1, i.e., the first vertex of G. In the first iteration of the inner while loop, CH2 is assigned to channel 2. CH1 is found to be at the bottom of CH2 so L is set to



(a) A nondirected graph.



(b) Adjacency structure for the graph in (a).

Figure 4.15: Example to clarify Adjacency Structure.

Creating-CCG Algorithm

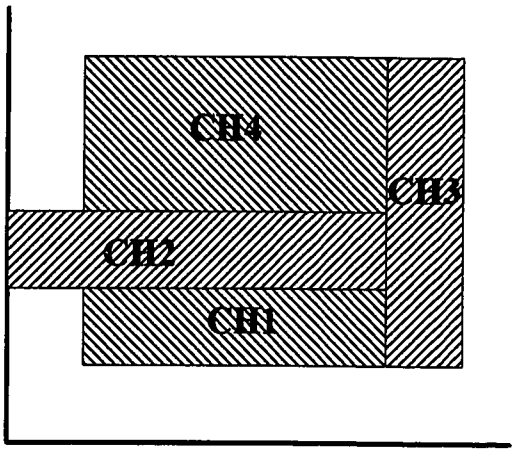
```
Load the set of channels (in any order) with thier coordinates in G adjacency
  structure;
Compute for each channel its horizontal and vertical capacity and save it in
  the channel record;
Initiate the EdgeList of each vertex of G to NULL;
CH1:=First Vertex in G;
While CH1 exists Do
 {
 CH2:=CH1->NextVertex;
 While CH2 exists do
  {
  L:=" "; Confirmed:=False;
  If CH1 is to the right or left of CH2 Then
   {
   L:="H";
   If CH1 and Ch2 have a common border Then
    Confirmed:=True;
   }
  If CH1 is to the top or bottom of CH2 Then
   {
   L:="V";
   If CH1 and CH2 have a common border Then
    Confirmed:=True;
   }
  If Confirmed Then
   {
   Append CH1 Edglist with new Edge such that:
        Successor:=CH2, Label:=L, NextEdge:=NULL;
   Append CH2 Edglist with new edge such that:
        Successor:=CH1, Label:=L, NextEdge:=NULL;
   }
  CH2:=CH2->NextVertex;
 }
CH1:=CH1->NextVertex;
}
```

Figure 4.16: Creating-CCG algorithm.

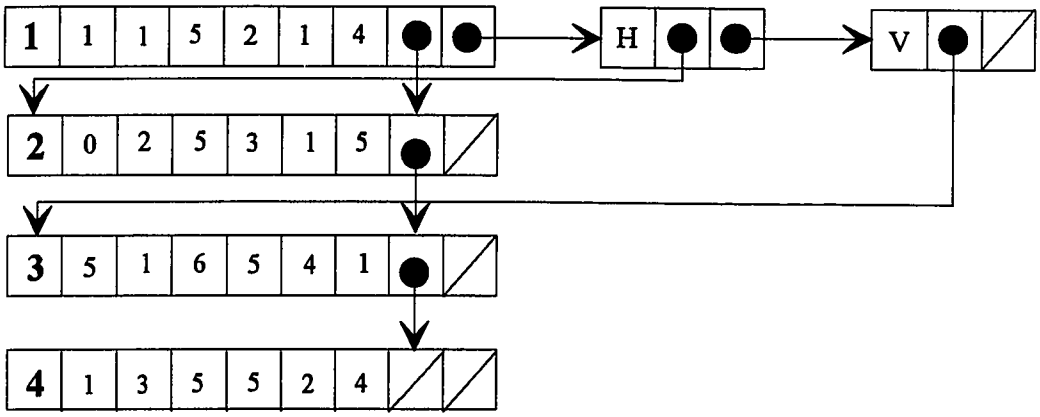
"V". They share a border so Confirmed is set to True. The EdgeList of CH1 is appended with a record whose successor points to channel 2, Label="V", and NextEdge is NULL. At the same time, CH2 EdgeList is appended with the same record but the successor is pointing to channel 1. In the second iteration of the inner while loop, CH2 is assigned to channel 3 which is found to be to the right of CH1 and sharing a border with it. This time, two records labeled with "H" are appended to CH1 and CH2 EdgeLists. The first one's successor points to channel 3, while the second one's successors points to channel 1. In the third iteration of the inner while loop, CH2 is assigned to channel 4. CH2 is found to be on top of CH1 so Label is set to "V". However, CH1 and CH2 do not share any border, thus Confirmed stays False and no records are appended to any vertex. This ends the first iteration of the outer while loop. The resulting adjacency structure is shown in Figure 4.17 (b).

3. In the second iteration of the outer while loop, CH1 is assigned to channel 2. The inner while loop starts by assigning CH2 to channel 3. Label="H" since CH2 is to the right of CH1 and Confirmed is set to True since they CH1 and CH2 share a border. Two records are added to the EdgeLists of CH1 and CH2. Then CH2 is assigned to channel 4 which is to the top of CH1. Label="V" and Confirmed is set to True. Another two records are added.

4. Finally the third iteration of the outer while loop assigns CH1 to channel 3. The inner while loop assigns CH2 to channel 4. CH2 is found to the left of CH1 so Label="H". They share a border so Confirmed=True. A record labeled "H" is added to the end of CH1 EdgeList. Its successor points to channel 4. Also another record labeled "H" is added to the end of CH2 EdgeList and its



(a) Layout of the Channels.



(b) Adjacency structure after the first iteration of the outer While loop is completed.

Figure 4.17: Illustration for Example 4.3.

successor points to channel 3.

5. CH1 is assigned to NULL so the outer while loop terminates and the algorithm ends.

The resulting adjacency structure and CCG are shown in Figure 4.18 (a) and (b).

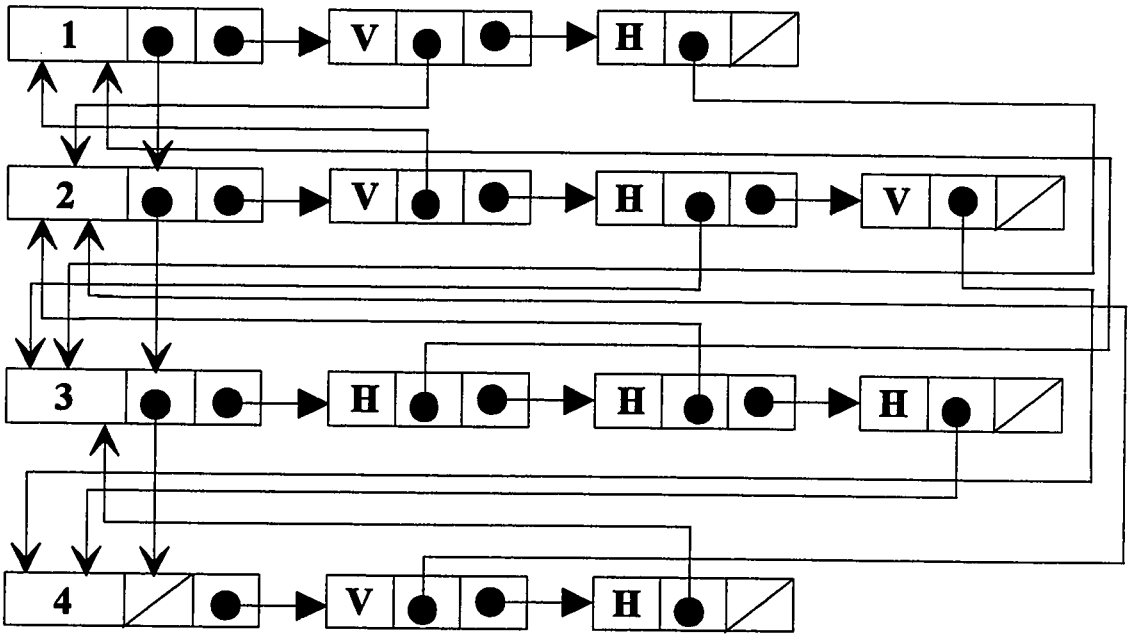
4.3.2 Determination of Wires Paths

The routing problem can be defined as follows: given a set of cells on a two-dimensional plane and a set of pairs of points, generate the coordinates of horizontal and/or vertical lines segments needed to connect every two points of a pair, such that no horizontal segments nor vertical segments overlay any cell or another horizontal or vertical line segment.

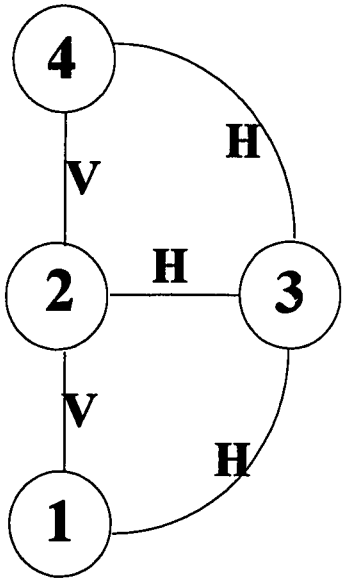
It is clear from the definition that every wire path on the layout must lie in the routing area of the layout. In previous section, a list of channels which covers all empty space in the layout (i.e., a space between a cell and its neighboring cells or a space between cells and layout boundaries) were identified. A wire path must lay on these channels in order to connect a pair of points.

Every point involved in a wire connection must be on the boundary of a cell, thus it must also be on the boundary of a channel. To predict the path of a wire connecting two points, we proceed as follows.

The system automatically finds the coordinates of each pair of points that must be connected together according to the netlist and cells coordinates resulting from the placement stage. Determination of wires paths procedure starts by assigning a channel for each point, say ch_s and ch_f standing for start and final channel consequently. Then, according to the CCG, a shortest path is found between these two



(a) Final Adjacency Structure.



(b) CCG for the final adjacency structure.

Figure 4.18: Final Adjacency Structure and CCG for Example 4.3.

assigned channels. Since the router objective is to minimize wire length, then it is expected to use this shortest path when connecting the corresponding pair of points.

For the channels of the CCG, every channel horizontal and vertical capacities were computed and saved in the record of the channel. When a wire segment passing by channel ch_i is vertical, then ch_i vertical capacity (CVC_i) is reduced by 1. On the other hand, if the wire segment is horizontal then ch_i horizontal capacity (CHC_i) is reduced by 1. The procedure adjusts these capacities starting from ch_f as follows:

- For *chf & chs*, if the point lays on the horizontal border of any of these channels then the corresponding channel CVC is reduced. Otherwise, the channel CHC is reduced.

- Assign *current* to the first channel in the shortest path between *chs* and *chf*. Thus, $current = chf$.

- If the arc between *current* and *current* $\rightarrow$ *prev* is labeled by "H" and the point on *chf* lays on a vertical border, then reduce *current.cvc*. Else if the arc is labeled by "V" and the point lays on a horizontal border, then reduce *current.chc*.

- set the variable *dir* to " ".

- While $current \neq chs$, check if the arc between *current* and *current* $\rightarrow$ *prev* is labeled by "V", then reduce *current* $\rightarrow$ *prev.cvc* and if $dir \neq$ "*V*" then reduce *current.cvc* and set $dir =$ "*V*". Else, if the arc between *current* and *current* $\rightarrow$ *prev* is labeled by "H", then reduce *current* $\rightarrow$ *prev.chc* and if $dir \neq$ "*H*" then reduce *current.chc* and set $dir =$ "*H*". Assign *current* to *current* $\rightarrow$ *prev* and loop until the "While" condition is false.

- If *dir* = "*H*" and the point on *chs* lays on a vertical border, then reduce *chs.cvc*. Else, if *dir* = "*V*" and the point lays on horizontal border of *chs*, then reduce *chs.chc*.

There exist many algorithms for finding the shortest path between a source vertex and a destination vertex in a non-directed graph. However, for CCG application, arcs do not have any weight to compute the length of path and decide upon the shortest one [24]. A path length will be measured by the number of vertices on the path. In case more than one path have the same number of nodes, another criteria is considered and that is to select the path which has minimum number of segments. A line segment is counted when an arc label connecting two vertices of a path is changed from the label of the arc that was connecting the previous two vertices. In other words, a path that has the minimum number of switching direction is preferred. The reason for this will be explained in the following chapter.

The algorithm of the above-outlined procedure is given in Figure 4.19. The syntax for reducing a variable by one is analogue to that in the C language (i.e., $--$). A small example consisting of three cells and three pairs of points is given below to illustrate the procedure.

Example 4.4 *Given the following pairs of points that must be connected on the layout of Figure 4.20,* [(4, 1), (8, 4)], [(4, 4), (8, 2)], [(2, 1), (2, 5)]*; and the following set of cells along with their coordinates and dimensions* {$C1$(1, 1, 2, 5), $C2$(7, 1, 5, 2), $C3$(1, 4, 2, 5)}*, show a list of the merged channels of the layout and create the CCG for these channels. Then use the algorithm in Figure 4.19 to predict the paths for the wires connecting each pair of points. Show a final list of all channels along with their new capacities values.*

Predicting-Wires-Paths Algorithm

```
chs = GetChannel(p1x,p1y);
chf = GetChannel(p2x,p2y);

If ( (p1x,p1y) lays on Horizontal_Border of chs ) Then
  { chs.vw--; }
Else
  { chs.hw--; }

If ( (p2x,p2y) lays on Horizontal_Border of chf ) Then
  { chf.vw--; }
Else
  { chf.hw--; }

current = ShortestPath(chs,chf);

If ( current and current->prev are Vertically connected &&
   (p2x,p2y) lays on Horizontal_Border of chf ) Then
  { chf.hw--; }
Else If ( current and current->prev are Horizontally connected &&
       (p2x,p2y) lays on Vertical_Border of chf ) Then
  { chf.vw--; }

dir = ' ';
While ( current != chs ) Do
  {
   If ( current and current->prev are Vertically connected ) Then
     {
      current->prev->vw--;
      If ( dir != 'v' ) Then
        { current->vw--; }
      dir = 'v';
     }
   Else   /* current and current->prev are Horizontally connected */
     {
      current->prev->hw--;
      If ( dir != 'h' ) Then
        { current->hw--; }
      dir = 'h';
     }
   current = current->prev;
  }

If ( dir='H' && (p1x,p1y) lays on Vertical_Border of chs ) Then
  { chs.vw--; }
Else If ( dir='V' && (p1x,p1y) lays on Horizontal_Border of chs ) Then
  { chs.hw--; }
```

Figure 4.19: The algorithm of Predicting-Wires-Path procedure.

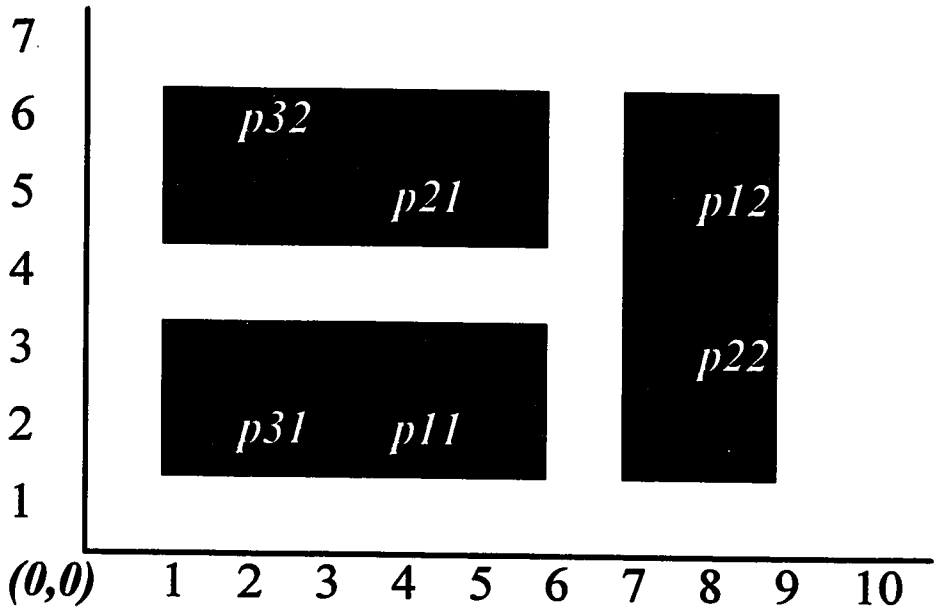


Figure 4.20: The layout of Example 4.4.

- *Solution:*

1. Identifying Channels and Merging Channels procedures are applied to the layout. Resulting channels are graphically presented in Figure 4.21, and a list of these channels along with their computed capacities is given in Figure 4.22.

2. Applying the Creating CCG procedure yields the non-directed graph in Figure 4.23.

3. For the first pair of points [$p11$(4,1), $p12$(8,4)], it is found that $p11$ lies on a vertical border of channel number 1, so $ch_s = ch1$ and chs VC is reduced, while $p12$ lays on a horizontal border of channel number 4, so $ch_f = ch4$ and chf HC is reduced. On the CCG, we find the shortest path between chf and chs. The path has one arc only which is labeled by "V", and since $p12$ lays on a vertical border of chf, then neither of the "If" statements before the

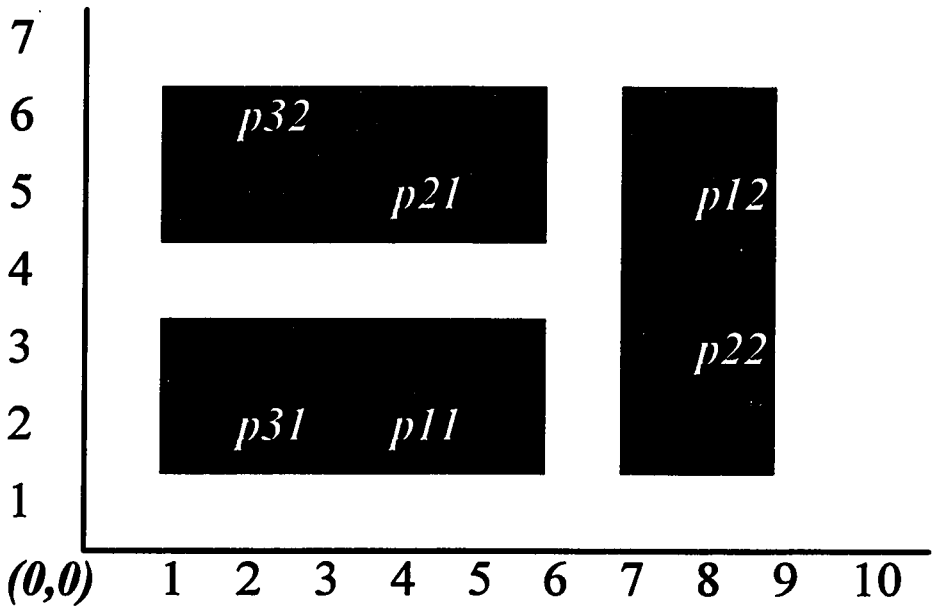


Figure 4.21: Graphical representation of the channels of the layout in Example 4.4.

CH #	min. X	min. Y	max. X	max. Y	CHC	CVC
1	0	0	10	1	1	10
2	0	1	1	7	6	1
3	6	1	7	7	6	1
4	9	1	10	7	6	1
5	1	3	6	4	1	5
6	1	6	6	7	1	5
7	7	6	9	7	1	2

Figure 4.22: Channel list for the layout in Example 4.4.

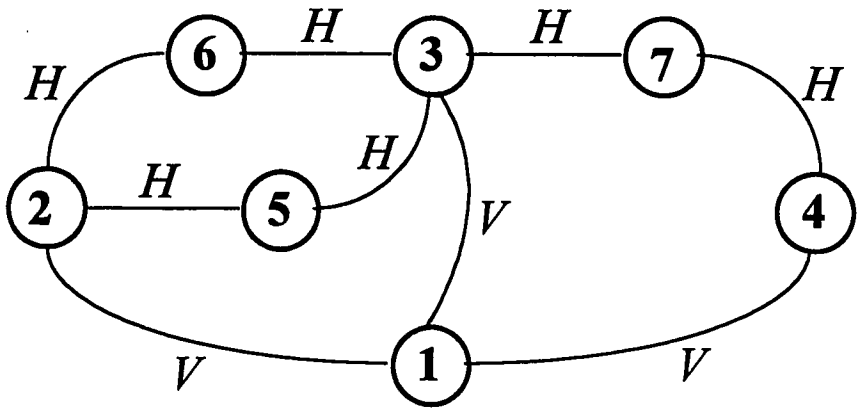


Figure 4.23: Channel Connectivity Graph for the layout in Example 4.4.

"While" loop are executed. *dir* is set to " " and the "While" is entered with *current* = *chf*. The arc between *current* and *current* → *prev* is labeled by "V" and *dir* ≠ "*V*" so both *chf* and *chs* VC's are reduced and *dir* is set to "V". *current* is set to *current* → *prev* which is *chs*, and the algorithm exits the "While" loop. *dir* = "*V*" and *p*11 lays on a horizontal border of *chs* so *chs* HC is reduced.

4. For the second pair of points [*p21*(4,4),*p22*(8,2)], we see that *p*21 lays on a horizontal border of channel number 5, so *chs* = *ch*5 and *chs* VC is reduced. *p*22 lays on a vertical border of channel number 4. So *chf* = *ch*4 and *chf* HC is reduced. There exist three paths on the CCG between *chf* and *chs* each of shortest length. These paths are: [4,1,2,5], [4,1,3,5], and [4,7,3,5]. Tracing the labels of the arcs in every path reveals that both the first and the second paths have a change in the direction while the third path has no changes, so the third path is chosen as the predicted path. The first arc in this path is labeled by "H", and since *p*22 lays on a vertical border of *chf*, then *chf* VC is reduced.

dir variable is set to " ", and the "While" loop is started with *current* = *chf* and *current* → *prev* = *ch*7. The first arc between *chf* and *ch*7 is labeled "H" and *dir* ≠ "*H*", so both *chf* and *ch*7 HC's are reduced. *dir* is set to "H" and *current* = *current* → *prev*. In the second iteration of the "While" loop, *current* = *ch*7 and *current* → *prev* = *ch*3. The arc in the path between *ch*7 and *ch*3 is also labeled by "H" as its predecessor arc (since *dir* = "*H*"), so this time the HC of *ch*3 only is reduced. In the third iteration, *current* = *ch*3 and *current* → *prev* = *ch*5. The arc between them is labeled by "H" so the HC of the *ch*5 only is reduced. *current* is set to *current* → *prev* = *chs*, so the algorithm exits the "While" loop. *dir* = "*H*" and since *p*21 lays on a horizontal border of *chs*, then neither of the "If" statements after the "While" loop are true and no further reduction is done.

5. For the last pair of points [*p31*(2,1),*p32*(2,5)], we see that *p*31 lays on a horizontal border of channel number 1, so *chs* = *ch*1 and *chs* VC is reduced. *p*32 lays on a horizontal border of channel number 6. So *chf* = *ch*6 and *chf* VC is reduced. There exist two paths on the CCG between *chf* and *chs* each of shortest length. These paths are: [6,2,1], [6,3,1]. Tracing the labels of the arcs in every path reveals that both paths have one change in the direction so the system chooses any one of them. Let us assume that the first path was chosen as the predicted path. The first arc in that path is between *chf* and *ch*2, and is labeled by "H". However, *p*32 lays on a horizontal border of *chf*, so neither of the "If" statements before the "While" loop are true. The "While" loop is entered with *current* = *ch*6 and *current* → *prev* = *ch*2 after *dir* is set to " ". Both *chf* and *ch*2 HC's are reduced and *dir* is set to "*H*". In the

CH #	CHC	CVC
1	-1	6
2	5	0
3	5	1
4	3	-1
5	0	4
6	0	4
7	0	2

Figure 4.24: List of channels in Example 4.4 after applying Predicting-Wires-Paths algorithm.

next iteration, the arc between $current = ch2$ and $current \rightarrow prev = chs$ is labeled by "V". $ch2$ VC is reduced and because $dir \neq$ "V", then chs VC is also reduced. $current = current \rightarrow prev = chs$, and the "While" loop is exit. $dir =$ "V" while $p31$ lays on a horizontal border of chs, so chs HC is reduced.

6. A final list of all channels showing resulting capacities values is given in Figure 4.24.

4.4 *Task III:* Routing Regions Adjustment

After all wires paths have been determined in *task II*, the system is now supposed to provide the required spacing in terms of channels CHC & CVC. The new channel table determines how much every channel CHC & CVC must be in order for each channel to accommodate expected wires routes.

A channel whose $CHC = h$ is less than zero, must be expanded vertically by $|h|$ grid unit(s). Similarly, a channel with $CVC = v$ where $v < 0$, must be expanded

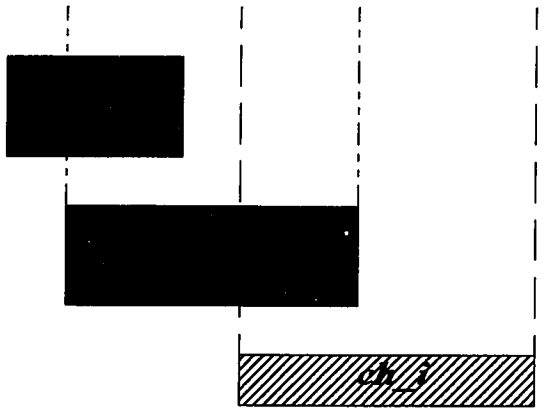
horizontally by $|v|$ grid units. This operation of expanding channels as required by *Task II* is called adjusting the layout.

4.4.1 CHCs Adjustment

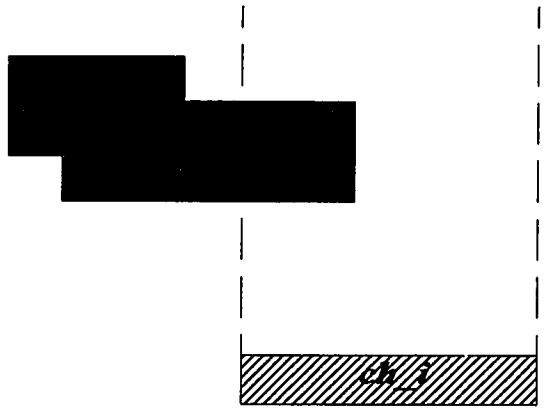
If a channel ch_i is found to have a negative CHC($= h$) after *task II*, then this channel must be expanded vertically by inserting $g = |h|$ row(s).

Insertion of rows is done by pushing all cells above ch_i upward by g grid unit(s). Pushing a cell upward by g units is done by adding g value to the cell Y-coordinate. Which cells must be pushed is an important issue and is carried as follows. If a cell Y-coordinate is greater than $ch_i(y) + ch_i(h)$ and interferes ch_i VF, then the cell is pushed and its *move* flag is set to **true**. This guarantees all cells in the VF of ch_i to be pushed up. But what about cells that are not in the VF of ch_i but in the VF of a cell c_i that has been pushed up? Pushing c_i could probably make it overlap it with other cells interfering with its VF but not interfering with ch_i VF. See for example the cell c_2 in Figure 4.25. Graph *(a)* shows the positions of *chi*, c1 and c2 before pushing c1 while graph *(b)* shows their positions after pushing cell c1 by 2 grid units.

As can be see in Figure 4.25 (b), pushing c1 only because it interferes with *chi* VF will result in an overlap between c1 and c2. To avoid such overlapping, the *pushing procedure* is done iteratively as follows. Once a cell *ci* is found to need *pushing* because it interferes with *chi*, its new Y-coordinate is saved, its *move* flag is set to **true**, and this cell becomes the active object for *pushing*. All other cells are checked if they interfere with *ci* VF using *ci* original Y-coordinate value. Now any cell interfering with *ci* VF is pushed-up by g units, and becomes the active object



(a) Positions before pushing-up.



(b) Positions after *C1* is pushed-up by 2 Grid units.

Figure 4.25: An example to illustrate the pushing operation.

after setting its *move* flag to true.

By setting the *move* flag of each cell, the procedure avoids pushing the same cell more than once even though it interferes with the VF of more than one active object.

4.4.2 CVCs Adjustment

The same terminology works when a channel ch_i is found to have a negative CVC($= v$) after *task II.* However, this time the channel must be expanded vertically by inserting $g = |v|$ column(s).

Insertion of columns is done by pushing all cells to the right of ch_i right-ward by g grid unit(s). Pushing a cell right-ward by g units is done by adding g value to the cell X-coordinate.

Again, which cells must be *pushed* is decided by checking which cells interferes with ch_i HF. If a cell does interfere with *chi* HF then it is *pushed*, its *move* flag is set to **true**, its new X-coordinate is saved, and it becomes the active object for *pushing.*

The algorithm for adjusting channels is given in Figure 4.26. It illustrates how to check mathematically if a cell interferes with an active object's vertical or horizontal fields.

Example 4.5 *Apply the algorithm given in Figure 4.26 to adjust the channels in the layout of example 4.4 according to the resulted channels CHC & CVC values. Show a list of the cells with their new coordinates.*

- *Solution:*

After cells data (i.e., X & Y coordinates, heights and widths) and channels data

Procedure Adjust_Layout

```
struct CELLS { x, y, nx, ny : Integer;
               h, w         : Integer;
               moved        : Boolean;
               next         : Pointer to next
                              element in CELLS; }

struct CHANNELS { x1, y1, x2, y2 : Integer;
                  hc, vc         : Integer;
                  next           : Pointer to next element
                                   in CHANNELS; }
Begin
 - Load Cells data in CELLS structure {nx = x, ny = y};
 - Load Channels data in CHANN structure;
 - Clear_Moved();
 - Assign ch to first channel in CHANNEL structure;
 - While ( ch )
     If (ch.hc < 0)
       Push_Cells_UP(ch.x1, ch.y1, ch.x2, ch.y2, - ch.hc);
     ch = ch.next;
 - Update y coordinate for every cell from ny;
 - Clear_Moved();
 - Assign ch to first channel in CHANNEL structure;
 - While ( ch )
     If (ch.vc < 0)
       Push_Cells_Right(ch.x1, ch.y1, ch.x2, ch.y2, - ch.vc);
     ch = ch.next;
 - Update x coordinate for every cell from nx;
END.
```

Figure 4.26: Adjust-Layout algorithm. Continued in Figure 4.27.

```
Procedure Clear_Moved()
begin
 - Assign c to first cell in CELLS structure;
 - While ( c )
     c.moved = False;
end;

Procedure Push_Cells_Up(x1, y1, x2, y2, g)
begin
 - Assign c to first cell in CELLS structure;
 - While ( c )
     If ( (c.moved=False) && (c.x < x2) &&
          (c.x+c.w > x1) && (c.y >= y2) )
       c.moved = True;
       c.ny = c.ny + g;
       Push_Cells_Up(c.x, x.y, c.x+c.w, c.y+c.h, g);
     c = c.next;
end;

Procedure Push_Cells_Right(x1, y1, x2, y2, g)
begin
 - Assign c to first cell in CELLS structure;
 - While ( c )
     If ( (c.moved=False) && (c.y < y2) &&
          (c.y+c.h > y1) && (c.x >= x2) )
       c.moved = True;
       c.nx = c.nx + g;
       Push_Cells_Right(c.x, x.y, c.x+c.w, c.y+c.h, g);
     c = c.next;
end;
```

Figure 4.27: Continuation of Figure 4.26.

(i.e., X & Y coordinates and CH & CV capacities) are loaded, the variable *moved* in all cells records is set to **False** and the algorithm starts the first while loop. The first channel that has a negative HC is $ch1$, and the procedure **Push_Cells_Up** is called with the following list of parameters: (0,0,10,1,1). The last parameter is the absolute value of $ch1$ HC. This procedure will start checking for all cells if any interferes with $ch1$ VF. It will detect that $C1$ interferes, so its *ny* coordinate is updated to become 2 and its *moved* variable is set to **True**. Now the procedure **Push_Cells_Up** is called recursively with a new list of parameters, that is: (1,1,6,3,1). The coordinates now are for cell $C1$ which was pushed in the first iteration. This time the procedure will start all over checking for all cells if any interferes with $C1$ VF. It will detect that $C3$ does, so $C3$ *ny* coordinate is updated and it becomes 5 and its *moved* variable is set to **True**. Again the procedure is called recursively with a new list of parameters (4,1,6,6,1) which stand this time for $C3$ coordinates. There does not exist any cell that interferes with $C3$ VF, so the procedure terminates and backup one step. Again, no other cells interfere with $C1$ VF, and the procedure backup one more step to continue checking for more cells interfering with $ch1$. It will detect that $C2$ does interfere with $ch1$ VF, so its *ny* coordinate is updated to 2 its *moved* variable is set to **True,** and recursively the procedure is called with the coordinates for $C2$ as its passing parameters. No cells interfere with $C2$ VF, so the procedure backup one step and continue searching for other cells interfering $ch1$ VF. Again, it will detect that $C3$ interferes but since its *moved* variable is set to **True,** then no action is taken and finally the procedure terminates. Back to the while loop, the algorithm

CELL #	*x*	*nx*	*y*	*ny*
1	1	1	1	2
2	7	7	1	2
3	1	1	4	5

Figure 4.28: The new coordinates for cells in Example 4.5 after applying Adjust Layout algorithm.

will exit since only *ch*1 was found to have a negative HC.

The algorithm next steps are to copy all cells *ny* values to the cells *y* coordinates, reset all cells *move* variable to **False**, and then to start the second while loop. The first channel that has a negative VC is *ch*4. The procedure **Push_Cells_Right** is called with the following list of parameters: (9,1,10,7,1). The last parameter is the absolute value of *ch*4 VC. This procedure will start checking for all cells if any interferes with *ch*4 HF. It will detect that no cell interferes, and the procedure terminates. *ch*4 is the only channel with negative VC, so the algorithm will exit the second while loop without any changes to cells coordinates and they all will retain the same value for their *move* variable which is **False**. A list of the cells new coordinates is shown in Figure 4.28.

Chapter 5

Detailed Routing

Detailed routing is the most time consuming stage between all stages of the process of automatic design of VLSI layouts. The task of detailed routing (routing for short) is to find precisely paths on the layout floor, on which conductors that carry electrical signals are run.

The large number of nets for most designs and large number of constraints make routing a difficult task. Moreover, routing highly depends on placement stage [2], [8].

5.1 Maze Routing

A class of general purpose routing algorithms which assumes the layout as a grided two-dimensional plane are named as *maze routers.* All ports, wires, and edges of bounding boxes that enclose the cells are aligned on the grid. The size of grid cells is defined such that wires belonging to different nets can be routed through adjacent cells without violating the width and spacing rules of wire. In *AutoVLSI* system,

grid size was set to $(8 \times 8)\lambda$ (refer to Chapter 2 for more details).

The most widely known maze routing algorithm for finding a path between two points on a grid that has obstacles is Lee algorithm [2], [18], [23]. When trying to route a path r for example, the obstacles are the cells, and other paths that were found on the layout ahead to routing r.

Lee algorithm guarantees finding a path between two points if one exists. In addition, if more than one path exists, it guarantees to pick the shortest one, which satisfies the third objective of the system, that is minimizing total wires length. The algorithm has three phases which are explained in the following subsection.

5.1.1 Phases of Lee Algorithm

Lee algorithm connects a pair of points at a time. It does so through the following three phases:

Phase I: Filling

The first phase is called filling or wave propagating because grids of the layout are filled with integers in a circular way analogous to waves resulting from dropping a stone in a still lake. First, the pair of grid cells to be connected are labeled by S and T. Then, a "while" loop is started with the variable $i = 1$ as follows. At iteration i, every unoccupied grid at Manhattan distance i from S is labeled by i. Then i is increased by one and the algorithm loops until on the j^{th} step one of the following is valid:

- The grid cell T is reached; or

- T is not reached at step j, but there are no empty grids adjacent to the grids filled with $j-1$; or
- T is not reached and j equals M, where M is an upper bound on the layout height and width in terms of grid cells.

If grid cell T was reached in step j during the filling phase, then there is a path between S and T points whose length is j. However, if at step j there were no empty grids adjacent to cells labeled with $j-1$, then the required path does not exist on the layout and thus could not be found. If $j = M$ where M is the upper bound on the path length while T is not reached, then again a path on the current layout does not exist. The process of filling is illustrated in Figure 5.1.

Phase II: Retrace

The second phase of Lee algorithm is called the *retrace phase.* The actual shortest path is found in this phase as follows. If grid cell labeled with T was reached in the filling phase at step i, then surely there exists an adjacent grid whose label is $j-1$. Likewise, there will be a grid labeled with $j-2$ adjacent to that labeled by $j-1$, and so on until a grid whose label is 1 which will be adjacent to grid cell S. The path is formed from those consecutive grid cells. For the example in Figure 5.1, since the target cell T was reached in the 8^{th} step, then there must be a grid adjacent to T whose label is 7. Likewise, there will exist a grid whose label is 6 next to that labeled with 7. By tracing the numbered cells in descending order from T to S, the desired shortest path is found. The grid cells of the retraced path are shaded in Figure 5.2.

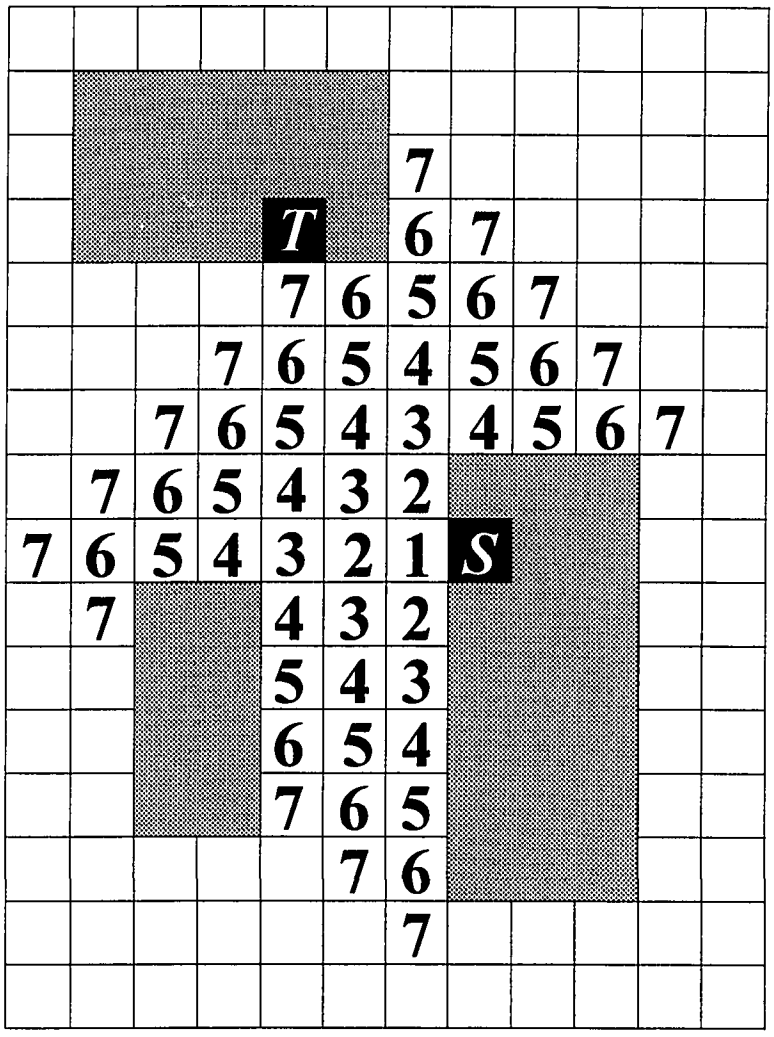


Figure 5.1: Small layout example to illustrate the filling phase in Lee algorithm.

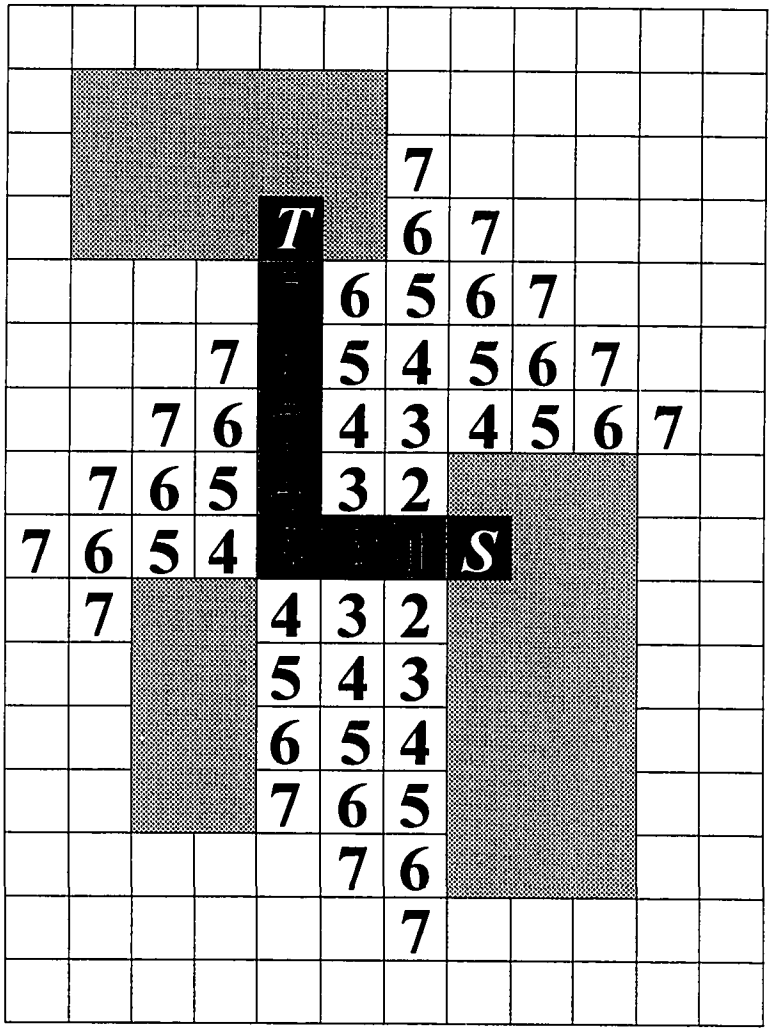


Figure 5.2: Retrace phase for the example in Figure 5.1.

Phase III: Label Clearance

In this phase, all labeled grid cells except those used for the path just found, are cleared for subsequent interconnections. The grid cells used for the path just found become obstacles in the layout in addition to cells of the layout and previously found paths.

5.1.2 Routing Layers

Routing layers is the number of conducting materials used to build the routes on the layout. In *AutoVLSI* system, two materials are used for routing, namely: *polysilicon*, and *metal*.

The concept of layers means that a piece of one layer is electrically isolated from a piece of another layer even if they were placed on top of each other. Thus, having more than one layer for routing facilitates the operation to a great extent.

For *AutoVLSI* system, *polysilicon* material is used to build vertical routes, while *metal* is used for horizontal ones. This means that horizontal and vertical routes are electrically isolated even if they cross each other. If the same route has one horizontal and one vertical segments that must be connected together, a *polycontact* is created at their corner of intersection. Figure 5.3 illustrates the concept of layers.

5.1.3 Not Found Paths

As mentioned above in the first phase of Lee algorithm, there are two cases where a path could not be found on the layout. The first is a result of blocking the path and can be solved, while the second is because of an upper bound on the layout height and width and this has no solution but to increase this upper bound [23].

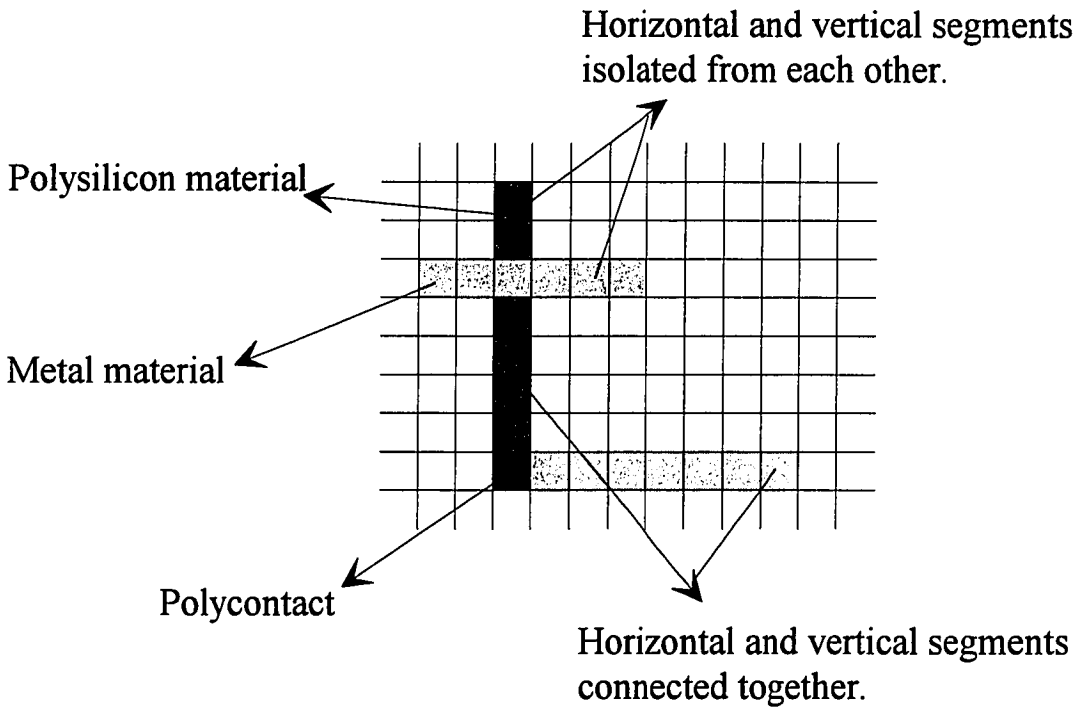


Figure 5.3: Illustration of the two layers used for routing.

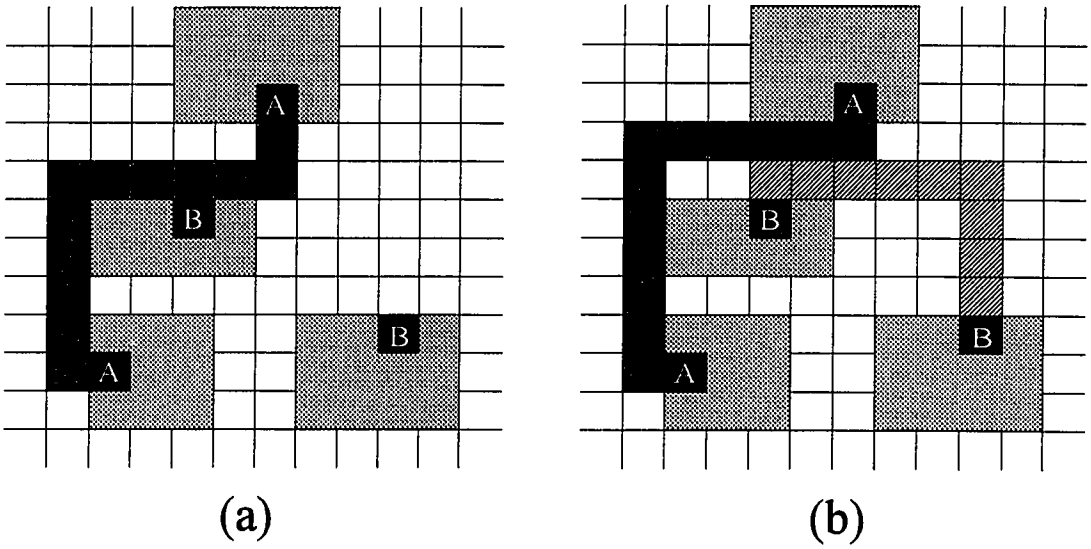


Figure 5.4: Path blocking example solved by switching the order of nets when routed.

A path is said blocked when, at step i of the filling phase, T is not reached and there are no more adjacent grid cells to grids labeled by $i-1$. There are two reasons for path blocking. The first reason is when a previous path blocks the port of a cell which is the start (S) or termination (T) point for another net. This is graphically illustrated in Figure 5.4 (a). This kind of blocking is usually solved by switching the order of the nets when routing, such that the blocked one is routed first. It can be seen in Figure 5.4 (b), that when path B is routed ahead to path A, blocking is solved.

The second reason for blocking a path, is a result of inadequate space. In Figure 5.5 (a), path B was blocked because path A used all the space between the two blocks. Switching the order of routing the two paths will result in the blocking of path A instead. The solution in such case is to increase the spacing between the two blocks as seen in Figure 5.5 (b).

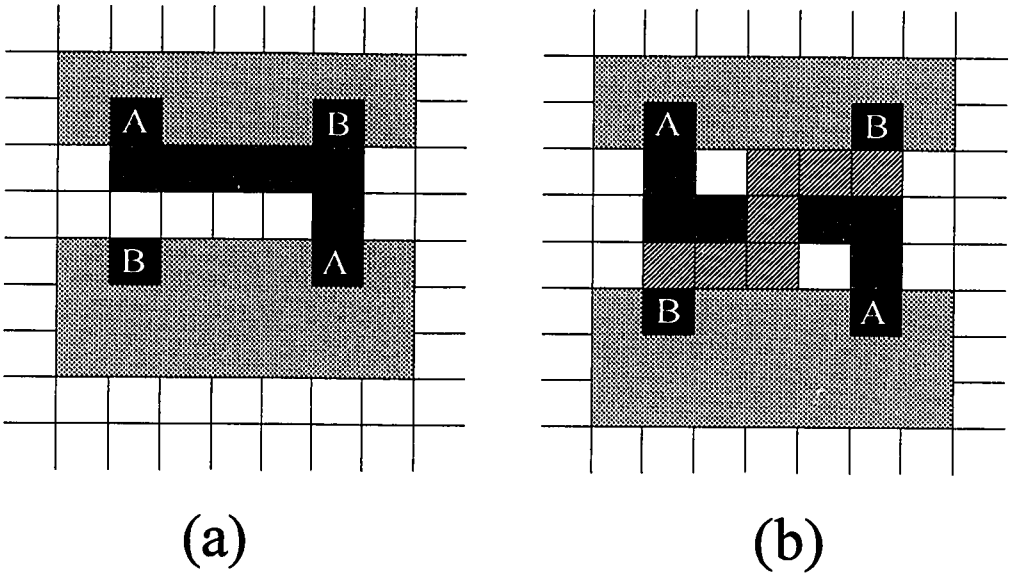


Figure 5.5: Path blocking example as a result of inadequate space.

5.2 Extracting Nets in *AutoVLSI*

As mentioned previously, after the Logic Assignment stage, the design is presented in the form of two files: *GP2.DAT* which has the gate types used in the design, and *IOP2.DAT* which has the interconnections between different ports of these gates. Also, when the system library of VLSI cells was prepared, a file named *POC.DAT* had to be created. This file has the coordinates in grid units of each port on every VLSI cell relative to the cell origin (0,0).

Extracting nets of a design is done in two steps. The first step is to map these interconnections in the *IOP2.DAT* file to the cells ports in *POC.DAT* and define the set of points which must be connected together. And the second step, is to find the shortest path between every set of these points. A path consists of several segments where each segment connects between two points.

5.2.1 Mapping Between Design Interconnections and Ports on VLSI cells

A software named **P4** receives the three files mentioned above as input and assigns a sequential net number (a negative integer) to every port on the cells according to the interconnections found in *IOP2.DAT* file. If one port exists more than once on the boundary of a cell, then every coordinate for it is given the same net number. Also, if two ports or more on different cells are interconnected which each other, then all coordinates for these ports are given the same net number to indicate their belonging to the same net. **P4** saves resulted data about the nets in an output file named *N9.DAT*. This last file has the coordinates of every point in every net.

Three softwares named **FIR**, **SEC**, and **THI** process *N9.DAT* file as input plus the two resulting files from the global routing stage (*PROCXY.ADJ* and *PROCHW.DAT*) and produce an output file named *SEPATHS.DAT*. Every line in this file is a record for one point belonging to a net. The record has four pieces of data: net number, its X-coordinate of a point, its Y-coordinate, and its cell number. The point coordinates are relative to the cell place in the layout. All points are sorted according to the net number such that points related to the same net are grouped together, and a line that has four " -1" numbers separates between points of two different nets.

5.2.2 Finding Shortest Paths

Minimal spanning tree technique is used to find the shortest path between a set of points which are to be connected together.

FOU software receives *SEPATHS.DAT* file as input and calculates the absolute Manhattan distance (AMD) between every point of a net and all other points which

belong to the same net [23]. AMD is calculated between two points P_1 and P_2 according to the following equation:

$$AMD = |(P_1\_X - P_2\_X)| + |(P_1\_Y - P_2\_Y)|$$

This Manhattan distance is considered as an estimate for wire length between these two points. This data is saved in a file named *ABSMAN.DAT* where every line is a record of six fields: AMD value, first point order number in the list of points of a net in *SEPATHS.DAT* file, second point order number, net number, first point cell number, and second point cell number. A line with six occurrences of a negative number (g) separates between every group of points related to the same net. The negative number g is equal to $-$[the number of points for the net] $- 1$.

A software named **KRUSKAL** uses both last output files, namely *SEPATHS.DAT* and *ABSMAN.DAT*, and finds the minimal spanning trees between the points of every net such that one point only from every cell is connected. The algorithm used is *Kruskal* algorithm which uses the computed AMD values as the weights for the branches of a tree. The output file *MST.DAT* will have the final set of segments for every path, where each segment is determined by the starting and ending points coordinates on the layout floor. Every line in that file is a record representing a segment and consists of seven pieces of data: net number, starting point X-coordinate, starting point Y-coordinate, ending point X-coordinate, ending point Y-coordinate, starting point cell number, and ending point cell number. A set of segments having the same net number constitute a route which must be represented on the layout by conducting materials. For further details about **FOU** and **KRUSKAL** algorithms, refer to [23].

For simplicity, the operation of extracting nets was integrated in one batch file called **GMSTADJ**, which stands for: generating minimal spanning tree for the cells in *Procxy.Adj* file. When the batch file **GMSTADJ** is called, the five softwares mentioned above (**FIR**, **SEC**, **THI**, **FOU**, and **KRUSKAL**) are executed in sequence. Note that **P4** software is not included and has to be executed ahead. After completion, all temporarily files are deleted. The flow-chart of the batch file **GMSTADJ** is given in Figure 5.6.

5.3 Detailed Routing in *AutoVLSI*

5.3.1 Running the Router Software

In the previous section, it has been shown how points of a net which constitute a path have been determined. The process of *detailed routing* is to find the exact coordinates on the layout floor for the routes that connect between every pair of points of every path. Thus for every record in *MST.DAT* file, a set of one or more routes have to be created where a route can be either horizontal or vertical, and is determined by its start and end points coordinates.

An available detailed router software which follows Lee algorithm and uses two layers was integrated with *AutoVLSI* system [23]. The software was named **ROUTER**. It receives three files as inputs, namely: *PROCXY.ADJ*, *PROCHW.DAT*, and *MST.DAT*. Two output files are generated. *COFGRID.DAT* file will have the coordinates of the routes, that is a set of points coordinates which must be connected by horizontal and vertical conducting materials. Every set of routes is preceded by a line of three integers. The first is a negative sequence number which shows the order

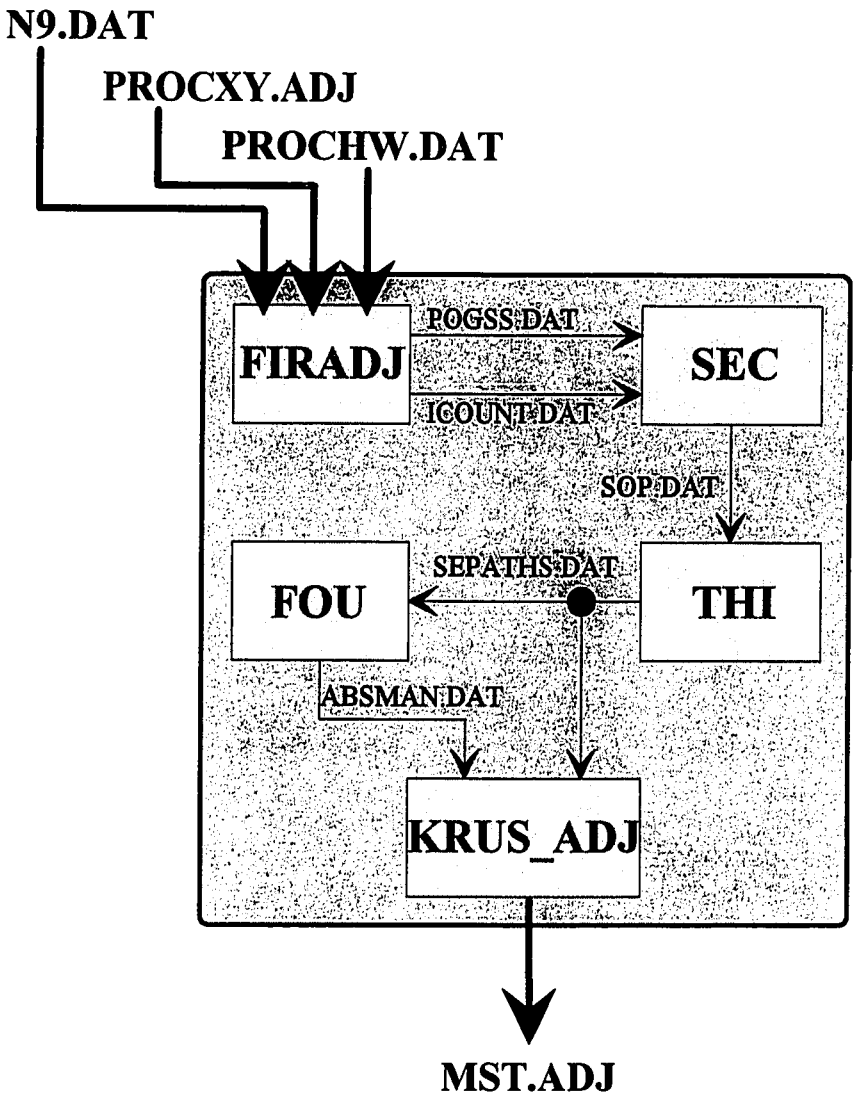


Figure 5.6: The algorithm for *GMSTADJ* batch program.

of the path segment when routed. The second integer is the net number the path belongs to, and finally the third integer which is the number of routes representing that path segment.

The second output file *NFOUND.DAT* generated by **ROUTER** software will be empty unless at least one of the path segments could not be routed. For every non-routed segment, a record will be found in that file showing the net number that the segment belongs to, the X- and Y-coordinates of the start and end points of that segment, and finally the cell numbers of the start and the end points of the segment are written. Also the same data will be echoed to the computer monitor while running the software **ROUTER**.

5.3.2 Non-Routed Segments

If after detailed routing some of the segments were not routed and the output file *NFOUND.DAT* was not empty, then some segments were blocked.

As mentioned earlier, there are two reasons for blocking. These can be solved either by reordering the segments and re-routing, or by pushing some cells to resolve a congested area. In *AutoVLSI* system, these problems are automatically solved as follows.

Reordering Segments for Routing

If the non-routed segment was not the first record in *MST.DAT* file, then presenting it at first to the detailed router will most probably make it routable while not affecting other segments. This is achieved by the **REROUTE** software. It reads the segments in *NFOUND.DAT* file and creates a new *MST.DAT* file such that they

are placed at the beginning of the list of segments.

ROUTER software has to be executed again after **REROUTE** to get the new coordinates of the routes. Also *NFOUND.DAT* file is checked again to see if different segments could not be routed at this iteration.

REROUTE software can be used several times to solve different segments blocking each other. However, if one of the following two conditions is true, then reordering will not be effective any more. The first condition is when the first segment in *MST.DAT* file is not routed. And the second condition is when the same pattern of one or more segments is appearing in *NFOUND.DAT* file during several iterations of the process.

In case any of the above two conditions exists, then blocking is a result of congested area in the layout and is solved by pushing cells as explained in the next sub-section.

Pushing Cells

Global routing stage function was to predict the place and amount of space needed for the detailed router to route segments of all paths. However, in many cases, the detailed router will not pick the same paths predicted and the layout will have some congested area.

This congestion can be solved by increasing the space around one or more of the cells which the non-routed segment is starting from or terminating at. This is done by pushing a cell upward and rightward by some grid units.

To help the user determine graphically where the congestion is on the layout, a software tool named **SRPLOTNF** will show a symbolic plot of the cells and the

routed segments. It will color the cells participating in a non-routed segment by red. Looking at the layout can tell where the congestion is and the user can pick one or more cells for the pushing operation.

PUSHCELL software receives *two parameters.* The first is the number of the pushed cell, and the second is the number of grid units that the cell must be pushed in both directions: upward and rightward. Pushing a cell is done exactly as expanding a channel vertically and horizontally. This is explained in Section 5.4.

Cell pushing operation will change the X- and Y-coordinates of the pushed cell and every cell in its horizontal or vertical fields. This means that routes coordinates are not valid any more, so the system must branch back to the step of generating the minimal spanning tree, that is calling the batch file **GMSTADJ**.

To explain the detailed routing stage more, the flow-chart is given in Figure 5.7.

5.4 Building the Layout

After detailed routing stage is completed, the VLSI layout created is described in four ASCII files. The first file is *GP2.DAT* which has the cells types. The second and third files are *PROCXY.ADJ* and *PROCHW.DAT* which have the X- and Y-coordinates, and the dimensions (i.e., width and height) for every cell in the layout floor. The fourth file *COFGRID.DAT* has the coordinates for the start and end points of every route in the layout floor.

AutoVLSI system extracts the data in these four files and creates three files in *MAGIC* format. The first file has the cells only placed according to their X- and Y-coordinates, while the second file has the routes. Vertical routes are built from *polysilicon* material while horizontal ones are built from *metal* material. When there

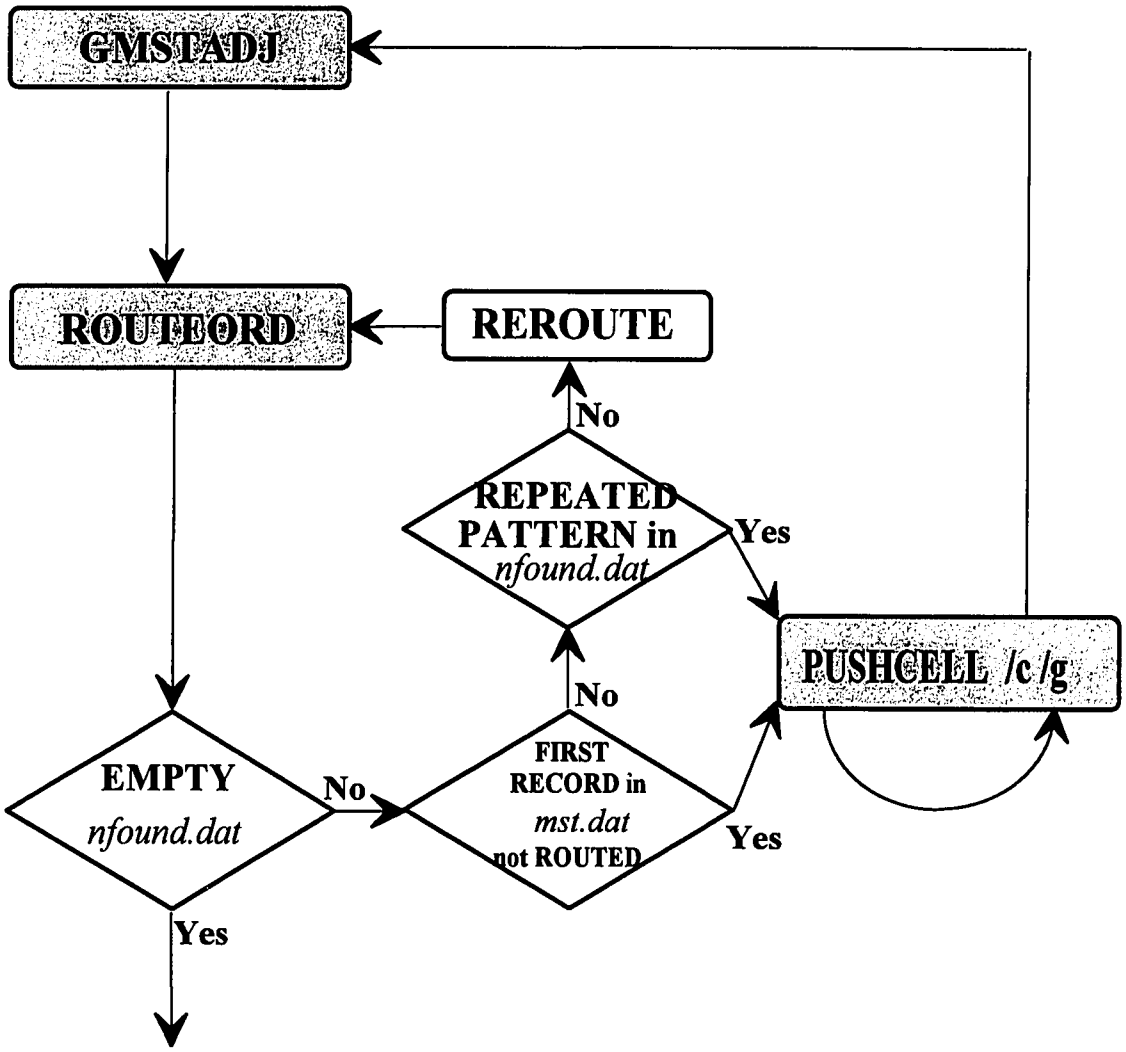


Figure 5.7: The Flow-Chart of the Detailed Routing stage in *AutoVLSI* system.

is a change from a horizontal to vertical route or vice versa in the same path segment, a polycontact is built to produce the electrical connectivity between the two routes.

Finally, the two files previously created are combined together such that the two layouts are overlaid on top of each other origin to origin. The resulting layout is saved in a third file.

5.4.1 Creating *MAGIC* Cells File

A software module called **MAGADJ** receives the three files: *GP2.DAT*, *PROCXY.ADJ*, and *PROCHW.DAT* as inputs and checks for every type found in *GP2.DAT* file that there exists a corresponding file with the same name and *MAG* extension available in a sub-directory called **MAG**. This is to insure that all *MAGIC* VLSI cells corresponding to gates of the design are available. Then according to the data in both *PROCXY.DAT* and *PROCHW.DAT* files, a *MAGIC* file called *CELLS.MAG* is created and saved in **MAG** sub-directory.

Cells X- and Y-coordinates created in the *MAGIC* file are given to λ units rather than grid units. If a cell C has x and y coordinates in *PROCXY.DAT*, then its new coordinates in *CELLS.MAG* file are going to be $(x \times 8) + 2$ and $(y \times 8) + 2$.

CELLS.MAG file refers to the *MAGIC* VLSI cells files available in the same directory by names. Thus, these files must not be deleted or named differently even after the **MAGADJ** software is executed.

5.4.2 Creating *MAGIC* Routes File

The data available in *COFGRID.DAT* file are the coordinates of the grids where a route starts and ends. These coordinates form either a horizontal or vertical line.

However, according to VLSI design rules, a route on the layout floor must be a rectangle whose width is 4λ.

The **RB** (routes builder) software reads in the coordinates in *COFGRID.DAT* file, transfers the grid coordinates to λ coordinates as done with the cells coordinates, and generates for every route a rectangle of width 4λ. If the route is vertical, then it is saved in the *MAGIC* output file as *polysilicon* material. On the other hand, if the route is horizontal, then it is saved as *metal* material.

The complete set of routes for one path segment is processed in one iteration. In case there is more than one route for a path, then after the second route is built, a *polycontact* at the shared grid between the two routes is built. Without this *polycontact*, the two routes would not be electrically connected since they are made of different layers. A new *polycontact* is built for every consecutive route in the set.

The data of vertical (*polysilicon*) and horizontal (*metal*) routes, and *polycontacts* are saved in a *MAGIC* output file named *ROUTES.MAG* in the **MAG** sub-directory. Also another output file named *ROUTES.PLO* is created in the main directory. This file is used by a supported tool called **RPLOT** to plot the cells and the routes of the layout on the monitor.

5.4.3 Combining the Cells and the Routes Files

Running the last software **COMBINE**, generates the final output file *CHIP.MAG* in the **MAG** sub-directory. Its function is to graphically overlay *CELLS.MAG* and *ROUTES.MAG* files on top of each other (origin to origin) to form the VLSI layout of the digital design described by *GP2.DAT* and *IOP2.DAT*.

The **COMBINE** software refers to both *CELLS.MAG* and *ROUTES.MAG* files

by names. Thus, they must be kept in **MAG** directory with these specific names.

5.5 Correctness & Simulation of the Layout

Through all stages of AutoVLSI system, VLSI design rules have been taken care of very carefully. Thus, generated layouts are correct by construction. However, functionality of the layout is related to the design initially given and to some time-constraints. It can not be guaranteed until the VLSI layout is simulated.

MAGIC is a sophisticated layout editor program which has a built-in simulator called **IRSIM**. MAGIC can be used to view the layout, to extract it into different formats (such as CIF format), to simulate the layout, and to plot it [1], [3], [4].

Chapter 6

Conclusion & Future Work

In this work, an extensive survey of different algorithms related to the tasks of the physical design problem of VLSI layouts was conducted. A set of selected algorithms was implemented to develop a system which automatically generates a VLSI general-cell layout for any digital system described in UAHPL. The developed system was named *AutoVLSI* and consists of four stages: logic assignment stage, placement stage, global routing stage, and detailed routing stage.

The system primary objective was to reduce turn-around time, which is the total execution time starting from a UAHPL code, until a VLSI layout is generated. Other objectives were to minimize total area of the layout and total wire length. The results for some digital circuits along with other statistical data are shown in Figure 6.1.

Turn-around time is given in minuets, while layout area is in λ^2. However, to evaluate the layout area according to placement and routing stages, space utilization (SU) ratio is computed as follows:

CIRCUIT	# of CELLS	# of NETS	# of ROUTES	# of POLY-CONTACTS	CELLS AREA	ROUTES AREA	LAYOUT AREA	SPACE UTILIZA-TION	TURN-AROUND TIME
EX30	32	42	151	109	3,171	1,556	8,526	55%	0:00:55
EX33-1	35	52	193	141	3,345	2,040	8,268	65%	0:03:36
EX33	51	68	272	204	3,633	2,823	14,883	43%	0:08:17
EX34	119	176	717	541	11,712	10,978	52,559	43%	0:30:40

Figure 6.1: Statisical data as a result of running *AutoVLSI* system on four different examples.

$$SU = \frac{\text{Cells Area} + \text{ Routes Area}}{\text{Layout Area}} \times 100$$

The larger space utilization ratio is, the more compact is the layout. In other words, large SU ratio means that less space of the layout is wasted.

6.1 Layout Compaction

Compaction is best defined by moving cells close to each other as most as possible in order to reduce layout area and thus increase space utilization. It is a common practice to try compacting a VLSI layout after it is completed. Layouts generated by AutoVLSI system can be sometimes compacted if an extra space was proposed in the global routing stage.

Global routing stage is not a hundred percent accurate procedure. It might **Push** cells up-ward or right-ward –to offer wiring space– more than required by

the detailed router since actual wire paths are not yet determined. After a general-cell VLSI layout has been created by *AutoVLSI* system and has been simulated, automatic compaction available in **Magic** system can be applied using the command "*PLOW*" [2], [3]. Another method, is to manually edit the cells locations X and Y coordinates in the file *PROCXY.ADJ* and modify any cell coordinates in order to compact the layout. However, if manual compaction is applied, then all steps of the detailed routing stage have to be repeated again to automatically fix routes paths according to the changes made to the layout.

6.2 Future Work

The following points raise some issues where future problems and solutions can be investigated:

- The system can be expanded to accept UAHPL modules having some combinational logic units (CLUs).
- A user graphical interface module can be integrated with *AutoVLSI* system to facilitate running its components and organizing circuit file structure.
- *Channels merging* sub-task should follow a more precise algorithm according to the design netlist. netlists. Current algorithm does merging broadly to channels neighboring each other and sometimes this creates a horizontal **wide** channel or a vertical **long** channel. When global routing predicts that a wire is to pass through one of these channels, it will assume that the wire will consume a complete (horizontal/vertical) track of the channel even though the wire might pass only in a small section of a track leaving other sections for

other wires. This means that in many cases, because every wire is assumed to consume a complete track, *routing region adjustment* task is over-expanding channels and thus increasing layout area and reducing space utilization ratio.

- Timing constraints can be added to other constraints of the system placement stage. This will insure generating VLSI layouts with no timing violation.

Appendix A

Summary of AutoVLSI System Components.

In this appendix, a brief description of every software used in *AutoVLSI* system is given. The name of every software is mentioned and followed by a list of parameters if it requires any. A parameter is proceeded by a slash (/) in the text, but when running the software, this slash should be removed. If a parameter is inclosed between brackets ([]), then it means that it is an optional parameter. Then an explanation for the name of the software is given when applicable.

After the software name, **Input(s)** record shows what files should be available when running the software, while **Output(s)** record shows what files will be created by the software. Then the software function is described.

A.1 Main Software Modules

- **p4**

Input(s): *GP2.DAT - IOP2.DAT - POC.DAT*

Output(s): *N9.DAT*

The output is a file that has each cell followed by the points on each cell were each one of these points has an I/O link in *IOP2.DAT* file. Each point is given a net number. If two points belong to the same I/O link, then both will be given the same number. The coordinates of these points are with reference to zero,zero (i.e., as if assuming that each cell will be placed at the origin of the layout).

- **GNETS (Generate Nets)**

Input(s): *N9.DAT*

Output(s): *NETS.DAT*

The output file has each net number preceding points, followed by cell numbers that has this net number as one of its entries. In another words, each net and its relating cells. This is needed for the next stage (Linear Ordering of Cells) where it is required to know what cells are related to each net rather than what nets are included in each cell.

- **LOC (Linear Order Cells)**

Input(s): *GP2.DAT - NETS.DAT*

Output(s): *CELLS.LOC*

The program does linear ordering for these nets in the input file according to number of cells related to each net. First, it linearly orders input cells (type 4018) alone because they are required to come in the beginning of the ordered list. At the same time, the program removes output cells (type 4013) because we would like to make sure they come at the end of the ordered list. After other cells than inputs or outputs are ordered, output cells are ordered and attached to the ordered list.

- **CG /n (Cluster Growth)**

Input(s): *GP2.DAT - AGHW.DAT - CELLS.LOC*

Output(s): *PROCXY.LAY - PROCHW.DAT*

This program is to give each cell an x and y coordinates to form the layout. It receives one parameter that determines the maximum width of the layout to be generated. *Procxy.Lay* file will have one cell at each row in the following format: cell#, cell X coordinate, and cell Y coordinate. *Prochw.Dat* file will have one cell at each row in the following format: cell#, cell height, and cell width.

- **GMSTLAY (Generate Minimal Spanning Tree)**

Input(s): *N9.DAT - PROCXY.LAY - PROCHW.DAT*

Output(s): *MST.DAT*

The minimal spanning tree is needed to know the shortest path for each net in the layout. Nets generated at this sage will be used later by the global router to determine the minimum spacing (horizontal and vertical) needed between each cell to allow the presence of connection material paths representing these nets.

- **CCHANS (Catch Channels ˜Identify Channels)**

Input(s): *PROCXY.LAY - PROCHW.DAT*

Output(s): *CHANS.CCH*

Catching channels (Channels Identification) is the first step toward global routing of nets. The output of this software is a file containing one channel in each row in the following format: ch.#, ch.ll.X, ch.ll.Y, ch.ur.X, ch.ur.Y, ch.hw, and ch.vw. (ch = channel, ll = lower left point, ur = upper right point, hw = horizontal weight, vw = vertical weight).

A weight of a channel is the number of tracks the channel contains. A track is the minimum space area needed for a connection material path.

- **UNIFYCHS (Unify Channels ~Merge Channels)**

Input(s): *CHANN.CCH*

Output(s): *CHANN.UNI*

Channels identified in the previous stage could be neighbors horizontally or vertically. These neighbored channels are unified (merged) together to reduce total number of channels used in the global routing stage.

- **GR (Global Router)**

Input(s): *PROCXY.LAY - PROCHW.DAT - MST.DAT - CHANN.UNI*

Output(s): *CHANN.GR*

The program starts by reading the unified channels file and creates a non directed graph whose nodes represent records of a channel and whose arcs represent adjacency between two nodes. Each arc has a label 'V' or 'H' telling that its relative nodes (channels) are either vertically or horizontally adjacent. The global router picks each net in *mst.dat* file which consists of two points (starting point and end

point) and assigns a graph node (channel) to each one of these points then finds the shortest path between the two assigned nodes. Starting from the first node in the shortest path, the horizontal/vertical weight of the channel represented by that node is reduced accordingly to reserve a track in this channel for the net being considered.

- **ADJL (Adjust Layout)**

Input(s): *PROCXY.LAY - PROCHW.DAT - CHANN.GR*

Output(s): *PROCXY.ADJ*

In this stage, channels are read from *chann.gr* file and processed according to their vertical and/or horizontal weights. If a horizontal weight of a channel is negative then it means that this channel height has to increase, so each cell falling in the vertical field of the channel is pushed up by n units, where $n = -$(the horizontal weight of the channel). The same thing goes for the vertical weight. Any cell falling in the horizontal field of a channel with a negative vertical weight has to be pushed right in order to increase the channel width. The pushing operation is achieved recursively. By recursively we mean that if $C1$ is pushed then every cell falling in the VF of $C1$ is also pushed.

- **GMSTADJ (Generate Minimal Spanning Tree)**

Input(s): *N9.DAT - PROCHW.DAT - PROCXY.ADJ*

Output(s): *MST.DAT*

In this stage, nets are generated for the router after cells have taken their final locations.

- **ROUTER (Detailed Router)**

Input(s): *PROCXY.ADJ - PROCHW.DAT - MST.ORD*

Output(s): *NFOUND.DAT - COFGRID1.DAT*

The detailed router job is to pick each net segment in the input file *MST.DAT* and tries to route it in the layout between cells connecting the two end points of this segment. The routing is done such that the total length of the segment is minimal. When it stops, *NFOUND.DAT* file will have these nets which the router was not able to route. *COFGRID1.DAT* file will have sequence of points in each line representing the corners of a routed net path. The first and second points will form the first segment of a net, then the second and third points form the second segment, and so on until that last point in the sequence which forms the net end point.

- **RB (Routes Builder)**

Input(s): *COFGRID1.DAT*

Output(s): *ROUTES.MAG - ROUTES.PLO*

This program builds routes from the specified connection material. It reads in the sequence of points in one line of the input file and generates a rectangle from every two consecutive points which form a segment. In this system, *polysilicon* is used for vertical segments of a net, and *metal1* is used for horizontal segments. The rectangle width for segments is a value saved previously in **RB** software. Whenever there is a switch of material for the same net, a *polycontact* is inserted. *ROUTES.MAG* output file is a **Magic** format file that holds routes of the layout. *ROUTES.PLO* output file is used by a utility named **RPLOT** to plot the layout and the routed segments.

- **MAGLAYOUT (Magic Layout)**

Input(s): *GP2.DAT - PROCXY.ADJ - PROCHW.DAT - CELLS.ORD*

Output(s): *CELLS.MAG*

Magic layout software will generate a **Magic** format file containing all cells in their corresponding coordinates according to *PROCXY.ADJ* file. Each cell is replaced by its VLSI mask available in the system library in **Magic** format.

- **COMBINE (Combine)**

Input(s): *CELLS.MAG - ROUTES.MAG*

Output(s): *CHIP.MAG*

This software graphically overlaps the two input files to generate the complete VLSI layout in **Magic** format.

A.2 Utility Software Modules

- **AREA (Layout area)**

Input(s): *PROCXY.LAY - PROCHW.DAT*

This software displays statistical data about layout height, width, and area; cells area; and space area, and space utilization ratio. The layout is read from the two input files.

- **PAREA /fn (Layout Area)**

Input(s): *PROCXY. fn - PROCHW.DAT*

fn: is a valid extension for a *PROCXY* file holding cells X,Y coordinates.

Same utility as **AREA** but the layout is read from a file *PROCXY* of the passed parameter extension and *PROCHW.DAT.*

- **CHAREA (Layout & Channels Area)**

Input(s): *PROCXY.LAY - PROCHW.DAT - CHANN.CCH*

Same utility as **AREA** plus displaying channels area. When channels area and space area are equal, it means that catching channels software (**CCHANS**) is covering all space area in the layout. The layout is read from the first two input files, while the channels structure is read from *CCHANN.CCH* file.

- **PCHAREA /fn (Layout & Channels Area)**

Input(s): *PROCXY.LAY - PROCHW.DAT - CHANN.fn*

fn: is a valid extension for a *CHANN* file holding channels structure of the layout.

Same as **CHAREA** software but this time the channels structure is read from a file *CCHANN* of the passed parameter extension.

- **PCPV /fn (Check Placement Violation)**

Input(s): *PROCXY.fn - PROCHW.DAT*

Output(s): *CXYXY.fn*

fn: is a valid extension for a *PROCXY* file holding cells X,Y coordinates.

This utility is to check if any two cells overlay each other as a result of running **CG** or **ADJL** programs. The layout is read from a file *PROCXY* of the passed parameter extension and *PROCHW.DAT* file. If any cell overlays any other cell, then both cells numbers are given in a message. The output file has the same extension of the passed parameter. It holds extended cells information in the following format: C#, C.ll.X, C.ll.Y, C.ur.X, and C.ur.Y. (C = cell, ll = lower left point, ur = upper right point).

- **PLOT (Plot Layout & Optionally Channels)**

Input(s): *PROCXY.LAY - PROCHW.DAT - [CHANN.CCH]*

This is a graphics utility that shows a symbolic plot of the layout read from the first two input files. If a question for plotting the channels structure is answered by yes, and the input file *CHANN.CCH* is available then the utility will plot channels in different color than cells in the layout.

- **PPLOT /fn1 [/fn2] (Plot Layout & Optionally Channels)**

Input(s): *PROCXY. fn1 - PROCHW.DAT - [CHANN. fn2]*

fn1: is a valid extension for a *PROCXY* file holding cells X– and Y–coordinates.

fn2: is a valid extension for a *CHANN* file holding channels of the layout.

Same as **PLOT** utility but the layout is read from a file *PROCXY* of the first passed parameter extension and *PROCHW.DAT*. Also channel structure (if to be plotted) will be read from a file *CHANN* of the second passed parameter extension.

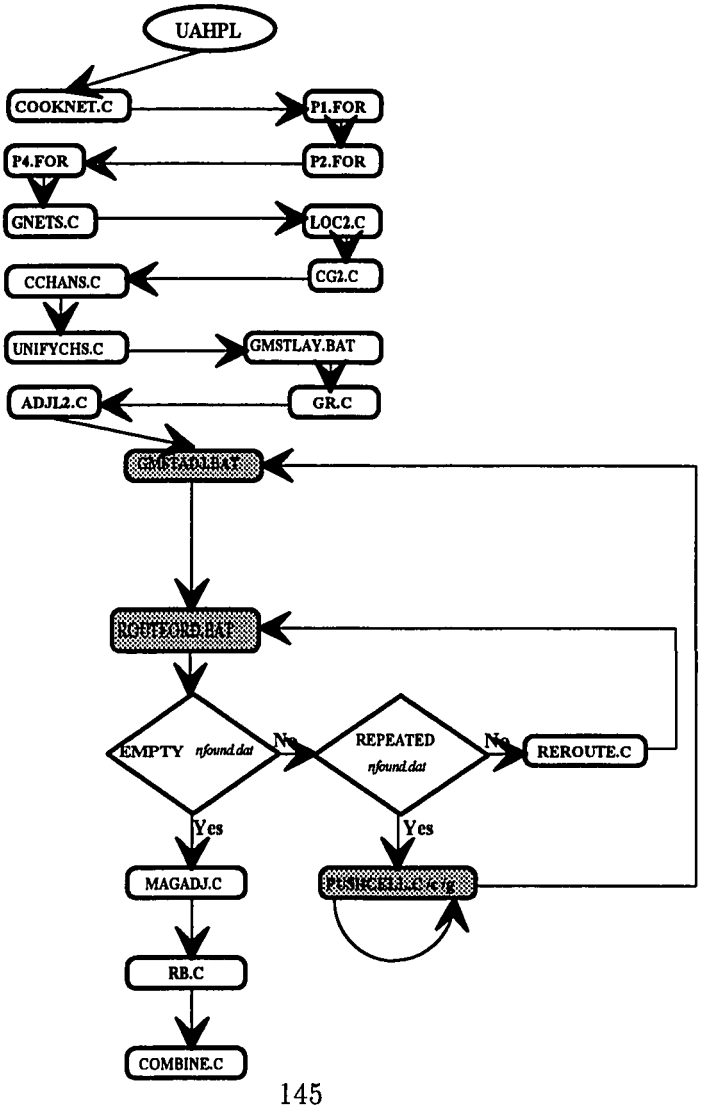
- **RPLOT (Plot Layout and Routes)**

Inputs: *PROCXY.ADJ - PROCHW.DAT - ROUTES.PLO*

Same as **PLOT** utility but layout is read from the input files *PROCXY.ADJ* and *PROCHW.DAT* plus that routes are read from the input file *ROUTES.PLO* and plotted.

Appendix B

Flow-chart of AutoVLSI System.



Bibliography

[1] Becket, W., 'VLSI Design Tools Manual', Release 3.1, NW Laboratory for Integrated Systems, University of Washington, Feb. 1987.

[2] Sadiq, M.S., and Habib, Y., 'VLSI Design Automation: Theory and Practice', McGraw-Hill Book Co., Europe, 1994.

[3] Mayo, R.N., et.al, 1990 DECWRL/Livermore Magic Release, Digital Western Research Laboratory, Sep. 1990.

[4] Terman, C., 'Users Guide to NET, PRESIM and RNL/NL', Cambridge, MA: Laboratory for Computer Science, M.I.T., MA.

[5] Shahookar, K., and Mazumder, P., "VLSI Cell Placement techniques", J. ACM Computing Surveys, Vol. 23, No. 2, Jun. 1991, pp. 143-220.

[6] Sait, M.S., "Integrating UAHPL-DA System with VLSI Design Tools", IEEE Trans. on Education, Vol. 35, No. 4, Sep. 1992.

[7] Sait, M.S., and Al-Khulaiwi, F.A., "Automatic Weinberger arrays synthesis from UAHPL description", Int. J. Electronics, 1990, Vol. 69, No. 2, pp. 211-224.

[8] Sahni, S., and Atul, B., "The Complexity of Design Automation Problems", J. ACM, 1980, pp. 402-411.

[9] Sutanthavibul, S., and Eugene, S., "JUNE:An Adaptive Timing-Driven Layout System", Tech. Report TR-89-37, Dep. of Computer Science, Univ. of Minnesota, 1989.

[10] Kuh, Ernest S., et.al, "Recent Advances in VLSI Layout", Trans. of IEEE, Vol. 78, No. 2, Feb. 1990.

[11] Cai, H., and Ralph, H.O., "Conflict-Free Channel Definition in Building-Block Layout", IEEE Trans. on Computer-Aided Design, Vol. 8, No. 9, Sep. 1989, pp. 981-988.

[12] Clow, G.W., "A Global Routing Algorithm for General Cells", IEEE 21st Design Automation Conference, 1984, Paper 3.3, pp. 45-51.

[13] Cai, Y., and Wong, D.F., "Channel/Switchbox Definition for VLSI Building-Block Layout", IEEE Trans. on Computer-Aided Design, Vol. 10, No. 12, Dec. 1991, pp. 1485-1493.

[14] Preas, P.T., and Vancleemput, W.M., "Placement Algorithms for Arbitrarily Shaped Blocks", Proc. of DAC, 1979, pp. 474-480.

[15] Goto, S., and Kuh, E.S., "An Approach to the Two-Dimensional Placement Problem in Circuit Layout", IEEE Trans. on Circuits and Systems, Vol. CAS-25, No. 4, Apr. 1978, pp. 208-214.

[16] Kang, S., "Linear Ordering and Application to Placement", IEEE 20th Design Automation Conference, Paper 30.2, 1983, pp. 457-464.

[17] Walter, S., and John, K.O., "Magic's Circuit Extractor", IEEE Design & Test, Feb. 1986, pp. 24-34.

[18] Luther, C.A., "On the Ordering of Connections for Automatic Wire Routing", IEEE Trans. on Computers, Nov. 1972, pp. 1227-1233.

[19] Hazem, M.A.,'An Integrated VLSI Layout Generation System', M.Sc. Thesis Proposal, KFUPM, Dep. of COE, 1993.

[20] Newton, A.R., and Sangiovanni-Vincentelli, A.L., "Computer-Aided Design for VLSI Circuits", IEEE Computer, Apr. 1986, pp. 38-60.

[21] Dutt, N.D., and Daniel, D.G., "Design Synthesis and Silicon Compilation", IEEE Design & Test of Computers, Dec. 1990, pp. 8-23.

[22] Hill, F.J., and Gerald, R.P., 'Introduction to Switching Theory & Logical Design', John Wiley & Sons (1981).

[23] Sait, M.S., 'VLSI Mask Generation from Register Transfer Level Descriptions: An Automated Approach', Ph.D. Thesis, KFUPM, Dep. of EE., 1987.

[24] Reingold, E.M., and Wilfred, J.H., 'Data Structures in Pascal', Little, Brown and Company, 1986.

Vita

Hazem Muhebbaddin Ahmad Naji Abu-Saleh:

- Born at Aleppo, Syria.
- Received Bachelor's degree in Computer Engineering from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in August 1991.
- Completed Master's degree requirements at King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia in February 1995.