





This is to certify that the

dissertation entitled

The Theory and Implementation of Computer Graphic  
Drawing Algorithms and Routing Tools in a VLSI  
Design Environment

presented by

YU-TSE CHEN

has been accepted towards fulfillment  
of the requirements for

Ph.D. degree in Electrical Engineering

  
Major professor

Date 11/1/89

**PLACE IN RETURN BOX** to remove this checkout from your record.  
**TO AVOID FINES** return on or before date due.

DATE DUE	DATE DUE	DATE DUE
APR 3 10 43	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

**MSU Is An Affirmative Action/Equal Opportunity Institution**

**THE THEORY AND IMPLEMENTATION OF COMPUTER GRAPHIC DRAWING  
ALGORITHMS AND ROUTING TOOLS IN A VLSI DESIGN ENVIRONMENT**

By

Yu-Tse Chen

A DISSERTATION

submitted to  
Michigan State University  
in partial fulfillment of the requirements  
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1989



## ABSTRACT

### THE THEORY AND IMPLEMENTATION OF COMPUTER GRAPHIC DRAWING ALGORITHMS AND ROUTING TOOLS IN A VLSI DESIGN ENVIRONMENT

By

Yu-Tse Chen

Exploiting VLSI technology for electronic designs can greatly increase product performance, but several design processes must be executed efficiently in order to produce a successful integrated circuit in a timely manner. In this research, we investigate two VLSI design related areas: the mapping of algorithms to architectures and the transforming of a netlist to a physical layout. Our goal is to apply VLSI technology to circuit design in order to efficiently increase product speed and reduce product area.

In the algorithm mapping investigation, we focus on developing an antialiased drawing engine. Three processes are needed in order to produce this high performance engine, i.e., algorithm development, architecture implementation, and performance estimation. By adopting operating concepts from both the incremental drawing technique and the area-antialiasing technique, an efficient antialiased drawing algorithm was developed. As a result, this algorithm produces realistic images faster than those of existing algorithms. According to the data flow of this algorithm, an architecture was implemented using fixed-point binary arithmetic. To estimate the performance of this engine, we constructed a delay model for the cells and implemented a prototype line-drawing engine assuming a 3  $\mu\text{m}$  CMOS technology. The timing estimation result shows that our antialiased drawing engine has a critical path delay of only 80 nanoseconds, which is much better than that of using printed-circuit board implementation.

In the VLSI design tools investigation, we focus on channel routing and mask generation problems in the standard-cell (or gate-array) implementation. The objective here is to develop an alternative channel routing approach which can produce acceptable routing results much more efficiently than those of existing routing approaches. We investigated a bottom-up routing approach which combines routing and mask generation. A numerical tiling method with several design rules was used in developing our mask generator. In conjunction with this mask generator, three efficient routers were developed using a heuristic approach and several effective processes to optimize routing parameters for three different routing cases, respectively. According to the results from several test-runs and comparisons, these routers as well as the mask generator are shown to be relatively efficient and flexible.

## **ACKNOWLEDGMENTS**

Thanks to my mother and my wife, Ling-Kuan Chen, and Jui-Ping Chen, for supporting me to complete this dissertation. Especially, thanks to my advisor, Dr. P. David Fisher, for giving me many constructive suggestions.

## TABLE OF CONTENTS

LIST OF TABLES .....	vii
LIST OF FIGURES .....	viii
Chapter 1: Introduction .....	1
1.1 Problem Statement .....	3
1.2 Research Goals and Objectives .....	7
1.2.1 A High Performance Graphic Engine .....	7
1.2.2 Efficient VLSI Layout Tools .....	8
1.3 Thesis Overview .....	9
Chapter 2: Existing Computer Graphic Drawing Algorithms .....	10
2.1 A High Performance Raster System Overview .....	12
2.2 Graphic Drawing Algorithms .....	15
2.2.1 Existing Line-Drawing Algorithms .....	17
2.2.2 The Curve-Drawing Technique .....	22
2.2.2.1 Scan-Covering Curves .....	23
2.2.2.2 Piecewise Curves .....	26
2.3 Existing Antialiasing Techniques .....	30
2.3.1 The Window Averaging Technique .....	31
2.3.2 The Area Antialiasing Technique .....	33
Chapter 3: The CF Antialiased Drawing Engine .....	40
3.1 The Algorithmic Development .....	41
3.1.1 The Line-Drawing Approach .....	41
3.1.2 The CFO Antialiasing Approach .....	42
3.1.3 Implementation of the CF Engine .....	51
3.2 The Architectural Implementation .....	55
3.3 Estimating the Performance of the CF engine .....	63
3.3.1 A Timing Estimation Technique .....	65
3.3.2 A Generic VLSI Implementation .....	67
3.3.3 Evaluating the Fabricated Chip .....	76
Chapter 4: Existing Two-Layer Channel Routing Approaches .....	81
4.1 Problem Formation .....	82
4.2 Existing Two-Layer Channel Routing Algorithms .....	89
4.3 Existing Mask Generators .....	107

<b>Chapter 5: The CF Two-Layer Channel Routing Approach .....</b>	<b>110</b>
<b>5.1 The CF Mask Generator .....</b>	<b>111</b>
<b>5.2 Three Efficient Routers .....</b>	<b>129</b>
5.2.1 The CF_1 Router .....	129
5.2.2 The CF_2 Router .....	138
5.2.3 The CF_3 Router .....	141
<b>Chapter 6: Summary and Conclusions .....</b>	<b>155</b>
<b>6.1 Summary .....</b>	<b>156</b>
<b>6.2 Future Research and Development .....</b>	<b>160</b>
<b>LIST OF REFERENCES .....</b>	<b>162</b>

## LIST OF TABLES

Table 3-1. Area result comparison of the ideal, CFO, PW, and GS thin-cone algorithms .....	49
Table 3-2. Simulation results of using CF engine to draw a line from (0,0) to (7,5) with $Wd=1$ pixel .....	62
Table 5-1. Test results of both the left-edge router and the CF_1 router .....	137
Table 5-2. Demonstration of the CF_2 routing results .....	142
Table 5-3. Demonstration of the CF_3 routing results .....	152
Table 6-1. Result comparison of the CF_1 router, the CF_2 router, and the CF_3 router .....	159

## LIST OF FIGURES

Figure 1-1.	Use of the Y-chart to represent the flow of the VLSI design process .....	2
Figure 1-2.	Example of a straight-line display on the pixel plane .....	5
Figure 2-1.	Functional model of a high-performance raster system .....	13
Figure 2-2.	Our research focus in a complete raster system organization .....	16
Figure 2-3.	Example of a straight line mapped onto a pixel plane .....	18
Figure 2-4.	Symmetrical property of a circle .....	24
Figure 2-5.	Two pixel-shading cases .....	35
Figure 3-1.	Illustration of $A_t$ , $W_x$ , $\bar{e}1$ , and $\bar{e}2$ corresponding to the starting column of a line .....	45
Figure 3-2.	Shaded-area results for three cases of $e_i$ .....	47
Figure 3-3.	Comparison graph of shaded area results .....	50
Figure 3-4.	CF algorithm with various line thickness .....	52
Figure 3-5.	Comparison of the execution speed of the PW, CF, and modified CF algorithms .....	54
Figure 3-6.	CF circle algorithm with $W_d=1$ pixel in octant $y \geq x \geq 0$ .....	56
Figure 3-7.	Block diagram of three incremental processes .....	58
Figure 3-8.	Block diagram of the pixel-intensity generator .....	59
Figure 3-9.	Block diagram of the $A_c$ generator under $W_x \leq 1$ case .....	61
Figure 3-10.	Block diagram of the address generator .....	64
Figure 3-11.	Illustration of a method to determine the timing delay for a cell .....	66
Figure 3-12.	Procedure for generating a SPICE deck from a magic file .....	68
Figure 3-13.	Procedure for constructing the timing equations of a circuit .....	69
Figure 3-14.	Block diagram of the line-drawing engine .....	71
Figure 3-15.	Block diagram of the bus interface and the 4-to-1 multiplexer .....	73
Figure 3-16.	Block diagram of the timing sequencer and the entire system timing chart .....	74
Figure 3-17.	Chip layout of our line-drawing engine .....	75
Figure 3-18.	Input and output test vectors of the line-drawing engine under a 16-bit operation case .....	78
Figure 4-1.	Example of a channel and its related features .....	83
Figure 4-2.	Example of a net list .....	83
Figure 4-3.	Illustration of the horizontal constraint graph and zone representation for the net list given in Figure 4-2 .....	85
Figure 4-4.	Illustration of the vertical constraint graph for the net list given in Figure 4-2 .....	85
Figure 4-5.	Examples of a cyclic routing case .....	87

Figure 4-6.	Illustration of the use of a wrong way horizontal segment .....	87
Figure 4-7.	Examples of four different routing cases .....	88
Figure 4-8.	Left-edge net assignment result .....	90
Figure 4-9.	Constrained left-edge net assignment result .....	92
Figure 4-10.	Illustration of the merging operations .....	94
Figure 4-11.	Illustrations of the matching operations .....	96
Figure 4-12.	Yoshimura net assignment algorithm .....	97
Figure 4-13.	Illustration of the Yoshimura net assignment algorithm .....	98
Figure 4-14.	Greedy routing result with the minimum jog length = 1 .....	100
Figure 4-15.	Illustration of the hierarchical routing .....	102
Figure 4-16.	Examples of the mazer1, mazer2, and mazer3 routings .....	104
Figure 4-17.	Examples of weak and strong modifications .....	106
Figure 4-18.	Examples of traditional symbolic routing and mask generation .....	109
Figure 5-1.	Tiling mask result for a given channel .....	112
Figure 5-2.	Illustration of the basic tiles and six tiling construction equations .....	114
Figure 5-3.	Thirteen basic tiles with the assigned numbers .....	115
Figure 5-4.	Example of using C_Rules 1.1 and 1.2 to construct the tiling array .....	117
Figure 5-5.	Example of using C_Rules 2.1 to 2.4 to construct the tiling array .....	119
Figure 5-6.	Illustrations of the routing violation cases .....	120
Figure 5-7.	Examples of using routing violation detecting rules to detect errors .....	122
Figure 5-8.	Examples of using the vias minimization .....	123
Figure 5-9.	Additional tiles with assigned numbers .....	125
Figure 5-10.	Vias minimization algorithm .....	126
Figure 5-11.	Algorithm used to calculate the total wiring length .....	127
Figure 5-12.	Algorithm used to calculate the number of vias .....	127
Figure 5-13.	Processing sequence of all processes in the CF mask generator .....	128
Figure 5-14.	Illustration of the routing without vertical constraints .....	131
Figure 5-15.	Comparison of two no-dogleg routing results for a given net list .....	132
Figure 5-16.	CF_1 routing algorithm .....	136
Figure 5-17.	CF_2 net assignment algorithm .....	140
Figure 5-18.	Illustration of doglegging operation for solving a given cyclic case .....	143
Figure 5-19.	Example of the subnet construction .....	145
Figure 5-20.	Local-density generation algorithm .....	146
Figure 5-21.	Wire compaction algorithm .....	149
Figure 5-22.	Sensitivity of $\alpha$ to the CF_3 routing results .....	151
Figure 5-23.	Mask result of the 169-column Deutsch example routed by the CF_3 router .....	153

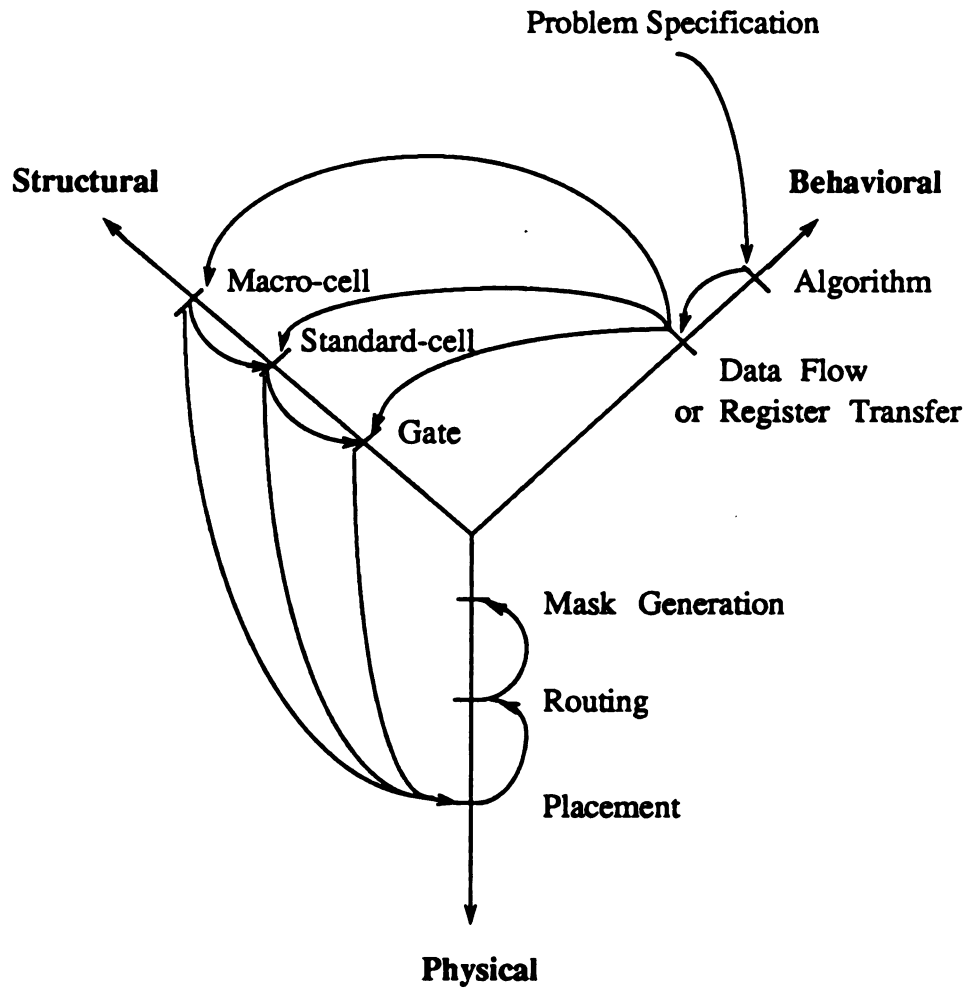


## **CHAPTER 1**

### **INTRODUCTION**

VLSI (Very Large Scale Integration) design holds the promise of reducing product area while increasing product execution speed. Basically, there are three principal tasks involved in VLSI design, i.e., develop the necessary algorithm, develop the architecture that implements this algorithm, and, finally physically implement this architecture in hardware/firmware/software. In order to generate successful products, designers have to correctly design their products at each task. But, the most important VLSI design step is the algorithmic level since a trivial design error at this level would require that the entire design to be redone. The main task at the algorithmic level is to develop an efficient algorithm to solve the given problem statement so that the algorithm can be easily mapped into an architecture. The design objective at the architectural level is to construct an architecture which performs the same function as the given data flow of the algorithm. This architecture is represented as a block diagram using a combination of functional cells. Finally, at the physical implementation level, designers need to translate the architecture into mask data for chip fabrication. The Y-chart is a very useful graph for expressing the relations among these three different design steps. Figure 1-1 illustrates the use of the Y-chart to represent the flow of VLSI design activity.

In order to reduce the design errors and the design time of the generated products, designers use VLSI automation processes linking powerful design strategies that can be executed automatically. VLSI automation processes at the physical design level typically use several layout tools to perform cell placement, routing of cell interconnections, and mask generation for the given architecture.



Note that each axis in the Y-chart represents the design representation;  
 → means mapping from one representation to another

Figure 1-1. Use of the Y-chart to represent the flow of the VLSI design process

In this research, we focus on two VLSI related design problems: VLSI implementation of computer graphic drawing engines at the algorithmic and architectural design levels and channel routing with mask generation tools for VLSI automation at the physical design level.

### 1.1 Problem Statement

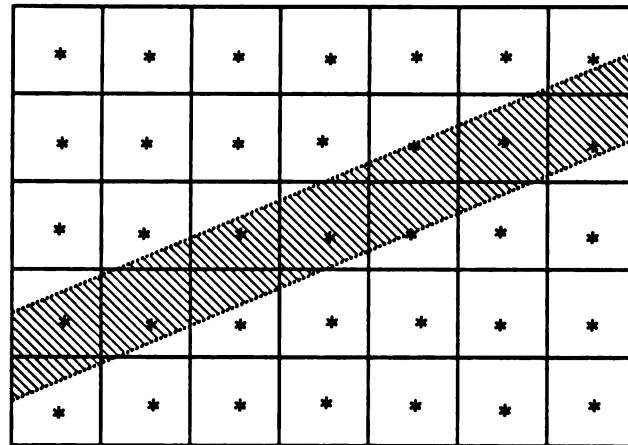
The objective of computer graphic drawing operations is to generate realistic images on raster displays at a very high drawing speed. There are two major computer graphic drawing techniques, parametric and nonparametric. In general, the parametric curve-drawing technique [1] is very flexible in representing a curve by setting a parameter. This parameter is used to generate the corresponding coordinate data. But one major disadvantage of this technique is its relatively high computational complexity since multi-valued calculations must be performed. The non-parametric curve-drawing technique [2] is very attractive for digital generation. It uses a decision-making process to generate approximate curves. With some numerical manipulation, this technique can be applied to scan line conversions used to generate pixels incrementally; i.e., the current drawing position can be determined by using its past drawing information. But this technique has a drawback, too; namely, it is relatively difficult to provide high order curve-drawings. By using some mathematical operations, both curve-drawing techniques can be represented by piecewise lines; thus, the performance of a drawing engine depends on the performance of the scan-converting lines in most raster systems. Thus, it is important to find out the suitable scan-converting line algorithm before constructing the drawing engine.

Although both curve-drawing techniques are useful for drawing operations, they tend to produce aliasing images while rendering geometric objects in computer graphics applications. Basically, aliasing is a display degradation problem resulting from

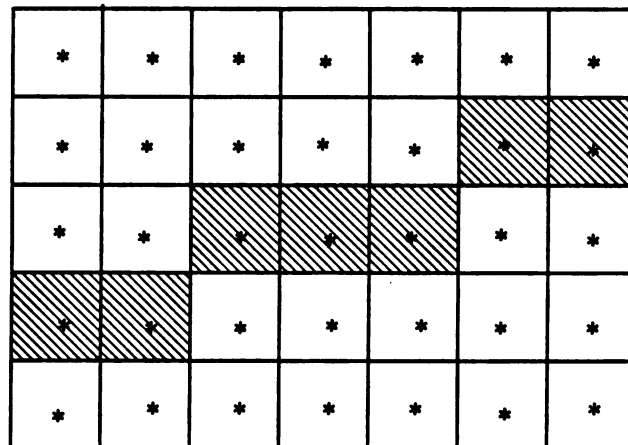
inaccurate digital sampling and filtering of the generated images. The result is known as "jaggy" or "staircase" appearance of lines on a raster display, as illustrated in Figure 1-2.

Antialiasing methods use filtering techniques to solve the problem, and the computer graphic system performs the required operations by processing multiple-intensity levels. The existing antialiasing methods differ in their operating speed and display quality, i.e., realism. Pixel averaging [3], a window-averaging method, is used to produce a pixel in a low-resolution image by averaging several pixels in a high-resolution image. This technique can achieve realistic antialiasing results but needs a very long processing time to manipulate the whole image buffer. Area antialiasing is another simple heuristic technique. It assigns an intensity to each pixel according to the area intersected by the image. Several existing approaches [4-7], which differ in their algorithmic complexities, use the idea of area antialiasing. The flexibility of selecting different display qualities at an acceptable operating speed, however, is the major problem in these antialiasing algorithms. Furthermore, when implemented in VLSI circuits, these existing graphic drawings, with or without antialiasing algorithms, need a lot of modifications or are impractical because their algorithmic structures are too complicated.

Another aspect of this research deals with VLSI automation at the physical design level. Although this automation should involve several different processes, i.e., placement, routing and mask generation, we limit our attention to the routing and mask generation of regular VLSI structures. These regular VLSI structures are standard-cell and gate-array structures, which are known to have a low design-time and high regularity. In general, two-layer channel routing is one of the major routing methods for these regular VLSI structures, and the goal is to find the optimal solution for the given netlist in each channel. The establishment of an optimal result for the two-layer channel routing problem has been shown to be NP-complete. There exists several two-layer channel routers that use heuristic approaches to generate optimal or near optimal routing results. Due to



(a)



(b)

Note that each "\*" represents the pixel center;  
each unit square in the columns represents the area of each pixel.

Figure 1-2. Examples of a straight line display on the pixel plane: (a) the ideal line;  
(b) the rasterized line.

different construction strategies of standard-cell and gate-array structures, several different channel routing cases can be found during the routing. These are: routing without vertical constraints, without doglegs, with restricted doglegs, and with unrestricted doglegs. The "left edge" channel router [8] attempts to minimize the placement of horizontal segments in each track. This algorithm can find a good solution if there are no vertical constraints, but it can not generate an optimal total wiring length for the given netlist. Several other existing methods, with or without doglegs, can be used to solve routings with vertical constraints. Basically, the use of graph theory to minimize the number of tracks has proven to be a very effective method [9-10]. It constructs the links and nodes of the graph according to the vertical sequence of given nets. Both dogleg and non-dogleg approaches can be applied to this method and can generate a reasonable result for the given routing with vertical constraint problems. The execution speed and the weighting assignment to each net, however, are the major problems with this method. Shin and Reed described a method which uses maze routing as well as rip-up and reroute approaches to reduce both the number of tracks and vias in the channel [11-12]. This method can generally be used in two-dimensional channel routing. It basically uses the dogleg approach to solve the vertical constraint problem but needs the correct weighting parameters set in order to find an optimal result. The main deficiencies of these methods are that they have long execution times and are inflexible for mixed-mode routing operations.

Finally, most channel routers use symbolic arrays to represent their routing results, and then use a mask generator to translate the symbolic arrays into mask results. Basically, the mask generator [13-14] uses the same method in generating functional components as well as wirings; it has to examine the symbolic array of the routing several times in order to generate the correct wiring masks. Thus, the whole execution time of this approach is relatively long, and it is difficult to verify the correctness of the mask

results unless the simulation tools are applied to the final layout.

## **1.2 Research Goals and Objectives**

The goal of this research is to apply VLSI technology to circuit design in order to increase product speed while reducing product area. To achieve this goal, we focus on the mapping of algorithms to architectures for graphic drawing engines and on the layout tools design for VLSI automation at the physical level. Our first research objective is to provide the user a high performance graphic drawing engine for quickly generating realistic drawings in a raster system. The second research objective is to provide the VLSI designer useful design tools to deal with VLSI layout problems at the physical level. Of course, these layout tools can be used to implement our drawing engine for translating its architecture to the physical layout. To achieve these two objectives, several related tasks have to be met.

### **1.2.1 The High Performance Graphic Engine**

In an advanced computer graphics system, the implementation of useful functional blocks in hardware can generally be used to improve the system performance. The efficient utilization of VLSI design methodologies can reduce the area and increase the execution speed of the product over that of the traditional printed-circuit board design. For developing a high performance drawing engine, the VLSI design methodology should be used throughout the entire design process. Therefore, our first task was to identify and investigate the suitability of using alternative drawing and antialiasing techniques. The intent here was to identify those algorithms, or portions of algorithms that could be best candidates for being efficiently and effectively mapped into VLSI architectures. In order to construct the drawing engine for a raster system, the architecture of the

raster system must also be investigated. Since the developed drawing engine should have the ability to quickly generate very realistic images, our second task was to develop a flexible antialiased drawing algorithm and its architecture. And, finally, our third task was to develop a technique for estimating the performance of the design. The approach used to realize these task objectives is described in the related chapters which follow.

### **1.2.2 Efficient VLSI Layout Tools**

In order to create an acceptable layout with a fast execution speed, VLSI layout tools are usually used in the physical implementation. Without considering the placement in the regular VLSI structural layout, we limited our attention to the two-layer channel routing with the mask generator problem. As stated previously, four routing cases can be obtained from the routing operation. Thus, our immediate objective was to develop several two-layer channel routers which can be used to deal with the different routing cases. Although our results should be general, our primary intent is to apply these tools to the physical implementation of our drawing engine.

Three tasks were defined to achieve this objective. We first identified and investigated the most promising set of routing and mask generation concepts from existing algorithms, which led to development of a much better router. A bottom-up layout approach was used in order to produce the testability feature for the routers. Our strategy was to first develop a powerful mask generator and then the routers. Thus, our second task was to develop the powerful mask generator, which provides a simple and clear interface with the routing operation. And, finally, our third task was to develop efficient two-layer channel routers to efficiently handle various routing cases. In this task, several useful channel-routing techniques were developed and refined. The approaches used to achieve these tasks are described in later chapters.



### **1.3 Thesis Overview**

Consistent with the stated objectives and tasks, this thesis is organized as follows: Chapter 2 provides an overview of the basic drawing methods, as well as several antialiasing techniques. It also presents several hardware organizations of raster systems. Chapter 3 presents a new approach in computer graphic drawing with antialiasing; it preserves the incremental drawing feature, while improving the display quality and executing speed of the developed antialiasing algorithm. Several comparisons of the developed antialiasing algorithm and the existing algorithms are also provided in Chapter 3. In addition, it contains a VLSI architectural design of this antialiasing algorithm and one generic VLSI implementation of the line-drawing algorithm. Chapter 4 provides the background information for several existing two-layer channel routers and mask generators. Chapter 5 describes the tiling mask generator and three efficient two-layer channel routers. It also includes the verification of the correctness of routing results and the evaluation of the performance of the routers by using several examples. Chapter 6 contains a summary of this research work and some recommendations for future research directions.

## **CHAPTER 2**

### **EXISTING COMPUTER GRAPHIC DRAWING AND ANTIALIASING ALGORITHMS**

The beginning of modern interactive graphics may be traced to Sutherland's work in 1963 [15]. At the same time, many engineering disciplines recognized the enormous potential for automating drafting and drawing activities in computer-aided design (CAD) and computer-aided manufacturing (CAM). By using the basic idea from television, the raster display is one of the important display components in interactive graphics systems. The image is formed from the raster, which is a set of horizontal lines each made up of individual light spots known as pixels. Thus, the raster is simply a matrix of pixels covering the entire screen area. The development of inexpensive solid-state memory made raster graphics feasible because they provide refresh buffers considerably larger and faster than those of decade ago. Thus, with these attractive features, raster graphics systems are widely used in today's computer applications.

In this chapter, we investigate several alternative architectures for the raster system in order to identify the functionality of our drawing engine. This functionality should be specified before proceeding with architectural and layout design phases. So, this chapter presents the interfaces of our drawing engine with both the frame buffer and curve-drawing operations in a suggested raster system architecture. In addition, in order to generate a high-performance drawing engine, this chapter outlines the approaches to identify suitable drawing and antialiasing techniques. Although most existing drawing algorithms can be used to generate aliased or antialiased results, their algorithmic complexities and display qualities are quite different. Thus, a reduction method is applied to the whole drawing algorithmic domain for selecting the suitable drawing technique and useful

drawing concepts. This selected drawing technique should have the operational flexibility to be capable of generating accurate antialiasing data with a high execution speed.

Therefore, the most important features of a high-performance graphic drawing engine are high drawing speed, antialiasing and extensibility. This drawing engine should be designed to draw display primitives in a relatively high speed in order to support the requirement of the high-resolution display. Since the line-drawing is the fundamental drawing primitive in a raster system, the line-drawing technique is the key topic of our entire investigation. By using either the piecewise or scan-converting curve methods, this line-drawing engine can be applied to curve-drawing operations. Therefore, this chapter includes the discussion of these approximated curve-drawing methods. Finally, in order to identify the suitable antialiasing technique, this chapter also presents two major antialiasing techniques, i.e. the window averaging technique and the area antialiasing technique.

In order to identify those useful drawing and antialiasing techniques, this chapter is organized as follows: Section 2.1 provides an overview of the functional model of a high-performance raster system. It includes the discussion of several functional representations and processing elements from the graphics applicational level to the final physical display level. Section 2.2 presents several line-drawing algorithms and two useful curve approximation techniques. It includes the identification of useful line-drawing concepts for our drawing engine. In addition, Section 2.2 also presents the interface between the line-drawing and complicated curve-drawing operations. In Section 2.3, we present a survey of useful antialiasing algorithms. These can be used to identify the best candidate for our development the antialiasing engine.

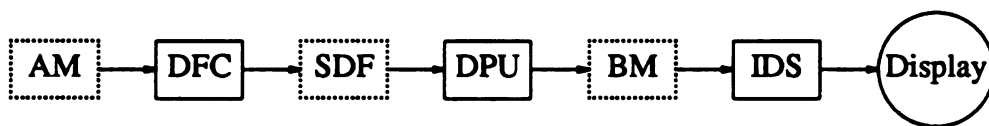
## **2.1 A High Performance Raster System Overview**



The design of graphics systems has been a challenging topic of study for several decades; the demand of ever-increasing performance has always pushed the available technology to its limits. Based on the different design emphases and strategies to organize the systems, several existing architectures [16-20] have been implemented in today's raster graphics systems. Although their input formats and output performances are quite different, the fundamental functional models of these systems are similar. This section includes a discussion of a basic functional model for raster systems as they related to existing architectures.

Figure 2-1 illustrates a basic functional model of high-performance raster systems [27]. This model can be treated as a generalized functional model of modern raster systems. The functionality of those representations and processing elements are as follows:

- The AM (the application model) contains a description of both the graphical and non-graphical properties of an object in a format determined by the application program and/or modeling package.
- The DFC (the display file compiler) is the part of the application program that contains the model traverser and calls to the graphics package for mapping the AM to the SDF.
- The SDF (the structured display file) contains a description of the graphical representation of the object.
- The DPU (the display processing unit) contains several drawing primitives used to map SDF to BM.

- The BM (the bit map) holds the scan-converted images and can be transformed to the display screen.
- The IDS (image display system) reads both bit maps and color-table maps and translates them into display monitor signals.



Note that  means the representation;  
 means the processing element.

**Figure 2-1. Functional model of a high-performance raster system:**  
 AM - Application Model; DFC - Display File Compiler; SDF - Structured Display File; DPU - Display Processing Unit; BM - Bit Map; IDS - Image Display System.

Usually, the AM and DFC are prepared and processed in the high computational unit such as a host computer because they require a high computational power to process all the possible objects. The SDF is generated after the execution of the DFC in the host computer. A portion of the memory in the raster system is always used for storing the SDF in order to provide related instructions to the DPU.

The design of the DPU is an important task since the quality and the speed of displayed images are heavily dependent on the design of the DPU. The use of built-in functional calls is a traditional method to implement the function of graphics primitives in the DPU. For example, Intel's 82720 [16] and 82786 and TI's 34010 [18] are imple-

mented by adopting this method. These graphics engines show a flexible feature in handling the display primitives and in linking with the host computer. However, because they do not contain the dedicated hardware for drawing primitives and antialiasing operations, their main deficiencies are low drawing speeds and poor realism results. Another useful method to implement the DPU is through the use of dedicated hardware for accelerating the speed of drawing operations. The SEILLAC-7 [19] utilizes a custom ECL DDA chip to achieve a high line drawing rate. This is a typical example of using the dedicated hardware to implement a drawing engine. Since a matrix encoder is included in the DDA circuit design, this chip can be used to alleviate the staircasing effects in drawing a line-segment. This design can only provide a small number of intensity levels for a line-drawing antialiasing operation. In addition, it can not provide various line-width options for users. Thus, the lack of good quality and flexibility of generated images are the main deficiencies of this DDA implementation. However, this method provides a useful concept for designing low-level functions, while all the geometric drawing functions are handled at the high level.

The design of the BM and IDS are much simpler than that of in the DPU. Basically, they are used to translate the images from the pixel-plane to raster displays. Their tasks include bit-map and color-map managements and the interfacing logic design. Typical examples of the BM and IDS designs can be found in the SEILLAC-7 [19] and MEGATEK's 7200 [20]. According to their design, the image translated from the frame buffer to a display is fast enough to provide a high resolution display.

Our research emphasizes the development of a crucial drawing element in order to provide a high-performance drawing engine. Actually, the functionality of this drawing element is to perform a low-level drawing operation of the DPU and to translate its results to the BM. Therefore, by following the similar architectural approach of the SEILLAC-7, our design focuses on the improvement of its short-comings in drawing

with the incorporation of antialiasing. Figure 2-2 illustrates our research focus in a complete raster system organization. The following two sections present a survey of existing line-drawing algorithms and antialiasing algorithms. In addition, in order to provide the useful extensions of the line-drawing design, the next section also presents several useful curve-drawing approximations.

## **2.2 Graphic Drawing Algorithms**

Since the 1960's, several curve drawing algorithms have been developed for drawing the geometric images on raster displays. Basically, two kinds of curve-drawing techniques are used to generate the graphic drawings, i.e., the parametric and non-parametric drawing techniques. By adopting these two techniques, several existing curve-drawing algorithms can be used to generate curves. But, they are different in the suitable application level, the algorithmic complexity and the execution speed. Because our attention is on the development of a high speed drawing engine, the survey of existing curve-drawing algorithms focuses on their adaptabilities with a basic drawing primitive.

In an advanced computer graphics system, a high speed drawing engine is always required for quickly displaying the images, and its fundamental drawing primitive is the line-drawing. Basically, this line-drawing element not only can be used to generate lines in the raster displays but also can be applied to other peripheral devices, such as digital plotters, etc.. Therefore, in Section 2.2.1, we provide a detail investigation of existing line-drawing algorithms, especially scan-converting line algorithms. And, Section 2.2.2 contains a general discussion of useful curve-drawing algorithms for the raster display in both parametric and non-parametric drawing techniques. It also provides several curve-drawing approximations useful for us to consider their interfaces with a basic line-drawing primitive.

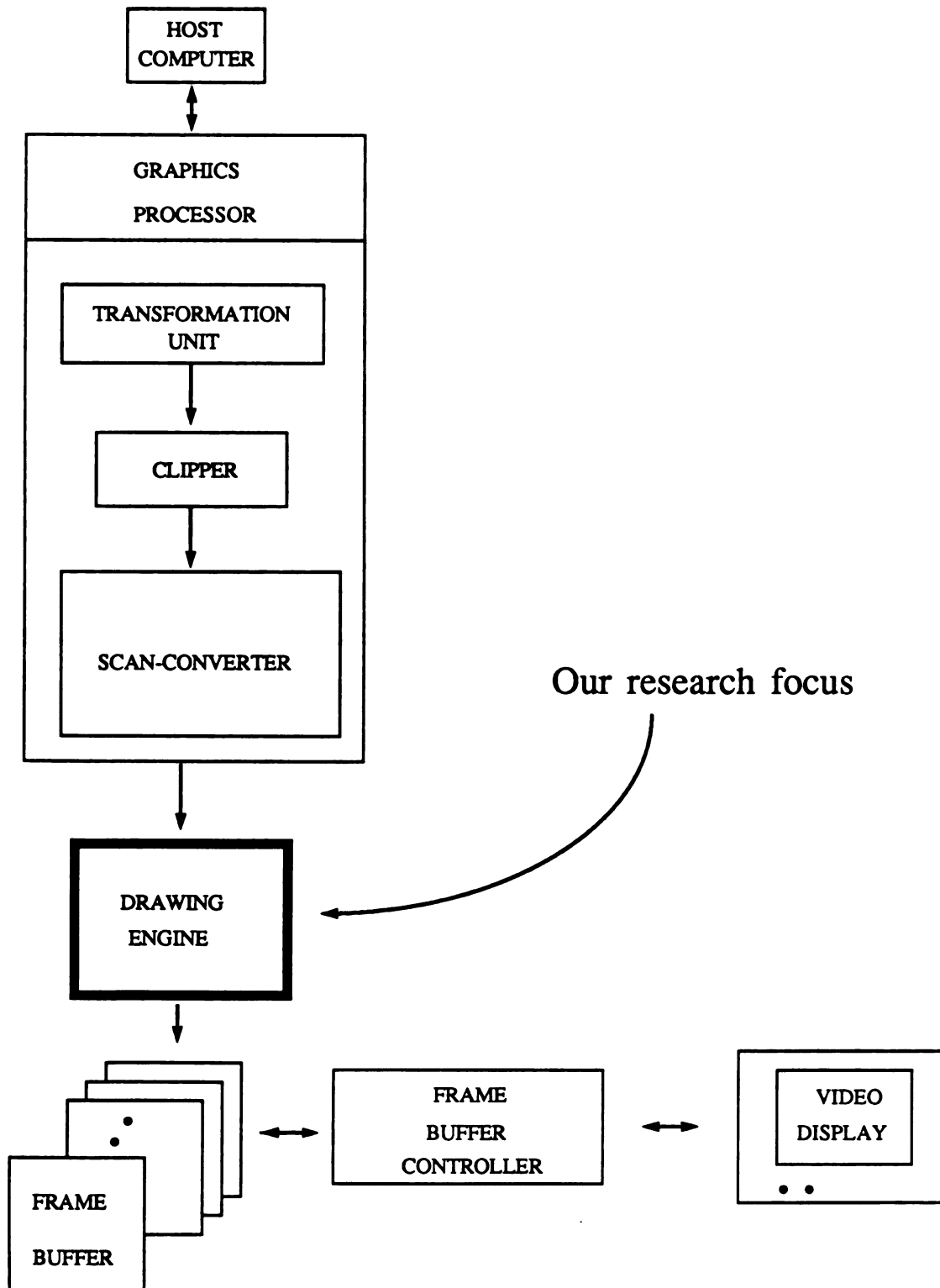


Figure 2-2. Our research focus in a complete raster system organization.



### 2.2.1 Existing Line-Drawing Algorithms

Since a cathode ray tube (CRT) raster display can be considered a matrix of discrete cells (pixels), it is not possible to directly draw a straight line from one point to another. The rasterization is a determination of which pixels will provide the best approximation to the desired drawing on a raster. The traditional line-drawing algorithms used in the line rasterization are digital differential analyzer (DDA) [21], the Bresenham line-drawing algorithm [22] and its extensions [4]. In order to increase the speed of the line-drawing, several alternative methods [23-25] have been developed by using memory mapping methods.

(1). The digital differential analyzer (DDA) algorithm:

The DDA is a technique uses a numerical method to solve the differential equations in order to obtain a rasterized straight line. For a straight line illustrated in Figure 2-3, its line-slope can be stated as:

$$m = \frac{dy}{dx} = \frac{y_2 - y_1}{x_2 - x_1}.$$

The rasterized solution of this line is

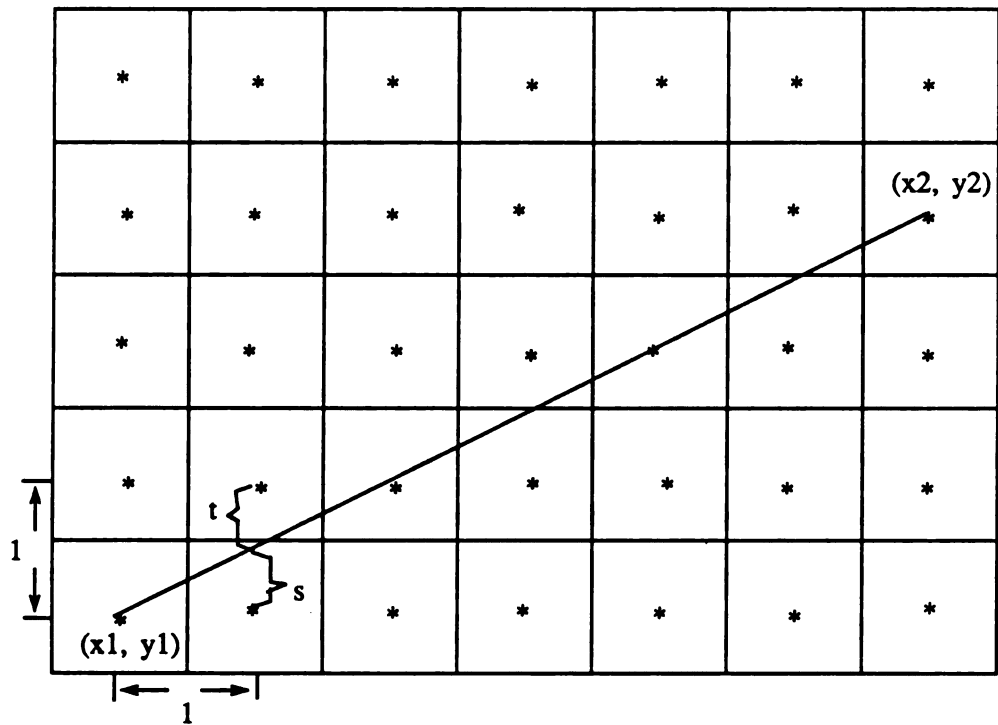
$$\begin{aligned} y_{i+1} &= y_i + dy; \\ y_{i+1} &= y_i + \frac{y_2 - y_1}{x_2 - x_1} dx, \end{aligned} \quad (2.1)$$

where  $x_1, y_1$  and  $x_2, y_2$  are the endpoints of the line,  $0 \leq m \leq 1$ , and  $y_i$  and  $y_{i+1}$  are the current and the next values of  $y$  at any given step. For  $dx = 1$ , Equation 2.1 can be rewritten as

$$y_{i+1} = y_i + m. \quad (2.2)$$

In fact, Equation 2.2 shows a recursion relation of successive values of  $y$  along the straight line. According to this method, the selected pixel-coordinates are represented in





Note that each "\*" represents the pixel center;  
each unit square represents the area of the pixel.

Figure 2-3. Example of a straight line mapped onto a pixel plane.

a non-integer format. Therefore, to find the closest integer value of the selected coordinate, a rounding operation is used by adding 0.5 to its original value and then processed by a truncation function. The listing of this simple DDA algorithm can be found in Roger's book [26].

Four different drawing cases need to be considered in the DDA, i.e.  $m > 1$ ,  $1 \geq m \geq 0$ ,  $0 \geq m \geq -1$  and  $m < -1$ . According to Equation 2.2, the calculation of finding  $y_{i+1}$  involves a floating-point operation because the maximum value of  $y$  can be  $2^{10}$ , for a screen with 1024X1024 pixels. Because floating-point is a relatively slow process, several modifications are necessary in order to provide a fast rasterized line-drawing speed. A revised algorithm of this DDA, the Bresenham line-drawing algorithm, is a very efficient and popular for drawing line segments with integral endpoints.

(2). The Bresenham line-drawing algorithm:

Although originally developed for digital plotters, Bresenham's algorithm [22] is equally suitable for use with CRT raster devices. With the same four operational cases described above, the Bresenham line-drawing algorithm uses integral arithmetic to incrementally generate the pixels needed to approximate a straight line. Except for this, the main difference between the DDA and Bresenham algorithms is in their initial phases. During the initialization phase of the Bresenham algorithm, the x-axis is used as the major incremental axis if  $dx$  is larger than  $dy$ . Whereas, the y-axis is used as the major incremental axis if  $dy$  is larger than  $dx$ . Note that we also use notations of the DDA algorithm in the discussion of the Bresenham algorithm. This algorithm can be classified as a two-point method because it uses a difference between  $s$  and  $t$  to select the pixel, where  $s$  and  $t$  are the distances from the lower and upper pixels to the straight line respectively. By using the same line example from the discussion of the DDA algorithm, Figure 2-3 also shows the relation of these two parameters with a line. Thus, a decision variable  $d_i$  is

used to select the pixels of the approximated line, where  $d_i = dx(s - t)$ . Assume the operation is derived under the  $0 \leq m \leq 1$  case. Then, the initial value of  $d_i$  is

$$\begin{aligned} d_1 &= dx(m - (1 - m)) \\ &= dx(2m - 1) \\ &= 2dy - dx. \end{aligned} \quad (2.3)$$

The relationship between two consecutive decision variables is determined by two cases; i.e.,  $d_i \geq 0$  and  $d_i < 0$ . If  $d_i \geq 0$ , then the upper pixel is selected, and

$$d_{i+1} = d_i + 2(dy - dx). \quad (2.4)$$

If  $d_i < 0$ , then the lower pixel is selected, and

$$d_{i+1} = d_i + 2dy. \quad (2.5)$$

The complete listing of Bresenham's line-drawing algorithm can be found in Foley and Van Dam [27]. Since the algorithm only requires integral additions and subtractions, its execution speed is relatively fast when compared with other approaches. But the main disadvantage of using the Bresenham line-drawing algorithm is the lack of antialiasing in the generated lines.

(3). The extension to the Bresenham line-drawing algorithm:

Pitteway and Watkinson [4] have developed an efficient extension to Bresenham's line-drawing algorithm, i.e., the PW algorithm. This algorithm can be used to reduce the aliasing and jaggling effect on the edge of a polygon of the raster display. Basically, it uses the sign of the difference between the mid-point of two pixels to the straight to select pixels of the rasterized line. Therefore, this algorithm can be classified as a mid-point method. In addition to using a decision variable,  $d_i$ , this method utilizes a reference,  $a$ , where

$$a = \begin{cases} 1 - m & \text{if } 0 \leq m \leq 1; \\ 1 - (1/m) & \text{if } m > 1; \\ 1 + m & \text{if } -1 \leq m < 0; \\ 1 + (1/m) & \text{if } m < -1. \end{cases} \quad (2.6)$$

At the initialization of the algorithm,  $d_0$  is set to  $1/2$ , i.e., the half scale of the distance between two consecutive pixels. With the consideration under the  $0 \leq m \leq 1$  case, the processes used to determine two consecutive decision variables are listed below. If  $d_i < a$ , then

$$d_{i+1} = d_i + m, \quad (2.7)$$

and the lower pixel is selected. If  $d_i \geq a$ , then

$$d_{i+1} = d_i - a, \quad (2.8)$$

and the upper pixel is selected.

Note that the value of  $d_i$  is controlled between  $-1$  and  $1$ . Basically, the structure of this PW algorithm is similar to that of the Bresenham line-drawing algorithm. But, because the decision variable is a constant at the initial phase of the PW algorithm, the initial calculating time of it is less than that of the Bresenham line-drawing algorithm. However, according to Equations 2.7 and 2.8, non-integral operations are required in the algorithm because  $m$  is a non-integer number and the value of  $d_i$  is used for edge-antialiasing. Section 2.3 provides the detail discussion of this PW algorithm applied to antialiasing operations.

#### (4). Alternative line-drawing algorithms:

Bresenham proposed an incremental line compaction method [23]. This method represents a line by using both the run length and the repeated pattern encoding techniques. By using several mapping tables, it can simultaneously generate multiple pixels in each decision cycle. But, this line compaction method needs several memory tables and techniques to send the various lengths of results to frame buffer. Therefore, the line compaction method may be fast in a long line, but for a short line, the method is slower than the incremental Bresenham line-drawing algorithm.

Sproull [24] uses program transformations to derive line-drawing algorithms. Basically, this transformation technique is similar to the translation of the DDA algorithm to the Bresenham line-drawing algorithm. This technique uses the  $n$ -step parallel incremental line-drawing operation for generating multiple pixels in one step. Thus, this transformation technique is useful in a multi-processing environment to generate several pixels in each execution cycle. The initialization times for all the processes, however, are totally different since different number of loops are used for calculating initial values of several variables. Also, the maximum deviation of a given result to an optimal line may be greater than one pixel. In addition, the line may have gaps or nonmonotonicities results while using this technique for drawing stroke patterns.

Wu and Rokne [25] introduced a technique used for a double-step incremental line-drawing generation. Basically, this technique uses basic concepts from both the Bresenham line-drawing algorithm and the memory mapping method to generate two consecutive pixels in each drawing step. But, in  $0 \leq m \leq 1$  operating condition, this algorithm needs to further separate the lines into two cases, i.e.,  $m < 1/2$  and  $m \geq 1/2$ . There are four fixed patterns of pixels and each of them contains two consecutive selected pixels. Once these four patterns of pixels are stored, this double-step incremental line-drawing algorithm can be executed faster than that of the Bresenham line-drawing algorithm. The irregularity of processing and the use of more registers than that of the Bresenham line-drawing algorithm are the main deficiencies of this algorithm. This double-step algorithm also uses three fixed intensity levels for solving aliasing problem of the line-drawing. As a result, poor realism of images may be generated by using this algorithm.

### 2.2.2 The Curve-Drawing Technique

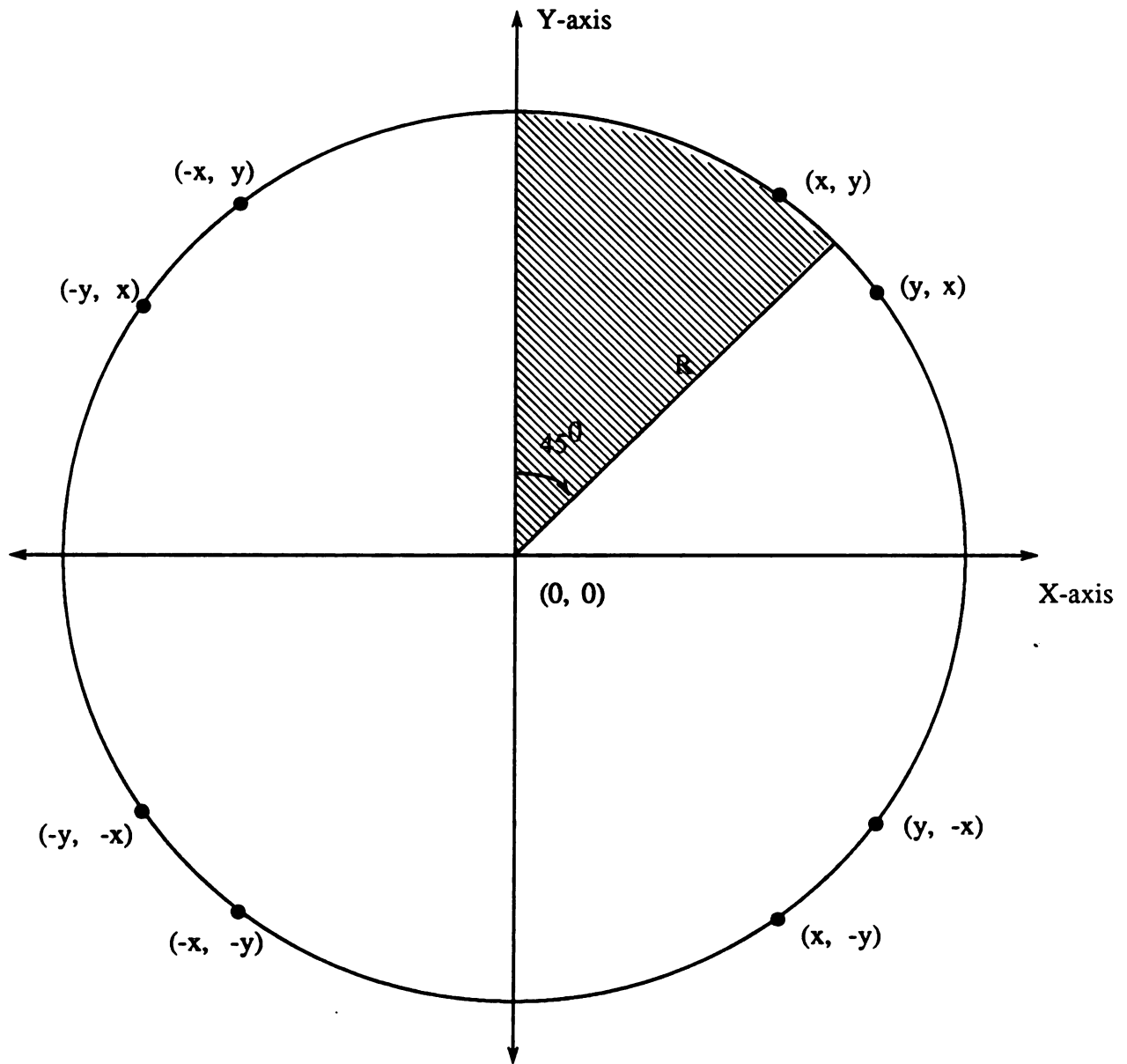
There are three methods that can be used to generate rasterized curves. Pixel map-

ping is the simplest method used to find all the related pixels of a curve according to the solution of its curve-equations. Since this method is mathematically impractical, this section does not include the discussion of this method. The piecewise linear approach is another method used to generate a number of key points on a curve. A rasterized curve is formed by connecting these points with related line-segments. By using this method, several useful curve-drawing algorithms can be implemented at the high drawing level of the DPU in order to generate approximated curves. A scan-curve conversion method is also useful for generating rasterized curves for raster systems. With the same operational strategies as that of the scan-line conversion, this scan-curve conversion is the most efficient curve-drawing method. According to the result of decision variables, it can incrementally generate pixels of a curve. In the next two sub-sections, we present several existing curve-drawing algorithms which use the scan-curve conversion and the piecewise approximation methods.

#### **2.2.2.1 Scan-Converting Curves**

Usually, the scan-curve conversion method is applied to the non-parametric curve-drawing algorithms to generate rasterized curves. For example, the Bresenham circle-drawing algorithm [28] is one of the most efficient and easiest algorithm of the circle-drawing algorithms. Because of symmetry, this algorithm only needs to find pixels of a circle in the second octant, i.e., a circle in  $y \geq x \geq 0$  region as shown in Figure 2-4. The rest of pixels can be generated by adding constant data to those pixels in the second octant of the circle. Thus, during the processing of the Bresenham circle-drawing algorithm, the x-axis is treated as an incremental axis, and it increments from  $x = 0$  to  $x = R/\sqrt{2}$  with step-size 1. Then, the y-axis is the decremental axis, and it decreases from  $y = R$  to  $R/\sqrt{2}$  by 1 or 0 each step. For simplicity, the center  $(x,y)$  of a circle is transformed to the origin  $(0,0)$ . By using a similar decision variable  $d_i$  of the Bresenham line-drawing





Note that the circle-center is  $(0, 0)$  and circle-radius is  $R$ ;

$(x, y)$  is any random point in the second octant of the circle.

Figure 2-4. Symmetrical property of a circle.

algorithm, this circle algorithm also uses the sign of  $d_i$  to choose pixels. The initial value of  $d_i$  is derived as below:

$$\begin{aligned} d_1 &= [(x_0 + 1)^2 + y_0^2] - R^2 + [(x_0 + 1)^2 + (y_0 - 1)^2] - R^2 \\ &= 3 - 2R, \end{aligned} \quad (2.9)$$

where  $(x_0, y_0)$  is equal  $(0, R)$ . The relationship between two consecutive decision variables is determined by two case,  $d_i \geq 0$  and  $d_i < 0$ . If  $d_i \geq 0$ , then the y-coordinate of the selected pixel is decreased by 1, and

$$\begin{aligned} d_{i+1} &= [(x_{i-1} + 2)^2 + (y_{i-1} - 1)^2] - R^2 \\ &\quad + [(x_{i-1} + 2)^2 + (y_{i-1} - 2)^2] - R^2 \\ &= d_i + 4(x_{i-1} - y_{i-1}) + 10. \end{aligned} \quad (2.10)$$

If  $d_i < 0$ , then the y-coordinate of the selected pixel is unchanged, and

$$\begin{aligned} d_{i+1} &= [(x_{i-1} + 2)^2 + (y_{i-1})^2] - R^2 \\ &\quad + [(x_{i-1} + 2)^2 + (y_{i-1} - 1)^2] - R^2 \\ &= d_i + 4x_{i-1} + 6. \end{aligned} \quad (2.11)$$

The complete listing of Bresenham's circle-drawing algorithm can be found in Foley and Van Dam [27]. Because this algorithm uses integral addition, subtraction and shifting operations to generate eight pixels each step, its execution speed is relatively fast. The main difference between Bresenham's circle and line algorithms is that an additional factor added to  $d_i$  is a variable in the Bresenham circle-drawing algorithm.

Jordan, et al. [2] developed a generalized curve-drawing technique directly from the non-parametric representation of the curve, i.e.,  $f(x, y) = 0$ . Their method uses continuous derivatives of the function,  $f(x, y)$ , and a direction variable to determine next selected point. While used for drawing circles, the Bresenham circle-drawing algorithm is faster than the generalized curve-drawing technique.

An efficient non-parametric curve-drawing technique [29] uses the mid-point

method to incrementally generate lines, circles, ellipses, parabolas and hyperbolas. This technique is much simpler and efficient than those of curve-drawing algorithms described above.

One main disadvantage of these scan-curve conversion algorithms is that the curve equation used is limited to second-order curves, and it is very difficult to extend them to much high order curves. In next section, we present a useful piecewise approximation method. This method can be used to generate much high order curves.

#### 2.2.2.2 Piecewise Curves

The recursive and interpolating techniques are two major methods used to generate piecewise curves. Both techniques are usually applied to the parametric curve-drawing algorithms since they can easily manipulate the parameters for generating the related points of the curve.

##### (1). The recursive technique:

The recursive technique uses a constant increment of the selected coordinate to generate several approximated points of a curve, where the number of points is defined by users. Basically, the next generated point and the current point are related by a function, i.e.,

$$\begin{aligned}x_{i+1} &= f(x_i, y_i); \\ y_{i+1} &= g(x_i, y_i),\end{aligned}\tag{2.12}$$

where  $f$ ,  $g$  are functions of both  $x$  and  $y$ . The use of the recursive technique for the circle-drawing is discussed below.

Assume a circle is described in polar coordinates, i.e, the circle parameters in angle  $\theta$  and radius  $r$ . The equations for circle are

$$\begin{aligned}x &= r \cos\theta + x_c; \\y &= r \sin\theta + y_c,\end{aligned}\tag{2.13}$$

where  $x_c$  and  $y_c$  are the coordinates of the circle-center. For the simplicity, the coordinates of the center are set to the origin (0,0). If we choose  $(x_i, y_i)$  as a starting point of a circle. With an increment angle  $\delta\theta$ , the coordinates of the next point are

$$\begin{aligned}x_{i+1} &= r \cos(\theta_i + \delta\theta); \\y_{i+1} &= r \sin(\theta_i + \delta\theta).\end{aligned}$$

Double-angle trigonometric formulas expand the right hand side to

$$\begin{aligned}x_{i+1} &= r (\cos\theta_i \cos\delta\theta - \sin\theta_i \sin\delta\theta); \\y_{i+1} &= r (\cos\theta_i \sin\delta\theta + \sin\theta_i \cos\delta\theta).\end{aligned}$$

Since  $x_i = r \cos\theta_i$  and  $y_i = r \sin\theta_i$ , the above equations can be rewritten as

$$x_{i+1} = x_i \cos\delta\theta - y_i \sin\delta\theta;\tag{2.14}$$

$$y_{i+1} = x_i \sin\delta\theta + y_i \cos\delta\theta.\tag{2.15}$$

Therefore, according to Equations 2.14 and 2.15,  $n$  points are generated from the entire operation, where  $n$  is a number defined by the user, and the value of  $\delta\theta$  is equal to  $2\pi/n$ . After connecting these sequentially generated points with line-segments, we have an approximate circle. As a result, the larger the  $n$  is assigned, the smoother the circle can be generated. Dewey [30] contains the subroutine of this circle-drawing algorithm and its extension to other curve-drawing operations.

## (2). The interpolating technique:

The interpolating technique is another method that can be applied to generate piece-wise curves. It uses a constant increment of a parameter to calculate several blending functions. These blending functions are used for curve interpolation. The number of blending functions and the number of generated points are also defined by the user. The

basic idea of this technique is to approximate a portion of an unknown curve by filling it with pieces of known curves which pass through the nearby sample points.

Suppose a curve is expressed in the parametric form:

$$\begin{aligned} x &= f_x(u); \\ y &= f_y(u), \end{aligned} \quad (2.16)$$

where  $f$  is the curve function and  $u$  is a parameter. Assume this curve passes through  $m$  sample points, i.e.,

$$(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m).$$

Then the curve can be approximated by using blending functions,  $B_i(u)$ , i.e.,

$$\begin{aligned} x &= f_x(u) = \sum_{i=1}^m x_i B_i(u); \\ y &= f_y(u) = \sum_{i=1}^m y_i B_i(u), \end{aligned} \quad (2.17)$$

For each value of  $u$ , these blending functions determine how much the  $i$ th sample point affects the position of the curve. In order to make the selection of blending functions simpler, the value of the first blending function is defined as:

$$\begin{aligned} B_1 &= 1, \text{ if } u = -1; \\ B_1 &= 0, \text{ if } u = 0, 1, 2, \dots, (m-1). \end{aligned} \quad (2.18)$$

Therefore, the equation of the first blending function can be written as:

$$B_1(u) = \frac{u(u-1)(u-2)\dots[u-(m-2)]}{(-1)(-2)(-3)\dots(1-m)}. \quad (2.19)$$

The use of general expressions of this method is so called Lagrange interpolation. Four of consecutive sample points are considered for constructing the approximated curve; thus, four blending functions are required for interpolation. For  $m = 4$ , equations of these four blending functions are written as:

$$B_1(u) = \frac{u(u-1)(u-2)}{(-1)(-2)(-3)} \quad (B_1 = 1, \text{ if } u = -1);$$

$$\begin{aligned}
B_2(u) &= \frac{(u+1)(u-1)(u-2)}{(1)(-2)(-3)} \quad (B_2 = 1, \text{ if } u = 0); \\
B_3(u) &= \frac{(u+1)(u)(u-2)}{(2)(1)(-2)} \quad (B_3 = 1, \text{ if } u = 1); \\
B_4(u) &= \frac{(u+1)(u)(u-1)}{(-1)(-2)(-3)} \quad (B_4 = 1, \text{ if } u = 2),
\end{aligned} \tag{2.20}$$

By using these functions and four sample points, a curve may be approximated. The equations of this curve are

$$x = x_1B_1(u) + x_2B_2(u) + x_3B_3(u) + x_4B_4(u); \tag{2.21}$$

$$y = y_1B_1(u) + y_2B_2(u) + y_3B_3(u) + y_4B_4(u). \tag{2.22}$$

Given four sample points, a region of this curve is approximated by using Equations 2.21 and 2.22. The approaches to find the good approximation of a curve are described below. The closest portion of this approximated curve lies between the second and third sample points, i.e., the value of  $u$  is between 0 and 1. Given four sample points (1, 2, 3, 4), the curve between middle two points (2, 3) is approximated by using  $n$  line segments, where  $n$  is defined by the user. Thus, we can get  $n$  consecutive sets of blending functions and points by applying different values of parameter  $u$  to the equations, where  $u = 1/n, 2/n, 3/n, \dots, 1$ .

The curve approximation in the first region (1, 2) is treated a little different than that in the region (2, 3). Now, the value of  $u$  is between -1 and 0, instead of 0 and 1. The calculation of the rest regions of the curve follows the same method of approximating region (2, 3). The only difference is it uses different sets of those four sample points. For example, to approximate the curve at region (3, 4), those given sample points will become (2, 3, 4, 5), and so on. The curve approximation in the final region ( $m-1, m$ ) uses the similar method as in the first region, where  $m$  is the last sample point. Now, the value of  $u$  is between 1 and 2, instead of -1 and 0, and those given sample points become ( $m-3, m-2, m-1, m$ ).

The algorithm mentioned above provides only the basic idea of interpolation by using blending functions. It does not consider the first-order continuity at all endpoints and the unity of blending functions,  $\sum B_i(u)=1$ . Several adjustments of the algorithm are introduced by Harrington [31] in order to provide better curve approximations. Instead of using approximated points, B splines uses several control points and a new set of blending functions to provide the unity of blending functions and both the first and the second-order continuities at all the generated endpoints. Another interpolation method is provided in Bezier curves [32]. It uses four control points and the approximated derivatives to generate all the endpoints with the first-order continuity feature and the approximated curve with a convex hull feature.

Therefore, the quality of the generated curve depends on the algorithm and the number of iterations we used. Because the line-drawing algorithm is used to connect those generated endpoints, the jaggy and staircase appearances still exist in the approximated curves. In order to improve this short-coming, several existing antialiasing algorithms are investigated in the next section.

## 2.3 Existing Antialiasing Techniques

Fundamentally, the appearance of aliasing effects is due to the fact that geometric drawings are continuous, whereas a raster device is discrete. Antialiasing techniques are generally used to reduce these display degradations generated from using the drawing operations. The simplest method is to increase the resolution of the raster to achieve the antialiasing. However, there is a limitation to the ability of CRT raster scan devices to display very fine rasters; presently, the practical limitation is about 2000 pixels per scan line. In addition to this method, two major antialiasing techniques are used for image antialiasing during or after executing a drawing operation. Section 2.3.1 presents the

window-averaging technique. It uses the post-filtering approach to apply a filtering function to the image after the image are drawn. Section 2.3.2 presents the area-antialiasing technique. It uses the pre-filtering approach to apply a filtering function to the image during the drawing operation.

### 2.3.1 The Window-Averaging Technique

The basic idea of the window-averaging is derived from increasing the sampling rate of the raster, and this technique is also called pixel averaging. It applies some types of averaging functions to the image at high resolution and obtains pixel attributes of this image at lower resolution [3, 33].

Basically, a window is similar to a filtering function during the averaging operation. In order to derive this window averaging technique, we list a two-dimensional expression of the discrete convolution below:

$$D(i, j) = \sum_{k=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} F(k, m) H(i-k, j-m), \quad (2.23)$$

where  $D$  represents the filtered scene produced by convolving the scene  $F$  with the filter  $H$ ;  $i$  and  $j$  are coordinates at the raster;  $k$  and  $m$  are parameters used for representing all the convolved area of the given point of an image. Since filtering operations in both the  $x$  and  $y$  directions are independent, this expression can be rewritten as

$$D(i, j) = \sum_{k=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} F(k, m) H_i(i-k) H_j(j-m), \quad (2.24)$$

where  $H_i$  and  $H_j$  are the filter functions at the  $x$  and  $y$  directions.

If each point of the image is approximated by a rectangular block, then the function  $F$  over any such block becomes constant. Thus, the function  $D$  can be simplified, and becomes



$$D(i, j) = \sum_{k=p_1}^{q_1} \sum_{m=r_1}^{s_1} C_1 H_i(i-k) H_j(j-m) + \dots \sum_{k=p_n}^{q_n} \sum_{m=r_n}^{s_n} C_n H_i(i-k) H_j(j-m), \quad (2.25)$$

where  $[p_n, q_n]$  and  $[r_n, s_n]$  represent the bounds of a given rectangular block of intensity  $C_n$ .  $n$  such blocks give the approximation to the filtered scene at point  $(i, j)$ .

If filter functions are considered separately for a given rectangular block, then the expression of the convolution can be rearranged as

$$D(i, j) = C_1 \sum_{k=p_1}^{q_1} H_i(i-k) \sum_{m=r_1}^{s_1} H_j(j-m) + \dots C_n \sum_{k=p_n}^{q_n} H_i(i-k) \sum_{m=r_n}^{s_n} H_j(j-m). \quad (2.26)$$

Therefore, by making the rectangular blocks arbitrarily small, we can obtain an arbitrarily good approximation to  $D$ .

Basically, there are two types of the averaging, uniform and weighted. A uniform averaging operation assigns a constant, such as  $1/n$ , to all the items of the filtering functions in the pixel-intensity expression given above, and each display pixel  $D(i, j)$  is determined by a summation of that expression. For example, by applying a 2X2 uniform window averaging operation to a image, the intensity of a pixel  $(i, j)$  is expressed as

$$D(i, j) = \frac{(C_1 + C_2 + C_3 + C_4)}{4}, \quad (2.27)$$

where  $C_{1-4}$  are the intensities of its surrounding sub-pixels.

The weighted averaging operation uses a distribution function to assign weights to the surrounding pixels according to their distance to the assigned pixel-center. The closer the pixel is to this assigned pixel-center, the higher the weight it is assigned. Thus, the values of each filtering function is approximated to the assigned weights divided by a summation of all the weights. For example, by applying a 3X3 weighted window averaging operation to a image, the intensity of a pixel  $(i, j)$  can be expressed as

$$D(i, j) = \frac{(4C_1 + 2C_2 + 2C_3 + 2C_4 + 2C_5 + C_6 + C_7 + C_8 + C_9)}{16}, \quad (2.28)$$

where  $C_1$  is the center pixel;  $C_{2-5}$  are the pixels next to the center pixel;  $C_{6-9}$  are the pixels next to the center pixel in a diagonal direction. The result of using this weighted averaging operation is illustrated by Crow [3]. It shows that the use of the window averaging technique can produce very realistic images.

Since both the uniform and weighted averaging operations need to process the entire image by a window, a lot of computations are required for their usage. Therefore, the use of this window averaging technique is at a lower operational level than that of the drawing operations in the raster system. In the next section, we present a much faster antialiasing technique for use in drawing operations.

### **2.3.2 The Area-Antialiasing Technique**

The area antialiasing technique uses a concept that the intensity of a pixel is proportional to the area of that pixel intersected by the image. It assumes a pixel has a non-zero area instead of being merely a mathematical point. Due to different assumptions on pixel-shapes and filtering functions, there are several area antialiasing algorithms used to calculate the intensity of each pixel along edges or lines of the image. In this section, we discuss several useful area antialiasing algorithms such as the PW algorithm, the GS algorithm, and several alternative algorithms.

#### **(1). The PW algorithm:**

As stated, Pitteway and Watkinson (PW) introduced an extension to the Bresenham line-drawing algorithm. This algorithm can also be used to incrementally generate the intensity of a selected pixel along a polygon edge for edge-antialiasing.

Basically, according to the mechanism of a CRT, the luminance of each pixel generated by the beam is a Gaussian distribution. Because overlappings exist among all the

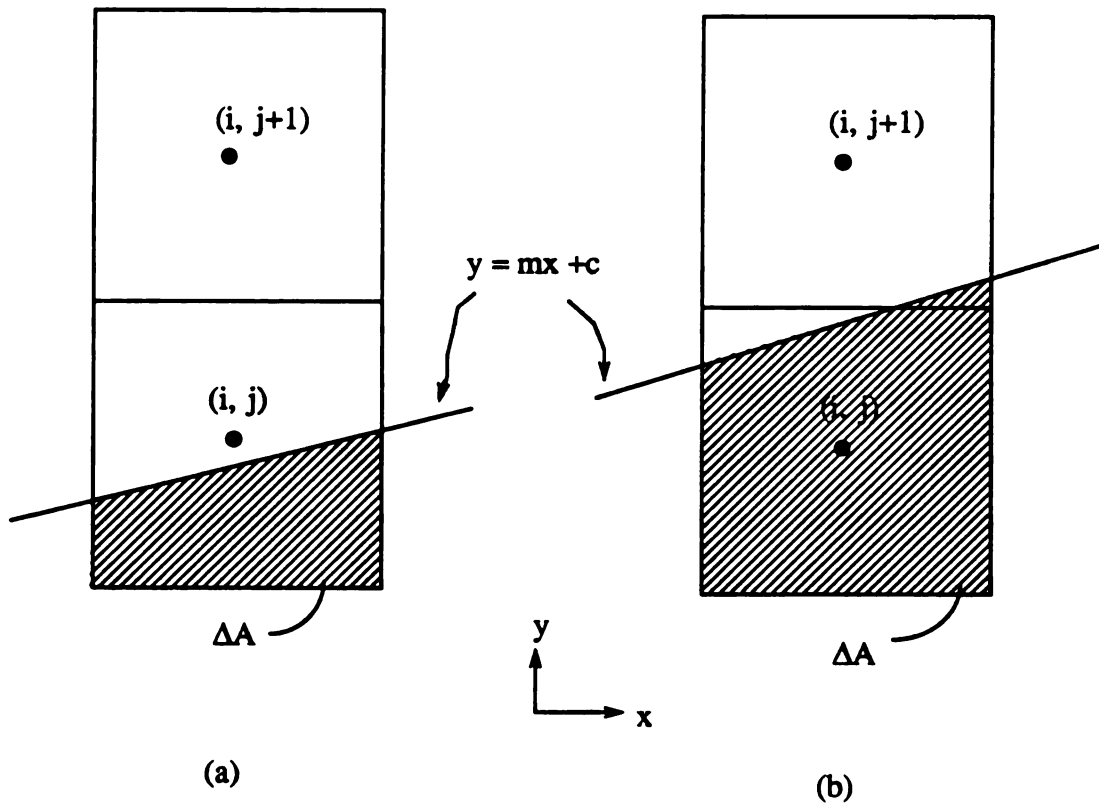
near-by pixels, a column of pixels produce a relatively smooth luminous across a group of raster lines [34]. Based on this result, the PW algorithm uses the assumption of a square pixel and an uniform distribution of the pixel-intensity of each pixel for achieving its operation.

As stated, the decision variable  $d$  is used to select pixels. In addition, the value of this variable is proportional to the area intersected by a polygon edge. In a pixel-selecting calculation,

$$\delta A = mi + c + 1/2 - j,$$

where  $\delta A$  is the intersected area of a pixel  $(i, j)$  below the line  $y = mx + c$ . And, it is the same value as  $d$  generated at that location. The above statement is true only when a line is intersected by the two vertical edges of a pixel. When a line intersection is at the upper boundary of a pixel, two pixels should be shaded, and the summation of their pixel-intensities should be equal to the value of  $d$ . But, according to the algorithm, only one pixel is shaded, and its pixel-intensity is the same as the value of  $d$ . Figure 2-5 illustrates these two pixel-shading cases. Thus, the PW algorithm neglects a portion of area covered by another pixel. The value of  $d$  is always between -1 and 1, and the maximum intensity of a pixel is assigned to 1, i.e., the area of a pixel is equal to 1. Therefore, during the incremental pixel-selecting calculation, the generated value of  $d$  each step is approximated by the intensity of that pixel.

The main advantage of using this algorithm is its simplistic since it uses only one decision variable to generate both the pixel-selection and pixel-intensity signals. We also use this useful concept in developing our drawing with antialiasing algorithm. According to its pixel-intensity calculation, only the polygon edge antialiasing is achieved by using the PW algorithm. Thus, it needs to execute the algorithm three times with different initializations in order to generate a unit-width antialiasing line-drawing. In addition to these inefficiencies, the inaccuracy of the generated pixel-intensity is also a key problem



Note that ● represents the pixel center;

$\Delta A$  represents the area of the pixel  $(i, j)$ .

Figure 2-5. Two pixel-shading cases: (a) one pixel is involved; (b) two pixels are involved.

of using the PW algorithm. In the next chapter, we present a general discussion of the performance of this algorithm and our developed antialiasing algorithm.

(2). The GS algorithm:

Gupta and Sproull (GS) [5] suggested treating the pixel as a cone with this pixel's intensity being proportional to the volume of the cone intersected by the line. This GS algorithm uses the distance between the pixel center and the line center and look-up tables to find the intensity of that pixel. Basically, this algorithm is also a variant of the Bresenham algorithm. It can incrementally update the distance variable  $p$ , similar to the decision variable  $d$  used in the Bresenham algorithm. By inputting this generated  $p$  and a line-slope  $m$  data, we can fetch the corresponding pixel's intensity from look-up tables. Of course, a pre-calculation process is needed to generate the intensity data and store it in these look-up tables. Thus, according to the GS algorithm, it is possible to assign intensities to several shaded pixels each step with a given line-width input. For example, there are three pixels are shaded each step for the line-width = 1, and one pixel is shaded for applying the algorithm to the polygon edge antialiasing.

With the initial value of  $p = 0$ , the the recursive process to generate  $p$  is stated below.

If ( $p \geq s$ ), then the upper pixel is selected as the next center pixel and

$$p = p + (m-1)*c; \quad (2.29)$$

else the lower pixel is selected as the next center pixel and

$$p = p + m*c, \quad (2.30)$$

where  $c = 1/\sqrt{1+m^2}$  is a factor. This factor is calculated from the perpendicular distance between the line-center and the pixel-center instead of the vertical distance used in Bresenham-like algorithms. And,  $s = (0.5-m)*c$  is a constant used to decide which value is selected for incrementing the distance variable.

The main advantage of using the GS algorithm is that it provides great flexibility in applying its antialiasing process to polygon edges, various line-thicknesses and line end-points. However, in order to support this flexibility and good antialiasing results, the size of look-up tables is relatively large. Furthermore, by comparing the structure of both the PW and GS algorithm, the GS algorithm requires more arithmetic operations than that of the PW algorithm. Also, in the next chapter, we present several result comparisons of using the GS, the PW and our algorithm.

(3). Alternative algorithms:

Fujimoto and Iwata [6] present another area antialiasing technique which uses a triangular intensity distribution of a pixel intersected by a line. Without using any look-up table, this algorithm incrementally generates intensities for up to three pixels each step. This algorithm uses an additional input parameter to assign the maximum intensity for each pixel-column. In addition, it has deficiencies of using several non-integral computations and of the limited line-widths option.

Turkowski [7] uses the coordinate transformation method to generate the effective convolution in terms of point-line distance for geometric antialiasings. The algorithm uses a non-incremental operation. This is quite different from those algorithms listed above. Basically, it assigns a Gaussian distribution to the pixel-intensity for convoluting with the line and polygon edge. And, these convolutions need a lot of numerical integrations. The major entries of the convolutions are the point-line distance and point-segment distance. Turkowski investigates several mathematical algorithms in order to provide the fastest calculation method for generating these two distances. As a result, the use of the Coordinate Rotation Digital Computer (CORDIC) rotation algorithm is suggested for calculating the point-line and point-segment distances. This algorithm can provide a very realistic result of the image. However, in order to increase the computing speed of the

algorithm, this algorithm must use the special-purposed hardware because several CORDIC iterations are used while calculating point-line or point-segment distances.

In this chapter, we described several existing drawing and antialiasing algorithms and several architectures for implementing these algorithms in raster systems. In summary, by comparing several different architectures for a high-performance raster system, we proposed the outline of our research focus, i.e., to develop a high-performance drawing engine by using a dedicated hardware approach. Criteria used for evaluating alternative drawing engines are realism, execution speed, operational flexibility, and VLSI implementation suitability.

According to the discussion of traditional line-drawing algorithms, we realize that the incremental and integral operation concepts are useful for developing a fast drawing algorithm. The other line-drawing algorithms are not suitable for developing our drawing engine because they have deficiencies in linking with the antialiasing operation. They also need a lot of extra memory and computational processes in order to provide fast line-drawing operations. In addition, we investigate the scan-curve conversion and piecewise curve-drawing methods which can be used to define the functionality of our drawing engine, i.e., line-drawing and curve-drawing functions.

Because it is fast, the area-antialiasing technique has the potential for generating pixels in real time; thus, it is much more useful than the window-averaging technique when speed is required. According to the discussion of existing area-antialiasing algorithms, we know that the PW and GS algorithms use the incremental pixel-intensity generation concept. This concept is useful for developing our drawing engine to construct simultaneously both the line-drawing and antialiasing operations. To have less algorithmic complexity is our main reason for selecting these two algorithms over that of alternative algorithms. But, trade-offs between the PW algorithm and the GS algorithm are the speed of execution, the flexibility to provide various functions, and the quality of

results, i.e., realism of the display. Our goal is to optimize the graphic engine design by treating these trade-offs as parameters in our investigation of alternative algorithms and architectures.



## **CHAPTER 3**

### **THE CF ANTIALIASED DRAWING ENGINE**

This chapter presents the development of our drawing engine, which is known as the CF engine. This development includes the functional implementation, the structural implementation, and the performance estimation of the CF engine. In the functional implementation, we concentrate more on developing the antialiasing algorithm than on developing the line-drawing algorithm. This is because the algorithmic structure of the line-drawing algorithm is similar to those of the traditional scan-converting line algorithms. We will demonstrate that our antialiasing algorithm is superior to alternative algorithms by a virtue of its operational flexibility and display quality. In the structural implementation of the CF engine, we adopt both parallel and pipeline architectures in our design. In this implementation, we also consider the maximum adaptability of this drawing engine to general curve-drawing operations. Finally, we use a suitable performance estimation technique to evaluate the performance of the CF engine. This technique is also useful for evaluating most VLSI implementations which use the standard-cell or custom designed cell approaches. Both the estimating and testing procedures are described in this chapter in order to verify the correctness of the timing model of basic cells and the timing estimation technique which we employed.

### 3.1 The Algorithmic Development

There are two kinds of interfaces for the CF engine to work with high-level curve-drawing functions. The most general one is its interface with piecewise curve-drawing functions. Because these functions are used to generate coordinates of several consecutive line-segments, inputs of the CF engine are simply specifications of line-segments. Thus, in this case, the CF engine is operated in its normal operational mode, i.e., line-drawing with antialiasing. Another interfacing of the CF engine is with scan-converting curve functions for quickly generating approximated curves. Because these functions provide pixel-selection signals, we can combine this information with our antialiasing results to generate realistic curves. Thus, in this case, the CF engine works as an antialiasing engine. In order to provide those required functions for the CF engine, we concentrate on the functional development of the CF engine in this section. It includes the line-drawing algorithm, the antialiasing algorithm, and the entire drawing algorithm of the CF engine. In addition, we compare our antialiasing results with those of other existing area-antialiasing algorithms.

#### 3.1.1 The Line-Drawing Approach

The line-drawing operation is a fundamental drawing primitive in the CF engine. In order to provide the fastest possible execution speed and the greatest possible extensibility to the antialiasing operation, we use the midpoint pixel-decision method. The midpoint method uses the sign of a vector from the actual line to related midpoints to select pixels of the rasterized line. The PW algorithm uses this midpoint method; thus, we use it as a basic building block and adopt it to fit our overall design criteria. In order to increase the execution speed of the line-drawing algorithm, we apply a replacement process to the PW algorithm. Thus, several major parameters and expressions of the algorithm are

changed and become more effective than the original ones. With those adjustments, the initial condition of  $d_i$  becomes

$$\begin{aligned} d_1 &= 0.5 - (1-m) \\ &= m - 0.5, \end{aligned} \tag{3.1}$$

and the reference  $a$  is now set to 0. Under the  $0 \leq m \leq 1$  case, the consecutive decision variable is determined by using the following expressions. If  $d_i < 0$ , then the lower pixel is selected, and

$$d_{i+1} = d_i + m.$$

Otherwise, if  $d_i \geq 0$ , then the upper pixel is selected, and

$$d_{i+1} = d_i + (1-m). \tag{3.2}$$

Note that we use the line-slope  $m$  in our line-drawing operation. The main reason for this is that  $m$  is an important factor for use in developing our antialiasing operation. By comparing the algorithmic structure of the two algorithms, we see that the use of this modified algorithm is faster than the original PW algorithm. This line-drawing operation is used as a decision element in the CF engine. Basically, the CF engine is constructed by both the line-drawing and antialiasing operations. Thus, we need a suitable antialiasing approach to link with this line-drawing operation in order to provide antialiased drawing features. In the next section, we present our antialiasing approach, which is known as the CFO antialiasing approach.

### 3.1.2 The CFO Antialiasing Approach

According to the conclusion of Chapter 2, the area-antialiasing appears to be the most suitable technique for developing our antialiasing algorithm. This is because the use of this technique can provide a fast execution speed and much realistic images, and it is

suitable for VLSI implementation. Thus, we must define two important factors in order to adopt this technique, i.e., the distribution of the pixel-intensity and the shape of a pixel on the raster. In the CFO antialiasing approach, the distribution of the pixel-intensity is assumed to be unweighted and uniform, and, a pixel is treated as a 1x1 square area on the raster. In addition, we need to construct the incremental process and exact pixel-intensity model for developing the CFO antialiasing algorithm. According to the given line-width  $Wd$  and line-slope  $m$  inputs, we first define several parameters for our CFO antialiasing algorithm as follows:

- (1)  $A_t$  represents the whole intersected area of a line with each pixel-column, where

$$A_t = Wd * \sqrt{1+m^2}. \quad (3.3)$$

Since the area of each pixel-column intersected by a line depends only on the given  $Wd$  and  $m$ ,  $A_t$  is a constant for each pixel-column of the given line-segment. It is also a vertical distance between two line-edges of the line.

- (2)  $W_x$  represents a factor to calculate the amount of pixels to be shaded in each pixel-column, where

$$W_x = \left\lfloor \frac{A_t}{2} \right\rfloor + 1. \quad (3.4)$$

By using geometric calculations, we can determine that number of shaded pixels is equal to  $2*W_x + 1$ . Also,  $W_x$  is a constant in the antialiased line-drawing operation. In addition, we can determine coordinates of the shaded pixels in each pixel-column by using this  $W_x$  and the pixel-decision signal.

- (3) There exist two error parameters, i.e.,  $\overline{e1}$  and  $\overline{e2}$ .  $\overline{e1}$  represents a vector from the upper pixel-midpoint to the upper line-edge in each pixel-column. If the direction of this vector is upward, then its value is positive; otherwise, its value is negative.  $\overline{e2}$  represents another vector from the lower pixel-midpoint to the lower line-edge in each pixel-column. If the direction of this vector is downward, then its value is positive; otherwise,

its value is negative. Thus,  $e_1$  and  $e_2$  are variables, and their values can be updated each step. By using geometric calculations, we express the initial values of  $e_1$  and  $e_2$  as

$$e_1 = e_2 = \frac{At}{2} - \left\lfloor \frac{At}{2} \right\rfloor - \frac{1}{2}. \quad (3.5)$$

Figure 3-1 illustrates the geometrical relationships among  $At$ ,  $W_x$ ,  $\overline{e_1}$  and  $\overline{e_2}$  corresponding to a line with the starting central pixel  $(i, j)$ , line-slope  $m$  and line-width  $W_d$ . Note that all of the definitions listed above are based on the line-drawing under the  $0 \leq m \leq 1$  case. Their extensions to other cases are the same as we described in Section 2.2.1 with the related values of  $m$ , i.e.,

$$m = \begin{cases} m & \text{if } 0 \leq m \leq 1; \\ (1/m) & \text{if } m > 1; \\ -m & \text{if } -1 \leq m < 0; \\ -(1/m) & \text{if } m < -1 \end{cases} \quad (3.6)$$

It is obvious that the midpoint method is used for our calculating  $e_1$  and  $e_2$ . So, the calculation of consecutive values of  $e_1$  and  $e_2$  can be executed incrementally:

If the upper pixel is selected, then

$$\begin{aligned} e_{1_{i+1}} &= e_{1_i} - (1-m); \\ e_{2_{i+1}} &= e_{2_i} + (1-m). \end{aligned} \quad (3.7)$$

Otherwise, if the lower pixel is selected, then

$$\begin{aligned} e_{1_{i+1}} &= e_{1_i} + m; \\ e_{2_{i+1}} &= e_{2_i} - m. \end{aligned} \quad (3.8)$$

The use of the uniform area antialiasing concept provides a direct and effective intensity calculation. By using this concept, we form an exact intensity solution expressed in terms of the error parameter  $e_i$ , where  $i = 1$  or  $2$ . Note that this antialiasing operation is also considered under the  $0 \leq m \leq 1$  case. Thus, there exists three possible cases for a line-edge intersected by a column of pixels, i.e.,  $e_i > m/2$ ,  $-m/2 \leq e_i \leq m/2$  and  $e_i < -m/2$ . The value two is the maximum number of pixels to be intersected by a line-

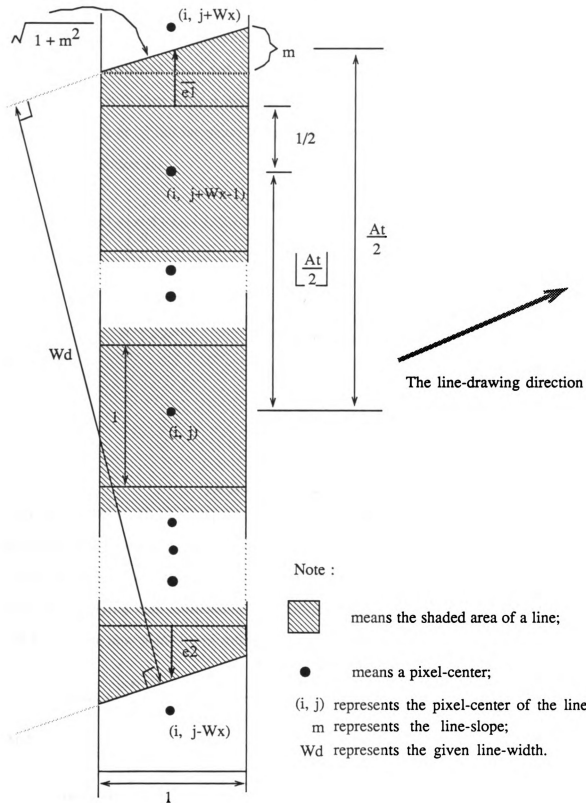


Figure 3-1. Illustration of  $At$ ,  $Wx$ ,  $\overline{e1}$  and  $\overline{e2}$  corresponding to the starting column of a line.

edge, and *Area 1* and *Area 2* represent the shaded area for each of these two pixels. Figure 3-2 illustrates results of *Area 1* and *Area 2* with three cases of  $e_i$ . Note, if any shaded-area result is equal to 0, then there is no pixel-intensity of that pixel. And, if it is equal to 1, then the pixel is at its full intensity. By using the geometric calculations, we express the shaded-area solution as:

$$Area\ 1 = \begin{cases} e_i & \text{if } e_i \geq m/2; \\ m/8 + e_i/2 + e_i^2/2m & \text{if } -m/2 < e_i < m/2; \\ 0 & \text{if } e_i \leq -m/2. \end{cases}$$

And

$$Area\ 2 = \begin{cases} 1 & \text{if } e_i \geq m/2; \\ 1 - m/8 + e_i/2 - e_i^2/2m & \text{if } -m/2 < e_i < m/2; \\ 1 + e_i & \text{if } e_i \leq -m/2. \end{cases} \quad (3.9)$$

Note that

$$-1 \leq e_i < 1,$$

$$0 \leq m \leq 1,$$

and

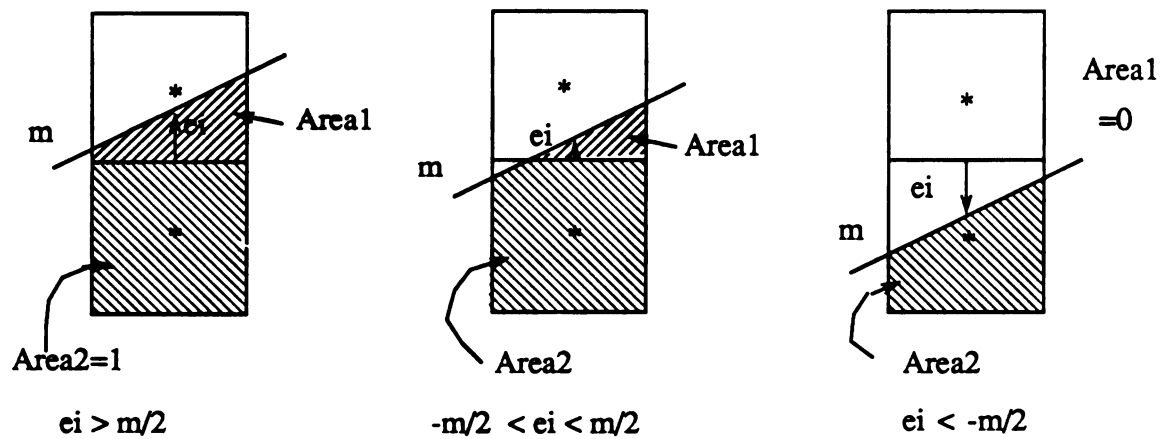
$$e_i^2/2m \ll 1, \text{ if } -m/4 < e_i < m/4.$$

And,  $e_i$  can be  $e_1$  or  $e_2$ . Thus, we can eliminate the complicated terms in the exact shaded area solutions in order to save its computational time. Also, to reduce the error generated from this elimination, a shifting process is applied to these shaded-area expressions. By applying both eliminating and shifting operations to the original solution, we can express the new shaded-area solution as:

$$Area\ 1 = \begin{cases} e_i & \text{if } e_i \geq m/4; \\ m/8 + e_i/2 & \text{if } -m/4 < e_i < m/4; \\ 0 & \text{if } e_i \leq -m/4. \end{cases}$$

And,

$$Area\ 2 = \begin{cases} 1 & \text{if } e_i \geq m/4; \\ 1 - m/8 + e_i/2 & \text{if } -m/4 < e_i < m/4; \\ 1 + e_i & \text{if } e_i \leq -m/4. \end{cases} \quad (3.10)$$



Note that each "\*" represents the pixel center;  
 $m$  represents the slope of the line;  
 each unit square in the columns represents the area of each pixel.

Figure 3-2. Shaded-area results for three cases of  $ei$ .



The maximum error of this solution is determined by calculating the difference between the approximated result and the exact result. We find this error is only  $1/32$ , i.e., 3.125%.

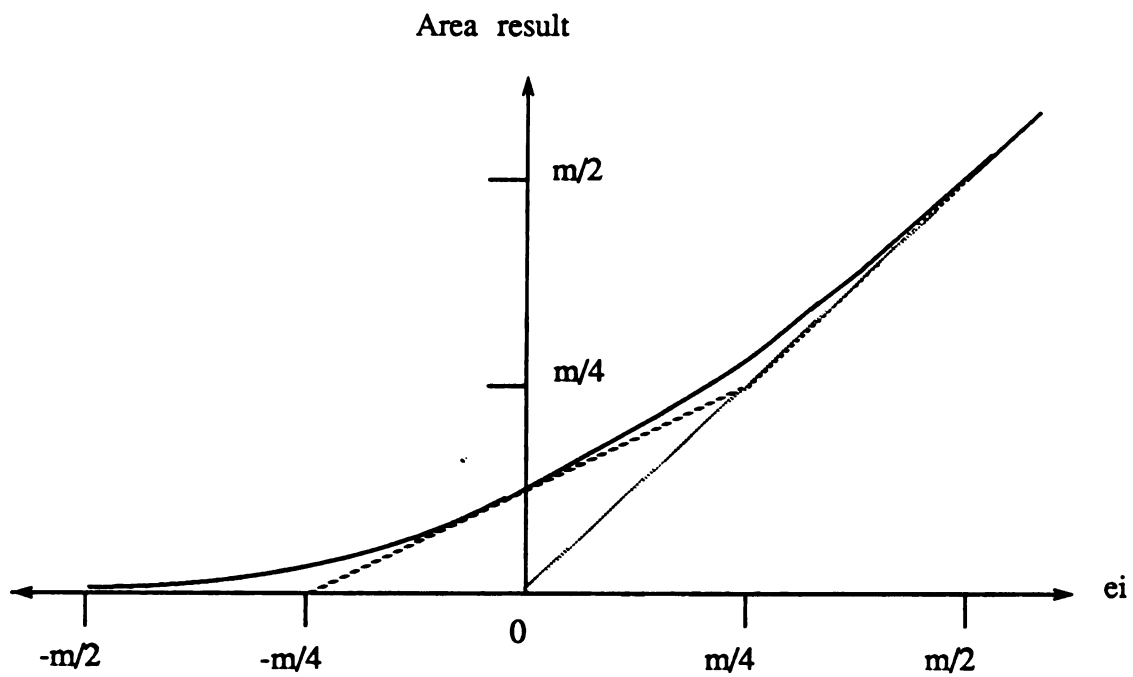
Note that inputs of this CFO algorithm are only the line-slope  $m$  and the line-width  $wd$ . These two parameters are useful for calculating the initial value of all the parameters. In the CFO algorithm, the values of all the parameters are restricted to the range  $[-1, 1]$ , and each generated pixel-intensity value is restricted to the range  $[0, 1]$ . These features are useful in the structural implementation of the CF engine. From the structure of the CFO algorithm, we know this algorithm has both incremental operation and multiple shaded-pixels generation features. Also, the CFO algorithm allows any positive real number used as the line-width input. Thus, the use of the CFO algorithm can provide various line-width options to users.

A comparison of various line antialiasing results is provided by Pitteway [35]. It contains the related pixel-intensity results of the idea calculation, the GS [5] and the PW [4] algorithms for a line with the line-width  $wd = 1$  and drawn from  $(0,0)$  to  $(7,5)$ . With the same condition, we use our CFO algorithm to calculate the related pixel-intensities. These antialiasing results of the ideal, CFO, PW and GS thin-cone algorithms are illustrated in Table 3-1. It shows that pixel-intensity results of our algorithm is more accurate than those of alternative area-antialiasing algorithms. In addition, because we use a constant intensity parameter  $A_t$  in our algorithm, results of using our antialiased line-drawing operation do not have non-constant line thickness problems, which occurs in the GS algorithm. Furthermore, in order to compare the maximum error of using our algorithm with that of the PW algorithm, Figure 3-3 illustrates their shaded area results corresponding to an error parameter  $\epsilon_i$ . It shows that the maximum area error of PW implementation is  $m/8$ ; however, the maximum area of CFO implementation is only  $m/32$ . Thus, the use of the CFO algorithm generates much more accurate pixel-intensity results for antialiasing operations.

Location	GS thin-cone result	PW result	CFO result	Ideal result
U1	0.126	0.114	0.147	0.156
B1	0.958	1.000	0.936	0.918
L1	0.126	0.114	0.147	0.156
U2	0.010	0.000	0.004	0.024
B2	0.833	0.829	0.825	0.805
L2	0.386	0.400	0.400	0.400
U3	0.550	0.543	0.543	0.543
B3	0.705	0.686	0.686	0.685
L3	0.000	0.000	0.000	0.001
U4	0.240	0.257	0.257	0.264
B4	0.924	0.972	0.897	0.889
L4	0.049	0.000	0.075	0.076
U5	0.049	0.000	0.075	0.076
B5	0.924	0.972	0.897	0.889
L5	0.240	0.257	0.257	0.264
U6	0.000	0.000	0.000	0.001
B6	0.705	0.686	0.686	0.685
L6	0.550	0.543	0.543	0.543
U7	0.386	0.400	0.400	0.400
B7	0.833	0.829	0.825	0.805
L7	0.010	0.000	0.004	0.024
U8	0.126	0.114	0.147	0.156
B8	0.958	1.000	0.936	0.918
L8	0.156	0.114	0.147	0.156

Note that the location of the pixel, GS and PW indicated here are the same as the definitions in [35].

Table 3-1. Area result comparison of the ideal, CFO, PW and GS thin-cone algorithms from (0,0) to (7,5) with  $wd = 1$  pixel.



Note:

- is the exact result;
- ..... is PW's result;
- is CFO's result.

Figure 3-3. Comparison graph of shaded area results.

### 3.1.3 Implementation of the CF algorithm

During the development of our antialiasing algorithm, we recognized that only the pixel-decision signal was missing to fulfill the entire drawing operation. Thus, both the line-drawing and the antialiasing algorithms should be link together in order to construct the CF algorithm. In the CFO algorithm, the operation of the error parameter  $e_i$  is similar to that of the decision variable  $d$  in the line-drawing operation. Thus, we can simply use the error parameter  $e_i$  to decide which pixel is to be selected. The procedure is described as follows: IF  $e_i \geq 0$ , then select the upper pixel; else, select the lower pixel. The incremental calculation of  $e_i$  is the same as we described in the previous section. This concept is useful in the line-drawings with non-integral endpoints because it is not necessary for  $e_i$  to equal  $1/2$  at the starting and ending points of the line-drawing operation. However, there are two error parameters in our antialiasing algorithm, and both can be used to generate different pixel selection signals. For example, by using  $e_1$ , we select the upper pixel; whereas, we may select the lower or upper pixels by using  $1-e_2$ . Both results are correct because they use two different pixel-midpoints as their bases. Therefore, we still use a line-drawing algorithm described in Section 3.1.1 to generate our pixel-decision signal and to construct the entire drawing algorithm. Figure 3-4 illustrates the basic configuration of the CF algorithm, where  $a_1$  and  $a_2$  represent pixel-intensities of two pixels at the upper line-edge.  $a_3$  and  $a_4$  represent pixel-intensities of two pixels at the lower line-edge. And, *Assign\_area* represents a subroutine used to generate two pixel-intensities according to the value of the error parameter input.

In order to verify the correctness of the CF algorithm, Chen and Fisher [36] illustrate image results by applying the algorithm to several examples. Basically, we use a concept similar to Thacker and Smith [37] to verify the algorithm. This testing concept applies several possible patterns of an image as input databases to the algorithms and then to evaluate their performance by inspecting their visual results.

```

LineAnt( x1, y1, x2, y2, Wd)

{ m = (y2 - y1)/( x2 - x1);
  w = 1 - m;
  At = Wd * SQRT( 1 + m 2);
  Wx = ⌊At/2⌋ + 1;
  d = 1/2 - w;
  e1 = At/2 - ⌊At/2⌋ - 1/2;
  e2 = e1;
  x = x1;
  for ( i = 1; i < (x2 - x1) + 2; i++)
  { Assign_area( e1, a1, a2);
    Assign_area( e2, a3, a4);
    if ( Wx > 1)
    { for (j = y1 - Wx + 2; j < y1 + Wx - 1; j++)
      display( x, j, 1);
      display( x, y1 + Wx - 1, a2);
      display( x, y1 - Wx + 1, a4);
    }
    else display( x, y1, At - a1 - a3);
    display( x, y1 + Wx, a1);
    display( x, y1 - Wx, a3);
    if ( d < 0)
    { s = 0; d = d + m;
      e1 = e1 + m; e2 = e2 - m; }
    else { s = 1; d = d - w;
      e1 = e1 - w; e2 = e2 + w; }
    x = x + 1;
    y1 = y1 + s;
  }
}

Assign_area( ei, ai1, ai2)
{ if ( ei >= m/4)
  { ai1 = ei; ai2 = 1; }
  else if ( ei >= (-m/4))
  { ai1 = m/8 + ei/2; ai2 = 1 - m/8 + ei/2; }
  else { ai1 = 0; ai2 = 1 + ei; }
}

```

Figure 3-4. CF algorithm with various line thickness.

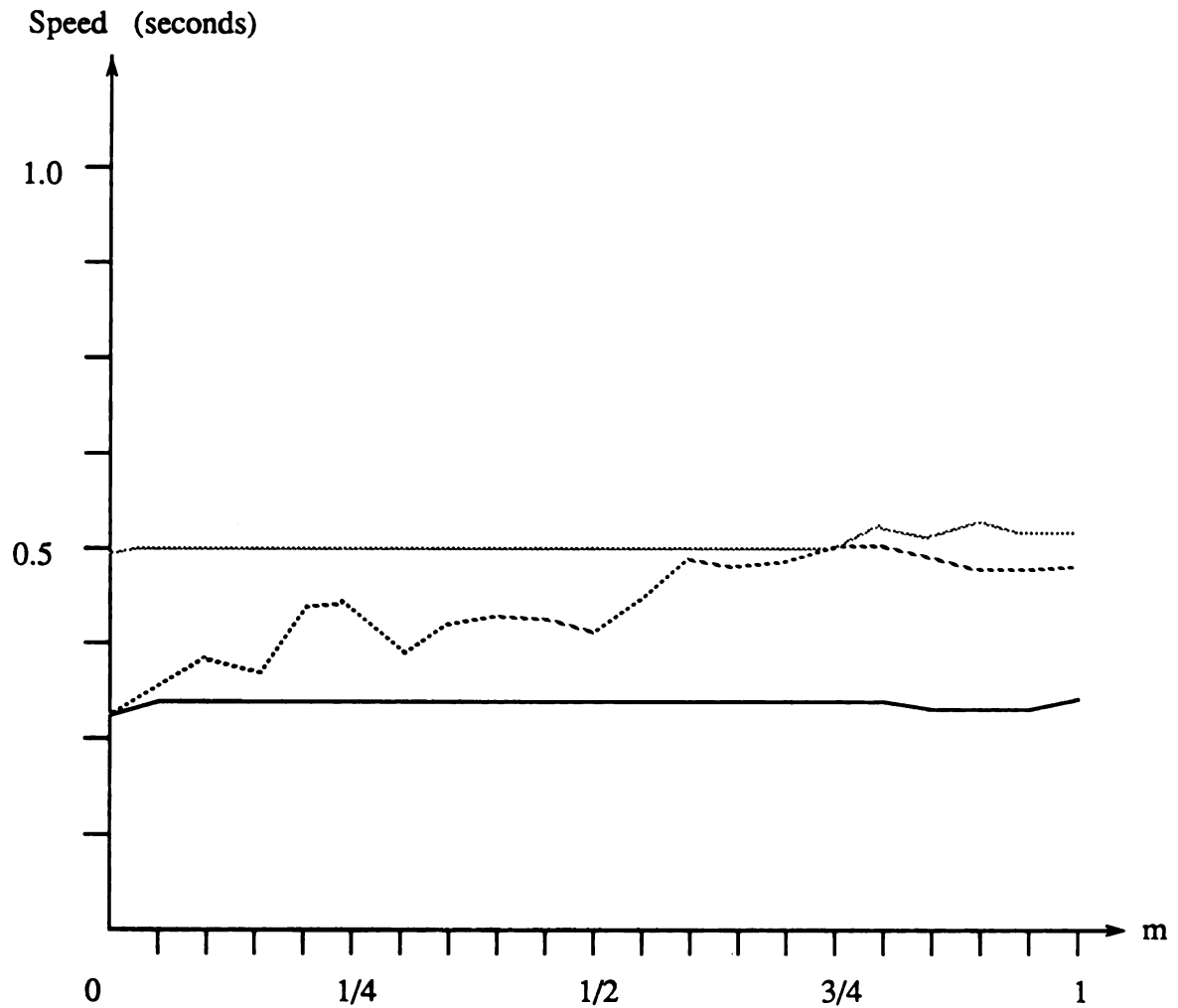
In addition to evaluating the correctness and realism of the CF algorithm, we compare its execution speed with that of alternative algorithms. The GS algorithm not only needs to calculate a distance variable each cycle but also needs a large amount of memory to store all the related pixel-intensities. The algorithmic complexity of the GS algorithm is similar to that of ours, but, the execution time of the GS algorithm requires additional time for fetching the related pixel-intensities from a memory look-up table. Thus, the execution speed of the CF algorithm is faster than that of the GS algorithm.

The PW algorithm is the simplest antialiasing algorithm, and it is useful for edge antialiasing. This algorithm must be operated three times and needs different initializations while being used for the antialiased line-drawing. We also modified the original CF algorithm in order to increase its execution speed. But, we accomplished this by sacrificing some realism. The pixel-intensity resulting from this modified algorithm is the same as that of the PW algorithm. This modified algorithm has the same structure as that shown in Figure 3-4, only the pixel-intensity assignment subroutine is changed to

$$\begin{aligned}
 & \text{Assign\_area}(e_i, a_1, a_2) \\
 & \{ \text{if } (e_i \geq 0) \\
 & \quad \text{if } (e_i \geq 0) \\
 & \quad \quad \{ a_1 = e_i; a_2 = 1; \} \\
 & \quad \text{else} \\
 & \quad \quad \{ a_1 = 0; a_2 = 1 + e_i; \} \\
 & \}
 \end{aligned} \tag{3.11}$$

We used several representative drawing examples to evaluate the execution speeds of the PW algorithm, our original algorithm, and the modified CF algorithm. A comparison of these results is provided in Figure 3-15. From this figure, we see that our original CF algorithm is faster than the PW algorithm, and the modified algorithm is the fastest one among all three algorithms.

If the CF algorithm is used in conjunction with piecewise curve operations, then the algorithm is operated in its normal mode. So, it is used to draw the given line-segments with an antialiasing feature. In conjunction with scan-curve conversion operations, the



Note:

- represents PW's result;
- ..... represents CFO's result;
- - - represents modified CFO's result;

all timing results are measured in CPU second on SUN-3;

x2 - x1 is equal to 800.

Figure 3-5. Comparison of the execution speed of the PW, CF and modified CF algorithms.

CF algorithm is operated in its antialiasing mode. It is used to generate pixel-intensities for antialiased curve-drawings. The information used here are the curvature of a curve and the pixel decision signal. For this case, the line-slope  $m$  and the constant intensity  $A_t$  in the algorithm should be replaced by two incremental variables. Figure 3-6 lists a typical antialiased circle-drawing algorithm with line-width  $W_d = 1$ . It combines both the Bresenham circle-drawing algorithm and our CFO antialiasing algorithm. Furthermore, by replacing the incremental line-slope with the exact one in each drawing cycle, we can derive an improved CF algorithm. This algorithm can be used to generate much more realistic circles. However, we need more complicated computations in the algorithm than that of the nominal algorithm. Chen and Fisher [36] demonstrated the results of using the aliased algorithm, the nominal algorithm and the improved CF circle-drawing algorithms.

By combining both the line-drawing and CFO antialiasing algorithms, we constructed the CF algorithm. From the comparisons and demonstrations listed above, the CF algorithm is the greatest in its execution speed, operational flexibility and realism among all the algorithms.

### 3.2 The Architectural Implementation

According to the data flow of the CF algorithm, we can easily identify three main parallel processes. These are the processes of the pixel decision variable  $d_i$ , and two error parameters  $e_1$  and  $e_2$ . The process of variable  $d_i$  is used to generate a pixel decision signal for determining the minor axis address and increments in both  $e_1$  and  $e_2$  operations. The processes of  $e_1$  and  $e_2$  are used to generate intensities of the shaded pixels around the upper and lower line-edges of a line. Thus, for each clock cycle, two identical pixel intensity generators with inputs  $e_1$  and  $e_2$  produce four pixel-intensities.



*CircleAnt*( *x1*, *y1*, *R*)

```

{
    y1 = y1 + R;
    xx = 0;
    yy = R;
    m = 0;
    w = 1;
    mstep = 1.414 / R;
    At = 1;
    Atstep = 0.414 * 1.414 / R;
    d = 3 - 2 * R;
    e1 = 0;
    e2 = e1;
    while ( xx <= yy)
    {
        Assign_area( e1, a1, a2);
        Assign_area( e2, a3, a4);
        display( x1, y1, At - a1 - a3);
        display( x1, y1 - 1, a1);
        display( x1, y1 + 1, a3);
        m = m + mstep;
        w = w - mstep;
        At = At + Atstep;
        e1 = At/2 - 1/2;
        e2 = e1;
        if( d < 0 )
        {
            s = 0;      d = d + 4*xx + 6;
            e1 = e1 + m;      e2 = e2 - m;
        }
        else
        {
            s = 1;      d = d + 4*( xx - yy) + 10;
            e1 = e1 - w;      e2 = e2 + w;
        }
        x1 = x1 + 1;      xx = xx + 1;
        y1 = y1 - s;      yy = yy - s;
    }
}

```

Figure 3-6. CF circle algorithm with  $W_d = 1$  pixel in octant  $y \geq x \geq 0$ .

Since the values of all the parameters in the CF algorithm are restricted to the range  $[-1, 1]$ , we use a fixed-point representation for all the parameters in our design. Also, we utilize two's complement arithmetic to calculate the consecutive values of these parameters. In our algorithm,  $m$  and  $1-m$  are the two increments used in the processes of  $d_i$ ,  $e_1$ . Because  $m$  is equal to a two's complement form of  $1-m$ , we use only  $m$  as our fixed increment. Also, in the process of  $e_2$ , we use  $\bar{m}$  as our fixed increment because the one's complement form of  $\bar{m}$  is equal to the two's complement form of  $1-m$ . For example, if  $m = .0100_2$ , i.e.,  $.25_{10}$ , and  $d_i = .1000_2$ , i.e.,  $.5_{10}$ , then the next decision variable is expressed as

$$\begin{aligned} d_{i+1} &= d_i - (1-m) \\ &= .5 - (1 - .25) \\ &= -.25 \end{aligned} \quad (3.12)$$

According to our drawing algorithm, the next decision variable is equal to

$$\begin{aligned} d_{i+1} &= .1000_2 + .0100_2 \\ &= .1100_2 \end{aligned} \quad (3.13)$$

Because no carry-out is generated in Equation (3.13), the final value of  $d_{i+1}$  should be treated as a negative value. According to two's complement arithmetic, Equation (3.13) can be interpreted as

$$d_{i+1} = -.0011_2 + -.0001_2 = -.0100_2 = -.25_{10} \quad (3.14)$$

Therefore, we can use integral computations to implement the CF engine; but, the precision of results depends on the number of bits used. Figure 3-7 illustrates the block diagrams of incremental processes of  $d_i$ ,  $e_1$  and  $e_2$ . These three processes must be synchronized by a same clock signal (CK). Each clock cycle, they generate new pixel selection and intensity calculation signals. The pixel selection signal is used to determine the coordinates of the shaded pixels and the sign-bits of  $e_1$  and  $e_2$ . Both error parameters are used for calculating the shaded pixel-intensities. By using either  $e_1$  or  $e_2$  as its input, Fig-

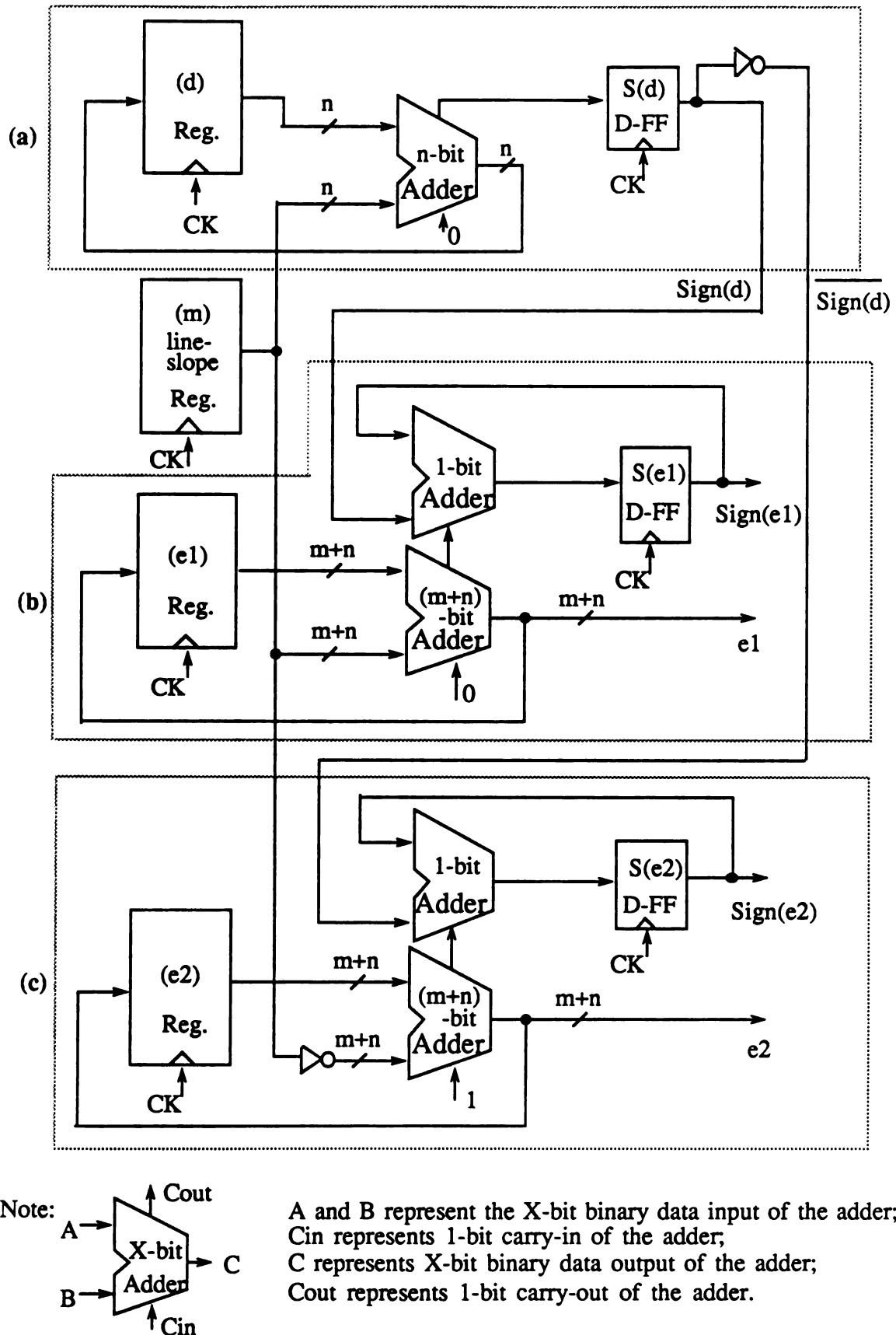
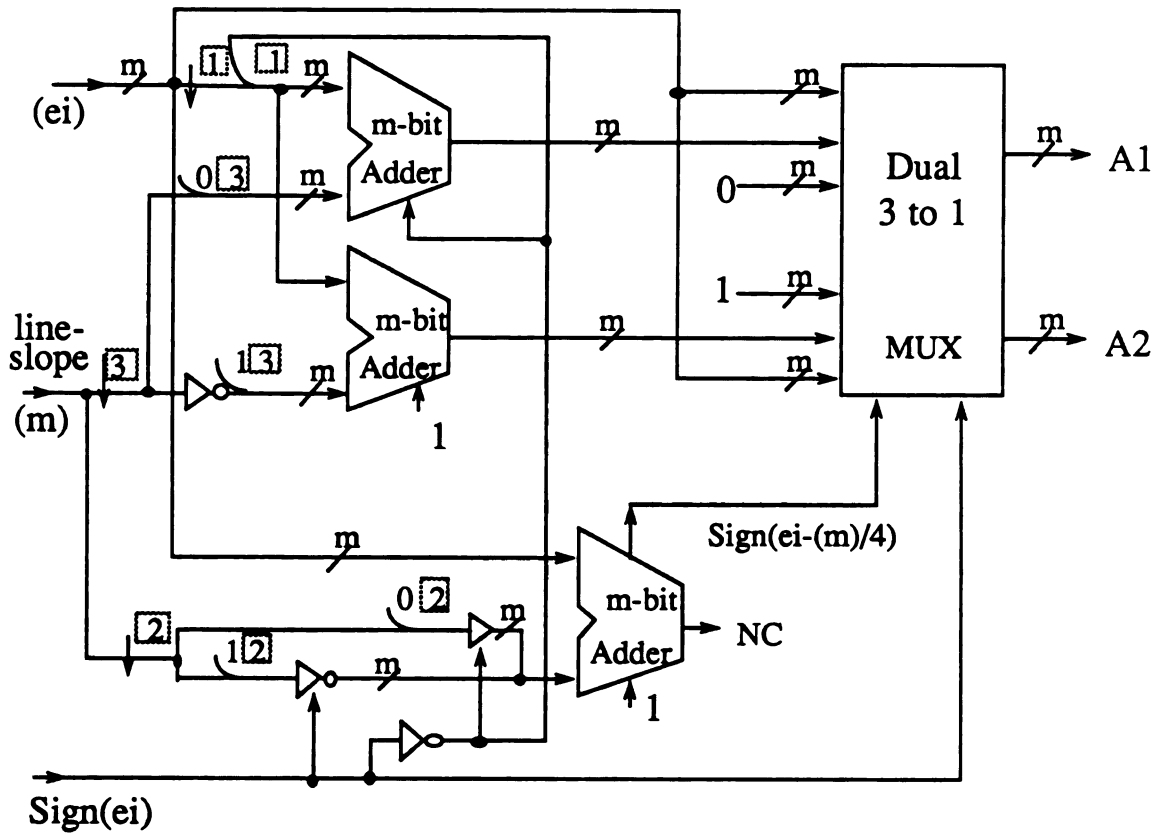


Figure 3-7. Block diagram of three incremental processes: (a) the decision variable  $d$ , (b) the upper error parameter  $e1$ , and (c) the error parameter  $e2$ .



Note:

$\xrightarrow{\boxed{n}}$  represents a n-bit shift operation of the data-bus.

$\xrightarrow{X\boxed{n}}$  represents a n-bit combination operation of the data-bus;  
X represents a data for n most significant bits.

Figure 3-8. Block diagram of the pixel-intensity generator.

ure 3-8 illustrates the block diagram of the intensity generator. Note that outputs of the multiplexer are  $A_1$  and  $A_2$ , where

$$A_1 = \begin{cases} ei & \text{if } (\text{Sign}(ei) \text{ AND } \text{Sign}(ei - m/4)) = 1; \\ ei/2 + m/8 & \text{if } (\text{Sign}(ei) \text{ XOR } \text{Sign}(ei - m/4)) = 1; \\ 0 & \text{if } (\text{Sign}(ei) \text{ NOR } \text{Sign}(ei - m/4)) = 1. \end{cases}$$

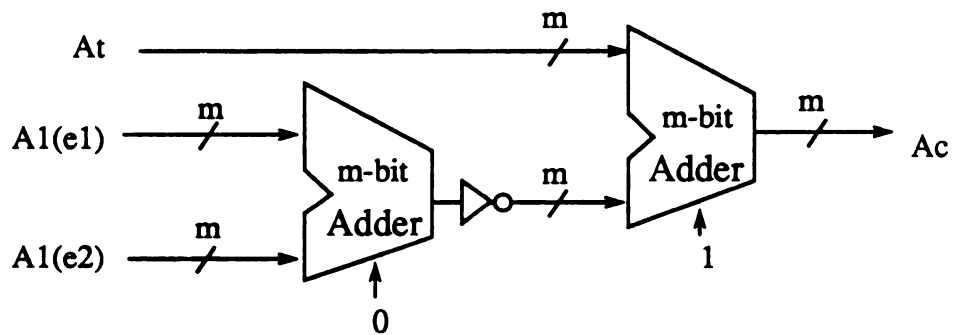
And,

$$A_2 = \begin{cases} 1 & \text{if } (\text{Sign}(ei) \text{ AND } \text{Sign}(ei - m/4)) = 1; \\ ei/2 - m/8 & \text{if } (\text{Sign}(ei) \text{ XOR } \text{Sign}(ei - m/4)) = 1; \\ ei & \text{if } (\text{Sign}(ei) \text{ NOR } \text{Sign}(ei - m/4)) = 1. \end{cases} \quad (3.15)$$

Figure 3-9 illustrates the part of intensity generator used to determine the intensity of the central-pixel each cycle under the  $W_x \leq 1$  case.

In Figures 3-7 to 3-9, we use three different sizes of adders, i.e., the  $n$ -bit adder, the  $(m+n)$ -bit adder and the  $m$ -bit adder. This is because we want to minimize rounding errors in the CF engine. For example, if a screen-size is less than  $1024 \times 1024$  pixel, then we can assign the minimal number 10 to  $n$ . Thus, the 10-bit adders are used in the decision variable operation. In addition, if the required number of intensity level is less than 16, we can assign the minimal number 4 to  $m$ . Thus, the 14-bit adders are used in error parameter operations, and the 4-bit adders are used in pixel-intensity generators. The critical path of this circuit is in the process of  $e_2$  because it uses the largest size of adders. With the same calculation in Table 3-1, Table 3-2 illustrates results of using the CF engine with  $m+n = 8$  and  $m = 5$ . It shows that main operations of the CF engine are executed in parallel and pipeline fashion. Multiple pixel-intensities can be generated in each drawing cycle. Because we only use 5-bit operations, the intensity results in Table 3-2 are less accurate than those in Table 3-1.

Figures 3-7 to 3-9 illustrate the main building blocks of the CF engine. Several supplementary designs must be implemented in order to produce a powerful and complete drawing engine. The extension to the curve-antialiasing operations is achieved by chang-



**Note:**

A1(e1) represents the result of A1 from executing the intensity generator with e1 input;  
 A1(e2) represents the result of A1 from executing the intensity generator with e2 input;  
 Ac represents the intensity of the center pixel.

Figure 3-9. Block diagram of the Ac generator under  $W_x \leq 1$  case.

	clock #0	clock #1	clock #2	clock #3	clock #4
$d_i$	X(10000000) =0.5	1(00110110) =0.211	0(11101100) =-0.078	1(10100010) =0.633	1(01011000) =0.344
e1	X	1(00011101) =0.113	0(11010011) =-0.176	1(10001001) =0.535	1(00111111) =0.246
e2	X	same as e1	1(01100111) =0.402	0(10110001) =0.309	0(11111011) =-0.020
A1(e1)	X	X	(00100) =0.125	(00000) =0	(10001) =0.531
A1(e2)	X	X	same as A1(e1)	(01100) =0.375	(00000000) =0
Ac	X	X	(11111) =0.969	(11011) =0.844	(10110) =0.688

	clock #5	clock #6	clock #7	clock #8	clock #9
$d_i$	1(00001110) =0.055	0(11000100) =-0.234	1(01111010) =0.477	X	X
e1	0(11110101) =-0.043	0(10101011) =-0.332	1(01100001) =0.379	1(00010111) =0.090	X
e2	1(01000101) =0.270	1(10001111) =0.559	0(11011001) =-0.152	1(00100011) =0.137	X
A1(e1)	(00111) =0.219	(00010) =0.063	(00000) =0	(01100) =0.375	(00100) =0.125
A1(e2)	(00010) =0.063	(01000) =0.250	(10001) =0.531	(00000) =0	(00100) =0.125
Ac	(11110) =0.938	(11101) =0.906	(10110) =0.688	(11011) =0.844	(11111) =0.969

Note that all the output data of  $d_i$ , e1 and e2 are represented in a 8-bit binary format; all the output data of A1(e1), A1(e2) and Ac are represented in a 5-bit binary format. X means donot care data.

Table 3-2. Simulation results of using the CF engine to draw a line from (0,0) to (7,5) with  $W_d = 1$ .

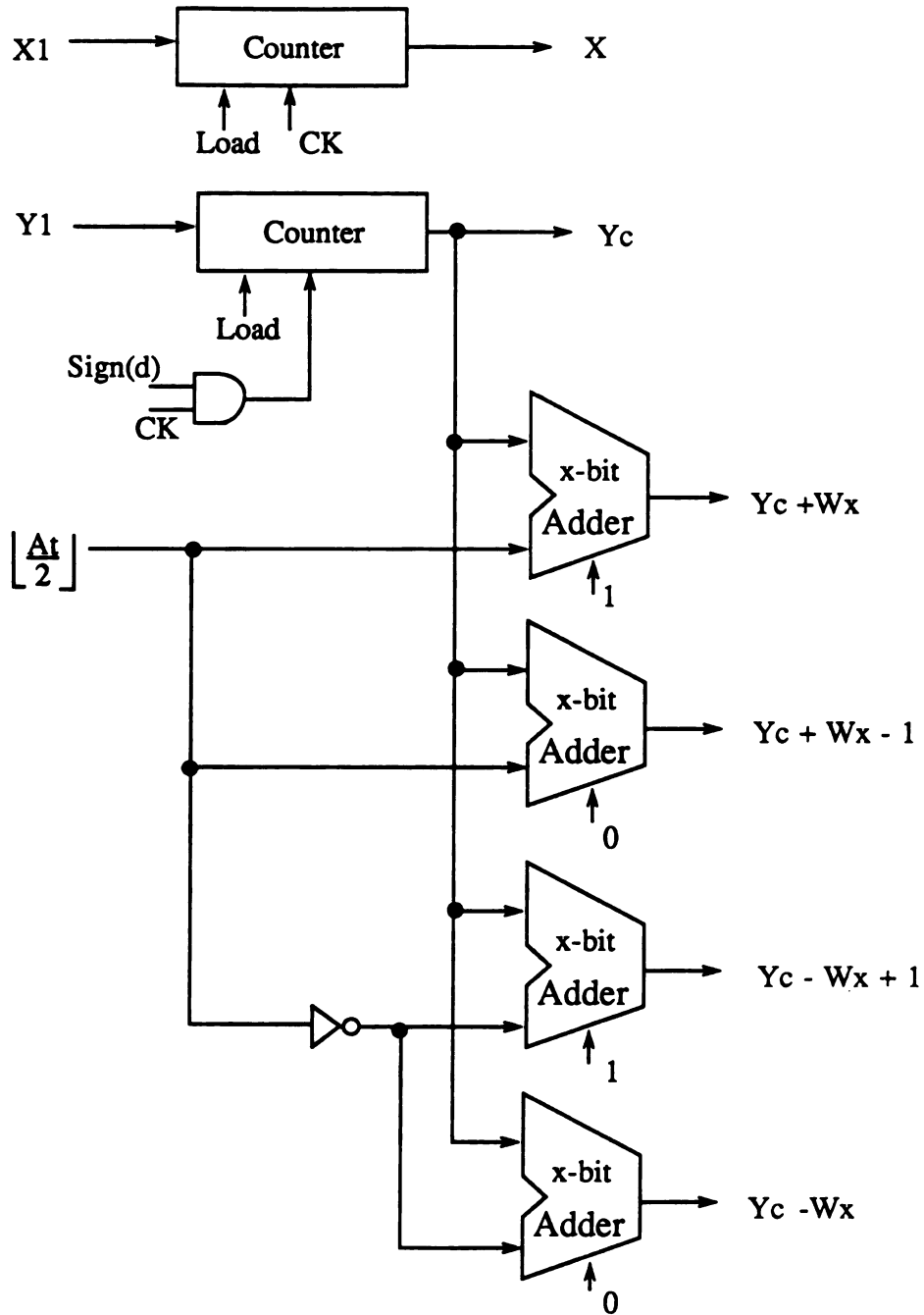
ing the fixed line-slope structure to an incremental one. Thus, users can select 0 or a constant as the increment data for the line-slope calculation each step. The control logic design of the CF engine is similar to that of the line-drawing engine. We will describe this in the next section. The design of the address generator depends on the structure of the frame buffers. We can use counters and adders to construct the basic parallel structure for this address generator. Figure 3-10 shows a basic block diagram of this address generator for generating addresses of the shaded pixels. The generation of the full-intensity pixels can be achieved by using a filling process in the high drawing level or a dedicated filling hardware which is synchronized by the CF engine.

In this section, we presented a structural implementation of the CF engine. Although truncation is used in our approximated integral operations, the results of using the CF engine are still very accurate. Because pixel-intensities are represented in binary, users can flexibly select a different number of intensity levels depending upon the application. If the user selects the  $n$  most significant bits of intensity outputs of the CF engine, then  $2^n$  is the number of distinct intensity levels available. For example, if we use a 4-bit operation in the pixel-intensity generator, there are 2, 4, 8 and 16 intensity levels are available for antialiasing operations.

### 3.3 Estimating the Performance of the CF Engine

Speed of operation and area required to implement the algorithm are two important parameters for evaluating the performance of an integrated circuit. Basically, the use of different technologies can greatly affect these performance results. For example, the result of using  $1.5\ \mu\text{m}$  CMOS for implementing a given circuit is faster and smaller in design than those of using  $3.0\ \mu\text{m}$  CMOS for implementing the same circuit. Thus, we use a fixed technology in our discussion of the performance estimation technique.





Note that  $X1$  represents the X-coordinate of the starting point;  
 $Y1$  represents the Y-coordinate of the starting point;  
 $Yc$  represents the Y-coordinate of the central-pixel each cycle;  
 $Sign(d)$  is the sign bit generated from the decision operation;  
 the value of  $x$  is proportional to the possible maximum input value of  $Wd$ .

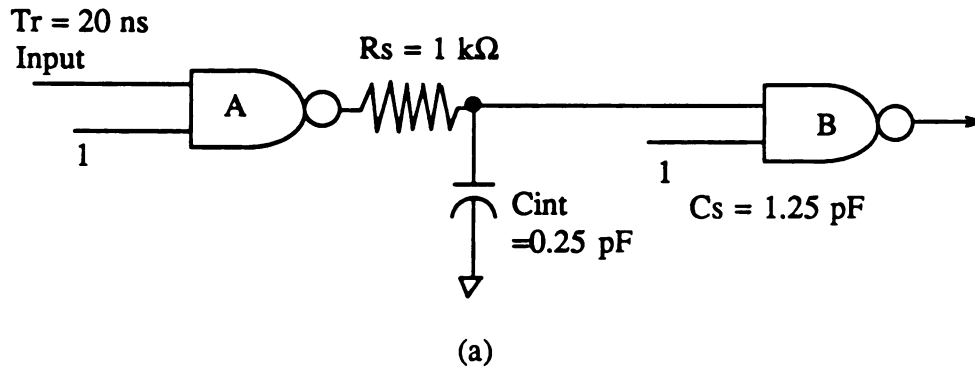
Figure 3-10. Block diagram of the address generator.

Because the accuracy of area estimation of a circuit heavily depends on the placement and routing approaches used, we only consider the timing here.

### 3.3.1 A Timing Estimation Technique

Heinbuch [38] provides several different timing equations for each cell in a CMOS 3 $\mu$ m cell library. These equations can be used to estimate timing delays of a cell in various operational conditions. Our timing estimation technique utilizes this concept to estimate the timing delay of a given circuit in VLSI physical implementation. In general, the entire chip is hierarchically constructed by several functional blocks (cells), and several interconnection wires are used to connect these blocks (cells). We usually use the placement and routing tools to determine physical locations of these cells and wires. In order to reduce the overall timing estimation time, fast placement and routing tools must be used for circuit layout operation.

Thus, according to the interconnection and fan-out information from the circuit layout, we can determine output loadings for all of the cells in an integrated circuit. These output loadings can be applied to the related timing equations of a cell for calculating its timing delay. For example, in Figure 3-11, we use the given timing equations and loading data to calculate the timing delay of a simple cell. Thus, the estimated timing result of the circuit can be determined by summing-up the timing delays of all the cells in the critical path. Note, the critical path of a circuit can be found by applying a switch-level timing simulator to the layout of that circuit. For example, we can use Crystal for estimating timing delays of CMOS circuits [39]. Now, we can estimate the timing result of a circuit if the timing equations of all the cells are available. However, if the timing equations of the required cells are not available, then we have to construct approximate timing equations for these cells.



Output delay:

$$P_d(0-1) = P_{dc}(0-1) + 0.13 (T_{r/f} - 10) + 0.66 R_s CL$$

$$P_d(1-0) = P_{dc}(1-0) + 0.22 (T_{r/f} - 10) + 0.65 R_s CL$$

Output rise time / fall time:

$$T_r = T_{rc} + 0.19 (T_{r/f} - 10) + 2.52 R_s CL$$

$$T_f = T_{fc} + 0.19 (T_{r/f} - 10) + 2.53 R_s CL$$

Basic timing equations:

$$P_{dc}(0-1) = 2.04 CL + 2.9$$

$$P_{dc}(1-0) = 2.08 CL + 3.96$$

$$T_{rc} = 7.36 CL + 3.08$$

$$T_{fc} = 7.32 CL + 2.76$$

(b)

$$CL = C_{int} + C_s = 0.25 + 1.25 = 1.5 \text{ PF}$$

$$T_{fc} = 7.32 * 1.5 + 2.76 = 13.74 \text{ ns}$$

$$P_{dc}(1-0) = 2.08 * 1.5 + 3.96 = 7.08 \text{ ns}$$

$$T_f = 13.74 + 0.19 (20 - 10) + 2.53 * 1 * 1.5 = 19.43 \text{ ns}$$

$$P_d(1-0) = 7.08 + 0.22 (20 - 10) + 0.65 * 1 * 1.5 = 10.25 \text{ ns}$$

(c)

Note that  $C_{int}$  represents the interconnection capacitance;

$C_s$  represents the input capacitance of gate B;

$R_s$  represents overall series resistance.

Figure 3-11. Illustrate of a method to determine the timing delay for a cell:

(a) given circuit diagram; (b) performance equations;

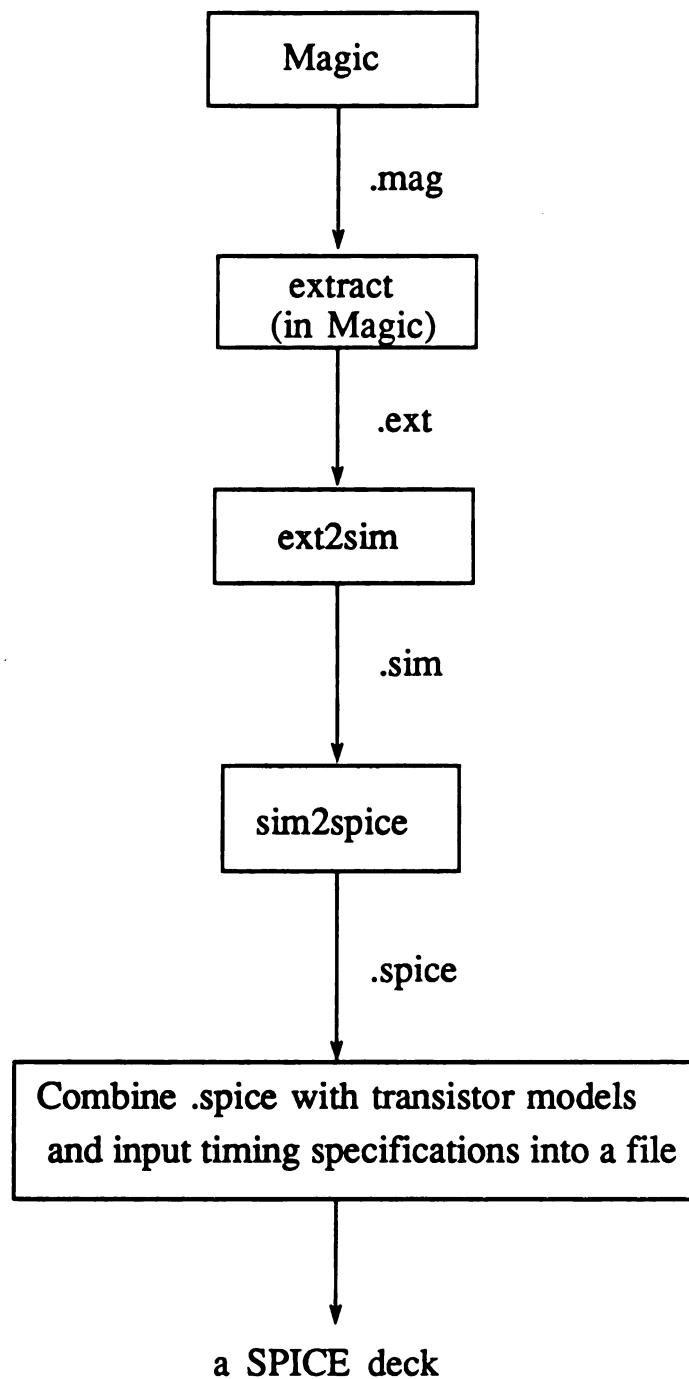
(c) timing delay results for gate A.

SPICE is a simulation tool used at the transistor level to identify the timing data of a circuit by specifying its input timing specifications and output loads. Because SPICE is very time consuming, we usually apply it to circuits less than a couple hundred transistors. Therefore, SPICE can be applied to all the standard-cells and custom designed cells if they contain a reasonable number of transistors. A complete SPICE deck includes transistor models, an extracted circuit information, and the user-specified input data. Basically, silicon foundries will provide SPICE transistor models for a user-specified technology. Figure 3-12 shows a procedure to generate a SPICE deck. It starts with Magic [40] for mask editing a circuit, through several transformation and combination processes, and generates a SPICE deck of that circuit. The important input data that affects the timing characterization of a cell are the rise time and its output loading. Figure 3-13 illustrates the procedure for constructing the timing equations of a circuit. By utilizing this procedure, we can form the approximated timing equations of all the cells used in a given circuit.

In this section, we stated our timing estimation technique. We will evaluate this technique by comparing the timing estimation result with the test result of a line-drawing chip. If the difference between both results are small, then we believe this estimation technique is correct; otherwise, we need to adjust the loading calculations and redo the simulation process.

### **3.3.2 A Generic VLSI Implementation of the Line-Drawing Algorithm**

The basic line-drawing algorithm used here is a modified PW algorithm, which in turn is a modification of the Bresenham line-drawing algorithm. In order to have an integral computation, all the parameters used in the algorithm are multiplied by a main-axis factor. In addition, we also use the replacement process to change expressions of




Note that  represents processes for Magic [40];  
.mag, .ext, .sim, .spice are files generated from those related processes.

Figure 3-12. Procedure for generating a SPICE deck from a magic file.

Step 1. The construction of basic delay equations.

a. Form input specifications:

1.  $T_r = 10$  ns;  $R_S = 0.2$  k $\Omega$ ;  $C_L = 1.5$  pF;
  2.  $T_r = 10$  ns;  $R_S = 0.2$  k $\Omega$ ;  $C_L = 4.0$  pF;
- where Temperature = 25 $^{\circ}$ , 95 $^{\circ}$  or 125 $^{\circ}$ .

b. List equations (one set for each temperature (T)):

$$T_{rc} = x_1 * C_L + x_2$$

$$T_{fc} = x_3 * C_L + x_4$$

$$P_{dc}(0-1) = x_5 * C_L + x_6$$

$$P_{dc}(1-0) = x_7 * C_L + x_8$$

c. Use the input specifications listed above to form two SPICE decks for each T.

After each SPICE run, we can determine  $T_{rc}$ ,  $T_{fc}$ ,  $P_{dc}(0-1)$  and  $P_{dc}(1-0)$ .

Therefore, we can use these two SPICE results to solve for each pair of parameters in the equations.

Step 2. The construction of output delay equations.

a. Form input specifications:

1.  $T_r = 30$  ns;  $R_S = 0.2$  k $\Omega$ ,  $C_L = 1.5$  pF;
2.  $T_r = 30$  ns;  $R_S = 0.2$  k $\Omega$ ,  $C_L = 4.0$  pF; where Temperature = 125 $^{\circ}$ .

b. List output equations:

$$T_r = T_{rc} + y_1 * (T_r / T_f - 10) + y_2 * R_S * C_L$$

$$T_f = T_{fc} + y_3 * (T_r / T_f - 10) + y_4 * R_S * C_L$$

$$P_d(0-1) = P_{dc}(0-1) + y_5 * (T_r / T_f - 10) + y_6 * R_S * C_L$$

$$P_d(1-0) = P_{dc}(1-0) + y_7 * (T_r / T_f - 10) + y_8 * R_S * C_L$$

c. By using the same concept as outlined in Step 1, we can solve for the parameters in these equations.

We use the input specifications listed in Step 2(a) to form the related SPICE decks.

The values of  $T_{rc}$ ,  $T_{fc}$ ,  $P_d(0-1)$  and  $P_{dc}(1-0)$  come from the SPICE results of Step 1.

Note that  $x_1$  to  $x_8$  and  $y_1$  to  $y_8$  are parameters of the given equations.

$T_r$  and  $T_f$  are the rise time and fall time, respectively;

$P_{dc}(0-1)$  and  $P_{dc}(1-0)$  are the propagation delays from 0 to 1 and from 1 to 0.

Figure 3-13. Procedure for constructing the timing equations of a circuit.

parameters in order to save the execution time of the algorithm. Parameters of the modified algorithm are listed below:

(1) The initial value of the decision variable  $d_i$  becomes

$$\begin{aligned} d_1 &= \frac{1}{2}dx - (1-m)dx; \\ &= dy - \frac{1}{2}dx, \end{aligned} \quad (3.16)$$

where  $dx$  is assumed as the main-axis distance factor for the line-drawing operation.

(2) The constant value of the reference  $a$  becomes 0.

(3) For the  $0 \leq m \leq 1$  case, the calculation of the next consecutive decision variable is as follows:

If  $d_i < 0$ , then

$$d_{i+1} = d_i + dy,$$

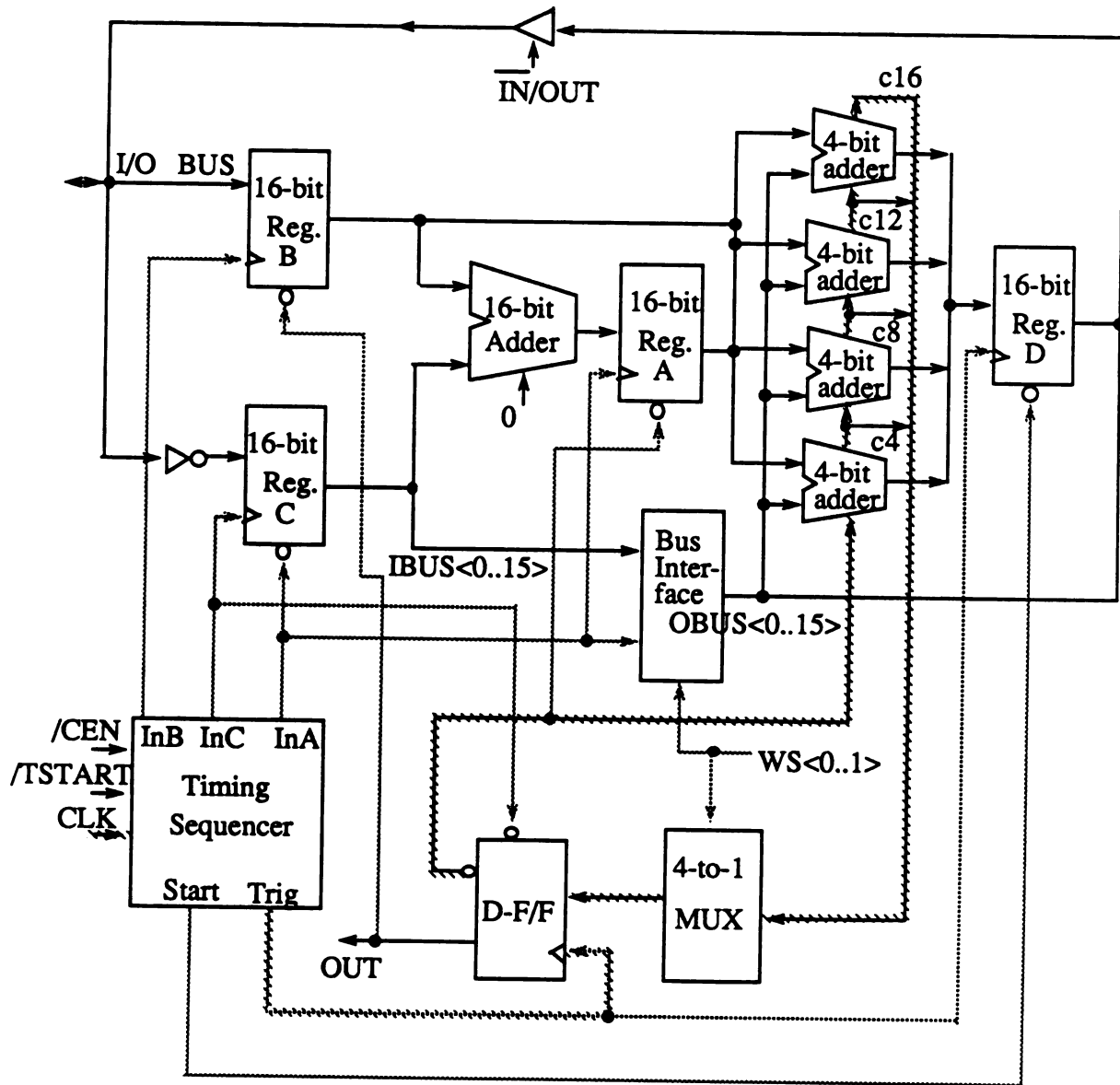
and the lower pixel is selected. If  $d_i \geq 0$ , then

$$\begin{aligned} d_{i+1} &= d_i - (1-m)dx, \\ &= d_i + dy - dx, \end{aligned} \quad (3.17)$$

and the upper pixel is selected.

From all the expressions listed above, the modified line-drawing algorithm uses only integral addition and subtraction operations to incrementally generate pixels for the rasterized line. Thus, this algorithm not only preserves the basic incremental process of the line-drawing but also has a much simpler computational complexity than that of alternative algorithms.

The structural implementation of this line-drawing algorithm follows the same approach as that of the CF engine. Figure 3-14 is a block diagram of this line-drawing engine. It includes both the data flow and the control flow of the line-drawing engine. Also, noted on the diagram is the critical path of this engine. In this implementation, we



Note:  $\rightarrow$  represents data flow;  $\dashrightarrow$  represents control flow;  
 $\sim\sim\sim$  and  $\sim\sim\sim$  represent the critical path of the circuit.

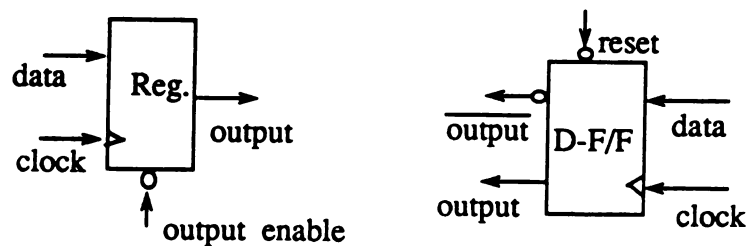


Figure 3-14. Block diagram of the line-drawing engine.

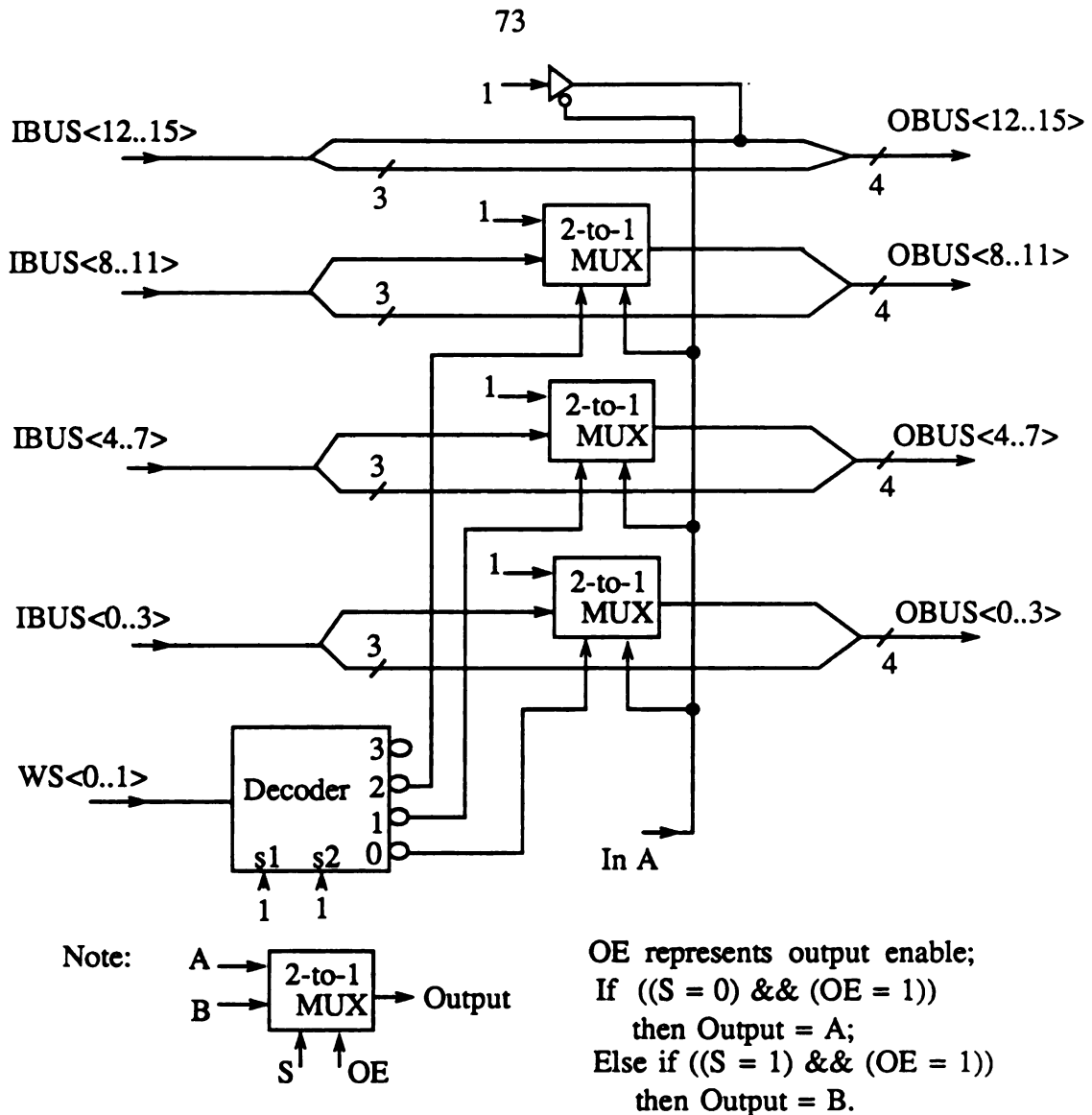


adopt both design flexibility and testability concepts. In order to achieve design testability, the internal bus of the architecture is controllable and observable, i.e., the internal data flow can be set and fetched by users during the output phase. Both the input bus and the internal bus of this architecture use the same I/O pins. Thus, its internal operation and output operation can be executed in parallel during the output phase. The design flexibility of this engine is achieved by using variable operating sizes to generate suitable output data. A shorter execution time in using this engine can be achieved when it is operated in a low-resolution as opposed to a high-resolution display. The user can select 4, 8,

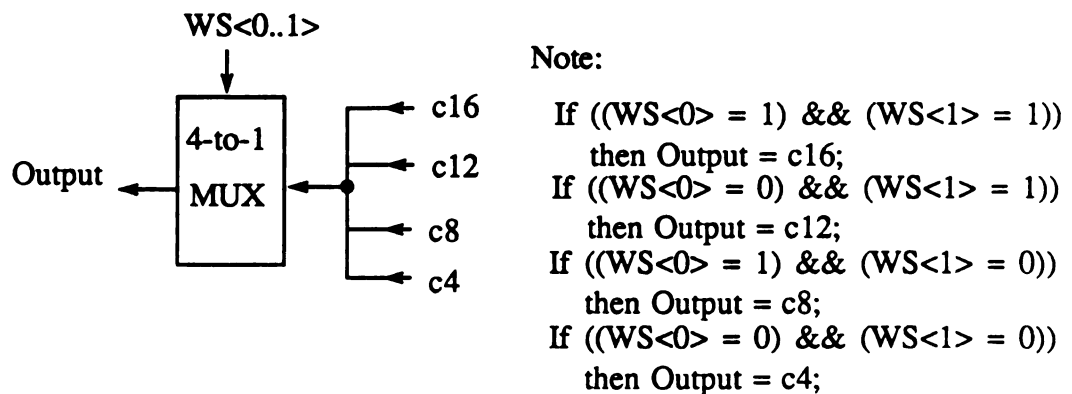
12, and 16-bit operating sizes by using the input switches  $WS<0..1>$ . These switches are used to control the data flow in the bus interface and 4-to-1 multiplexer. Figure 3-15 produces block diagrams of the bus interface and multiplexer.

The timing sequencer is the central part of this circuit. It determines the execution sequence of all the cells in the line-drawing operation. At the beginning of the loading phase, the first output signal and the increment term are generated simultaneously. Then, operations required to generate each output signal and to perform the next computation are executed in a pipeline fashion. Figure 3-16 illustrates the block diagram of this control flow generator and the entire system timing chart.

At the physical level, the layout of a static D-F/F, a 4-bit static carry look-ahead adder and several gates are used as the basic cells for hierarchically organizing the entire circuit. The functional correctness and the timing data of these cells were verified and characterized using SPICE. Since all of the loading data of those cells can be fetched directly from the mask editing results, we only need one SPICE run to get the approximate timing data. The estimated timing delay of this line-drawing engine uses the sum of the timing delay of all cells in the critical path. Final chip layout of our line-drawing engine is shown in Figure 3-17. This 40-pin chip uses the CMOS 3 $\mu$ m technology and occupies an area of 4600  $\mu$ m by 6800  $\mu$ m. Actually, only 25 pin-outs are needed to



(a)



(b)

Figure 3-15. Block diagram of (a) the bus interface and (b) the 4-to-1 multiplexer.

**Figure 3-16. Block diagram of the timing sequencer and the entire system timing chart.**

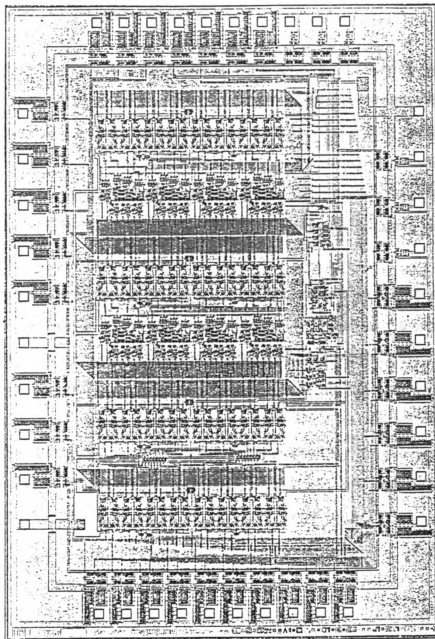


Figure 3-17. Chip layout of the line-drawing engine.

implement the entire design; the extra pins are used for testing.

### 3.3.3 Evaluating the Fabricated Chip

Our test goal is to evaluate the actual performance of the CF engine. Specifically, our objectives were to verify the functional correctness of the line-drawing engine and to evaluate the correctness of our timing estimation technique. This section presents the test procedure and test results of our line-drawing chip. By comparing the estimated results with the test results, we can determine the actual timing and loading data for the cells. Then, we can use these data and the timing technique to estimate the timing results of the CF engine.

We used the Topaz system [41] to test our line-drawing chip. The Topaz system verifies the conformance of a VLSI circuit to design characteristics dynamically by applying a set of digital stimulus vectors to input pins of the device. Resultant device output vectors are captured for analysis or compared directly in real time with expected response vectors. The stimulus and expected response vectors are created by reading a sequence of pattern words from Topaz memory. The original source of the pattern words is most often a vector file created by the device simulator used during the VLSI development to simulate the device.

A device has three general categories of signal pins, i.e., input pins, output pins, and input/output (I/O) pins. In order for the Topaz system to test a device, a Topaz channel must be connected to each device pin involved in the verification, and that channel must be set up to function as an input, output, or I/O channel as appropriate.

The Topaz system operates in conjunction with an IBM PC/AT or compatible personal computer. The user can prepare the desired test patterns and timing delays by filling in the related on-screen data charts. These data are used to set up the test

configuration of the Topaz system. During the testing phase, the user can issue the test runs from the computer. The resultant data will be generated and evaluated in the test unit. Then, these resultant data and the related message will be sent to the computer for display.

A typical test procedure is as follows:

- Step 1:** Allocate and set up Topaz channel, delay, and power sources. This is a planning and initialization step. A Topaz channel must be assigned to each device signal pin involved in the verification. Note that each channel contains two bytes, and these two bytes can be controlled separately. But, all the bits in each byte must have the same data type. For example, if the most significant byte is used as input, the least significant byte can be used as output. A Topaz power supply must be assigned to satisfy the power requirement of the tested device. Topaz delays must be assigned to meet the various timing requirement of the testing. For example, the power source of the tested chip is set to 5 V. We assign the least significant byte of channel E as input, which contains input signals /TSTART, /CEN, and WS<0..1> of the line-drawing engine in Figure 3-14. The input timing delay of this least significant byte is set to 20 ns. This means when the system clock is activated, the related input data will be sent to the chip with a 20 ns delay.
- Step 2:** Enter the test pattern sequence. Both the input data and expected data output sequences need to be created and stored in the Topaz system memory. Each pattern bit must be stored in the channel bit position and assigned to the device pin with which that pattern bit is associated. Each pattern word must contain data appropriate for a particular clock period of the sequence. For example, Figure 3-18 illustrates all the input and output test vectors of the line-



```

      CLK: 010101010101010101 <-- input
/TSTART: 100111111111111111 <-- input delay 20 ns
      /CEN: 111111000000000111 <-- input delay 20 ns
      WS<1>: 111111111111111111 <-- input
      WS<0>: 111111111111111111 <-- input
      IN/OUT: 000000111111111111 <-- input
I/O BUS<0>: x x 0011 x 10011001111 <-- input/output
I/O BUS<1>: x x 0011 x 00011110000 <-- input/output
I/O BUS<2>: x x 1100 x 10011000000 <-- input/output
I/O BUS<3>: x x 0000 x 10011000000 <-- input/output
I/O BUS<4>: x x 0000 x 10011000000 <-- input/output
I/O BUS<5>: x x 0000 x 10011000000 <-- input/output
I/O BUS<6>: x x 0000 x 10011000000 <-- input/output
I/O BUS<7>: x x 0000 x 10011000000 <-- input/output
I/O BUS<8>: x x 0000 x 10011000000 <-- input/output
I/O BUS<9>: x x 0000 x 10011000000 <-- input/output
I/O BUS<10>: x x 0000 x 10011000000 <-- input/output
I/O BUS<11>: x x 0000 x 10011000000 <-- input/output
I/O BUS<12>: x x 0000 x 10011000000 <-- input/output
I/O BUS<13>: x x 0000 x 10011000000 <-- input/output
I/O BUS<14>: x x 0000 x 10011000000 <-- input/output
I/O BUS<15>: x x 0000 x 10011000000 <-- input/output
      OUT: x x 0000011001111111 <-- output

```

Note that all the symbols listed above are the same as those shown in Figure 3-16;  
 "x" represents a tri-state data;  
 "0" and "1" represent the binary data.

Figure 3-18 . Input and output test vectors of the line-drawing engine under a 16-bit operation case.



drawing engine under a 16-bit operation case. These input and resultant vectors are used and generated from the functional simulation of our line-drawing engine at the physical implementation level. Thus, we can enter these vectors as test vectors in the Topaz system.

- Step 3: Wire the test board. In this step, the user connects the pins of the tested chip to the related channel on the test board as he planned in Step 1.
- Step 4: Install the test board.
- Step 5: Plug in the device.
- Step 6: Run and evaluate the tests. We can use single-step operations to check the functional correctness of the chip, and use real-time operations to find the timing results of the chip. In the single-step operation, if the input data is applied to the input pins in the current step, then the related output data will be displayed in the next step. The duration of each step is controlled by the user. By comparing the single-step results with the expected output data, we can verify the functional correctness of the tested chip. In the real-time operation, both the test and comparison operations can be performed simultaneously. The comparison operation is used to detect the errors by comparing the generated results with the expected data. If there exists any error, the location of that error will be reported. In addition, in order to detect the maximum clock rate of the chip, we must apply several different internal clock rates to the Topaz system and do the test several times. This maximum clock rate is the highest clock rate used for producing the correct outputs of the tested chip.

By following this test procedure and using the same data in Figure 3-18, we determined that all of the internal blocks of the line-drawing engine were functioning correctly. In our test, only the I/O pad drive presents an abnormal operation, which is latched in its output mode all the time. This can be fixed by replacing it with the correct I/O pad. Also, by applying several different internal clocks, we measured that the peak throughput of this engine is about 12.5-M pixels per second. This is close to our timing estimated result of the line-drawing engine, i.e., its critical path delay is about 80 ns. Thus, our timing estimation model is very realistic.

## CHAPTER 4

### EXISTING TWO-LAYER CHANNEL ROUTING APPROACHES

Another aspect of the research described here deals with two-layer channel routing and mask generation in the VLSI physical implementation. The immediate goal here is to develop fast routers for use in the physical layout of the CF antialiased drawing engine. Also, these routers can be applied to standard-cell and gate-array designs for solving their embedded channel routing problem.

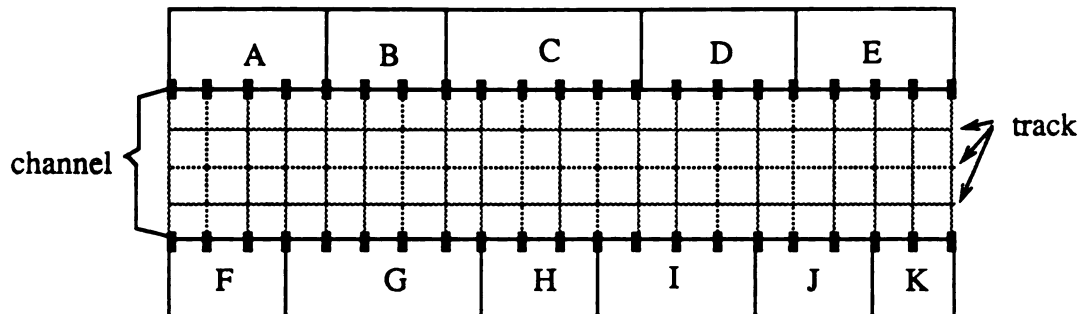
Because the two-layer channel routing problem is NP-complete [43], several existing routing algorithms use different heuristic approaches. In addition, most existing channel routers use a symbolic routing method to generate their routing results in terms of symbols. According to these symbolic data, the traditional mask generator is used for producing the related routing masks. In this chapter, we investigate existing two-layer channel routing and mask generation algorithms in order to identify useful routing and mask generation concepts. These concepts will steer the development of improved two-layer channel routing algorithms.

The minimal routing area, minimal total wiring length, and minimal required vias are three important performance data used to evaluate channel routers. Also, the complexity of their routing algorithms is another factor used to identify the useful routers in the investigation. In the evaluation of the traditional mask generator, we emphasize its ability to verify the correctness of the routing masks as well as its operational flexibility for integrated routing systems.

## 4.1 Problem Formulation

The basic terms used to define the channel routing problem are stated as follows. The rectangular region between two parallel rows of cells is called a **channel**, and it is used for connections among the cells. One channel may consist of several tracks, and each **track** may consist of a row of horizontal grid line segments. The **pins** are located on the top and bottom edges aligned on the grid of the channel. Figure 4-1 illustrates an example of a channel and its related features. A **net** is a set of pins to be electrically connected, and we usually assign a positive integer number to the pins of each net. If a zero is assigned to a pin, then no net will be connected to this pin. In the channel, if the maximum assigned number is  $n$ , then  $n$  is also the maximum number of nets to be routed. Therefore, the channel routing problem can be stated as a **netlist**. It gives the net numbers to be connected to the related top and bottom pins from left to right. For example, Figure 4-2 is a netlist which consists of twelve columns and seven nets.

Usually, two layers are used to complete all of the routing paths, and one layer (e.g., metal\_1) is used for horizontal segments and another layer (e.g., metal\_2) is used for vertical segments. A connection between these two layers uses a **via**. The horizontal segment of a net is determined by its leftmost and rightmost pin connections. Let  $S(i)$  be the set of nets whose horizontal segments intersect column  $i$ . The number of elements in each  $S(i)$  is called the **local density** of column  $i$ . The **channel density** is the largest number of elements among all of the local densities within a channel. This number is also equal to the minimum number of tracks necessary to route a channel. For example, in Figure 4-2,  $S(1) = \{1, 2\}$ ,  $S(2) = \{1, 2, 3\}$ , and  $S(3) = \{1, 2, 3, 4\}$ , etc.. So, the local density of column 1 is equal to 2, and the local density of column 2 is equal to 3, and so on. By examining all of the local densities, we find that the channel density of this channel is equal to 4.



Note that the crossing of two dash lines represents a symbolic grid point;  
 cells A-K represent standard cells;  
 each ■ represents a pin in the channel.

Figure 4-1. Example of a channel and its related features.

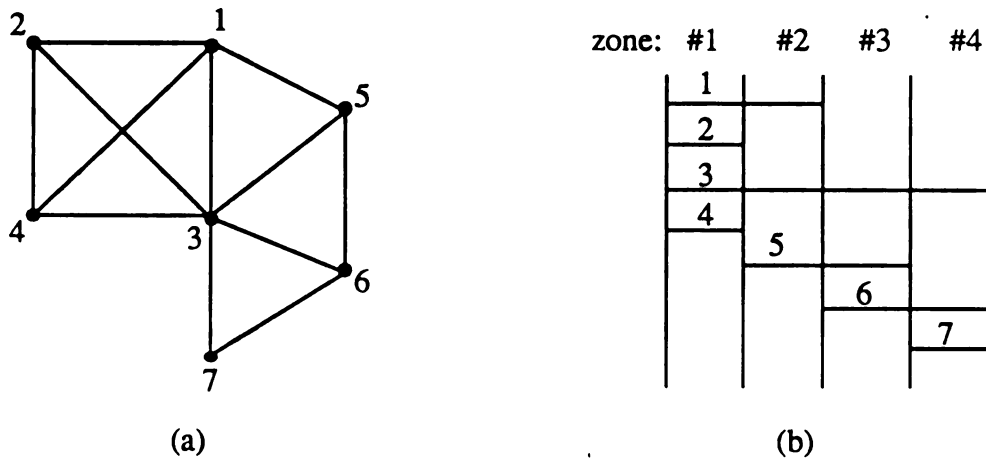
Top: 3 1 3 0 0 5 6 0 3 0 0 0  
 Bottom: 1 2 4 2 4 1 5 7 0 7 6 0

Note that all pins with the same assigned number belong to the same net.

Figure 4-2. Example of a netlist.

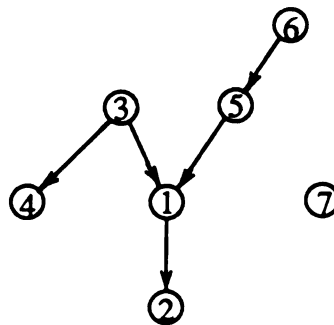
In most of existing two-layer channel routing algorithms, several definitions are used to deal with channel routing problems. They are listed as follows.

- **horizontal constraint:** A constraint that exists when two nets cross the same column.
- **horizontal constraint graph:** A graph in which each node represents a net. An edge between node  $i$  and node  $j$  represents these two nets have a horizontal constraint. For example, Figure 4-3(a) illustrates a horizontal constraint graph of the netlist given in Figure 4-2. It shows that net 1 and net 2 are connected by an edge because they cross the same columns 2, 3, and 4.
- **zone representation:** Another graphical representation of horizontal constraints in a channel. It uses one column to represent a zone and each horizontal line segment to represent a net. Each zone represents the maximal set  $S(i)$ . So, a net may cross several zones and a zone may consist of several nets. For example, Figure 4-3(b) illustrates a zone representation of the netlist given in Figure 4-2. It shows that zone 1 consists of nets 1, 2, 3, and 4 because  $S(3) = \{1, 2, 3, 4\}$  and  $S(3)$  is the maximal set among five consecutive  $S(i)$ s, where  $i = 1, 2, 3, 4$ , and 5.
- **vertical constraint:** A constraint that exists when two nets consist of a pin in the same column. The net connected to the top pin must have its horizontal segment above that of the net connected to the bottom pin in that column.
- **vertical constraint graph:** A graph in which each node represents a horizontal segment or net. A directed vector from node  $i$  to node  $j$  means that horizontal segment  $i$  must be placed above horizontal segment  $j$  because of a vertical constraint. For example, Figure 4-4 illustrates a vertical constraint graph of the netlist given in Figure 4-2. According to the leftmost column of the netlist, net 3 should be placed above net 1. This is the reason why a vector is drawn from net 3 to net 1 in this vertical constraint



Note that each number represents a net.

Figure 4-3. Illustration of (a) the horizontal constraint graph, and (b) the zone representation for the netlist given in Figure 4-2.



Note that  $\textcircled{n}$  represents net  $n$ ;  
 $\rightarrow$  represents a vector.

Figure 4-4. Illustration of the vertical constraint graph for the netlist given in Figure 4-2.

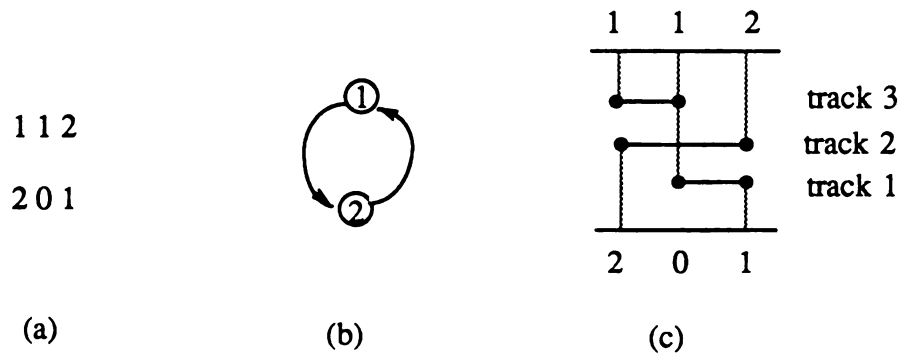
graph.

- **cyclic vertical constraint:** A loop shown in the vertical constraint graph. For example, Figure 4-5(a) shows a given cyclic netlist, and Figure 4-5(b) is the vertical constraint graph of this problem.
- **doglegging:** A solution to the cyclic vertical constraint case. It is also useful to reduce the number of tracks. For example, Figure 4-5(c) shows a solution using doglegs to solve the given cycle problem in a channel.
- **wrong way horizontal segment:** A wire segment which uses the metal\_2 layer instead of the original metal\_1 layer. It can be used to reduce the number of required tracks for routing and/or to eliminate cycle vertical constraints. For example, Figure 4-6 illustrates the use of a wrong way horizontal segment to reduce the number of wiring tracks from 3 to 2. However, most of the routers avoid using this wrong way routing since it induces the wire capacitance in the overlapped area of both layers.

Except for the wrong way routing, four channel routing cases can be found, i.e., routing without vertical constraints, without doglegs, with **restricted doglegs**, and with **unrestricted doglegs**. A restricted dogleg is a dogleg which can only be used in columns that contains a pin of the selected net. Whereas, an unrestricted dogleg is a dogleg which can be used in any free space in the channel. Figure 4-7 illustrates the example of these four routing cases. The differences among these four routing cases are on their requirement of the routing area, total wiring length, and number of vias.

The goal of the channel router is to obtain the minimal routing area, minimal total wiring length, and minimal required vias in the routing result. The execution speed and memory requirement are two other factors used to evaluate routers. In the next section, we discuss several existing routing strategies and their trade-offs.





Note: — represents a vertical wire segment in the first layer;  
 — represents a horizontal wire segment in the second layer;  
 ● represents a via (contact).

Figure 4-5. Examples of a cyclic routing cases: (a) given netlist, (b) vertical constraint graph, and (c) the routing solution.

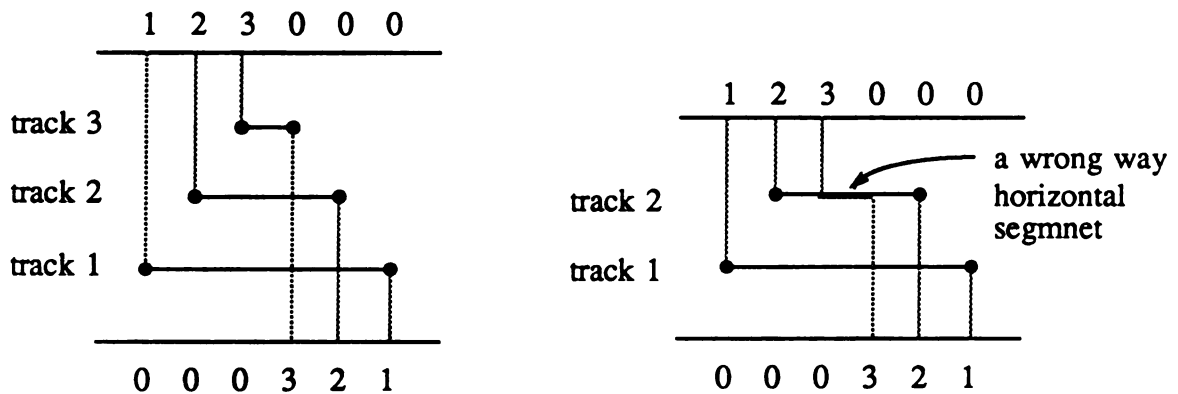
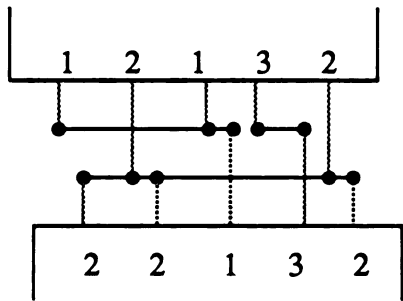
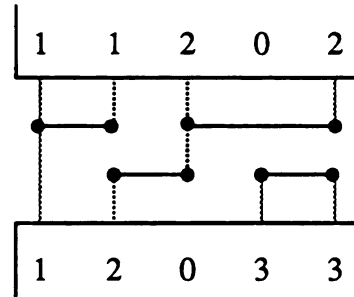


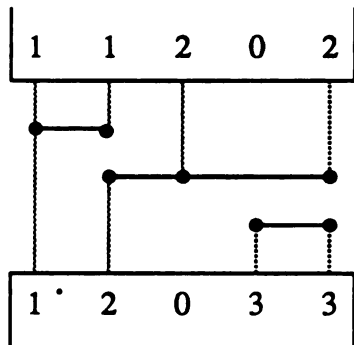
Figure 4-6. Illustration of the use of a wrong way horizontal segment: (a) before the operation; (b) after the operation.



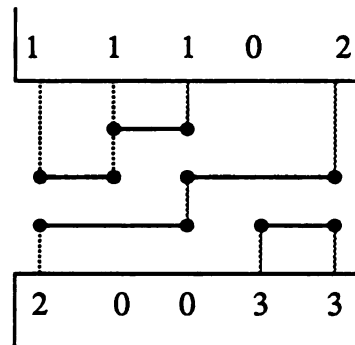
(a)



(c)



(b)



(d)

Figure 4-7. Examples of four different routing cases: (a) without vertical constraints; (b) without doglegs; (c) with restricted doglegs; (d) with unrestricted doglegs.

## 4.2 Existing Two-Layer Channel Routing Algorithms

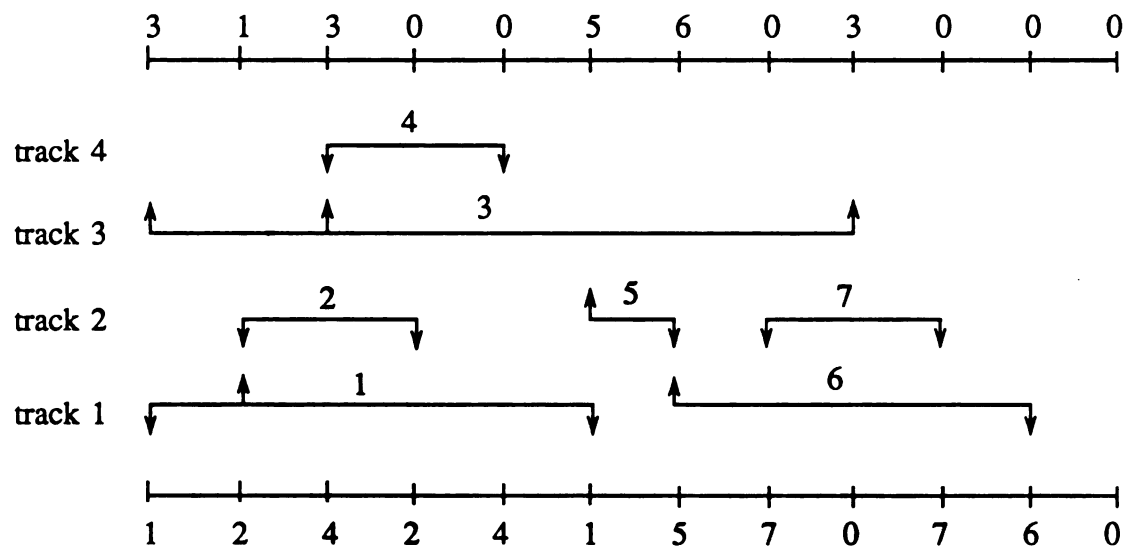
### (1) The left-edge algorithm:

Hashimoto and Stevens [8] attempted to maximize the placement of horizontal segments in each track. The edges by the left endpoint of each segment are sorted. For example, the sorted list of the netlist in Figure 4-2 is

1 3 2 4 5 6 7.

The algorithm selects the first edge, 1, and places it in the first track of the routing region. Net 1 is deleted from the sorted list. By scanning the remaining sorted list from left to right, the algorithm then assigns the next suitable net to track 1. This selected net should not have any horizontal constraint with net 1. According to the zone representation given in Figure 4-3(b), this selected net is net 6. The above process is repeated until no more nets can be placed on track 1. The algorithm starts again using the remaining nets in the sorted list and filling track 2, and so on. The final net assignment made by the left-edge algorithm is shown in Figure 4-8.

One serious problem of this routing solution is that the algorithm does not consider any vertical constraints. For example, in Figure 4-8, net 3 and net 4 will be shorted at the third column if we connect the pins of net 3 and the pins of net 4. However, this left-edge algorithm finds a channel density solution if there are no vertical constraints. Note that this channel density result is the minimal number of tracks one can get in any two-layer channel routing operation. Thus, the basic routing concept of this algorithm is useful for the routing without the vertical constraints case. Because the left-edge algorithm uses this "from left to right" placing rule, the use of this algorithm may not generate an optimal total vertical length for a given routing problem. This is the main disadvantage of the left-edge algorithm.



Note that the netlist used here is the same as that in Figure 4-2.

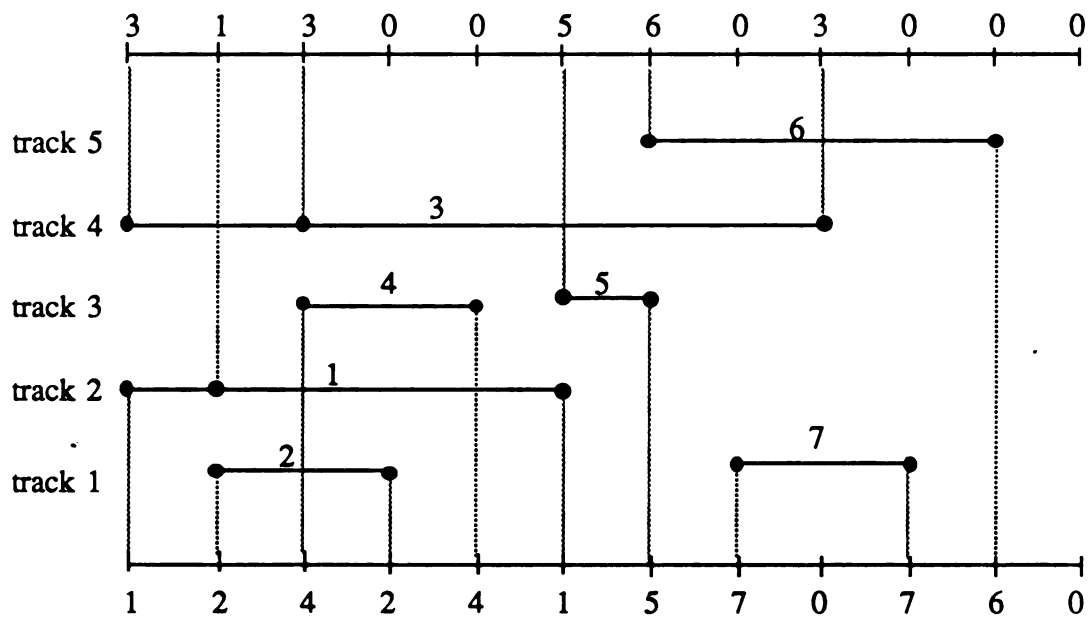
Figure 4-8. Left-edge net assignment result.

(2) The constrained left-edge algorithm:

By using the same routing concept of the left-edge algorithm, Perskey, et al. [44] suggested a constrained left-edge algorithm. In addition to the left-edge algorithm, this algorithm is realized by using a vertical constraint graph. In this graph, the nets have no descendants and meet the left-edge selection requirements are the best candidates for the net assignment operation. If this algorithm is applied to the netlist in Figure 4-2, its final net assignment result is illustrated in Figure 4-9. Although the generated number of tracks is equal to the channel density plus one, it is still an optimal solution. Thus, this constrained left-edge algorithm can be applied to route the routing without doglegs case. However, this algorithm can not be used for cycle vertical constraint routings. In addition, this algorithm has the same problem as that of the original left-edge algorithm.

(3) The dogleg algorithm:

The dogleg algorithm proposed by Deutsch [45] uses doglegging to avoid vertical constraint loops and to decrease the density of the channel. This algorithm breaks each multiple-pin net into several individual horizontal segments. Each break is a possible candidate to form a restricted dogleg in the channel. By using a realized vertical constraint graph, the algorithm first selects suitable horizontal segments for the bottom track. These suitable segments are the nodes without any descendants in the graph. The routing strategy used here is similar to that of the restricted left-edge algorithm. Then, the dogleg algorithm selects another suitable horizontal segments for the top track. These horizontal segments are the nodes without any ancestors in the graph. Now, it uses a reverse restricted left-edge operation, which places horizontal segments from right to left. Then, the algorithm processes the second-from-the-bottom track; and son on, until all segments are placed. This dogleg algorithm can be used to produce less tracks results than that of the constrained left-edge algorithm. However, the use of the dogleg algorithm will generate



Note that the netlist used here is the same as that in Figure 4-2.

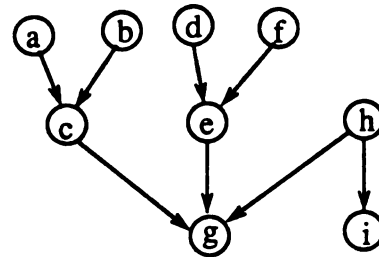
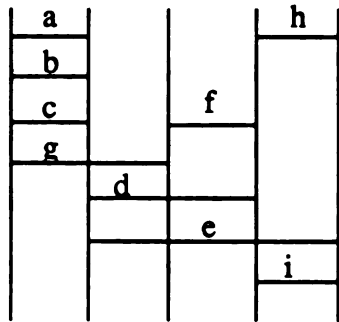
Figure 4-9. Constrained left-edge net assignment result.

much more vias.

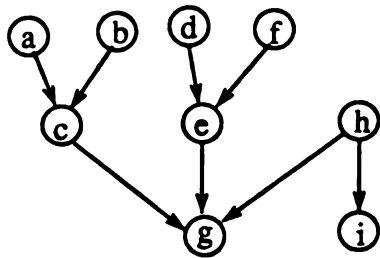
(4) Three efficient channel routers:

Two algorithms by Yoshimura and Kuh [9] and one algorithm by Yoshimura [10] are used to provide alternative net assignment strategies for the channel routing. The first Y&K algorithm uses both the zone representation and vertical construction graph to minimize the longest path in the vertical constraint graph. In order to minimize the longest path, the algorithm utilizes a merging process. Starting from the first zone, this merging process is operated between two consecutive zones. Its basic operational concept is that if two net have no horizontal or vertical constraints, then they can be merged, i.e., these two nets can be placed on the same track. The result of each merging is an update of the zone representation and vertical constraint graphs. This process is finished when no further merging takes place. For example, Figure 4-10 illustrates the merging operations of a given zone representation and vertical constraint graph. A cost function is setup in order to find the suitable two nets for each merging operation.

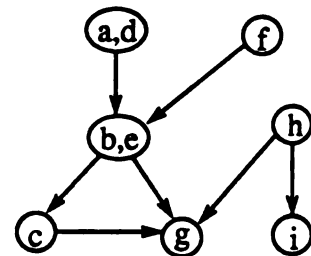
Since the first Y&K algorithm uses the sequential update operation, a merging of two nets may block subsequent mergings. The second Y&K algorithm attempts to provide a better merging process than the first algorithm. It uses a bipartite graph  $G_A$ , where a node represents a net and an edge between two nets signifies they can be merged. A merging is expressed by a matching on the graph, and it can be updated dynamically. This matching process is also operated one zone at a time. But, several possible pairs of nets are considered for merging. During each matching operation, the algorithm detects and deletes the infeasible edges in the graph  $G_A$ . These infeasible edges contains pairs of nets which may violate the vertical constraint graph in the matching operation. The maximum matching for each net is determined by using the same cost function defined in the first algorithm. Finally, two nodes are matched in the graph when both nets are ter-



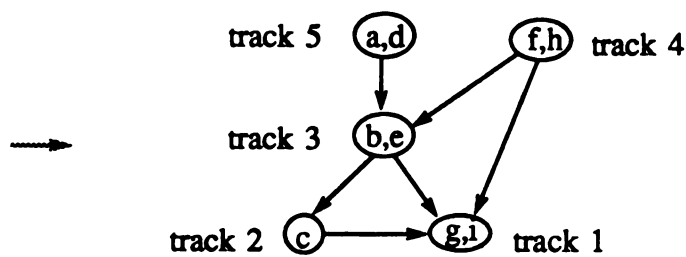
(a)



VG for zone 1



VG for zone 2 and zone 3



VG for zone 4 and the final track assignment

(b)

Note: VG represents the vertical constraint graph.

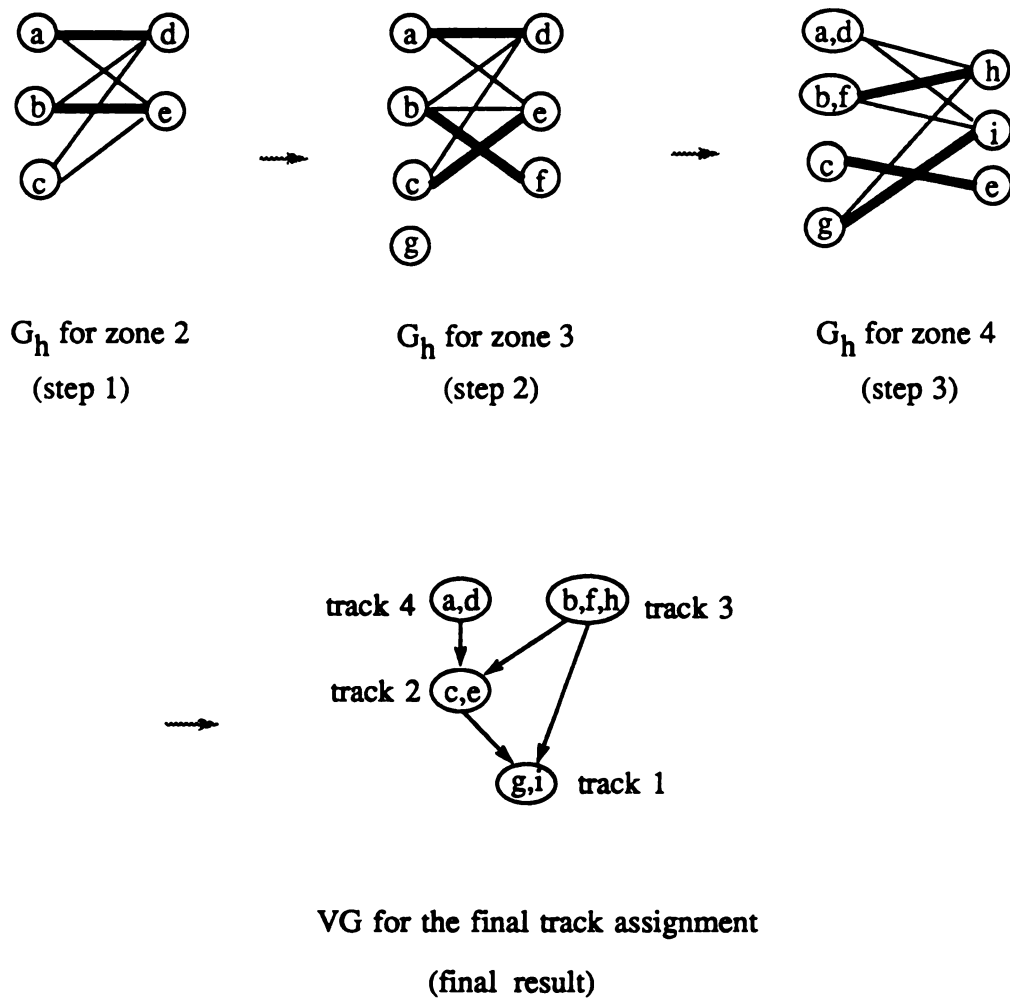
Figure 4-10. Illustration of the merging operations:  
 (a) given zone representation and vertical constraint graph;  
 (b) merging operations.



minated. For example, Figure 4-11 shows the matching operations using the same zone representation and vertical constraint graph given in Figure 4-10. It shows that the use of the second Y&K algorithm can generate better routing results than that of the first algorithm. However, this second algorithm needs a large amount of CPU time and memory space for checking several possible merging paths.

The Yoshimura efficient algorithm [10] attempts to reduce the total number of tracks by maximizing the net assignment. It constructs both the zone representation and vertical constraint graphs. In addition, it uses a weighting function to assign net weights. A set of nets are called the available nets if they have no ancestors in the vertical constraint graph. A net assignment algorithm is used to process these available nets for calculating the longest path length. This algorithm uses a path graph which shows the relationship between the zone representation and the available nets. In this graph, a longest path is formed by selecting the nets with the highest combination of weights. This net assignment algorithm is listed in Figure 4-12. By applying this algorithm to an example, its related net assignment operations and results are shown in Figure 4-13. The entire algorithm is based on a row by row approach. Thus, the net weight assignment process is crucial because it greatly affects the final location of the nets. Despite this net weight assignment process, the third algorithm is the fastest algorithm among all three efficient algorithms.

By dealing with the nets or the subnets, these three algorithms can be used for the without doglegs routing or with restricted doglegs routing. All of these algorithms use heuristic weighting or cost functions to assign net weights. So, they are heavily depend on the weights in order to generate good results. Actually, for a given netlist, these three efficient routers generate similar routing results.



Note: VG represents the vertical constraint graph;  $G_h$  represents a bipartite graph;  
 — represents the possible merge; — represents a maximum matching.

Figure 4-11. Illustration of the matching operations for the given zone representation and vertical constraint graph in Figure 4-10.

Definition:

$ZS(N)$  : leftmost zone of net  $N$   
 $ZE(N)$  : rightmost zone of net  $N$   
 $W(N)$  : net weight  
 $ZN(Z)$  : set of available nets whose left ends are in zone  $Z$   
 $ZP(Z)$  : zone potential  
 $ZA(Z)$  : list of current selected net in zone  $Z$   
 $\#Z$  : number of zones

Phase 1:

```

begin
  ZP(0) = 0;
  ZA(0) = 0;
  for I = 1 to #Z do
    begin
      ZP(I) = ZP(I-1);
      ZA(I) = ZA(I-1);
      for N  $\in$  ZN(I) do
        begin
          P = ZP(ZS(N)) + W(N);
          if P  $\geq$  ZP(I) then
            begin
              ZP(I) = P;
              ZA(I) = N;
            end
          end
        end
      end
    end
  end

```

Phase 2:

```

begin
  Na = { };
  N = ZA(#Z);
  while N  $\neq$  0 do
    begin
      Na = Na  $\cup$  {N};
      N = ZA(ZS(N));
    end
  end
end

```

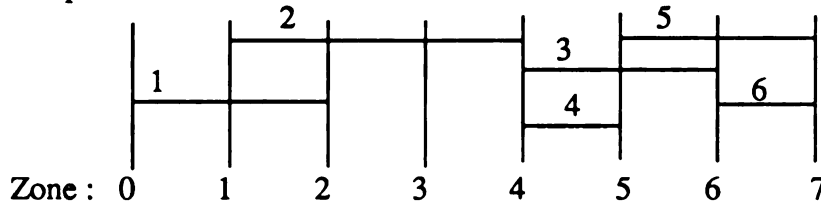
Note that the final selected nets for assignment are in  $Na$ .

Figure 4-12. Yoshimura net assignment algorithm.

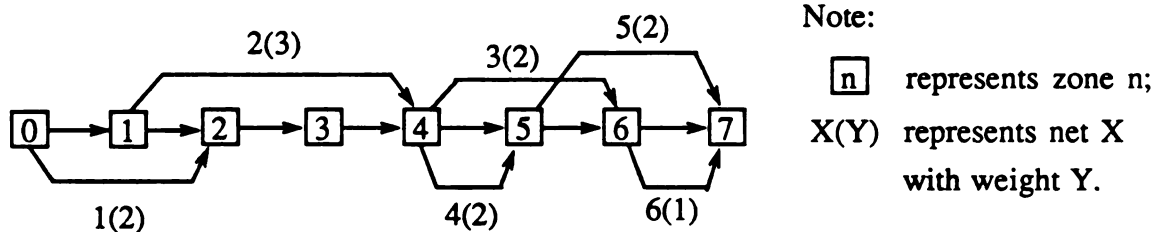
Available nets : 1, 2, 3, 4, 5, 6.

Net weights:  $W(1) = 2$ ,  $W(2) = 3$ ,  $W(3) = 2$ ,  $W(4) = 2$ ,  $W(5) = 2$ ,  $W(6) = 1$ .

Zone representation:



(a)



(b)

Initial data:

N	1	2	3	4	5	6
ZS(N)	0	1	4	4	5	6
ZE(N)	2	4	6	5	7	7

Z	0	1	2	3	4	5	6	7
ZN(Z)	0	0	1	0	2	4	3	5,6

Phase 1:

Z	0	1	2	3	4	5	6	7
ZP(Z)	0	0	2	2	3	5	5	7
ZA(Z)	0	0	1	1	2	4	3	5

Phase 2:

$$N_a = \{ 5, 4, 2 \}$$

(c)

Note that all the notations used here have the same definitions as in Figure 4-11.

Figure 4-13. Illustration of the Yoshimura net assignment algorithm:

(a) given net information; (b) path graph; (c) related operations and results.

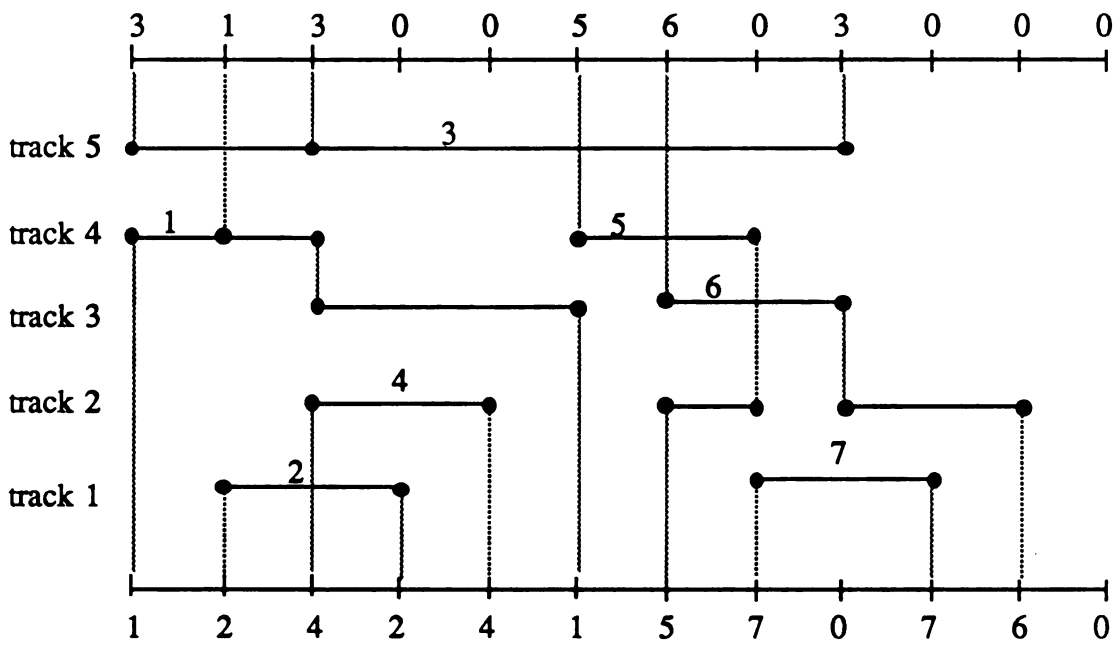
(5) The Greedy channel router:

Rivest and Fiduccia [46] proposed a column by column approach to route a channel instead of using the traditional row by row approach. The operation starts from the left side of the channel. In each column, the router tries to maximize the number of tracks available in the next vertical column by using a sequence of heuristics. According to the pin locations of the routed nets, these routing strategies are used to determine the best wiring patterns for the nets at each column.

The Greedy router uses a jog to provide an additional track while a net tries to move closer to its next pin location. So, for using this router, the user needs to specify the value of the minimum jog length for each given netlist. No jogs are made when the free vertical wiring distances are shorter than this minimum jog length. A higher value of minimum jog length reduces the number of vias, and a lower value minimizes the number of tracks in the routing. It is obvious that this router can be used for the routing with unrestricted doglegs case. For the given same netlist as in Figure 4-2, Figure 4-14 illustrates the final routing result by using the Greedy router. It shows that the use of this router generates the same number of tracks as those of the previous algorithms. However, this router uses more vias and total wiring length than those of the previous algorithms. In addition, the number of tracks used for routing must be defined before the routing operation starts. The operation is failure if the provided routing space is not enough. For this case, the router must be re-executed again with an increased number of tracks.

(6) The hierarchical channel router:

Burstein [47] proposed a divide and conquer approach to reduce the channel routing problem to the case of a  $(2 \times n)$  grid, where  $n$  is the number of columns in a channel. Consider the generalized problem for a  $(m \times n)$  grid. Partition the grid into two parts:  $((m/2) \times n)$  and  $((m/2) \times n)$  sub grids. This router treats each part as a  $(1 \times n)$  grid; so, the problem is



Note that the netlist used here is the same as that in Figure 4-2.

Figure 4-14. Greedy routing result with the minimum jog length = 1.

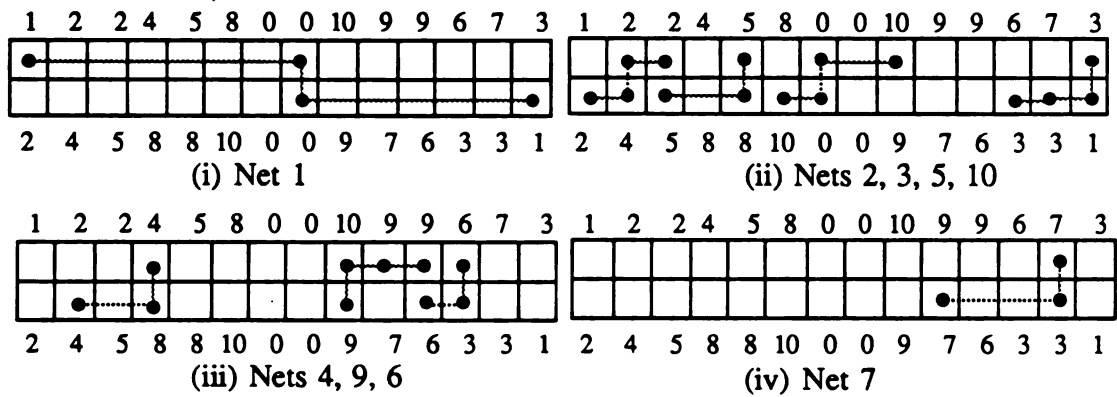
reduced to the case of the  $(2 \times n)$  grid.

Initially, the router needs to set the costs of crossing all of the horizontal and vertical boundaries in the realized  $(2 \times n)$  grid. It routes one net at a time, and this net can be selected randomly. A wiring path of a net is selected if its overall cost is the cheapest one among those of all the possible paths. Then, the costs of boundaries in the  $(2 \times n)$  grid is updated according to the path of this routed net. Figure 4-15 illustrates the operations and result of the hierarchical router for a given netlist. It shows that the hierarchical router can be used for the routing with unrestricted cases. In addition, With a good cost assignment, this router can generate an optimal tracks results. However, the hierarchical router uses a large number of vias; and, its algorithm complexity can be very high, i.e., the upper bound is  $(Nn \log_2 m)$ , where  $N$  is the number of nets and  $m$  is the number of tracks used.

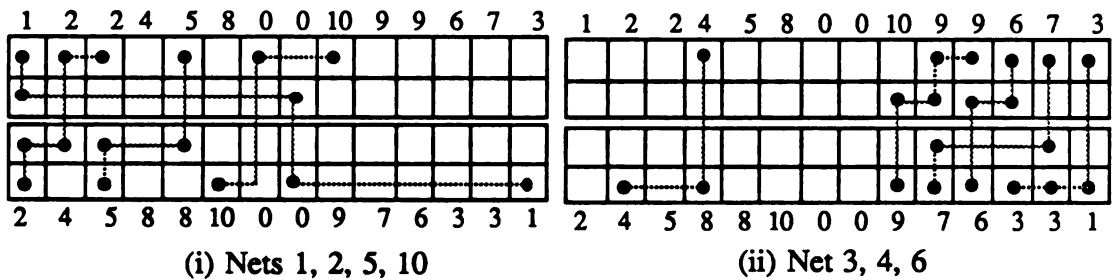
(7). The YACR2 router:

Reed, et al. suggested a new symbolic router: YACR2 [11]. This router can route channels with cyclic constraints and uses a virtual grid. Basically, it uses one layer for vertical interconnections and the other layer for horizontal interconnections as other channel routers mentioned previously. However, YACR2 uses the wrong way horizontal segments to reduce the number of routing tracks.

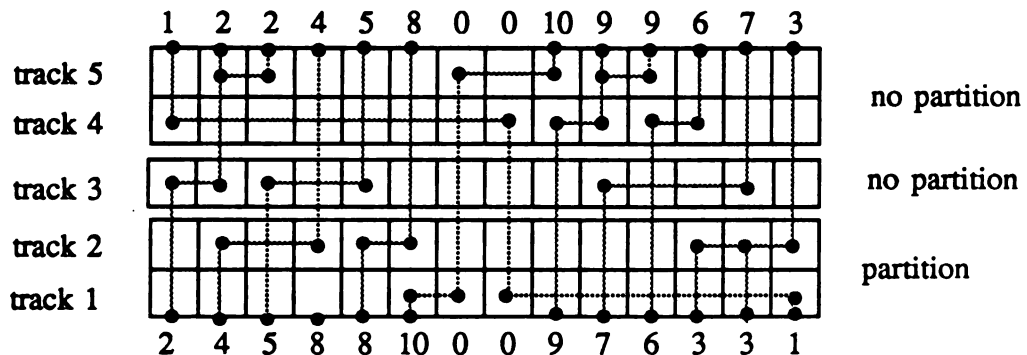
The router first selects and assigns all the nets to the suitable tracks, which is related the first, second, and third phases of the algorithm. Then, it uses the simplified maze routers to complete the routing, which is related to the fourth phase of the algorithm. Note that a cost matrix is used to determine which net is selected, and which track is assigned. In order to support these operations, this router adopted both the left-edge and maze routing strategies. It modifies the left-edge algorithm in order to reduce the overall vertical constraint violations and to be easy for maze router operations. At the final rout-



(a)



(b)



(c)

Note:  $\square$  represents one grid;  $---$  represents the assigned connection;  
 • represents a turning point or a pin.

Figure 4-15. Illustration of the hierarchical routing: (a) first partition;  
 (b) second partition, and (c) third partition and final result.



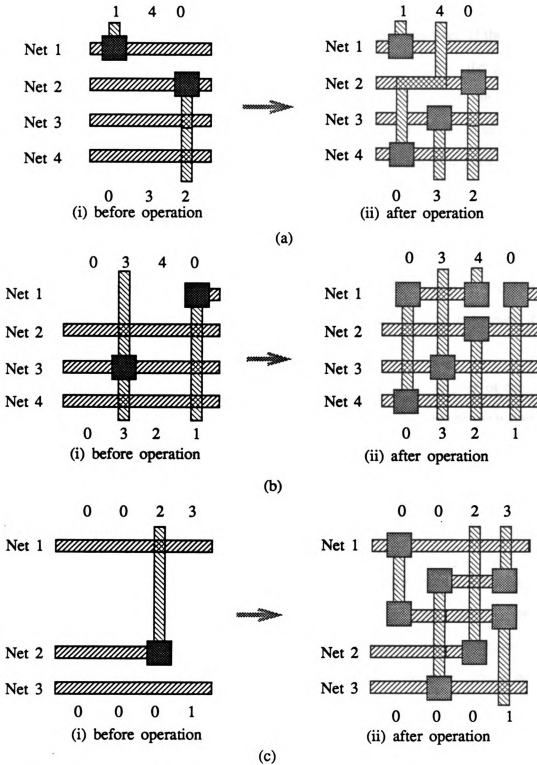
ing phase, the nets and their pins without vertical violations will be routed first. Then, three types of maze routers are used to connect the rest of the nets and pins with vertical violations. These three types of maze routers are maze1, maze2, and maze3. Note that maze1 does not introduce any vias per violation, maze2 introduces at most two additional vias per violation, and maze3 introduces at most four additional vias per violation. Figure 4-16 illustrates the application of these three maze routers. If all of three maze routers can not route any of the given net, then one additional row will be added, and the entire algorithm may need to be applied again. Once all the nets and pins are connected, the router terminates.

As a result, the use of the YACR2 router can generate a small number of vias and tracks for a given netlist. Also, with an enough routing space, the execution speed of the YACR2 router is faster than that of the hierarchical router. However, the execution speed will be slow down if the routing should be re-operated again, or the cost matrix used for both net selections and net assignments is improperly set.

#### (8). The Mighty switch-box router:

Since the use of switch-box router can generally solve the two-dimensional routing problem, it can also solve the two-layer channel routing problem. Shin and Sangiovanni-Vincentelli [12] proposed a routing technique to complete the routing within a switch-box. This technique uses the pre-defined costs of each wire and via to select the best path for routing. It also uses the wrong way horizontal segments for constructing the shortest path for the routed nets. In addition, it allows modifications and rip-ups of nets when an existing shortest path is not optimal or when no path exists.

Four main processes are used in the Mighty router: path finder, path conformer, weak modifier, and strong modifier. A path finder searches for a minimum-cost path among all available pins. A maze router finds this minimum-cost path. When all the nets



Note: ▨ represents the first layer, ▩ represents the second layer, ■ represents a via (contact).

Figure 4-16. Examples of (a) mazer1 routing, (b) mazer2 routing, and (3) maze3 routing.

have been processed by the path finder, the path conformer takes over. If the path can be routed without any violations, the path is implemented. Otherwise, the path finder is invoked again to find a new path. If a costly path needs to be routed, the modifiers are used. First, the weak modifier is used to push the existing paths or vias in all of the possible directions. This may create free spaces for routing the current path. When weak modification fails to find a path, then strong modification is called. During a strong modification, part of existing connections are removed so that the blocked net can be connected. After the path is connected, all the nets disconnected during the rip-up process are re-scheduled and re-connected. If the strong modification also fails to find a path, then the router reports failure and exits. Figure 4-17 illustrates some examples of operations of weak and strong modifications.

As a result, the use of the Mighty router can find a small number of total wiring length and tracks for most of the routing cases. However, the computational complexity of this router is  $O(k^3pnL)$ , where  $k$  is number of nets,  $p$  is the number of pins,  $n$  is the number of columns,  $L = O(mn)$ , and  $m$  is the number rows. This is much higher than that of any one of the channel routers described previously. In addition, a failure condition may occur when the user selects a bad cost for input.

In summary, we have identified several useful routing concepts from the existing channel routing algorithms described in this section. The left-edge algorithm provides a fast and complete operational concept to deal with the routing without the vertical constraints case. The Yoshimura efficient algorithm provides an efficient routing strategy to deal with routing with vertical constraints case. It can generate a better routing result than that of the constrained left-edge algorithm. Also, this efficient algorithm is faster than that of the first Y&K algorithm and the second Y&K algorithm. In order to generate a much compact routing result, the routers must use the unrestricted doglegging strategy. In this case, the maze routing concept of the YACR2 and Mighty routers is superior to

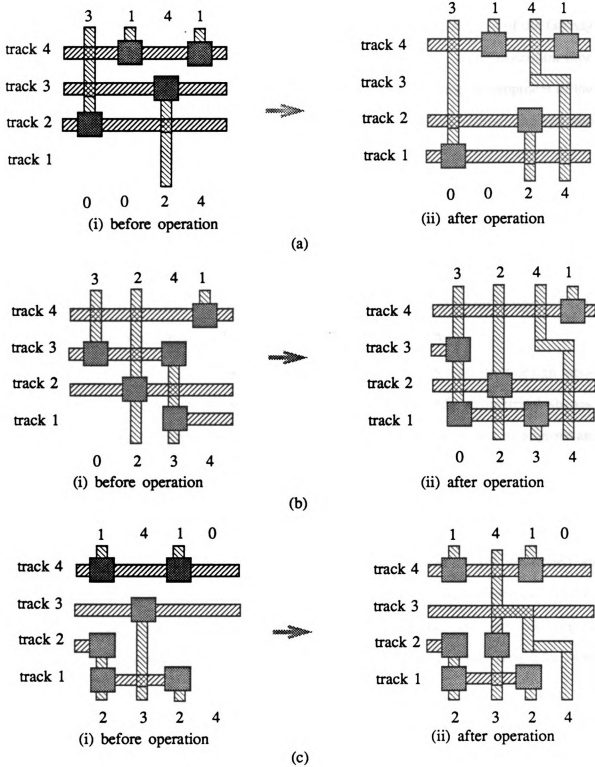


Figure 4-17. Examples of weak and strong modifications: (a) unit-push (down);  
(b) jump-push (down); (c) rip-up and reroute.

both the partitioning method of the hierarchical router and jogging method of the Greedy router. However, the use of maze routing usually requires a much longer execution time and also requires the wrong-way routing capability. Therefore, in the development of the routers, we focus on the routing without vertical constraints, routing without doglegs, and routing with restricted doglegs.

### 4.3 Existing Mask Generators

We now turn our attention to the problem of mask generation. Our goal is to use our knowledge about channel routing to generate a powerful and efficient design automation tool which fully integrates the routing and mask generation tasks.

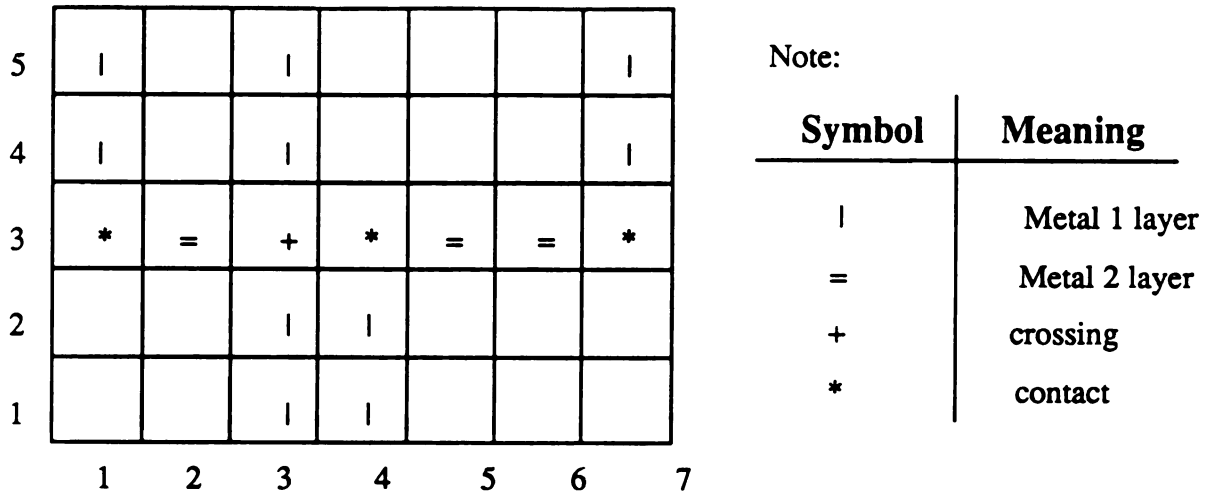
The basic symbolic layout technique uses a set of symbols to construct design topological layout schematic. Each layout can be mapped into specific geometrical shapes. These shapes can be defined by sizing data and subsequently subjected to appropriate modification, depending on contextual relationships with the adjacent orthogonal and diagonal layout symbols. Larsen [5] suggested several symbol-to-geometrical mappings which can be easily comprehended by device designers and the requisite algorithms are amenable to computer implementation. In addition, Rogers, et al. [6] developed an integrated symbolic design system. This system provides several useful features: technology independent layout tools, scale independent circuit designs, fast layout debugging using symbolic level circuit simulation, and fully automated mask generation and automated chip assembly.

A subset of this symbolic-to-mask operation can be applied to the routing mask generation. According to the given symbolic routing data and sizes of the wires and via (contact), a routing mask generator is used to produce its related routing masks. These masks are represented by the mask generation statement, and it can be used in the chip

fabrication. By realizing the traditional mask generation operation, Figure 4-18 shows the example of the symbolic routing, mask generation statement, and related mask result. Note that the sizes of the wires and via are set without violating any design rules.

According to the traditional approach, a symbolic array is used in both the routing operation and the mask generation. Once the symbolic array is formed, there is no way to detect routing errors in the array unless we simulate the entire circuit. Thus, the main disadvantage of using this approach is that it is hard to detect any routing violations in the generated masks. In addition, when used in the interactive mask editing system, this approach can be difficult to perform mixed-mode routing operations. Mixed-mode routing is a combination of automatic routing and user-defined with the latter usually used to reduce the total wiring length of a specified net.

In the next chapter, we will present a new routing approach which is used for integrating both the router and the mask generator. In addition to improving the routing performance, we also adopt the mixed-mode routing operation and the routing verification concepts in our development.



(a)

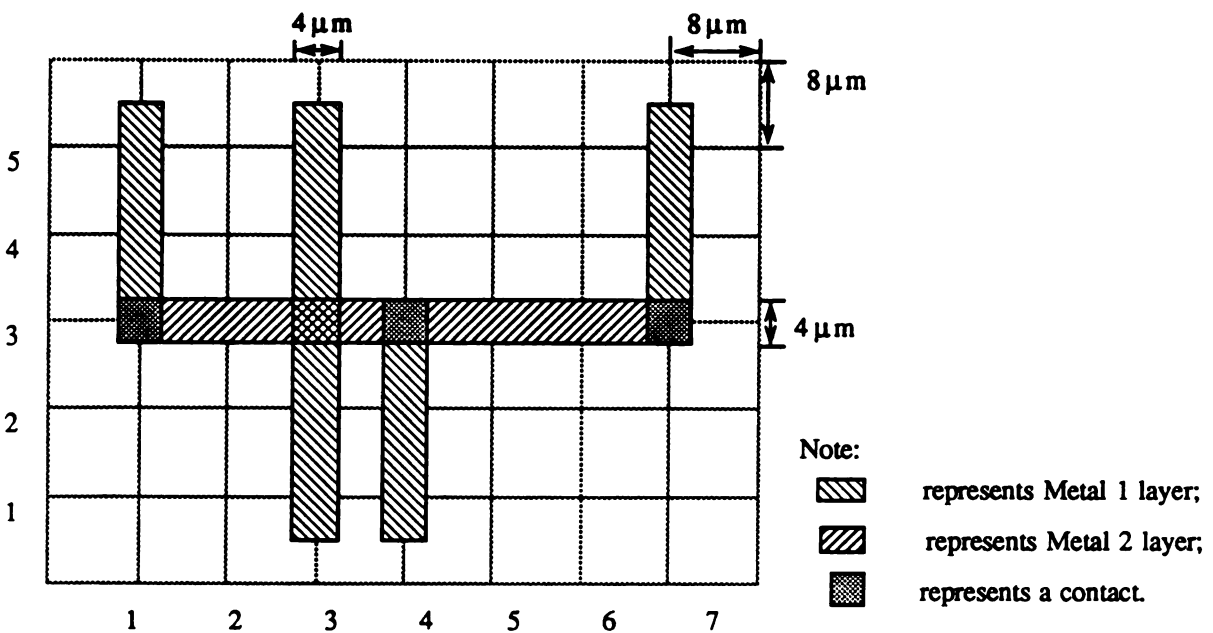
```

# Routing example
#
# Metal 1 layer wires
wire   Metal_1   w=4μm, l=8μm (1,4) (1,5)
      "          "          (3,1) (3,5)
      "          "          (4,1) (4,2)
      "          "          (7,4) (7,5)

# Metal 2 layer wires
wire   Metal_2   w=4μm, l=8μm (2,3) (3,3)
      "          "          (5,3) (6,3)

# Contacts
contact M1_M2    w=4μm, l=4μm (1,3)
      "          "          (4,3)
      "          "          (7,3)
  
```

(b)



(c)

Figure 4-18. Examples of traditional symbolic routing and mask generation:

(a) symbolic routing; (b) mask generation statement; (c) final mask result.

## **CHAPTER 5**

### **THE CF TWO-LAYER CHANNEL ROUTING APPROACH**

This chapter presents our two-layer channel routing approach. The objective here is to efficiently generate the acceptable routing masks for the given channel. Two major operations are involved in the routing approach, i.e., the intermediate routing operation and the mask generation. The intermediate routing operation is used to produce a sequence of intermediate routing result according to the given netlist. Based on different routing cases, we develop three different routing strategies. These routers can be readily applied to the mixed-mode operation for minimizing the wiring length of the user-specified net. In addition, they can generate optimal or near optimal routing results for several well-known examples.

The mask generator translates the intermediate routing data into masks. It utilizes a tiling technique and several operating rules for reducing the number of vias and verifying the correctness of the routing result. Also, a routing performance measurement process is used to determine the number of vias and total wiring length in the routing result. The routing verification process is implemented in the mask generation level in order to detect all the possible routing errors resulting from the routers.



## 5.1 The CF Mask Generator

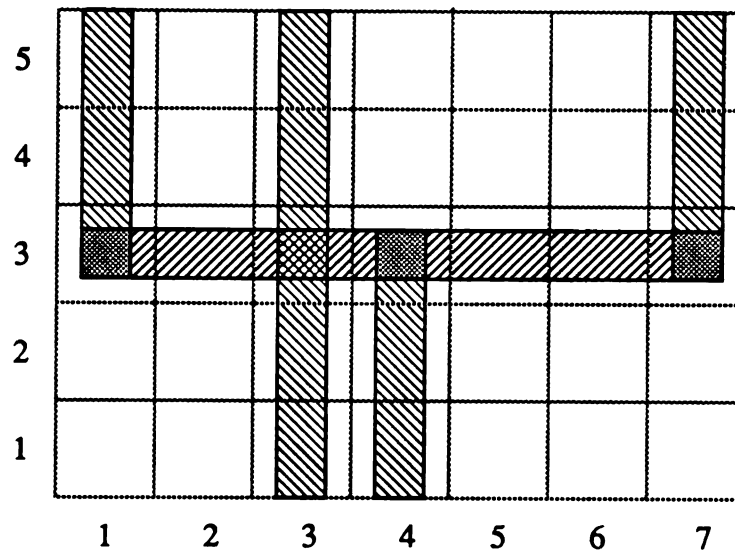
In this section, we present several important data structures and processes in order to construct the CF mask generator. These data structures and processes are listed as follows: (1) the intermediate routing data, (2) the tiling assignment, (3) the tiling construction rules, (4) the routing violation detection rules, (5) the vias minimization, and (6) the performance data generation.

### (1) Intermediate routing data:

We define the format of the intermediate routing data so that the interface between the high level routing process and the low level mask generation can be simplified. An intermediate routing data is defined as  $(x_1, y_1, x_2, y_2, t)$ , where  $(x_1, y_1)$  and  $(x_2, y_2)$  represent the starting and ending coordinates of a wiring path in a grid-based channel, respectively. And,  $t$  represents the assigned track number for this wiring path. For example, the intermediate routing data,  $(1, 5, 4, 1, 3)$ , means a wiring path which starts from grid  $(1, 5)$  and ends at grid  $(4, 1)$  by passing through track 3. Figure 5-1 shows three wiring paths used to construct the mask result for a given channel, and their related intermediate routing data are  $(1, 5, 4, 1, 3)$ ,  $(4, 1, 7, 5, 3)$ , and  $(3, 5, 3, 1, 0)$ , respectively. Note that coordinate  $(1, 1)$  is the base coordinate of the bottom-left grid in the channel, and  $t = 0$ , means that the related wiring path is a straight wire. An important feature of this intermediate routing format is that it can be readily used for mixed-mode operations. The user can specify the wiring path of any two points by editing the related routing data interactively.

### (2) Tiling assignment:

In Figure 5-1, the entire routing mask is partitioned into several small blocks. Each block of the mask is called a tile, and each coordinate (grid) in the channel is related to



Note:

Each block represents a tile;



represents Metal 1 layer;



represents Metal 2 layer;

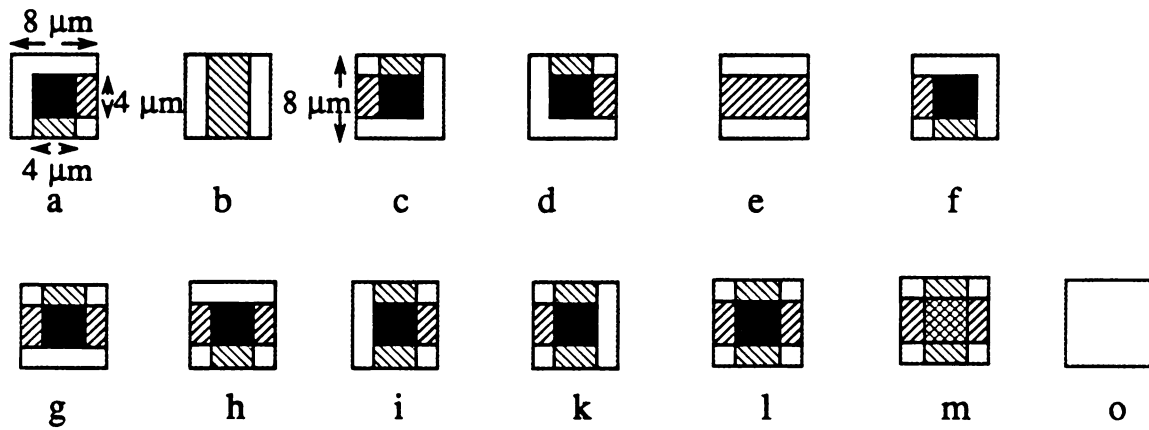





represents a via (contact).

Figure 5-1. Tiling mask result for a given channel.

only one tile. The array structure of the tiles in the channel is called a tiling array. This tiling array can be used as a cell in the final layout of a circuit. We assume that the routing operation follows the traditional strategy in using layers, i.e., one layer for the horizontal wires and the other layer for the vertical wires. Based on this assumption, we can pick up several outstanding tiles from the mask result as our basic tiles. For example, in Figure 5-1, the mask result consists of 35 tiles, where it only needs 7 basic tiles to represent its mask result. Except for the wrong way routing, we evaluate the masks in most of the routing results and determine all the possible basic tiles. As a result, thirteen basic tiles are selected and can be used to construct most of the mask results. Figure 5-2(a) shows these basic tiles along with their temporary assigned symbols. Basically, we assign a normalized wire-size to all of the wires and vias in the tiles, and it is based on a 3-um technology.

For the routing without vertical constraints and without dogleg cases, six equations are used to determine the relationships among these thirteen basic tiles. These equations are established by following a from left-to-right and from top-to-bottom wiring construction sequence. Figure 5-2(b) shows these six tiling construction equations. By solving these equations, thirteen numbers, from 0 to 12, are assigned to symbols of these tiles, respectively. Note that tile number 0 represents an empty tile. These basic tiles are separated into two groups, i.e., the simple tile group and the constructed tile group. Each tile in the constructed tile group is combined by two or more tiles from the simple tile group. Figure 5-3 illustrates thirteen basic tiles with the assigned numbers, and they are separated into the simple tile group and the constructed tile group. Note that the spaces inside each tile are used to avoid the design rule violations. This tiling assignment is correct because the constructed equations listed in Figure 5-3 match the symbolic equations listed in Figure 5-2.



Note:  means Metal 1 layer;  means Metal 2 layer;  
 means the contact between Metal 1 layer and Metal 2 layer.

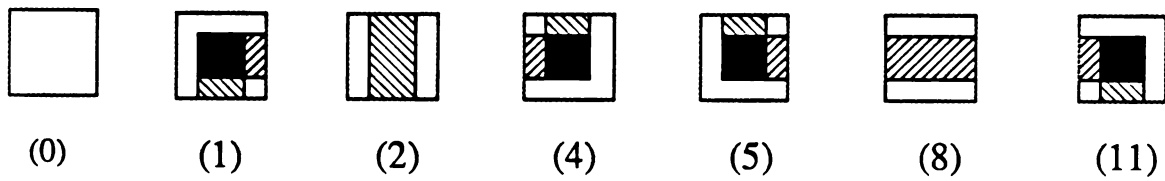
(a)

1.  $c + d = g$ ;
2.  $f + a = h$ ;
3.  $b + a = i$ ;
4.  $c + b = k$ ;
5.  $k + a = l$ ;
6.  $e + b = m$ .

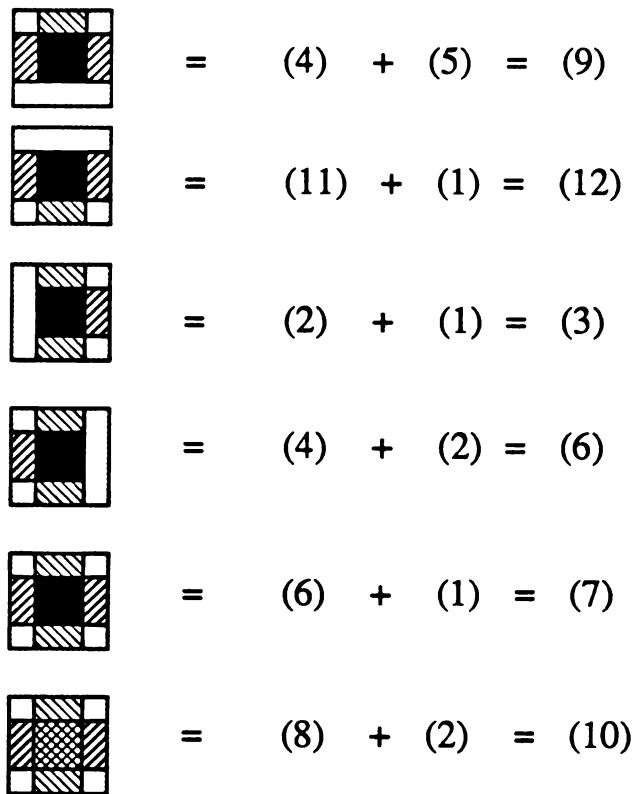
(b)

Figure 5-2. Illustration of the basic tiles and six tiling construction equations:

- (a) thirteen basic tiles with their temporary assigned symbols;
- (b) six tiling construction equations .



(a)

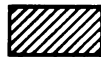


(b)

Note:



means Metal 1 layer;



means Metal 2 layer;



means the contact.

Figure 5-3. Thirteen basic tiles with the assigned numbers: (a) simple tiles; (b) constructed tiles.

For the with doglegs routing cases, we still use these thirteen tiles; however, some constraints must be set for their tiling construction operations. Also, the number of tiles must be increased when we use vias minimization operations.

(3) Tiling construction rules:

For a given channel, we need to determine which tile should be used at any of specified locations. The tiling construction rules are used to determine the tiles along a wiring path; one wiring path corresponds to one intermediate routing datum. Three types of tiles are used in tiling construction operations, i.e., resulting tile, desired tile, and existing tile. The resulting tile number,  $T(i,j)$ , is determined by the desired tile number,  $D(i,j)$ , and the existing tile number,  $E(i,j)$ . Note that  $(i,j)$  can be any coordinate in the tiling array. The desired tile number is a tile number used to represent one of the grids along the wiring path, and it is determined without considering the content of the tiling array. The existing tile number is a tile number of a grid, which already exists in the tiling array. Initially, all the existing tile numbers are set to 0.

For the routing without dogleg case, two tiling construction rules are established, which are derived directly from the six construction equations listed in Figure 5-2(b).

**C\_Rule 1.1:** If  $E(i,j) = D(i,j)$  or  $E(i,j) = 10$ , then  $T(i,j) = E(i,j)$ .

**C\_Rule 1.2:** Except for C\_Rule 1.1,  $T(i,j) = D(i,j) + E(i,j)$ .

The processing sequence for each the wiring path is the same as the input sequence of the intermediate routing data. According to each intermediate routing data input, we form a set of tiles for the related wiring path. Figure 5-4 illustrates an example using these construction rules. In each processing step, the tiling array is efficiently filled with one set of tiles for each wiring path. The final tiling array can be directly mapped to the same routing mask result as shown in Figure 5-1.

Wire 1: (1, 5, 4, 1, 3)

Wire 2: (3, 5, 3, 1, 0)

Wire 3: (4, 1, 7, 5, 3)

(a)

Step 1:

(wire 1 construction)

5	2	0	0	0	0	0	0
4	2	0	0	0	0	0	0
3	5	8	8	11	0	0	0
2	0	0	0	2	0	0	0
1	0	0	0	2	0	0	0
	1	2	3	4	5	6	7

Step 2:

(wire 2 construction)

5	2	0	2	0	0	0	0
4	2	0	2	0	0	0	0
3	5	8	10	11	0	0	0
2	0	0	2	2	0	0	0
1	0	0	2	2	0	0	0
	1	2	3	4	5	6	7

Note:  $D(3,3) = 2$  and  $E(3,3) = 8$ ;  
then,  $T(3,3) = 10$ .  
(C\_Rule 1.2)

Step 3:

(wire 3 construction)

5	2	0	2	0	0	0	2
4	2	0	2	0	0	0	2
3	5	8	10	12	8	8	4
2	0	0	2	2	0	0	0
1	0	0	2	2	0	0	0
	1	2	3	4	5	6	7

Note:  $D(4,1) = 2$  and  $E(4,1) = 2$ ;  
then,  $T(4,1) = 2$ .  
(C\_Rule 1.1)

Final tiling array

(b)

Figure 5-4. Example of using C\_Rules 1.1 and 1.2 to construct the tiling array:  
(a) given intermediate routing data; (b) tiling array constructions.

For the routing with dogleg case, the tiling construction rules are more complicated than those of the routing without dogleg case. This is because that the subnets of each net can be assigned with different tracks for routing with dogleg operations. These construction rules are described as follows:

**C\_Rule 2.1:** If  $E(i,j) + D(i,j) = 13$  and ( $E(i,j) = 11$  or  $E(i,j) = 2$ ), then  $T(i,j) = 6$ .

**C\_Rule 2.2:** If  $E(i,j) + D(i,j) = 7$  and ( $D(i,j) = 2$  or  $D(i,j) = 5$ ), then  $T(i,j) = 3$ .

**C\_Rule 2.3:** If  $E(i,j) = D(i,j)$  or  $E(i,j) = 10$ , then  $T(i,j) = E(i,j)$ .

**C\_Rule 2.4:** Except for C\_Rules 2.1 to 2.3,  $T(i,j) = D(i,j) + E(i,j)$ .

Note that C\_Rule 2.3 is the same as C\_Rule 1.1, and C\_Rule 2.4 is the same as C\_Rule 1.2. Thus, C\_Rules 2.1 to 2.4 are the generalized tiling construction rules for most of the routing cases. Figure 5-5 illustrates an example using these construction rules. It also shows the final routing mask for this given routing with dogleg channel.

#### (4) Routing violation detection rules:

Basically, for any two different nets, if the horizontal (vertical) wires of their wiring paths are overlapped, then these two wiring paths consist of horizontal (vertical) violations. During each of the tiling construction process, the CF mask generator can detect the horizontal and vertical routing violations using several rules. In order to determine these routing violation detection rules, we first establish all the possible horizontal and vertical routing violation cases. Figure 5-6 shows these cases, which include two horizontal violation cases and eight vertical violation cases. Then, according to the relationships among the tiles in each of these cases, we form the violation detection rules below:

**HD\_Rule:** If  $E(i,j) \diamond 2$  and  $E(i,j) \diamond 0$ , then the routing has horizontal violations, where  $(i,j)$  is one of the coordinates along the horizontal wire of the current wiring path.



Wire 1: (1, 5, 4, 1, 2)  
 Wire 2: (4, 1, 5, 5, 3)  
 Wire 3: (5, 5, 7, 1, 2)

(a)

5	2	0	0	0	0	0	0
4	2	0	0	0	0	0	0
3	2	0	0	0	0	0	0
2	5	8	8	11	0	0	0
1	0	0	0	2	0	0	0
	1	2	3	4	5	6	7

5	2	0	0	0	2	0	0
4	2	0	0	0	2	0	0
3	2	0	0	1	4	0	0
2	5	8	8	6	0	0	0
1	0	0	0	2	0	0	0
	1	2	3	4	5	6	7

5	2	0	0	0	2	0	0
4	2	0	0	0	3	8	11
3	2	0	0	1	4	0	2
2	5	8	8	6	0	0	2
1	0	0	0	2	0	0	2
	1	2	3	4	5	6	7

Step 1: (wire 1 construction)

Step 2: (wire 2 construction)

Step 3: (wire 3 construction and  
the final tiling array)

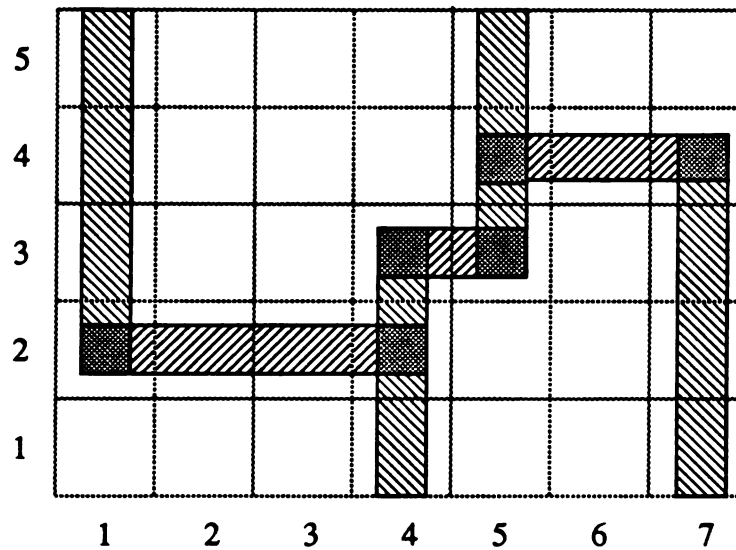
Note:

T(4,2) uses C\_Rule 2.1.

Note:

T(5,4) uses C\_Rule 2.2.

(b)

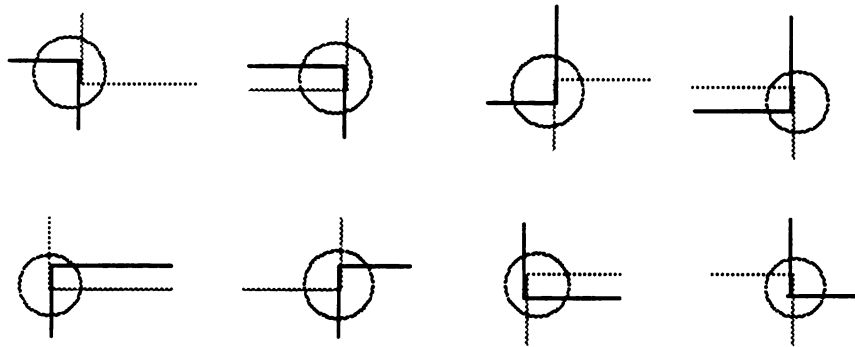


(c)

Figure 5-5. Example of using C\_Rules 2.1 to 2.4 to construct the tiling array:  
 (a) given intermediate routing data; (b) tiling array construction;  
 (c) routing mask result.



(a)



(b)

Note that each circle represents the place where the violation occurs;  
 ..... represents the first wire (Wire 1);  
 ——— represents the second wire (Wire 2).

Figure 5-6. Illustrations of the routing violation cases: (a) two horizontal violation cases; (b) eight vertical violation cases.

- VD\_Rule 1:** If  $E(i,j) = 2$ , then  $\text{Left\_count} = \text{Left\_count} + 1$ , where  $(i,j)$  is a coordinate of the left corner of the current wiring path.
- VD\_Rule 2:** If  $E(i,j) = 2$ , then  $\text{Right\_count} = \text{Right\_count} + 1$ , where  $(i,j)$  is a coordinate of the right corner of the current wiring path.
- VD\_Rule 3:** If  $E(i,j) = 1$  or 4 or 5, or 11, then  $\text{Left\_count} + 1$ , where  $(i,j)$  is one of the coordinates along the left-vertical wire of the current wiring path.
- VD\_Rule 4:** If  $E(i,j) = 1$  or 4 or 5, or 11, then  $\text{Right\_count} + 1$ , where  $(i,j)$  is one of the coordinates along the right-vertical wire of the current wiring path.
- VD\_Rule 5:** If  $\text{Left\_count} = 2$  or  $\text{Right\_count} = 2$ , then the routing consists of vertical violations. Note that this rule is used to check the value of  $\text{Left\_count}$  and  $\text{Right\_count}$  at the end of each wiring path. Initially, both  $\text{Left\_count}$  and  $\text{Right\_count}$  are set to 0.

Figure 5-7 illustrates three examples using these rules to detect their horizontal and vertical routing violations. It shows that we not only can detect any routing violations but also can easily identify the locations of routing violations. These outstanding features will lead us to verify the correctness of the routing result. Also, we can quickly adjust the routing process if the CF mask generator locates a routing error.

#### (5) Vias minimization:

In order to reduce the number of vias in the routing result, we apply a vias minimization process to the tiling array. This process tries to change the layer used for horizontal wires into another layer used for the vertical wires if no errors exist. Figure 5-8 illustrates two examples using vias minimization operations. It shows that the number of vias can be reduced by one or two vias for each of the successful wires. One deficiency of using this vias minimization technique is that we need to use some new tiles. Also, we need two supplementary tiles for connecting the I/O pins of cells with the tiling masks.

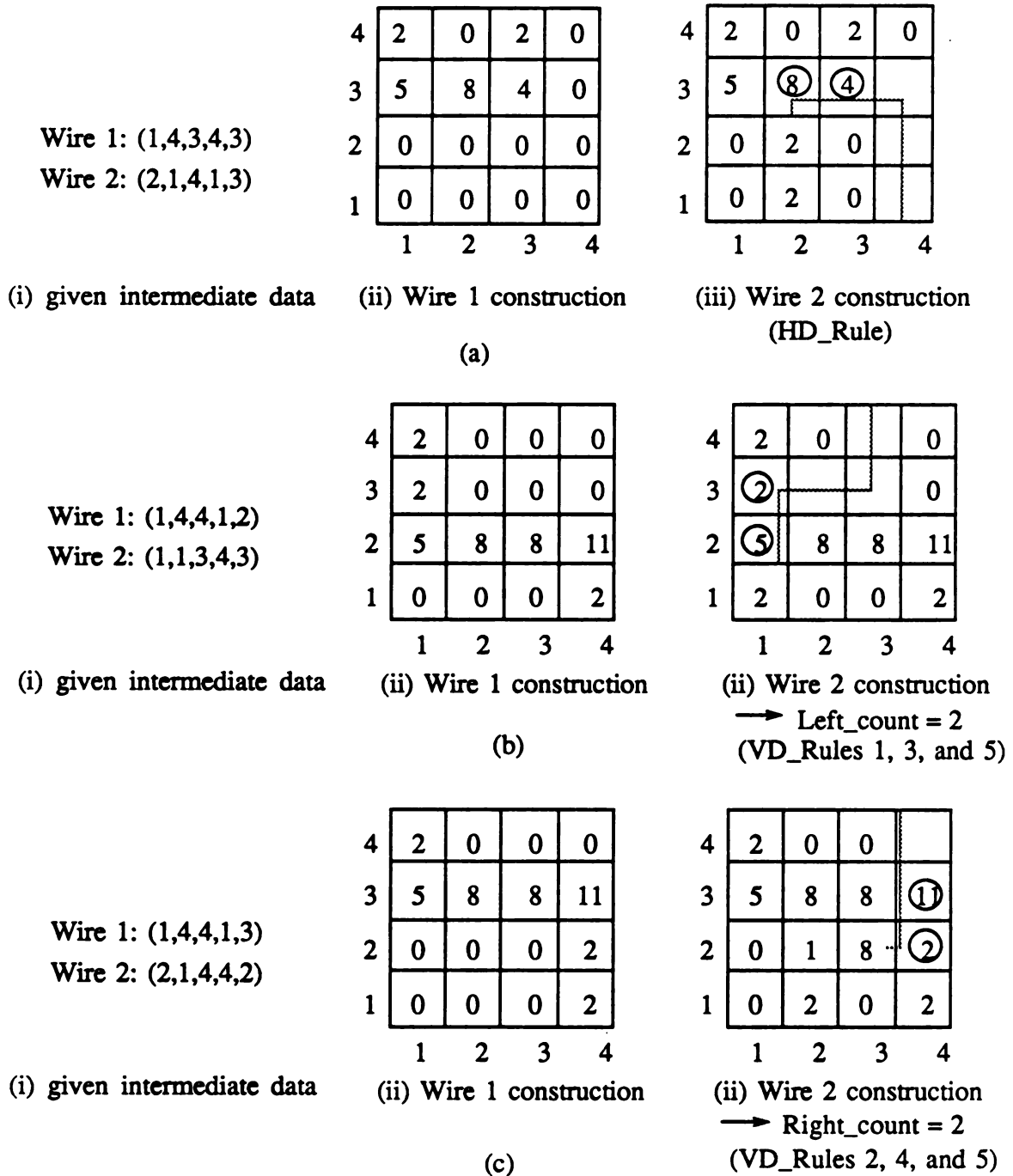
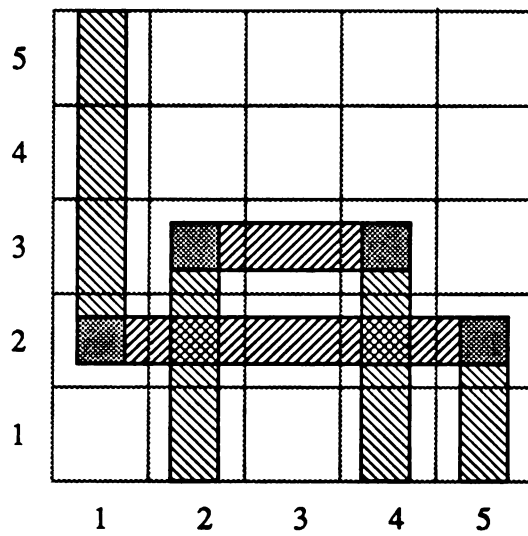
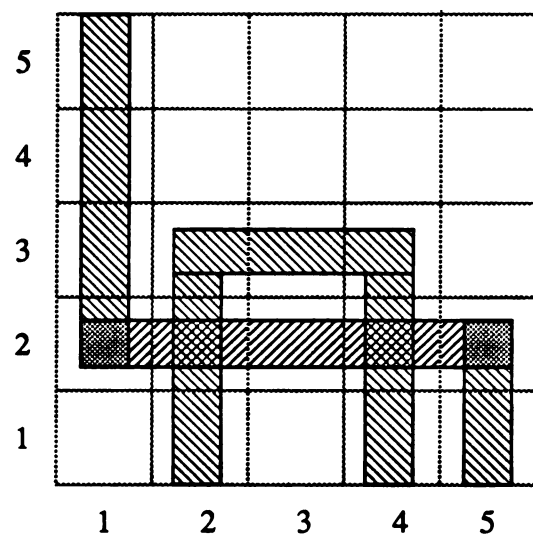


Figure 5-7. Examples of using routing violation detecting rules to detect errors:  
(a) for a horizontal violation; (b) for a left vertical violation;  
(c) for a right vertical violation.

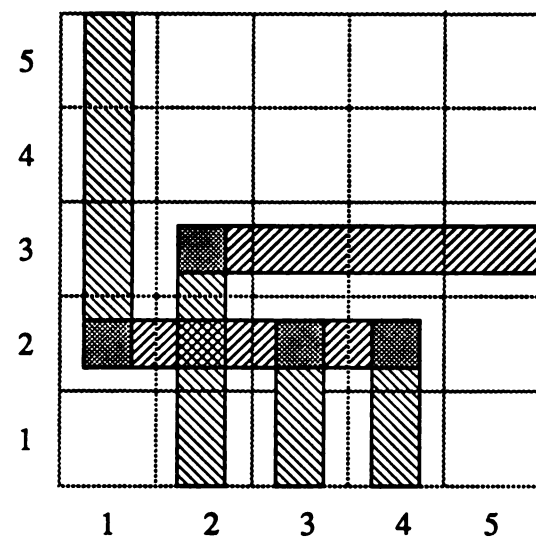


(i) before the vias minimization

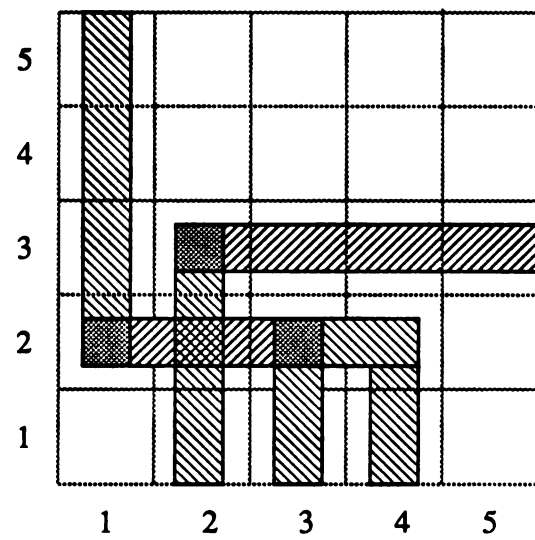


(ii) after the vias minimization

(a)



(i) before the vias minimization



(ii) after the vias minimization

(b)

Figure 5-8. Examples of using the vias minimization: (a) for two vias reductions; (b) for one via reduction.

Figure 5-9 shows these additional tiles with assigned numbers. Also, in Figure 5-10, we list the vias minimization algorithm.

(6) Performance data generation:

The generation of the routing performance data can lead to the selection of acceptable routing processes and results. Two important routing performance parameters are extracted from the constructed tiling array, i.e., total number of vias and total wiring length. These two parameters can be easily detected by summing up the related performance data embedded in all of the tiles in the tiling array. Figure 5-11 is the algorithm used to calculate the total wiring length of the routing result, and Figure 5-12 is the algorithm used to calculate the total number of vias of the routing result.

Finally, in Figure 5-13, we list the processing sequence of all the processes in the CF mask generator. The input of the CF mask generator requires a file with intermediate routing data and the maximum assigned track number. The final output process is used to put all the tiles from the tiling array to a file, and currently this file can be directly accessed by Magic [40]. The program implementing the CF mask generator consists of about 355 lines of C. When applied to the Deutsch difficult example, this program generates the routing masks in about 1.87 seconds CPU time on a VAX 8600 under the UNIX operating system. The distribution of this CPU time for the related processes is also shown in Figure 5-13. The input data used here is generated from a router that will be presented in the next section. Each of these algorithms have the same computational complexity, i.e.  $O(mn)$ , where  $n$  is the number of columns and  $m$  is the number of rows of a given channel. Thus, the computational complexity of the entire CF mask generation algorithm is also bounded by  $O(mn)$ .

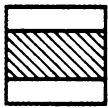


(13)

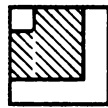


(14)

(a)



(15)



(16)



(17)



(18)



(19)



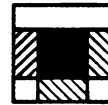
(20)



(21)



(22)



(23)



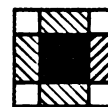
(24)



(25)





(26)



(27)

(b)

Note:  means Metal 1 layer;  means Metal 2 layer;


 means the contact between Metal 1 layer and Metal 2 layer.

Figure 5-9. Additional tiles with assigned numbers: (a) for top and bottom pin connections; (b) for vias minimizations.

```

for (y = 1; y < max_track + 1; y++)
{
    flag = 0;
    for (x = 1; x < max_column + 1; x++)
    {

/* Set conditions to determine the starting point for vias minimizations */

        if ( T[x][y] == 5 || T[x][y] == 1 || T[x][y] == 3 || T[x][y] == 7 ||
            T[x][y] == 9 || T[x][y] == 12)
        {
            st = x;
            flag = 1;
        }
        else if (T[x][y] != 8)
            flag = 0;

/* Check the rest successful tiles in the horizontal direction, and replace the tiles*/

        if ((T[x][y] == 4 || T[x][y] == 11 || T[x][y] == 6) && flag == 1)
        {
            if (T[x][y] == 4) T[x][y] = 16;
            if (T[x][y] == 11) T[x][y] = 17;
            if (T[x][y] == 6) T[x][y] = 18;
            if (T[st][y] == 5) T[st][y] = 19;
            if (T[st][y] == 1) T[st][y] = 20;
            if (T[st][y] == 3) T[st][y] = 21;
            if (T[st][y] == 9) T[st][y] = 22;
            if (T[st][y] == 12) T[st][y] = 23;
            if (T[st][y] == 7) T[st][y] = 24;
            for ( i = st + 1; i < x; i++)
                T[i][y] = 15;
        }
        if ((T[x][y] == 9 || T[x][y] == 12 || T[x][y] == 7) && (flag == 1) &&
            (T[st][y] == 5 || T[st][y] == 1 || T[st][y] == 3))
        {
            if (T[x][y] == 9) T[x][y] = 25;
            if (T[x][y] == 12) T[x][y] = 26;
            if (T[x][y] == 7) T[x][y] = 27;
            if (T[st][y] == 5) T[st][y] = 19;
            if (T[st][y] == 1) T[st][y] = 20;
            if (T[st][y] == 3) T[st][y] = 21;
            for ( i = st + 1; i < x; i++) T[i][y] = 15;
        }
    }
}

```

Figure 5-10. Vias minimization algorithm.



```

/* Initialization */
Total_length = 0;

/* Examine the tiling array to accumulate the total wiring length */
for (x = 1; x < max_column + 1; x++)
{
    for (y = 1; y < max_track + 1; y++)
    {
        /* Add the related wire length for the tiles in the top and the bottom tracks */
        if (y == 1 || y == max_track)
        {
            if (T[x][y] != 8 && T[x][y] != 0)
                Total_length = Total_length + 0.5;
        }

        /* Add the related wire length for the rest of tiles */
        if (T[x][y] == 10)    Total_length = Total_length + 2;
        if (T[x][y] == 7)    Total_length = Total_length + 2;
        if (T[x][y] == 6 || T[x][y] == 3)
            Total_length = Total_length + 1.5;
        if (T[x][y] == 9 || T[x][y] == 12)
            Total_length = Total_length + 1.5;
        if (T[x][y] == 1 || T[x][y] == 4 || T[x][y] == 5 || T[x][y] == 11)
            Total_length = Total_length + 1;
        if (T[x][y] == 2)    Total_length = Total_length + 1;
        if (T[x][y] == 8)    Total_length = Total_length + 1;
    }
}

```

Figure 5-11. The algorithm used to calculate the total wiring length.

```

/* Initialization */
Total_vias = 0;

/* Examine the tiling array to accumulate the number of vias */
for (x = 1; x < max_column + 1; x++)
{
    for (y = 1; y < max_track + 1; y++)
    {
        if (T[x][y] != 8 && T[x][y] != 0 && T[x][y] != 10 && T[x][y] != 2 &&
            T[x][y] != 15 && T[x][y] != 16 && T[x][y] != 17 && T[x][y] != 18 &&
            T[x][y] != 19 && T[x][y] != 21 && T[x][y] != 21)
            Total_vias = Total_vias + 1;
    }
}

```

Figure 5-12. Algorithm used to calculate the number of vias.

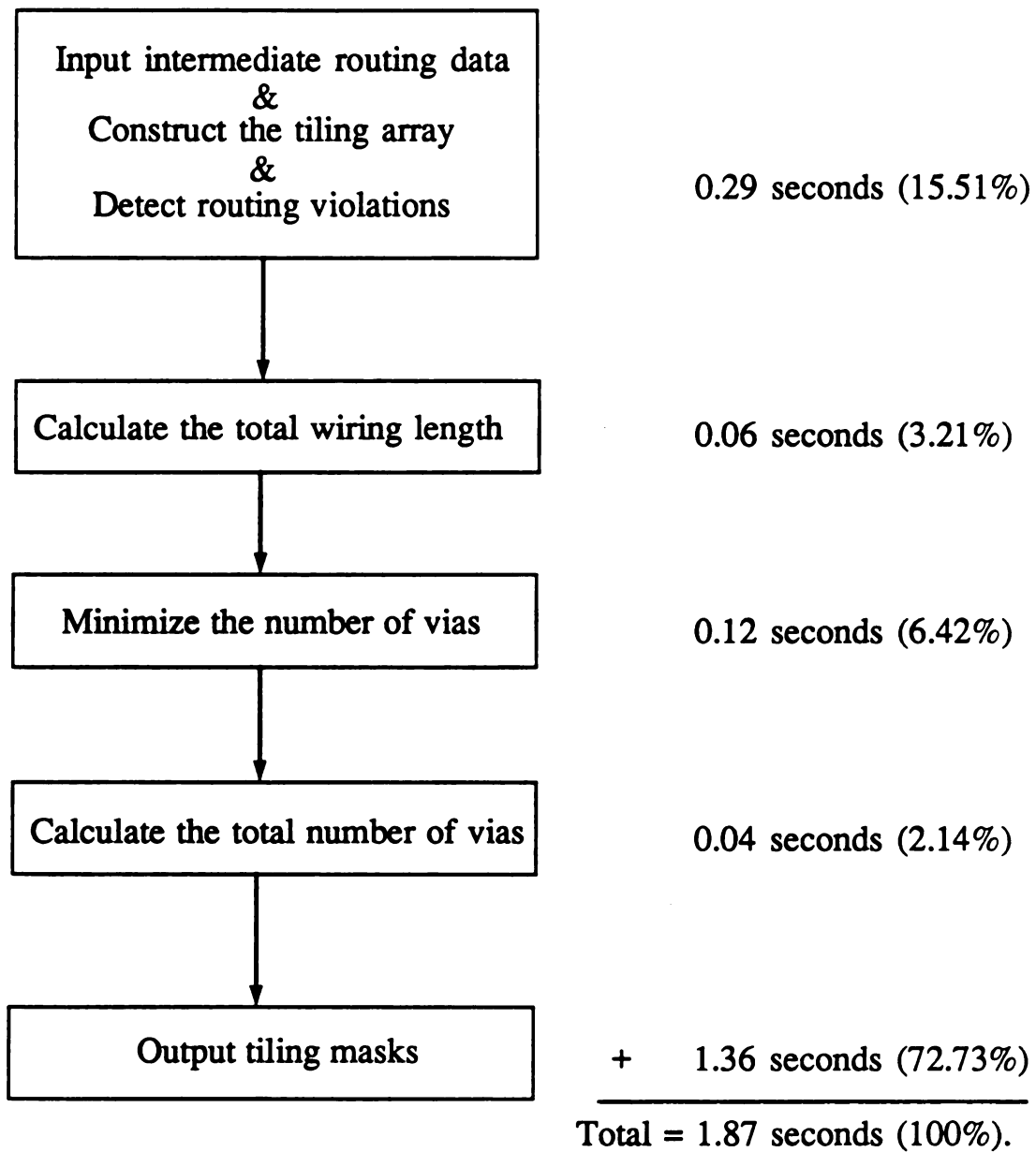


Figure 5-13. Processing sequence of all processes in the CF mask generator.  
(including their realted CPU time results for the Deutsch example)

## 5.2 Three Efficient Routers

In this section, we present the development of three routers for the routing without vertical constraint, routing without dogleg, and routing with dogleg cases, respectively. These routers are used in conjunction with the CF mask generator to generate the routing masks for a given netlist. The notation used throughout this section is the same as that used in Section 4.1.

The input of these routers is a netlist, and the output of the routers are the intermediate routing data and the maximum assigned track number. The number of tracks, the number of vias, and the total wiring length are three important factors in our router development. Because there exists some trade-offs among these factors, each of the routers will emphasize the minimization of one or two factors in the routing result. Also, the router will try to reduce the rest of the factors as small as possible. In addition, by using a special net-weight assignment, all of these routers can easily achieve the mixed-mode routing operation. Finally, we will use the CF mask generator to verify the correctness of the routers and to evaluate the performance of their routing results.

### 5.2.1 The CF\_1 Router

The channel routing problem can be simplified if there exists no vertical constraints in the channel. This without dogleg routing case can be found in a restricted standard-cell approach, such as the cell structure suggested by Jennings [48]. In this standard-cell approach, the region between two neighbor pins of a channel is equal to one wiring space. Then, the channel without vertical constraints is formed by shifting all the bottom cells to the right-hand side with one half of the wiring space. The most important feature of this routing environment is that both the non-cyclic and the cyclic routing cases will be treated equally by the router. In addition, their routing results are always equal to the

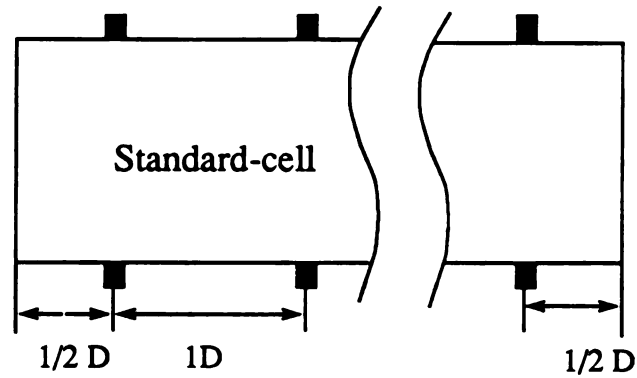
channel densities for the given netlists. For example, Figure 5-14 illustrates the example of the I/O pin assignment for a restricted standard-cell and the routing results for the cyclic and non-cyclic netlists.

The left-edge algorithm [8] is the simplest method used to deal with the routing without vertical constraints. It can generate the number of tracks equal to the channel density for a given netlist. However, the total wiring length of its routing result may not be minimized. For example, Figure 5-15 is a comparison of the left-edge routing result and the optimal routing result for a given netlist. It shows that the total wiring length in the left-edge routing result is much larger than that of the best routing result. Thus, we present an improved router to deal with the routing without vertical constraint, which is known as the CF\_1 channel router. In order to take advantage of this given routing environment, the CF\_1 router does not use any doglegs. Thus, this router tries to minimize the total wiring length after the resulting number of tracks and vias have been minimized.

Basically, the CF\_1 router uses a zone representation technique to identify the outstanding zones for net assignments. It also uses a new net-weight assignment process to reduce the total wiring length and to achieve mixed-mode routing operations. Overall, the CF\_1 router includes the following processes: (1) input the netlist, (2) generate the zone representation, (3) identify the best outstanding zone, (4) assign the nets, (5) output the intermediate routing data.

#### (1) Input the netlist:

The router sequentially reads the original netlist and forms a new net list by inserting two zeros for each pair of input pins. Each pair of input pins is determined by one pin in the top netlist and the related pin in the bottom netlist. For example, the original netlist is shown as



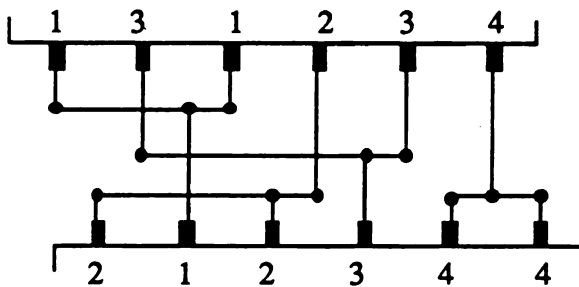
Note that "D" represents the unit of a wiring space;

■ represent I/O pin of the stand-cell.

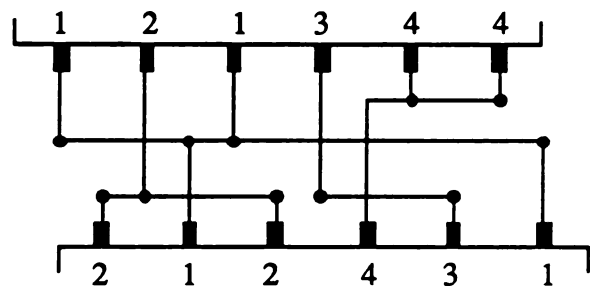
(i)

Top netlist: 1 3 1 2 3 4  
Bottom netlist: 2 1 2 3 4 4

Top netlist: 1 2 1 3 4 4  
Bottom netlist: 2 1 2 4 3 1



(ii)



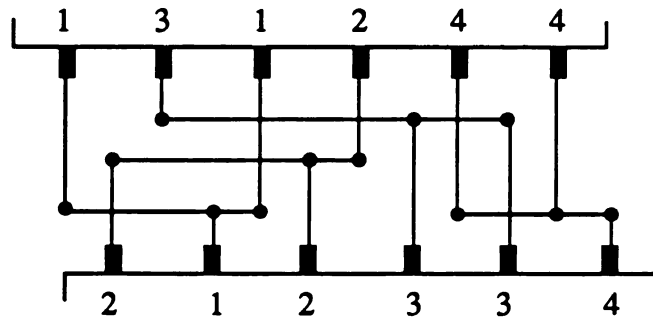
(iii)

Figure 5-14. Illustration of the routing without vertical constraints:

(i) I/O pin assignment for a restricted standard-cell;

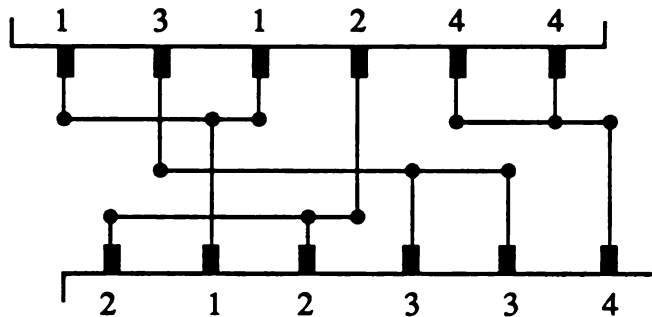
(ii) given a non-cyclic netlist and its routing result;

(iii) given cyclic netlist and its routing result.



Total wiring length =  $9.5 + 27 = 36.5$  D.

(i)



Total wiring length =  $9.5 + 21 = 30.5$  D.

(ii)

Note:

"D" represents one unit of the wiring space in both vertical and horizontal directions.

Figure 5-15. Comparison of two no-dogleg routing results for a given netlist:

(i) left-edge routing result; (ii) optimal routing result.

1 2 3 4 5

2 3 4 5 1.

Then, the new netlist becomes

1 0 2 0 3 0 4 0 5 0

0 2 0 3 0 4 0 5 0 1.

Note that this is a key operation to construct a channel without vertical constraints. In addition, during the netlist input, several important pieces of information are generated. They are the starting column,  $start[i]$ , the ending column,  $end[i]$ , the total number of the top pins,  $n_{top}[i]$ , and the total number of the bottom pins,  $n_{bottom}[i]$ , where  $i$  represents the net number. The  $start[i]$  and  $end[i]$  are used to determine the number of zones covered by net  $i$ , and the  $n_{top}[i]$  and  $n_{bottom}[i]$  are used to determine the weight for net  $i$ ,  $w[i]$ , where

$$w[i] = n_{top}[i] - n_{bottom}[i].$$

Therefore, a positive large value of  $w[i]$  means that net  $i$  should be placed close to the top track in order to reduce its wiring length. For the mixed-mode routing operation, the user-specified net is assigned with a new net-weight. According to the value of  $w[j]$ , we define the new net-weight,  $w'[j]$ , as follows.

$$\text{If } w[j] \geq 0, w'[j] = INF.$$

$$\text{Else, } w'[j] = -INF.$$

Where net  $j$  is the user-specified net, and  $INF = 2 * (\text{the maximum column number in the channel})$ . In other words, the user-specified net will be placed close to the top track if the number of its top pins larger that of its bottom pins. Otherwise, this net will be placed close to the bottom track.

**(2) Generate the zone representation:**

The generation of the zone representation uses the same zone-construction concept as described in Section 4.1. Two operations are involved in this generation, i.e., the zone formation and the zone compaction. In zone formation, each column in the channel represents one zone, and each zone consists of one set of nets which intersect this column. Also, the maximum assigned track number is determined in this operation, and it is equal to the channel density for the given netlist. The zone compaction operation is used to identify the outstanding columns (zones) which consist of an unique set of nets in the channel. All of these zones will be used for net assignment operations later. Also, the zone-weight for each outstanding zone is determined by summing up all of its net-weights.

**(3) Identify the best outstanding zone:**

The zone-weight information of all the outstanding zones is used to identify the best outstanding zone as the first zone for processing in the net assignment. This best outstanding zone is determined by having the highest zone-weight among all of the outstanding zones.

**(4) Assign the nets:**

The net assignment process is used to assign tracks to the nets in each of the outstanding zones. It starts from the best outstanding zone and moving toward both channel ends. In order to identify the net assignment sequence, all the nets in each outstanding zone are sorted according to the values of their net-weights. The higher the value of the net-weight is, the closer the net is placed to the top track. Basically, the CF\_1 router will assign a suitable track to a feasible net at each processing step. The feasible net is a net which has not been assigned a track in the current zone. The suitable track is a track which is used for the net assignment without causing any horizontal violations. The net



assignment process includes two phrases, i.e. the top net assignment phrase and the bottom net assignment phrase. In the top net assignment phrase, the operating sequence to determine the suitable track and the feasible net is from the top to the bottom tracks and from the highest-weight to the lowest-weight nets, respectively. Whereas, it is from the bottom to the top tracks and from the lowest-weight to the highest-weight nets in the bottom net assignment phrase.

(5) Output the intermediate routing data:

The output process is used to construct all the intermediate routing data. Each intermediate routing data is a description of the wiring path of a subnet, which includes the locations of the starting and ending pins and the assigned track number. The track number assigned to a subnet is the same as the track number assigned to its original net. The locations of the starting and ending pins for the subnet can be determined by matching the current processed net number with the numbers in the netlist.

Figure 5-16 lists the CF\_1 routing algorithm and includes all of the processes described above. It shows that the computational complexity of the net assignment process is the highest one among all of the processes in the algorithm. Therefore, the computational complexity of the CF\_1 routing algorithm is bounded by  $O(mNz)$ , where  $m$  is the number of rows,  $N$  is the number of nets and  $z$  is the number of zones in a given channel. The program implementing the CF\_1 router consists of about 500 lines of C. Three netlists from Yoshimura and Kuh [9] and one netlist from Deutsch [45] are used to test this router, which are denoted as the YK\_a, YK\_b, YK\_c and Deutsch net lists. Table 5-1 illustrates the test results of both the CF\_1 router and the Left-edge router for these netlists. Note that the implementation of this left-edge router uses the same I/O processes as in the CF\_1 router. According to this table, the number of vias and tracks resulting from both routers are very close. It also shows that the run time of the left-edge router is faster

```

CF_1 algorithm()

/* Input netlist */
while (!= EOF)
{ insert 0s to the neighbors of the top and bottom pins; /* for no vertical constraints */
  construct net information; }
for all of the nets {construct net-weights; }
/* Form the zones (each column represents one zone) */
processing all the columns in the channel from left to right
{ construct one set of zone-nets, {Z(i)}, which cross the current column i;
  determine the maximum track number; }
/* Compact the zones */
all the columns in the channel from left to right (initially, i = 1 & j = 2)
{ if {Z(i)}  $\subset$  {Z(j)}
  { {Z(i)} = {Z(j)};
    {Z(j)} = {Z(j+1)}; }
  if {Z(i)}  $\supset$  {Z(j)}
  { {Z(j)} = {Z(j+1)}; }
  else
  { accept {Z(i)} as one of the outstanding zones;
    construct the zone-weight of {Z(i)};
    i = j; j = j+1; }
}
/* Find the starting zone location: ZS*/
processing all the outstanding zone
{ find ZS which has the largest zone-weight among all of the outstanding zones; }
/* Assign the nets */
from ZS to the right-end of the channel and from (ZS-1) to the left-end of the channel
{ Sort the nets with their net-weight in the current zone from the largest to the smallest one;
  /* Top net assignment ( for the nets with net-weights  $\geq 0$ ) */
  from the net with the highest net-weight down to the last net with 0 net-weight
  { from 1 to the maximum track number
    assign a suitable track to the current net; }
  /* Bottom net assignment ( for the nets with net-weights  $< 0$ ) */
  from the net with the smallest net-weight up to the last net with 0 net-weight
  { from the maximum track number to 1
    assign a suitable track to the current net; }
}
/* Output intermediate routing data */
processing all the nets
{ examine the netlist from the starting column to the ending column of the current net
  {construct the intermediate routing data for each of the subnets in this net; }
}

```

Figure 5-16. CF\_1 routing algorithm.

Router	Problem (CD)	#Tracks	#Vias	Total wiring length (D)	Run time (sec.)
LF	YK_a (15)	15	117	2655.5	0.1
CF_1		15	115	2449.5	0.3
LF	YK_b (17)	17	122	2928	0.16
CF_1		17	122	2777.5	0.35
LF	YK_c (18)	18	149	3934.5	0.22
CF_1		18	139	3604	0.45
LF	Deutsch (19)	19	285	8018	0.42
CF_1		19	285	7547	1.0

Note that "LF" represents the left-edge routing algorithm [8];  
"CD" represents the channel density of the given netlist;  
"D" represents the wiring length unit.

Table 5-1. Test results of both the left-edge router and the CF\_1 router.

than that of the CF\_1 router. However, the total wiring length results of the CF\_1 router are much less than those of the left-edge router.

### 5.2.2 The CF\_2 Router

For the normal standard-cell approach, the router needs to deal with vertical constraints in the channel. Thus, our second router, which is known as the CF\_2 router, is used for the routing with vertical constraints but without doglegs. The objective of this router is to minimize the maximum output track number after the number of vias has been minimized.

The processes used to construct the CF\_2 router are: (1) input the netlist, (2) generate the net-weights, (3) assign the nets, and (4) output the intermediate routing data. Except for forming the new netlist, the CF\_2 router uses the similar input and output processes as those of the CF\_1 router. In addition, a vertical constraint graph is constructed during the input process, and it is used to determine one of the weighting data,  $len[i]$ . In the graph,  $len[i]$  represents the from-top-level-to-bottom-level count for net  $i$ . Also, another weighting data is  $d[i]$ , which represents the distance between the starting and the ending pins of net  $i$ . The net-weight for net  $i$  is defined as

$$w[i] = d[i] + len[i].$$

Let net  $j$  be a user-specified net in the mixed-mode routing operation, then we define the net-weight for net  $j$  as follows.

$$\text{If } (n_{top}[j] - n_{bottom}[j]) \geq 0, \text{ then } w'[j] = INF.$$

$$\text{Else, } w'[j] = 1,$$

where  $n_{top}[j]$ ,  $n_{bottom}[j]$  and  $INF$  have the same definitions as used in the CF\_1 router.

The qualified nets for each net assignment process are the nets without any descen-

dants in the vertical constraint graph. By using the information of the qualified nets, the CF\_2 router generates several required data for the net assignment process. The number of total points,  $K$ , is equal to twice the number of the qualified nets, and each point is related to the location of the starting pin or ending pin of the nets. These points are sorted according to their related locations from the lowest to the highest. Then, the router assigns a point-weight to each of the points. The point-weight for point  $i$  is denoted as  $ZW[i]$ , where  $ZW(i) = 0$  if point  $i$  is a point related to the location of any starting pins. Else,  $ZW(i) = w[j]$  if point  $i$  is a point related to the location of the ending pin of net  $j$ . Also, the related net number is assigned to a point-net,  $ZN(i)$ . For example, if 2 and 7 are related to the starting and ending pin locations of a qualified net, net 5, and  $w[5] = 15$ . Then,  $ZW(2) = 0$ ,  $ZW(7) = 15$ ,  $ZN(2) = 5$ , and  $ZN(7) = 5$ .

The CF\_2 router uses a net assignment algorithm similar to that of the Yoshimura router [10]. Instead of using the zone representation technique, the CF\_2 router directly uses the related starting and ending pins of the qualified nets to determine the maximum net-weight path. Since any two pins of the qualified nets will not occupy the same location, the construction of the maximum net-weight path becomes straightforward. After processing each net assignment, all the nets in the maximum net-weight path will be assigned with a same track number. Note that the router sequentially produces one new track for each net assignment run from the top track to the bottom track. Then, the CF\_2 router deletes these nets from the vertical constraint graph. Thus, a new set of the qualified nets become available for the next net assignment operation. Figure 5-17 lists this net assignment algorithm. The computational complexity of this algorithm is only bounded by  $O(N)$ , where  $N$  is the number of nets in a given channel.

Overall, the computational complexity of the CF\_2 routing algorithm is bounded by  $O(mN \log_2 N)$ , where  $m$  is the number of rows and  $N$  is the number of nets in a given channel. This is also the computation complexity of the sorting algorithm used in the main net

## Definition:

$K$  : number of total points in the path = 2\*number of qualified nets.  
 $ZW(i)$  : weight at point  $i$ , where  $0 < i < K$ .  
 If  $ZW(i) = 0$ , it means point  $i$  is the starting point of a net;  
 else, it means point  $i$  is the ending point of a net.  
 $ZN(i)$  : point-net which is a net with starting or ending pins located at point  $i$ .  
 $ZP(i)$  : point potential (the largest accumulated weights at point  $i$ ).  
 $ZA(i)$  : current selected net at point  $i$ .  
 $ST(j)$  : starting point of net  $j$ .  
 Temp\_we : current weight.

## Phase 1:

```

{
  ZP(0) = 0;
  ZA(0) = 0;
  for (i = 1; i < K+1; i++)
  {
    if (ZW(i) == 0)
    { ZP(i) = ZP(i-1); ZA(i) = ZA(i-1); }
    else /* find a net in the path */
    { SP = ST(ZN(i)); /* determine the starting point of ZN(i) */
      if (ZW(i) + ZP(SP) > ZP(i-1))
      { ZP(i) = ZW(i) + ZP(SP); /* accumulate weights */
        ZA(i) = ZN(SP); }
      else /* give up this net in the path */
      { ZP(i) = ZP(i-1);
        ZA(i) = ZA(i-1); }
    }
  }
}

```

## Phase 2:

```

{
  Temp_we = ZP(K);
  while (Temp_we != 0)
  {
    for (i = K; i > 0; i--)
    { if (ZP(i) == Temp_we)
      { assign the current track to the selected net, ZA(i);
        delete net ZA(i) from the vertical constraint graph;
        Temp_we = ZP(i) - ZW(i); }
    }
  }
}

```

Figure 5-17. CF<sub>2</sub> net assignment algorithm.

assignment loop. The program implementing the CF\_2 router consists of about 400 lines of code written in the C programming language. The test results of this router for the YK\_a, YK\_b, YK\_c, and Deutsch netlists are listed in Table 5-2. It includes the number of tracks resulting from the Y&K routers and the left-edge router. Table 5-2 also shows that the CF\_2 router can generate the optimal or near optimal routing results in a very fast execution speed. The limitation of using these non-doglegging routers is that it can only be applied to the non-cyclic routing operations. Also, they generate a larger number of tracks than the channel density for the given netlist. For example, for the Deutsch netlist, the CF\_2 router generates 28 tracks which is close to the optimal solution of using the non-doglegging routing; however, the channel density for the Deutsch example is only 19 tracks.

### 5.2.3 The CF\_3 Router

Basically, the use of doglegs can reduce the maximum track number in the routing result. Also, it can reduce the possibility to encounter the cyclic routing problem. For example, the netlist in Figure 5-18 is the cyclic routing case if the non-doglegging router is used to solve for its routing result. However, it becomes a non-cyclic routing case if the doglegging router is used. Thus, in this section, we presents a doglegging channel router which is known as the CF\_3 router.

Because it uses doglegs, the CF\_3 router consists of three principal tasks, i.e., the formation of subnets, the subnet-weight assignment, and the wire compaction. The formation of subnets is used to construct the vertical constraint relationships among all the subnets. The subnet-weight assignment uses a heuristic method to determine all the subnet-weights. The wire compaction process is used as a final refinement operation to reduce the total wiring length for the given channel. Overall, the CF\_3 router consists of

Router	Problem(col., net)	#Tracks	#Vias	Total wiring length (D)	Run time (sec.)
CF_2	YK_a (76, 45)	15*	114	1646.5	0.07
CF_2	YK_b (74, 47)	18	118	1889	0.06
CF_2	YK_c (103, 54)	18*	146	2404	0.10
CF_2	Deutsch (175, 72)	28*	276	6281.5	0.15

Note that "\*" represents using the reversed netlist  
(the top and bottom netlists are switched);  
"D" has the same definition as in Figure 5-15.

(i)

Router	Problem (CD)	#Tracks
CF_2	YK_a (15)	15
Y&K_1		15
Y&K_2		15
LF		18
CF_2	YK_b (17)	18
Y&K_1		17
Y&K_2		17
LF		20
CF_2	YK_c (18)	18
Y&K_1		18
Y&K_2		18
LF		19
CF_2	Deutsch (19)	28
Y&K_1		30
Y&K_2		28
LF		39

Note that "Y&K\_1" represents the first Yoshimura and Kuh routing algorithm [9];  
"Y&K\_2" represents the second Yoshimura and Kuh routing algorithm [9];  
"LF" represents the left-edge routing algorithm [8];  
"CD" represents the channel density of the given netlist.

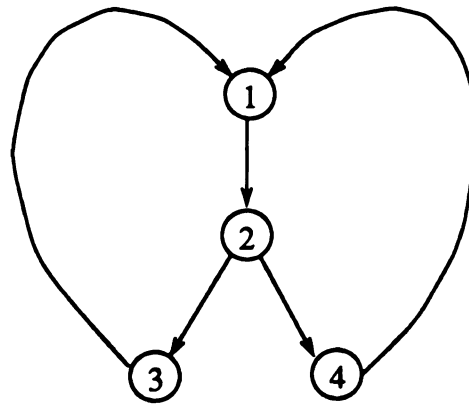
(ii)

Table 5-2. Demonstration of the CF\_2 routing results: (i) results for several netlists;  
(ii) result comparison table.

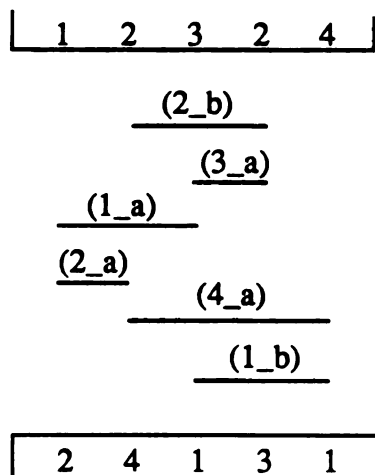


Top netlist : 1 2 3 2 4  
 Bottom netlist : 2 4 1 3 1

(i)



(ii)



(iii)

Note that subnets 1\_a and 1\_b are related to net 1;  
 subnets 2\_a and 2\_b are related to net 2;  
 subnet 3\_a is related to net 3; subnet 4\_a is related to net 4.

Figure 5-18. Illustrate of doglegging operation for solving a given cyclic case:

- (i) given netlist; (ii) cyclic vertical constraint graph for non-doglegging operation;
- (iii) related subnets and non-cyclic vertical constraint graph for doglegging operation.

the following processes: (1) input the net list, (2) generate the local densities, (3) assign the subnet-weights, (4) assign the subnets, (5) the wire compaction, (6) output the intermediate routing data.

(1) Input the netlist:

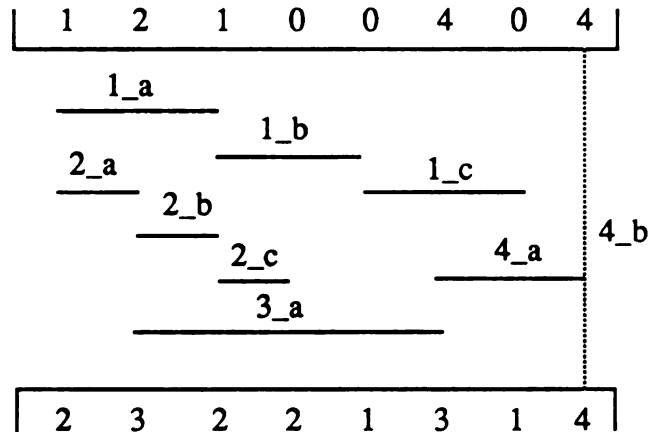
According to the given netlist, this process is used to construct the subnet information. It includes the starting and ending pin locations, and the originated net number for each subnet. Three subnet groups are constructed during the input process, i.e., the upper-subnet group, the middle-subnet group, and the lower-subnet group. A upper-subnet is formed by having two top pins, one top pin and one bottom pin with a zero in the same column, or two bottom pins with both zeros in the columns. A middle-subnet is formed by having one top pin and one bottom, or two bottom pins with a zero in one of the columns. A lower-subnet is formed by two bottom pins without any zeros in their columns of the netlist. Figure 5-19 shows some examples of the upper subnets, the middle-subnets, and the lower-subnets. This formation of the subnets is a realization of the vertical constraint. For example, none of the upper-subnets will have ancestors in the vertical constraint graph, and none of the lower-subnets will have descendants in the vertical constraint graph. All of the middle-subnets represent the intermediate nodes in the vertical constraint graph.

(2) Generate the local densities:

The local density of a subnet is one of the factors used to determine its subnet-weight. We first calculate the number of subnets cross each column of the subnet. Then, the largest number among all of the results is the local density for this subnet. Figure 5-20 lists the algorithm to generate the local densities for all the subnets.

Top netlist : 1 2 1 0 0 1 0  
 Bottom netlist : 2 3 2 2 1 3 1

(i)



(ii)

Upper-subnets : 1\_a, 1\_b, 1\_c, 1\_d, and 4\_a.  
 Middle-subnets : 2\_a, 2\_b, and 2\_c.  
 Lower-subnets : 3\_a.

Note that subnet 4\_b is a straight connection, so it need not be classified.

(iii)

Figure 5-19. Example of the subnet construction: (i) given netlist;  
 (ii) formation of subnets; (iii) classification of the subnets.

```

Local_density_generation()
{
    for ( $i = 1; i < K+1; i++$ )
    {
        /* Select the subnets */
        determine UP, which is the number of upper-subnets cross column  $i$ ;
        put these upper subnets in {UP_N};
        determine MD, which is the number of middle-subnets cross column  $i$ ;
        put these middle subnets in {MD_N};
        determine LO, which is the number of lower-subnets cross column  $i$ ;
        put these lower subnets in {LO_N};
        TL = UP + MD + LO;

        /* Update local densities of the selected subnets */

        for ( $j \in \{UP\_N\} \cup \{MD\_N\} \cup \{LO\_N\}$ )
        {
            if ( $loc[j] < TL$ )
            {
                 $loc[j] = TL$ ;
            }
        }
    }
}

```

Note that  $K$  represents the maximum column number in the channel;

$loc[j]$  represents the local density of subnet  $j$ .

Figure 5-20. Local-density generation algorithm.

(3) Assign the subnet-weights:

The weighting data for each subnet is used to determine the priority of assigning a track to the subnet. Since the dogleg channel routing is a NP-complete problem [43], the CF\_3 router presents a new heuristic approach for the subnet-weight assignment. Let subnet  $i$  be an upper-subnet, then we define its subnet-weight as follows.

$$w_u[i] = 2 * loc[i] + \alpha * dis[i],$$

where  $loc[i]$  represents the local density for subnet  $i$ , and  $dis[i]$  represents the distance between the starting pin and the ending pin for subnet  $i$ . Also,  $\alpha$  is an integer number between 1 and the channel density. Usually, the acceptable result is generated when  $\alpha$  is set about one half of the channel density. Let subnet  $j$  be a middle-subnet or a lower-subnet, then we define its subnet-weight as follows.

$$w_{m,l}[j] = loc[j] + \alpha * dis[j].$$

For the mixed-mode routing operation, we define the new subnet-weight as follows.

If subnet  $k$  is a subnet in the upper-subnet group, then  $w_u[k] = INF$ .

Else, if subnet  $k$  is a subnet in the lower-subnet group, then  $w_{m,l}'[k] = 1$ .

Note that subnet  $k$  is a user-specified net, and  $INF = (\alpha + 2 * (\text{channel density})) * (\text{the maximum column number in the channel})$ . Thus, each of subnets in the user-specified net has a highest priority to be selected in the subnet assignments.

(4) Assign the subnets:

The subnet assignment algorithm is similar to the net assignment algorithm as used in the CF\_2 router. However, the CF\_3 router uses the qualified subnets instead of the qualified nets, and all the subnets in the upper-subnet group are the qualified subnets. The result of this subnet assignment is a set of accepted subnets with a highest accumulated weight among all of the possible net-weight combinations. These subnets are assigned

with a current available track number. Then, the router deletes these accepted subnets from the upper-subnet group and rearranges the related subnets in the subnet groups. Each related subnet is a subnet that has at least one pin located at the same column as that of one of the accepted subnets. These related subnets will be moved from the middle-subnet group to the upper-subnet group, or from the lower-subnet group to the middle-subnet group. The subnet assignment process will be continued until no more subnets is left in the upper-subnet group. Finally, if there still exists the unassigned subnets in either the middle-subnet group or the lower-subnet group, then it means that the given netlist has a cyclic routing problem. We will discuss this problem in the next chapter.

(5) Wire compaction:

If the CF\_3 router only uses this row-by-row subnet assignment, then its total wiring length result may not be reduced. Therefore, a wire compaction process is used to minimize the total wiring length result. This process tries to push all of the lower-subnets as close to the bottom track as possible without overlapping any horizontal wires. Note that this bottom track number is resulting from the previous subnet assignment process. Figure 5-21 lists this wire compaction algorithm.

(6) Output the intermediate routing data:

During the input process, the information of all the subnets have been stored. Also, the assigned track number for each subnet is determined after we execute the wire compaction process. Thus, the intermediate routing data can be easily constructed by combining these given subnet information.

The CF\_3 router and Yoshimura efficient router [10] are similar; however, the CF\_3 router uses a much simpler algorithm to assign subnet-weights and subnets than those of the Yoshimura router. The computational complexity of the CF\_3 routing algorithm is

```

Wire compaction()
{
    for (i = max_track; i > 1; i--)
    {
        for (j = i-1; j > 0; j--)
        {
            for (x ∈ {T_LO_N[j]});
            {
                if (put x in {T_TL_N[i]} that will not generate any horizontal violations)
                {
                    delete x from {T_LO_N[j]};
                    insert x to {T_TL_N[i]};
                }
            }
        }
    }
}

```

Final intermediate routing data are in {T\_TL\_N[i]}, where i = 1 to max\_track.

Note:

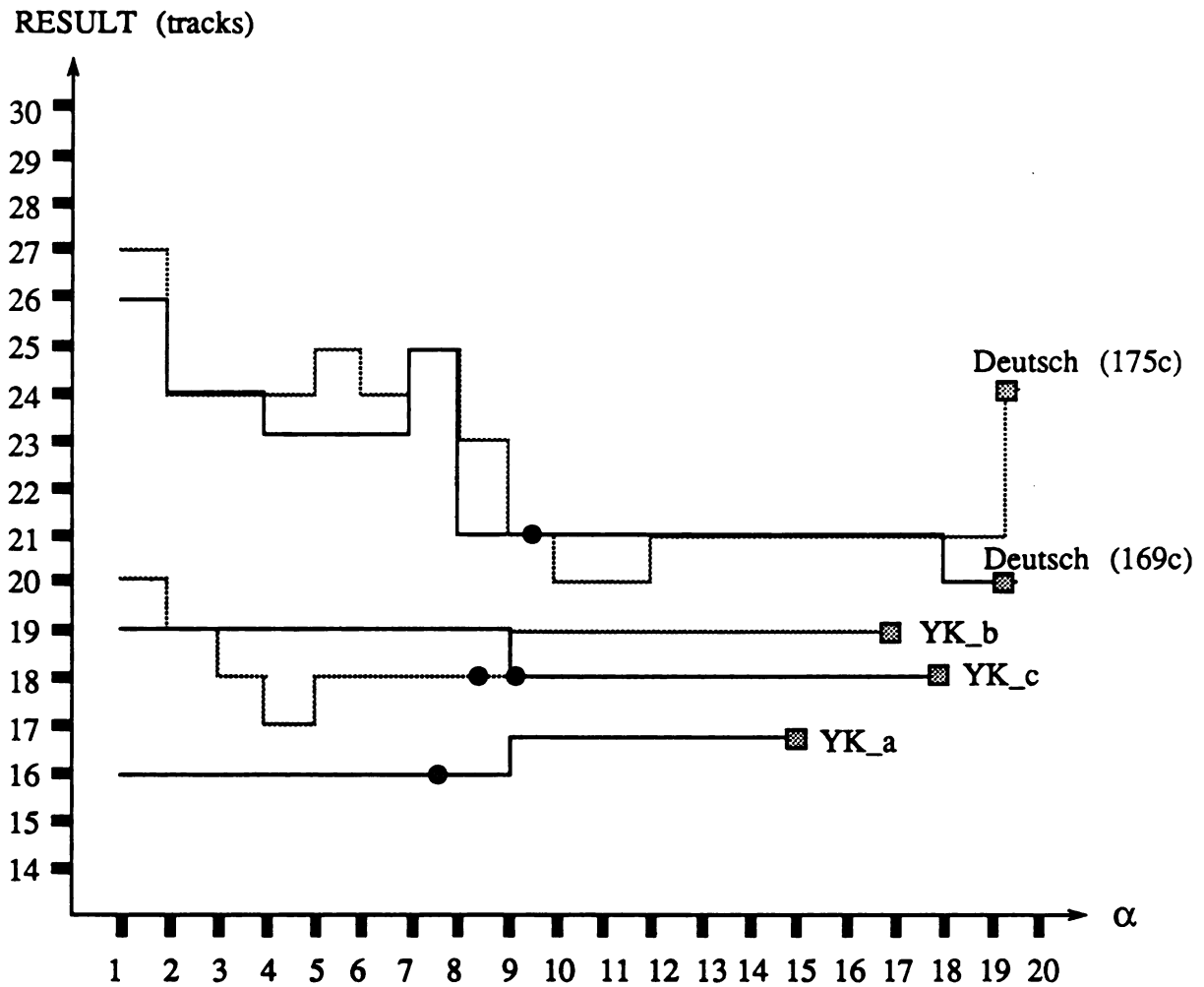
Max\_track represents the maximum track number resulting from the previous processes;  
 {T\_LO\_N[i]} represents a set of lower subnets with the same assigned track number;  
 {T\_MD\_N[i]} represents a set of middle subnets with the same assigned track number;  
 {T\_UP\_N[i]} represents a set of upper subnets with the same assigned track number;  
 $\{T\_TL\_N[i]\} = \{T\_UP\_N[i]\} \cup \{T\_MD\_N[i]\} \cup \{T\_LO\_N[i]\}.$

Figure 5-21. Wire compaction algorithm.

similar to that of the CF\_2 routing algorithm. However, instead of using the number of nets, the CF\_3 router uses the total number of subnets in a given channel. The computational complexity of the CF\_3 routing algorithm is bounded by  $O(mM \log_2 M)$ , where  $m$  is the number of rows and  $M$  is the total number of subnets in the channel. The program implementing the CF\_3 router consists of about 750 lines of C. For several routing examples, Figure 5-22 illustrates the CF\_3 routing results corresponding to different values of  $\alpha$ . If we define the acceptable track result is about 110% of the channel density for the given netlist, then the value of  $\alpha$  is close to one half of the channel density. The best test results of this router for the YK\_a, YK\_b, YK\_c, and Deutsch netlists are listed in Table 5-3. Also, Table 5-3 lists the result comparisons of the CF\_3 router and other existing routers for the same the YK\_c and Deutsch netlists. It shows that the CF\_3 router generates a little more vias than those of the routers; however, the CF\_3 router can produce much less total wiring length than those of most of the routers. Figure 5-23 is the mask result for the 169-column Deutsch example routed by the CF\_3 router. The total CPU time is about 3.2 seconds on a VAX 8600 under the UNIX operating system, which includes the execution time for both the CF\_3 router and the CF mask generator.

In this chapter, we have presented three efficient routers and a powerful mask generator. These routers can be used for mixed-mode routing operations. The CF\_1 router is used to deal with channel routing problems in the restricted standard-cell approach. It can generate the minimum number of tracks and vias by using the without doglegging and without vertical constraint operations. The total wiring length of the routing result is reduced because the CF\_1 router used an effective weight assignment process. Both the CF\_2 router and the CF\_3 router can be used to deal with channel routing problems in the normal standard-cell approach. The execution speed of the CF\_2 router is the fastest one among all of these three routers. The CF\_2 router can generate the minimum number of vias by using a without doglegging operation. Also, It utilized an improved net-





Note:

- represents the place where  $\alpha \equiv 1/2 * \text{the channel density of the given netlist}$ ;
  - represents the place where  $\alpha = \text{the channel density of the given netlist}$ .
- $\alpha$  is a user-defined factor in the weighting function.

Figure 5-22. Sensitivity of  $\alpha$  to the CF<sub>3</sub> routing results.

Router	Problem(col., net)	#Tracks	#Vias	Total wiring length (D)	CPU time (sec.)
CF_3	YK_a (76, 45)	16	129	1594	0.25
CF_3	YK_b (74, 47)	17	135	1870.5	0.24
CF_3	YK_c (103, 54)	18	164	2423.5	0.38
CF_3	Deutsch (175, 72)	20	350	5233.5	1.35
CF_3	Deutsch (169,72)	20	335	4924	1.33

Note that "D" has the same definition as shown in Figure 5-15.

(i)

Router	Problem(col., net)	#Track	#Vias	Total wiring length (D)
CF_3	YK_c (103, 54)	18	164	2423.5
YACR2		18 *	152	2448

Note: "\*" means that one more track was required for the automatic execution.

Router	Problem(col., net)	#Track	#Vias	Total wiring length (D)
CF_3	Deutsch (169,72)	20	335	4924
Y&K [9]		20	308	5075
Hierarchical [46]		19	354	5023
YACR2 [11]		19	287	5020
Mighty [12]		19	301	4812

Note that Hierarchical, YACR2, and Mighty routers use the unrestricted doglegging; YACR2 and Mighty use the wrong way routing approach.

(ii)

Table 5-3. Demonstration of the CF\_3 routing results: (i) best results of the CF\_3 router; (ii) two result comparison tables.

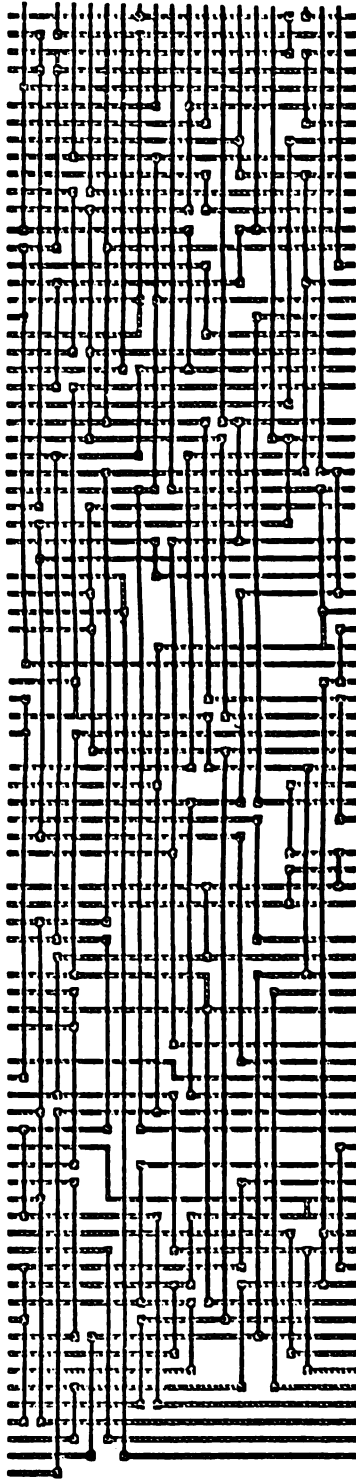


Figure 5-23. Mask result of the 169-column Deutsch example routed by the CF\_3 router.

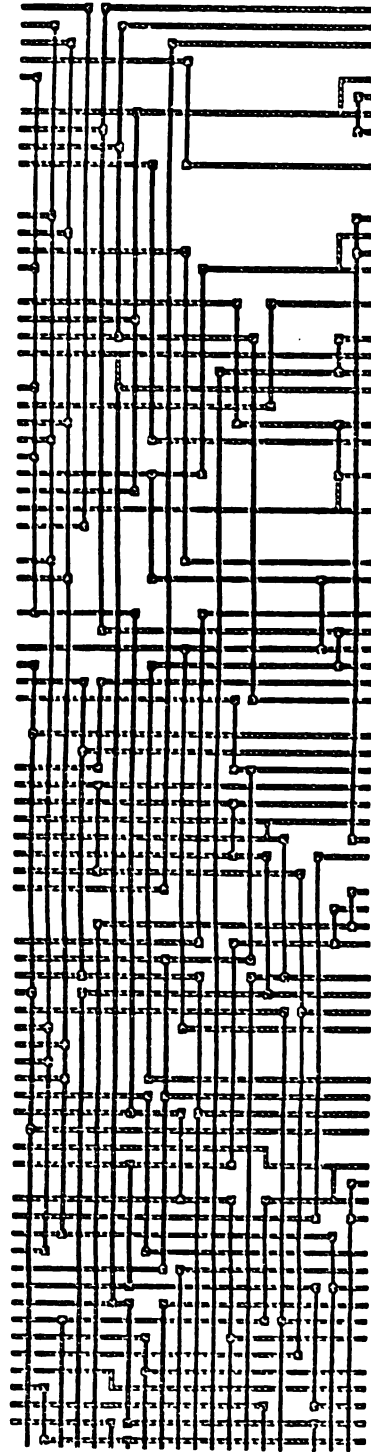


Figure 5-23. (continued)

assignment algorithm to reduce both the number of tracks and the total wiring length results. Basically, the CF\_3 router is a restricted doglegging router, and it used several new processes in the routing operation, i.e., the formation of the subnet groups, the assignment of subnet-weights, and the compaction of wires. By selecting a good parameter in the subnet-weight assignment, CF\_3 router can generate much less tracks and total wiring length results than those of the CF\_2 router and the CF\_1 router. However, the CF\_3 router will generate more vias than those of these two routers. From several routing result comparisons, we see that all three routers can efficiently generate optimal or near optimal results, respectively.

The CF mask generator used a tiling technique to efficiently generate the final routing masks for the given intermediate routing data. In addition to producing the routing performance data, it also used several error-detection and via-minimization rules to evaluate and improve the routing result of each router. All of three routers have been tested by the CF mask generator on several well-known routing examples including a benchmark example. As a result, these routers are proven to be correct.

Possible extensions of our routing approach will be discussed in the next chapter. These extensions involve methods to deal with the cyclic vertical constraint routing, irregular-channel routing, variable wire-widths routing, and mask output reduction.

## CHAPTER 6

### SUMMARY AND CONCLUSIONS

Two VLSI design related areas were investigated in this research: the mapping of algorithms to architectures and the transforming of netlists into physical layouts. Our goal is to apply VLSI technology to circuit design in order to efficiently increase product execution speed and reduce product area.

A computer graphics drawing engine was chosen for our algorithm mapping investigation because VLSI technology has the potential to greatly increase the computer drawing ability for advanced computer graphics systems. The objective here was to develop a high-performance antialiased drawing engine. To achieve this objective, we developed an efficient drawing algorithm and mapped it to an architecture that is well suited for VLSI implementation. In addition, we estimated the performance of this antialiased drawing engine in order to provide a specific performance index for the designer.

In the VLSI automation tools investigation, we focused on the regularity issue of the VLSI physical implementation because it can lead to highly testable and compact integrated circuits. Based on this requirement, the designer usually selects the standard-cell and/or gate-array implementation. Without considering the floor-plan and cell placement problems in the implementation, we investigated the integration of embedded channel routing and mask generation. Thus, the objective of this investigation was to develop an efficient channel routing approach to facilitate the routing process for a given netlist. This approach can not only be used to implement our drawing engine but also can be applied to most channel routing applications.

## 6.1 Summary

For developing a graphics drawing engine, we evaluated existing line-drawing and antialiasing algorithms to determine the fastest drawing techniques which possessed antialiasing capabilities. The non-parametric line-drawing technique is suitable for use in the hardware implementation because it provides a faster execution speed than that of the parametric one. A typical algorithm, which uses this technique, is the Bresenham line-drawing algorithm. With some modifications of the Bresenham line-drawing algorithm, an efficient and flexible line-drawing algorithm was developed. This algorithm uses several simplified operating parameters to rapidly generate the decision results for a given line. In addition, it can easily be used in conjunction with an antialiasing algorithm for generating realistic lines and curves.

The area-antialiasing technique can eliminate the aliasing appearance of the drawings on a raster display more quickly than other antialiasing techniques. Based on a square pixel-shape and uniform distribution of the pixel-intensity assumption for screen pixels, we established an exact area-antialiasing model. It defines the relationship between the values of the decision parameter and the output intensities. To increase the execution speed, the CFO antialiasing algorithm was developed by using a simplified area-antialiasing model. As a result, the maximum error of its intensity result is only 3.125%. The CF antialiased drawing algorithm was constructed by combining the line-drawing algorithm and the CFO antialiasing algorithm. According to the results from several comparisons and demonstrations described in Chapter 3, this algorithm is the best because of its execution speed, operational flexibility, and realism.

Based on the data flow of the CF algorithm, we implemented an antialiased line-drawing architecture, i.e., the CF drawing engine. This drawing engine consists of both pipeline and parallel processing features, and it generates a realized line with adjustable

line-widths and intensity-levels. According to the resulting block diagram of the CF drawing engine, each block can be viewed as a cell which can be a custom-designed, a standard-cell, or a gate. This simplifies the process for physically implementing the CF drawing engine.

To estimate the performance of this design, we developed a cell-delay model for all of the constructed cells assuming a 3- $\mu$ m CMOS technology. These constructed cells were used in a prototype line-drawing engine implementation; they can also be applied to implement the CF drawing engine. The objective of this line-drawing engine implementation is to set up an accurate delay data for all of the cells. Furthermore, the interconnection wire delay was determined by extracting the loading data from the actual circuit layout. Therefore, by using all of the cell-delay and the interconnection loading data, we were able to estimate the performance of the circuit. The timing delay of the CF drawing engine was found to be close to that of the line-drawing engine, i.e., the critical path delay is about 80 nanoseconds. Thus, the throughput of the CF drawing engine can be  $12.5P$ -M pixels per second, where  $P$  is the number of pixels to be assigned with intensities at each drawing step.

With respect to channel routing problems, we evaluated existing routing and mask generation algorithms to identify the useful routing algorithms. As a result, the left-edge routing algorithm [8] and Yoshimura routing algorithm [10] were selected as good candidates in dealing with channel routing problems. We also constructed a systematic method for developing several routers as well as a mask generator, i.e., the CF routing approach. First, a clear routing interface was defined between the routers and the mask generator thereby providing an entry point for mixed-mode routing operations. This interface includes a set of intermediate routing data, and each data represents a wiring path. The CF mask generator was developed to produce the correct mask result for a given set of intermediate routing data. By utilizing a tiling approach and several design rules, the CF

mask generator consists of several efficient processes, i.e., a routing error detection process, a via minimization process, and a routing performance data generator. The use of these processes can lead us to produce a simple and highly testable routing mask result. Overall, the computational complexity of the CF mask generator is only bounded by the area of a given channel, i.e.,  $O(mn)$ , where  $m$  represents the number of rows and  $n$  represents the number of columns in the channel.

To deal with different routing cases, we developed three efficient routers, i.e., the CF\_1 router, the CF\_2 router, and the CF\_3 router. The CF\_1 router is used in the restricted standard-cell environment, where the routing is done without vertical constraints. The CF\_2 router is used in the normal standard-cell environment where the routing is done with vertical constraints but without doglegs. The CF\_3 router is used in the normal standard-cell (or gate-array) environment where the routing is done with both vertical constraints and doglegs.

Since we adopted a heuristic approach for developing these routing algorithms, several weighting functions are used to determine their best net assignment sequences. In addition to using the intermediate routing data, the use of this net-weight assignment method gives us another entry point for mixed-mode routing operations. Several efficient processes were included in these routers, i.e., the modified left-edge process, the new net-weight assignment process, the modified net assignment process, and the wire-compaction process. These are used to improve the routing speed, number of vias, number of tracks, and total wiring length results.

In Table 6-1, we summarize the routing results of these routers for a benchmark example and list the computational complexities of their routing algorithms. The CF\_1 router can generate the smallest number of tracks and a relatively small number of vias, and this router can guarantee a 100% routing completion rate. However, because the CF\_1 router uses a without vertical constraint approach, its total wiring length result is



Router	Problem (CD)	#Tracks	#Vias	Total wire length (D)	Run time (seconds)	Algorithmic complexity
CF_1	Deutsch (19)	19	285	7547	1.0	$O(mNz)$
CF_2	Deutsch (19)	28	276	6281.5	0.15	$O(mN\log_2 N)$
CF_3	Deutsch (19)	20	350	5233.5	1.35	$O(mM\log_2 M)$

Note that "CD" represents the channel density of the given net list;

"D" represents the wiring length unit;

$m$  represents the number of rows in a channel;

$N$  represents the number of nets in a channel;

$z$  represents the number of zones in a channel;

$M$  represents the number of subnets in a channel.

Table 6-1. Result comparison of the CF\_1 router, the CF\_2 router,  
and the CF\_3 router .

the largest among those of the routers. Due to using the without dogleg approach, the CF\_2 router can generate a minimum number of vias for normal standard-cell routing operations. The CF\_2 router is also the fastest router among all of these routers. But, its total wiring length result is larger than that of the CF\_3 router. The CF\_3 router can generate the smallest wiring length result; however, its vias result is larger than that of the CF\_2 router. Both the CF\_2 router and the CF\_3 router have a 100% success rate for non-cyclic routings. However, the routing completion rate of the CF\_3 router is much higher than that of the CF\_2 router for the given random netlists. This is because the CF\_3 router uses a restricted doglegging operation to avoid most of the cyclic routing problems. Overall, according to several routing result comparisons from Chapter 5, the CF routing approach is the most effective in dealing with channel routing problems on the point of trade-off between the execution speed and performance.

To conclude, the CF drawing algorithm and architecture provide the quick, realism and flexible features in generating computer graphics drawings. Also, the method to map its algorithm to an architecture is very useful for use in VLSI applications. Furthermore, the CF routing approach provides the designer an efficient and flexible way to deal with two-layer channel routing and mask generating problems in VLSI design.

## 6.2 Future Research and Development

In the computer graphics drawing area, we developed a high performance antialiased drawing engine to improve the performance of the traditional graphics system. Other processing units may become the new bottle-necks of the computer graphics system. Thus, further research should be focused on utilizing the VLSI technology to increase the performance of the frame buffer and/or high-level graphics drawing facilities.

In the area of VLSI design automation, future research should be focused on several possible extensions of the CF routing approach. To achieve unrestricted doglegging operations, we may utilize the unassigned pins and possible empty spaces in a channel. This routing strategy can be combined in the CF routing approach to solve for cyclic routing cases. By using a suitable weighting data in the constructed irregular channel, one can easily apply the CF routing approach to the irregular-channel routing. To provide the adjustable wire-widths routing, one needs to modify both the format of intermediate routing data and the CF mask generator. Based on these modifications, the mask generator can easily adjust the wire-widths inside of the tiles in a tiling array. Also, the mask output process (see Fig. 5-13) occupies a high percentage of CPU time (72.73%) of the entire CF mask generation operation. This can be reduced by using the collection of the masks for all of the horizontal wires, vertical wires and contacts in the tiling array. In addition to reducing the CPU time of mask generation, this entire operation can also minimize the size of the memory occupied by the resulting masks.

In Chapter 3, we proposed a timing estimation method which required a rough layout of the circuit. Thus, future research should also be focused on utilizing this method to produce an efficient timing estimator for a given VLSI design. Simple and accurate models for both the interconnection loading data and the cell timing delay data should be the main focus of this research.

## LIST OF REFERENCES

- [1] Brodlie, K. W., "A Review of Methods for Curve and Function Drawing," *Mathematical Methods in Computer Graphics and Design*, Academic Press, 1980, pp. 1-37.
- [2] Jordan, B. W., W. J. Lennon, B. C. Holm, "An Improved Algorithm for Generation of Non-Parametric Curves," *IEEE Trans. on Computers*, Vol. C-22, No. 12, December 1973, pp. 1052-1060.
- [3] Crow, F. C., "A Comparison of Antialiasing Techniques," *IEEE Computer Graphics and Applications*, Vol. 1, 1981, pp. 40-47.
- [4] Pitteway, M. L. V. and Watkinson, D. J., "Bresenham's Algorithm with Grey Scale," *Communication of ACM*, Vol. 23, No. 11, Nov. 1980, pp. 625-626.
- [5] Gupta, S. and Sproull, R. F., "Filtering Edges for Gray-Scale Displays," *Communication of ACM*, Vol. 15, No. 3, August 1981, pp. 1-5.
- [6] Fujimoto, A. and Iwata, K., "Jag-Free Images on Raster Displays," *IEEE Computer Graphics and Applications*, Dec. 1983, pp. 26-34.
- [7] Turkowski, K., "Anti-Aliasing through the Use of Coordinate Transformations," *ACM Trans. on Graphics*, Vol. 1, No. 3, July 1982, pp. 215-234.
- [8] Hashimoto, A. and Stevens, J., "Wire Routing by Optimizing Channel Assignment within Large Apertures," *Proc. 8th Design Automation Workshop*, 1971, pp. 155-169.
- [9] Yoshimura, T. and Kuh, E. S., "Efficient Algorithms for Channel Routing," *IEEE Trans. on CAD of Integrated Circuits and Systems*, Vol. CAD-1, No. 1, Jan. 1982, pp. 25-35.
- [10] Yoshimura, T., "An efficient Channel Router," *Proc. 21st Design Automation Conf.*, 1984, pp. 38-44.
- [11] Reed, J., Sangiovanni-Vincentelli, A. and Santomuro, M., "A New Symbolic Channel Router: YACR2," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-4, No. 3, July, 1985, pp. 208-219.
- [12] Shin, H. and Sangiovanni-Vincentelli, A., "A Detailed Router Based on Incremental Routing Modifications: Mighty," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-6, No. 6, November, 1985, pp. 942-955.
- [13] Larsen, R. P., "Versatile Mask Generation Techniques for Custom Microelectronic Devices," *Proc. 15th Design Automation Conf.*, 1978, pp. 193-198.

- [14] Rogers, C. D., "MCNC's Vertically Integrated Symbolic Design System," *Proc. 22nd Design Automation Conf.*, 1985, pp. 62-68.
- [15] Sutherland, I. E., SKETCHPAD: A Man-Machine Graphical Communication System, *SJCC 1963*, Spartan Books, Baltimore, Md. pp. 329.
- [16] DePalma, G., Olson, M. and Jollis, R., "Dedicated VLSI Chip Lightens Graphics Display Design Load," *Electronic Design*, January 20, 1983, pp. 415-422.
- [17] Shires, G., "A New VLSI Graphics Coprocessor - The Intel 82786," *IEEE Computer Graphics and Applications*, October, 1986, pp. 49-55.
- [18] Asal, M., Short, G., Preston, T., Simpson, R., Roskell, D., and Gutttag, K., "The Texas Instruments 34010 Graphics System Processor," *IEEE Computer Graphics and Applications*, October, 1986, pp. 24-39.
- [19] Tkedo, T., "High-Speed Techniques for a 3D Color Graphics Terminal," *IEEE Computer Graphics and Applications*, Vol. 4, no. 5, 1984, pp. 46-58.
- [20] Megatek Corporation, "Megatek 7200 Graphics Engine," *San Diego, CA.*, 1981.
- [21] Sizer, T. H. R., *The Digital Differential Analyzer*, London: Chapman and Hall, 1968.
- [22] Bresenham, J. E., "Algorithm for Computer Control of Digital Plotter," *IBM syst. J.* 4 (1), 1965, pp. 25-30.
- [23] Bresenham, J. E., "Incremental Line Compaction," *The Computer Journal*, Vol. 25, No. 1, 1982, pp. 116-120.
- [24] Sproull, R. F., "Using Program Transformations to Derive Line-Drawing Algorithms," *ACM Trans. on Graphics*, Vol. 1, No. 4, October 1982, pp. 259-273.
- [25] Wu, X. and Rokne, J. G., "Double-Step Incremental Generation of Lines and Circles," *Computer Vision, Graphics and Image Processing*, Vol. 37, 1987, pp. 331-344.
- [26] Roger, D. F., *Procedural Elements for Computer Graphics*, McGraw-Hill Book Company, 1985, pp. 30-31.
- [27] Foley, J. D. and Van Dam, A., *Fundamental of Interactive Computer Graphics*, Addison-Wesley Publishing Company, 1982, pp. 433-436.
- [28] Bresenham, J. E., "A linear Algorithm for Incremental Digital Display of Circle Arcs," *Communications of ACM*, Vol. 20, No. 2, February 1977, pp. 100-106.
- [29] Van Aken, Jerry and Novak, M., "Curve-Drawing Algorithms for Raster Displays," *ACM Trans. on Graphics*, Vol. 4, No. 2, April 1985, pp. 147-169.

- [30] Dewey, B. R., *Computer Graphics for Engineers*, Harper & Row, Publishers, Inc., 1988, pp. 135-143.
- [31] Harrington, S., *Computer Graphics: A Programming Approach*, McGraw-Hill Book Company, 1987, pp. 397-424.
- [32] Bezier, P., "Numerical Control - Mathematics and Applications," *John Wiley and Sons, London*, 1972.
- [33] Crow, F. C., "The Aliasing Problem in Computer-Generated Shaded Images," *Communications of ACM*, Vol. 20, No. 11, Nov. 1977, pp. 799-805.
- [34] Conrac Corporation, *Raster Graphics Handbook*, Conrac Division, Covina, CA., 1980, pp. 9-6 to 9-9.
- [35] Pitteway, M. L. V., "On Filtering Edges for Grey-Scale Displays," *Computer Graphics*, Vol. 15, No. 4, December 1981, pp. 322-326.
- [36] Chen, Y., Fisher, P. D. and Olinger, M. D., "The Application of Area Antialiasing on Raster Image Displays," *Graphics Interface '88*, June 1988, pp. 211-216.
- [37] Thacker, W. E. and Smith, R. P., "Chip Simulation Is All a Matter of Image," *ESD: The Electronic System Design Magazine*, November 1988, pp. 65-70.
- [38] Heinbuch, D. V., "CMOS3 Cell Library," *Addison-Wesley Publishing Company*, 1988.
- [39] Ousterhout, J. K., "A Switch-Level Timing Verifier for Digital MOS VLSI," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-4, No. 3, July 1985, pp. 336-349.
- [40] Scott, W. S., et al., "1985 VLSI Tools: More Works by the Original Artists," *Computer Science Division, EECS Department, U.C. Berkeley*, March 1985.
- [41] Hilevel Technology, Inc., "TOPAZ Series User's Manual," CA., 1987.
- [42] Lee, C. Y., "An Algorithm for Path Connection and Its Applications," *IRE Trans. Electron. Comput.*, Vol. EC-10, 1961, pp. 346-365.
- [43] Szymanski, T. G., "Dogleg channel routing is NP-complete," *IEEE Trans. Computer-Aided Design*, Vol. CAD-4, 1985, pp. 31-41.
- [44] Persky, G., Deutsch, D., and Schweikert, D. G., "LTX - A Minicomputer-based system for automatic LSI layout," *J. Design Automation and Fault Tolerant Computing*, Vol. 1, No. 3, May, 1977, pp. 217-255.
- [45] Deutsch, D., "A Dogleg Channel Router," *Proc. 13th Design Automation Conf.*, 1976, pp. 425-433.

- [46] Rivest, R. L. and Fiduccia, C. M., "A Greedy Channel Router," *Proc. 19th Design Automation Conf.*, 1982, pp. 418-424.
- [47] Burstein, M., "Hierarchical Channel Router," *Proc. 20th Design Automation Conf.*, 1983, pp. 591-597.
- [48] Jennings, P. "A Topology for Semicustom Array-Struct LSI Devices, and Their Automatic Customisation," *Proc. 20th Design Automation Conf.*, 1983, pp. 675-681.