Copyright © 1985, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

A ROUTING TOOLBOX FOR
INTERACTIVE CUSTOM IC LAYOUT

by

D. E. Ryan

Memorandum No. UCB/ERL M85/40

15 May 1985

A ROUTING TOOLBOX FOR INTERACTIVE CUSTOM IC LAYOUT

by

D. E. Ryan

Memorandum No. UCB/ERL M85/40

15 May 1985

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

A ROUTING TOOLBOX FOR INTERACTIVE CUSTOM IC LAYOUT

by

Deirdre E. Ryan

5/12/85

Abstract

Several interactive graphics routines have been developed that enhance the routing capabilities of the Hawk/Squid package. The Hawk editor and underlying Squid database are suited for the design and development of custom integrated circuits. Hawk supports multiple windows for the editing and representation of multiple views of a circuit. Interactive routing routines have been added to the Hawk/Squid package to automate time consuming routing tasks typical in IC layout. These routines are described in this report. In addition to three bus-oriented routers, an interface to the channel router YACR2 was implemented to provide interactive channel routing in Hawk. The routing routines used by the routers are parameterized so that layout design rules are satisfied and specified at run time. Each routing routine is described in this report. One of the routines is presented in complete detail as an example for future integration of tools within the Hawk/Squid framework.

Acknowledgements

I would like to thank the following people for helping me at Berkeley. I would like to thank my advisor, Richard Newton, for providing guidance in the project. I owe an unspeakable amount to him for the opportunities that he has given me. Secondly, I would like to thank Alberto Sangiovanni-Vincentelli for convincing me to come to Berkeley (the capital of CAD), and for being, in general, an inspiration. I'd like to recognize Ken Keller as my mentor during my stay here. I'd like to thank Hopper for explaining things to me that would "only take 20 minutes." Thanks to Jeff Burns for reading my thesis and providing valuable feedback. I'd like to thank Jim Reed and Rick Rudell and everyone else in the cadgroup, especially those who answered all of my questions. Most of all, I'd like to thank Chris Marino, who kept me constantly laughing and without whom I would not have survived the cadgroup. Peter Moore, for telling me I'm "damn good". And, of course, Ron Gyrsick, for sending clicks to my terminal and Ken Kundert for taking all of my phone messages. This work was supported in part by DARPA under grant N00039-83-C-0107. Their support is gratefully acknowledged.

Table Of Contents

Chapter 1. Introduction 1

Chapter 2. Routing Problem Definition and Analysis. 3

2.1 Hawk and Squid. 3

2.2 Project Motivation. 5

2.3 Design Loop. 5

2.4 Alternatives. 8

2.5 SOAR Routing. 9

Chapter 3. Routers in the Toolbox. 11

3.1 Requirements of the Routers. 11

3.2 Routers 13

3.2.1 Pitch Change 13

3.2.2 Lturn. 17

3.2.3 Cable Router. 20

3.2.4 Channel Routing. 24

3.3 Design Rule Independence. 32

Chapter 4. Programming in the Hawk/Squid Environment. 36

4.1 Flow of the router. Pitch Change. 36

4.1.1 Invoking a command. 37

4.1.2 Reading Design Rules. 38

4.1.3 Get User Control Points. 38

4.1.4 Squid Data Retrieval. 39

4.1.5 Solving the Routing Problem. 42

4.1.6 Reporting Diagnostics. 43

4.1.7 Convert the Solution to Squid. 43

4.1.7.1 Open Routed Cell. 43

4.1.7.2 Place Geometries in Cell. 44

4.1.7.3 Save Routed Cell. 45

4.1.7.4 Place Instance of Route in Hawk View. 46

4.2 Debugging Hawk/Squid Routines. 47

4.3 YACR2/Hawk Interface 50

4.3.1 Operation and Description. 50

4.3.2 Suggestions for the YACR2/Hawk interface. 51

Chapter 5. Evaluation of the Hawk/Squid Programming Environment. 52

5.1 Documentation. 52

5.2 Learning to Program in Hawk and Squid. 53

5.3 Debugging Hawk and Squid Routines. 53

5.4 The Path Mechanism. 53

5.5 Exploiting Properties and Parameters. 54

5.6 Wish List of Features for Hawk and Squid. 54

Chapter 6. Conclusion 57

Appendix A. Code For The Pitch Change. 59

Appendix B. Routing Toolbox Users Guide. 74

CHAPTER 1

Introduction

Several interactive routing routines are described in this report. These routers are designed to speed up the floor-plan level physical routing of integrated circuits, a bottleneck in custom and semi-custom IC design. Different aspects of the routers and details of their development are presented.

First, the CAD tool environment will be examined as it applies to the tool users and the tool developers. The routing tools were installed into the Hawk/Squid[1] package, an IC CAD framework. They were developed in close conjunction with IC designers working in the Hawk/Squid framework to develop and complete a chip design. This provided valuable input and feedback to tool development from the users of the tools. Each router was designed to automate a recurrent routing chore in chip-level layout, as evident in the routing of the CMOS implementation of the SOAR (Smalltalk on a RISC)[2,3]. In Chapter 2, the CAD environment surrounding the routers is described.

In Chapter 3, the requirements of the routers are presented. Each router is described functionally. Three of the routers are designed to automate bus routing, and a fourth interactive routine provides an interface to the channel router YACR2[4]. The routing design rules used by the routines are parameterized so that the routers may be invoked with a set of design rules for almost any technology. The rules file that contains the design rule specifications is also described.

In Chapter 4, the mechanism of installing a routine into the Hawk/Squid framework is described. One of the routing routines is used as a detailed example for the future integration of CAD tools into Hawk/Squid. In addition, the Hawk/YACR2

interface is described and hints for writing and debugging these "client" routines are delineated.

Before concluding in Chapter 6, comments on the Hawk/Squid framework are given in Chapter 5.

CHAPTER 2

Routing Problem Definition and Analysis

As the scale of integration of VLSI circuits increases, the physical layout of a chip at a graphics editor becomes an increasingly unmanageable task. Many routing tasks that are repetitive are well suited for automation but editing tools often do not satisfy the needs of the custom IC designer. Large regions in a custom or semi-custom IC layout contain bus and signal routing. When routing an entire chip, the capabilities of the layout editor must exceed that of placing one geometry at a time, because the layout must frequently be updated to reflect the revisions inherent in a maturing design. Layout and design verification can be accelerated with computer aids; the guided routing toolbox was developed to assist in this floorplan layout of integrated circuits. In this chapter, the issues effecting the development of the guided routers are studied including:

(1) the Hawk/Squid layout environment.

(2) the motivation behind the project: the routing problem

(3) the alternatives considered for its solution,

(4) the design loop that uses chip-level layout, and

(5) the layout of SOAR.

2.1. Hawk and Squid.

The Hawk viewport graphics editor and underlying Squid database comprise an IC design *framework* upon which both CAD engineers and IC designers can build design systems. The package enables editing and management of the many different circuit representations, or *circuit views* of a design such as the physical and schematic views of

a cell. The design is entered and edited at a graphics terminal by invoking Hawk. The Squid database manages the underlying storage and retrieval of the hierarchical design being constructed.

In a hierarchical design, cells may be embedded in larger cells to reflect design partitioning. For example, several copies, or *instances*, of a nand gate may be called or *placed*, in an ALU cell. In the Squid database, the IC design is stored in the UNIX file system as described in [1]. In addition to circuit representations, non-circuit representations of a design may also be created and managed in the Hawk/Squid framework. The non-circuit representations, or *stranger views*, are not regulated or supported directly by the package, but may be manipulated by client routines that reside inside the Hawk/Squid framework. An example of a stranger view is a simulation view of a design, such as a textual SPICE input file that is derived from the extraction of a physical layout description or from the Squid netlist data structure.

The focus of this project has been on the enhancement of the physical layout capabilities of the Hawk graphics editor through the installation of the routing routines as client routines in the framework. Hawk/Squid supports physical layout with the following features: invocation of commands via keyboard and graphical input, pop-up windows, hierarchical menus, and multiple windows displaying multiple objects. From the menu or key commands, the user can create, edit, and save the cells being displayed in Hawk's multiple windows. In those windows, he can place, stretch, copy, and move geometries on different layers of a physical view to produce the mask layout of a chip. The system provides the access routines to manipulate geometries and cells. If a client routine is not linked to the Hawk load module prior to execution time, then during execution, the object file is dynamically linked and loaded when a command in the object file is invoked. A client tool is implemented with one of these subroutine commands. Tools may be integrated with the system through interface rou-

tines supplied by the package. The routers were added to the system using the interface routines.

2.2. Project Motivation.

At the cell level, the ability to place and edit geometry rectangles is satisfied by the Hawk editor with such commands as the *rectangle*, *terminal*, *move*, *copy*, and *place* commands[5]. Above the cell level, there were no tools integrated with the package to reduce the magnitude of time needed to complete the placement and routing of those circuit cells at the floorplan level. There is a general need in IC design for simple and incremental routing tools to ease the bottleneck that floorplan-level routing imposes on VLSI design. The layout and simulation of the CMOS SOAR chip was undertaken to drive the development and refinement of Hawk's layout facilities and test the effectiveness of a unified circuit design framework. Before a complex chip floorplan layout could be performed in a reasonable amount of time in the Hawk/Squid framework, chip-level routing tools had to be added. The routing needs were ranked by priority. The SOAR project's immediate need for routers prohibited the development of a full place and route system; the tools created were those that automated low-level commonly repeated tasks and required a short design and development period. A complete routing system would be useless to the SOAR project if the tool took a year to develop because chip routing would already have been completed. In a dynamic design loop, incremental routers are favored.

2.3. Design Loop.

To understand how the layout effort fits into the process of executing a VLSI design, one must look at a design loop of an IC:

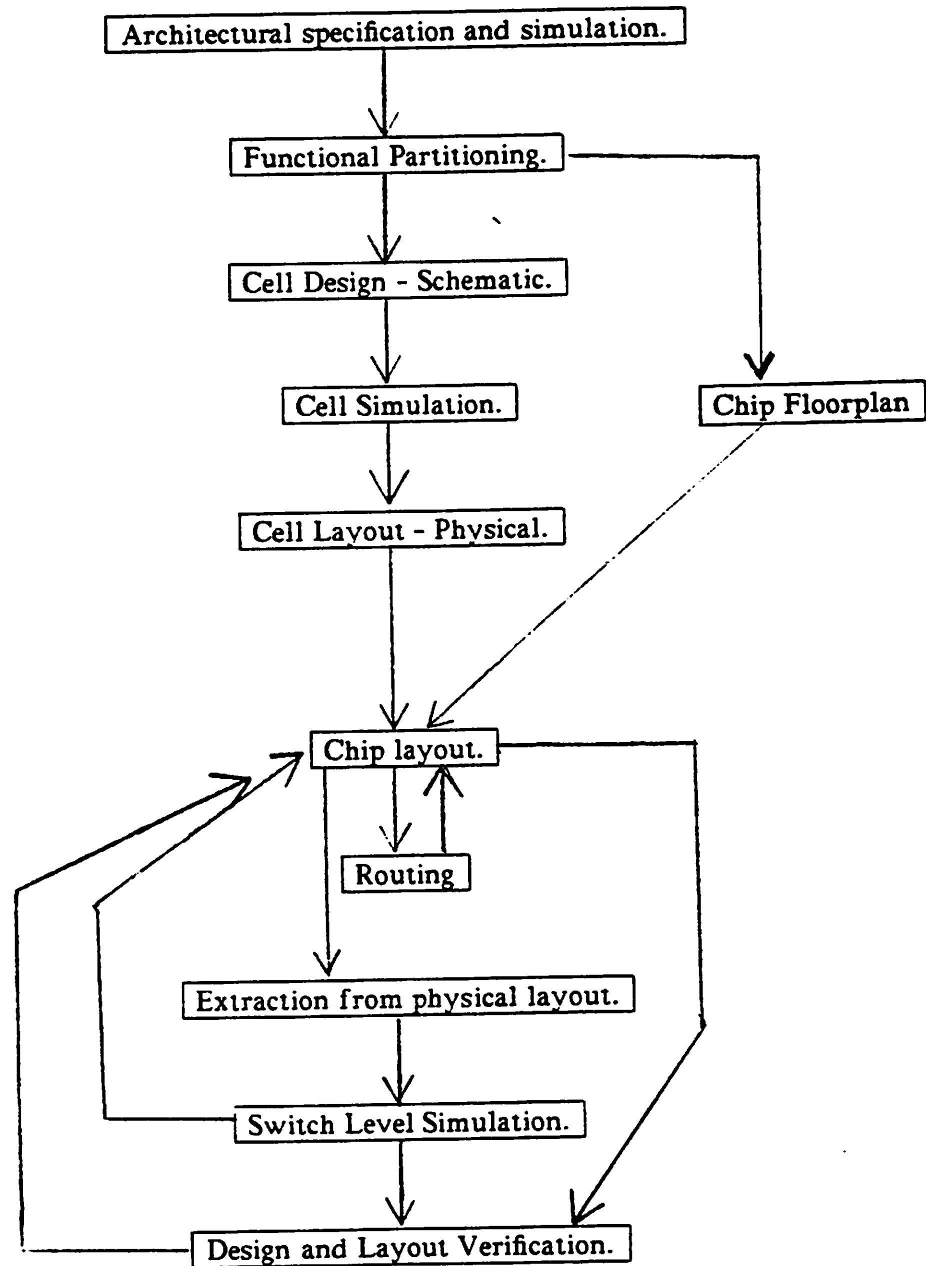

Figure 2.1. IC Design Loop.

In industry, design verification is often performed before the chip is committed to a specific layout. High performance industrial IC's such as Hewlett-Packard's FOCUS[6] chip are completely designed and simulated at the cell layout level before floorplan layout is performed. When a tradeoff is made between high performance and a short design time, designs may be assembled from pre-designed and pre-simulated cell libraries into either standard cell or gate array layouts.

In UC Berkeley's IC design environment, layout-verification and design-verification is performed after physical circuit cells are designed and placed in a layout. In SOAR, no cell library existed from which to select pre-designed and pre-simulated cells. Cells such as latches, nand gates, nor gates, and registers were built up in a SOAR library and used in the chip design wherever possible. After cell design, placement and routing was complicated by the irregular shape of circuit modules typical in a full-custom design like SOAR. Circuit extractors, layout rule checkers and other simulation tools use the circuit description derived from this physical layout. The results of these simulation and verification tools reveal layout errors and design flaws that require changes in the layout. The errors are then fixed and the design is resimulated. The design loop of "layout, simulation, layout, . . ." when fixing an error in the SOAR chip includes:

(1) approximately 100 CPU minutes for converting Squid to the intermediate format CIF[7]. (This step is a historical artifact of working with CIF in earlier designs. When all of the simulation tools work directly with Squid, this step will be reduced significantly.)

(2) 100 minutes of CPU time for circuit extraction.

(3) approximately 15 CPU minutes on a VAX 11/780 running a 40 cycle ESIM[2] simulation.

(4) time spent finding an error and its solution.

(5) and time spent making the layout change.

The designer iterates in this verification loop until the design is error free. In SOAR, approximately 30 of these verification iterations were performed, each requiring changes in the circuit layout. The routers help minimize design verification time by providing the ability to make incremental changes in a layout easily. The designer should be encouraged to make performance enhancement changes based on the results of simulation: it should not be a painful task to alter a layout. If it were necessary to re-run a global place and route program just to alter one region of a chip, the verification loop would be lengthened considerably. It is desirable to have the ability to interactively and incrementally change only certain regions of a layout.

2.4. Alternatives.

Given sufficient development time, however, it is difficult to create a place and route package that captures the intents and intelligence of the custom IC designer. Expressing electrical and circuit constraints to a global router is difficult. For example, routing heuristics may split a bus and route its sub-busses differently. The designer knows that the resulting signal timing skew is unacceptable. Global routers may not allow the designer the freedom to alter regions in a routing solution or they may slow down a critical clock signal with routing runs on the slow layer. When control of a router is only through initial circuit placement, it is not always obvious how altering one circuit's placement at the input will impact a router's solution. Changing a module's size or placement may change the entire route, creating routing problems elsewhere in the chip. When the designer can express layout specifications in a rule-based or constraint-oriented format, the router must internally rank the priority of those constraints. There is no guarantee that the router will satisfy them acceptably or at all. In very large designs, a limit on chip area may impose a hard constraint on routing

area: global routers may be simply unable to solve the routing problem in the space allotted.

If automatic routing tools were integrated into the package, yet unable to complete a route given the routing constraints, routing would have to be performed incrementally by the designer. The designer then needs a set of tools to assist in custom routing. The routines in the routing toolbox provide the intermediate routing capability between placing the routing geometries one at a time and routing the entire chip automatically.

2.5. Routing for SOAR.

The CMOS SOAR floorplan-level routing effort drove the development of the guided routing commands. SOAR's chip dimensions are 0.8cm. x 0.9cm. It contains roughly 34,000 transistors. It was through extensive interaction with the CMOS SOAR team that the tools were developed. Complaints about tedious tasks prompted the design of the several of the commands and feedback about developing tools were essential to the evolution of a tool's features. The tools were created in response to the designers using the system to maximize the utility of the tools being developed.

SOAR routing involves bus routing and a considerable amount of random signal routing. Commonly repeated tasks in the SOAR layout were automated in the routing toolbox to expedite chip layout. The **line** command was developed first. This command enables the user to draw a Manhattan-jogged wire by pointing to the succession of jog points. The wire command cuts the layout time of wires by significantly reducing the pointing events needed to place a wire. An extension of the wire command is a pitch change command that automates a change in the pitch between the wires in a bus. By hand, this is an error-prone and time-consuming task. The **pitch change, cable** and **LTurn** commands allow the user to manipulate busses of geometries quickly.

The routing tools influenced the SOAR chip floorplan. The interface to the YACR2 channel router provided fast rip-up and re-route capability, encouraging its use when routing control and logic regions: a channel area provides a good medium to route this random logic which is subject to change and whose nets are varied in destination. Many PLA's and NAND gates in the control section were placed on the border of rectangular channel regions and routed using the YACR2.

The tools in the routing toolbox do not perform the entire routing task. They are designed to fit into the layout environment by replacing some commonly-repeated routing event sequences with menu-driven routines. The layout artist guides the routines by interactively relaying his intent to the tools. It is therefore the designer, not a heuristic algorithm that is architecting the interconnection topography.

CHAPTER 3

Routers in the Toolbox

In this chapter, the routers in the toolbox are described from the user's perspective. First the routing requirements are reviewed. Then each router is described with example routes, and finally the design rule file is described. In this file, the routing design rule parameters used by the routines are specified.

3.1. Requirements of the Routers.

The routers described in this chapter fulfill the requirements of the designer for interactive layout. They are invoked via menu selection from the Hawk viewport graphics editor. The commands are highly-interactive; the editor prompts the designer throughout the routing process. Upon completion of the route, the solution that the router returns is usually the route that the designer would have drawn himself had he placed every rectangle. The intent of the routing tools is to help produce an efficient layout quickly by automating repetitive steps in the layout process. The designer guides routes by specifying "control points" when prompted.

The routers exploit the interactive Hawk environment by reporting pertinent diagnostics and routing information to the user. The YACR2 channel router interface reports such information as the name of the channel's longest net, any signal pin that does not have a matching pin in the channel, and whether the channel width should be grown or shrunk based on the channel's density. In the cable route, the user is notified if there is not a one-to-one correspondence between the initial and final terminals in the cable route. Pertinent information is reported during the course of the route and saved in a file for later reference. In the past, some CAD routing tools would just say "sorry" when a route was in error or could not be completed by the algorithm. The

purpose of the routing toolbox is to help the layout artist in every possible way. The detailed and informative diagnostics produced by the routers described here help keep the designer on the right track.

The routers emulate the designer's routing strategies. The cable and channel routers have a preferred routing layer and maximize runs on that preferred layer. While making jogs, the pitch change and the cable commands route wires at the minimum design rule spacing. The routing routines essentially return the route that the designer would have constructed had he routed the region rectangle by rectangle.

Each routing routine creates a cell master filled with the routing solution's geometries. A pitch change, LTurn, cable route, or channel route instance of the master is placed in the cell being edited in Hawk. Placing the geometries hierarchically as a cell, rather than placing flat geometries saves the context of the route. The entire routing cell may then easily be manipulated or deleted as routes are re-run. Selecting and deleting flattened geometries slows down the design verification loop.

The Hawk viewport manager may be invoked under various layout technologies. Current technologies in use at UC Berkeley are the CMOS-PW and the NMOS fabrication technologies. Each technology has its own layout design spacing rules and routing layers. The routing routines are not constrained to a particular technology-dependent set of rules and layers. They are flexible enough to route between any two layers of a particular technology. The spacing rules and routing layers are assigned at run time rather than at the compile time of the routers. While a particular layout and design style may employ only a subset of allowed routing strategies, the routers provide the mechanism to route in any number of ways.

The ".cadrc" file resides in the user's home directory and stores tool-specific information in the UC Berkeley CAD environment. Each ".cadrc" file is divided into several sections. The **HAWKROUTERS** section of this file contains routing information

that tunes the routers to a particular routing policy. Some fabrication lines support routing only in first-layer metal and in polysilicon. UC Berkeley's CMOS fabrication process recently introduced second layer of metal to its line. Via the rules file, the routers can make the transition from routing channels in polysilicon and metal1 to routing them in metal1 and metal2.

3.2. Routers.

3.2.1. Bus Routing.

Often in a bus oriented architecture like the SOAR 32-bit microprocessor, the layout artist thinks of a bus of nets and its associated route as one entity, but is forced to route each net in that bus individually. For example, it is necessary to bring the n output signals of a PLA from their wide output spacing to a minimum spacing in order to squeeze the bus through a tight routing region. The user typically routes one net at a time, performing wire jogs at minimum spacing. Routing a bus element by element is a tedious chore. The algorithm that the layout artist is performing when he lays down each geometry in the bus is easily encoded. The pitch change and L Turn commands were written to capture the layout engineer's intent and automate bus routing.

3.2.2. Pitch Change.

The pitch change command enables the user to direct a change in the pitch of a *bus* of nets from any regular or irregular spacing to a user-specified regular spacing. Here a *bus* of nets is defined as a parallel set geometries on the *same* layer ending roughly at the same edge as illustrated in Figure 3.1. The pitch change command may be used to fan out a minimally-spaced group of geometries to a contact-to-contact spacing or squeeze an irregular spacing down to a minimum spacing. In a pitch change, there is no order change in the bus's nets. The bus elements are simply river-routed to a new pitch.

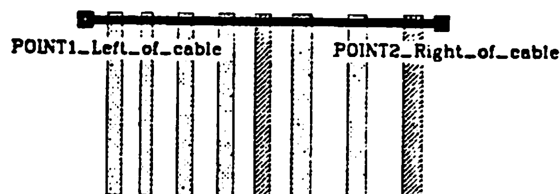

Figure 3.1. Initial Bus Geometries.

3.2.2.1. Supporting Rule File.

To perform a pitch change, it is necessary to specify the design rules associated with the layout routing layers with the inclusion of the HAWKROUTERS section of the ".cadrc" file. The pitch change may be performed on one of the two routing layers specified in that HAWKROUTERS section. Four ".cadrc" design rule parameters are necessary to perform a pitch change. These are the two possible routing layers' names and their minimum spacing. Here is an example for the mosis CMOS_PW technology:

```
/* ".cadrc" parameters for the pitch change command */
begin HAWKROUTERS
  NAME LAYER1_NAME "CP"
  NAME LAYER2_NAME "CM"
  VALUE LAYER1_MINSPACING 3
  VALUE LAYER2_MINSPACING 4
end
```

The two routing layers are "CM", CMOS metal, and "CP", CMOS polysilicon. A CMOS polysilicon geometry must be spaced at least 3 lambda from another polysilicon

geometry, while a CMOS metal geometry must be at lease 4 lambda from another metal geometry. Please refer to the ".cadrc" section of this chapter for more details on this design rule file.

3.2.2.2. Majority Selection.

A pitch change is performed on only one routing layer. If the program's database search returns geometries on two layers, a pitch change is performed on the layer with the majority count in the bus. For example, the pitch change selection region in Figure 3.1 has six polysilicon and two metal elements. The pitch change is performed on the six polysilicon elements. Majority rule is used because the objects on the other layer are usually not intended to be in the bus; they are unavoidably in the selection region.

3.2.2.3. Invoking the Pitch Change.

The pitch change command is found in Hawk's Routing sub-menu. Upon **pitchChange** menu selection, the user must specify which geometries he wishes to include and extend in the pitch change. He is prompted to cut across the bus of nets *where* he wants to start the pitch change, by pointing at the bottom and top of the bus. The top and bottom points are the endpoints of a *cut line* across the pitch geometries. The program performs the database search to find the geometries under that cut line on layers LAYER1_NAME and LAYER2_NAME; it decides which geometries to include in the pitch change by the majority rule. The user is then prompted to mark the point where he wants the bottom bus element to route to (final pitch point), and type the new *pitch* or separation of the wires of the bus in lambda units. The layout is then determined by the bottom point and the new pitch. Each geometry is extended in the pitch change at its original width. The new geometry spacing is the user-specified pitch. Jogs in the pitch change are performed at the routing layer's minimum spacing. A pitch change of the initial geometries of Figure 3.1 is shown in Figure 3.2 with the control

points marked.

3.2.2.4. Squid data storage.

The pitch change is stored in its own cell and a copy of that cell is placed in the Hawk view being edited. This eases changes in the layout. When the route is complete, the user is prompted to type a cell-name and cell-view identifying that routed cell. The cell is created in the Squid database and an instance of the cell is placed in the Squid cell from which the pitch change was invoked. The Hawk screen is redisplayed showing the newly routed pitch change.

3.2.2.5. Errors.

Any of several errors prohibit the completion of a pitch change. In case on an error, the user is notified what is preventing the pitch change and that error information is stored in the **route_error** file. The pitch change is not performed when no

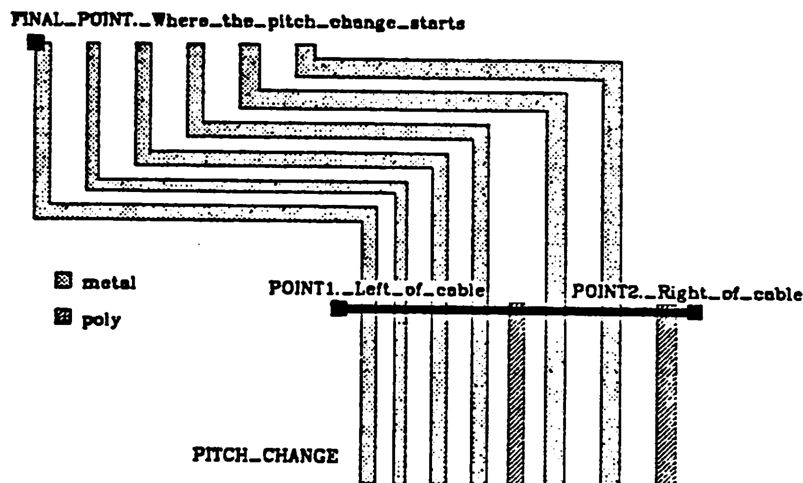

Figure 3.2. Pitch change with control points marked.

geometries are found under the cut line. The routine also aborts when the required ".cadrc" parameters are absent from the HAWKROUTERS section of the ".cadrc" file. Similarly, the pitch change is not performed when the user does not provide enough room between the initial cut-line and the final pitch point. The user is told how much more space is needed in lambda units to perform the jogs of the pitch change.

The pitch change command is described in more detail later in this report. It is used to review and describe the Hawk and Squid routines used in the guided routers, providing a working template for application programming in the Hawk/Squid framework.

3.2.3. LTurn.

The L-Turn router is very similar to the **pitchChange** router. It enables the user to perform an *L-Turn* on a bus of nets from any regular or irregular spacing to a user-specified regular spacing. Once again, a *bus* of nets is defined as a set of parallel geometries on the *same* layer ending roughly at the same edge. (see Figure 3.1.) Examples of the usage is routing a bus of nets around a circuit module, or routing the inputs to a PLA. The **LTurn** command may be invoked when there is no order swap in the geometries of the bus; therefore, no layer changes have to be made and no terminals are needed to imply routing connectivity. Geometries are extended around one turn in their original order and with their original widths as illustrated in Figure 3.3.

An "L-turn" may be performed on either of the two layers specified in the HAWKROUTERS section of the ".cadrc" file, but, as with the pitch change command, it is only performed on one of those layers. The program searches for geometries on the layers LAYER1_NAME and LAYER2_NAME. The L-Turn command requires only that the two routing layers be present in the HAWKROUTERS section of the ".cadrc" file.

```
/* ".cadrc" parameters for the LTurn routine */
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
```

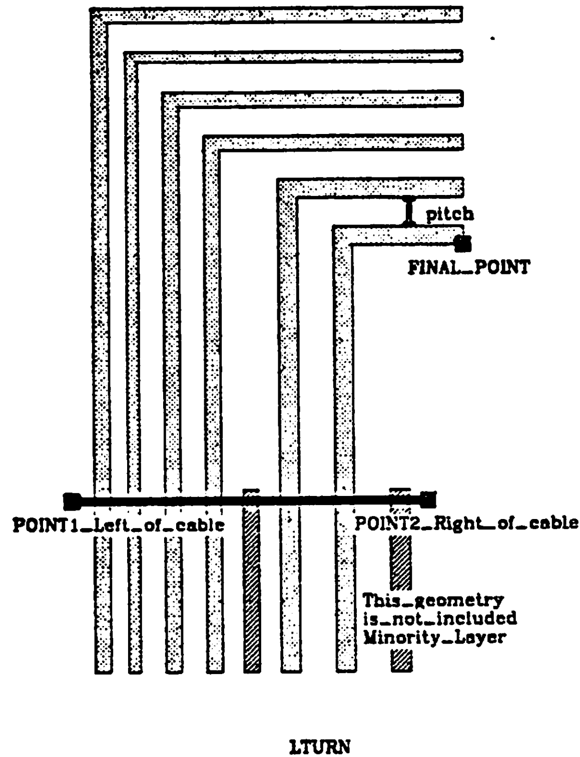

Figure 3.3. L Turn with control points marked.

```
NAME LAYER2_NAME "CM"
end
```

notifying the routine which of two possible layers the L-Turn may be performed on. An L-Turn routine is aborted if the two routing layers are not included in the ".cadrc" file.

Upon **LTurn** menu selection the user is prompted to cut the bus of nets *where* he wants to start the L-Turn by pointing to the left and right of the bus. The user would typically point to the left and right *edges* of the bus; he may alternatively point before the edge to do an L-Turn on geometries whose ends do not line up. The user is then prompted to point to where he wants the bottom of the L-Turn to begin and type the

new spacing of the wires in lambda units. The L-Turn command is then executed. The width of each geometry is extended in the L-Turn. The user is prompted to type a cell name and view name to give the L-Turn. That cell is created and placed in the current Hawk cell from which the L-Turn was called.

The pitch and LTurn commands are interactive incremental bus routing tools that have proven very useful in the layout of the CMOS SOAR 32 bit microprocessor. The pitch change command was used in approximately 40 places and the LTurn command was used in approximately 20 places in the layout of the CMOS SOAR chip. The routines were run many more times than that. accommodating changes in the layout. In Figure 3.4. SOAR bus routing is illustrated where pitch changes and LTurns are used.

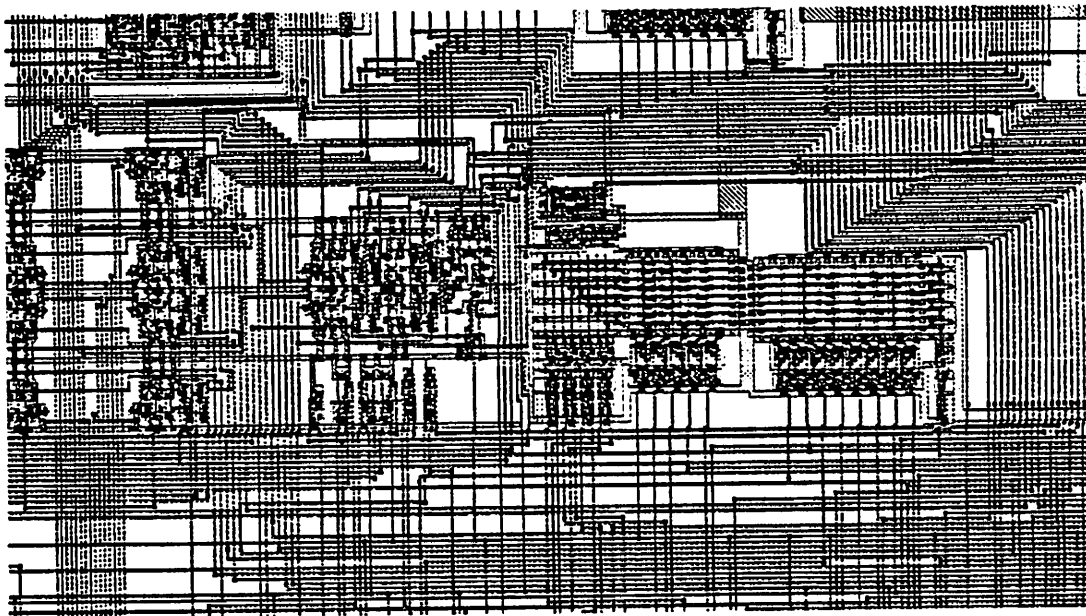

Figure 3.4. SOAR pitch change and LTurn.

3.2.4. Cable Router.

The cable command is used to route a bus of nets around obstacles. It is not a general river router: it routes each signal in a bus between one source terminal and one destination terminal. Like the pitch change and L turn routers, there must be a one to one correspondence between connections in the bus. Unlike those commands, the cable routine allows the signals to switch physical order from the source to the destination. If a switch in signal order is needed, the command performs the order change at the last jog in the bus's path. In the interactive cable command, bus specification is simple. Terminals specify:

(1) signal connectivity,

(2) cable layer, and

(3) the width of wires in the cable.

3.2.4.1. Signal Connectivity.

Because the physical signal order may be permuted from the initial to the final terminals, signal connectivity is specified with terminals at the beginning and end of the cable. Matched terminal names imply connectivity. Each signal in the cable has a terminal in the initial terminal set and in the final terminal set. It is necessary to place these terminals before running the cable command. Note that when there is no swap in signal order from the initial to the final terminals, it is easier to use the pitch and L turn commands, because in those commands it is not necessary to place signal terminals.

3.2.4.2. Routing Layer.

The designer selects the routing layer with the terminal layer. The initial and final terminal layers and the associated layer parameters must be specified in the HAWKROUTERS section of the user's ".cadrc" file. All of the initial terminals must be on

the same layer and all of the final terminals must be on the same layer.

(1) Case A. The initial terminal and the final terminal sets are on the *same* layer and that layer is described in the rules file. At the last jog in the cable, a via route onto an alternate routing layer is made to accommodate any signal order change. The design rules for the routing layers are described in the rules file. The layer shared by the initial and final terminals is maximized as shown in Figure 3.5.

(2) Case B. The initial and final terminal sets are on the two *different* layers specified in the rules file. In this case, the initial terminal set's layer is maximized as shown in Figure 3.6. The layer change from the initial terminal set's layer to the

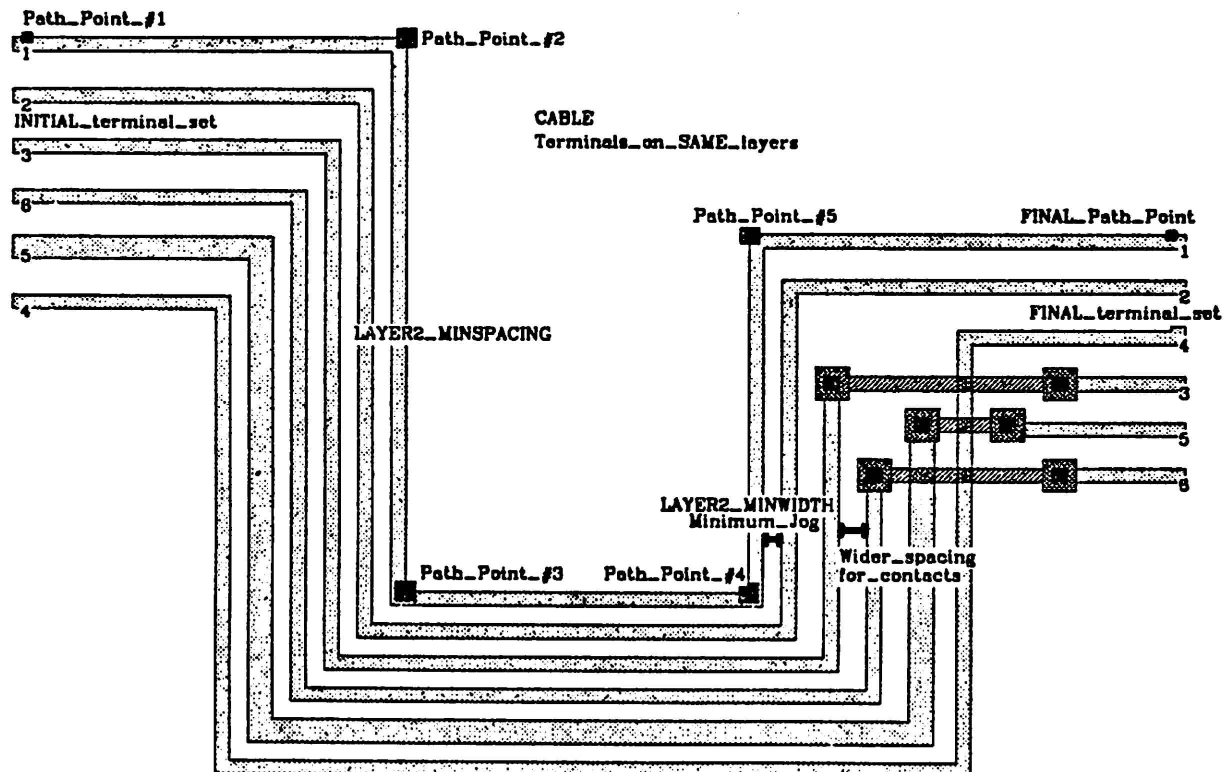

Figure 3.5. Cable Route. Same terminal layers at ends.

final terminal set's layer is made at the final jog.

In both cases, Squid instances of a contact master are automatically placed at layer changes in the cable solution. As with the channel routing command, it is necessary that the user have a Squid file describing the physical view of a via connecting the two routing layers.

3.2.4.3. Cable Path Specification.

The cable command allows the user to guide the bus's route around obstacles through a variable number of jogs. In the cable command, the user is prompted through the command as he guides the route. First the user points to the initial

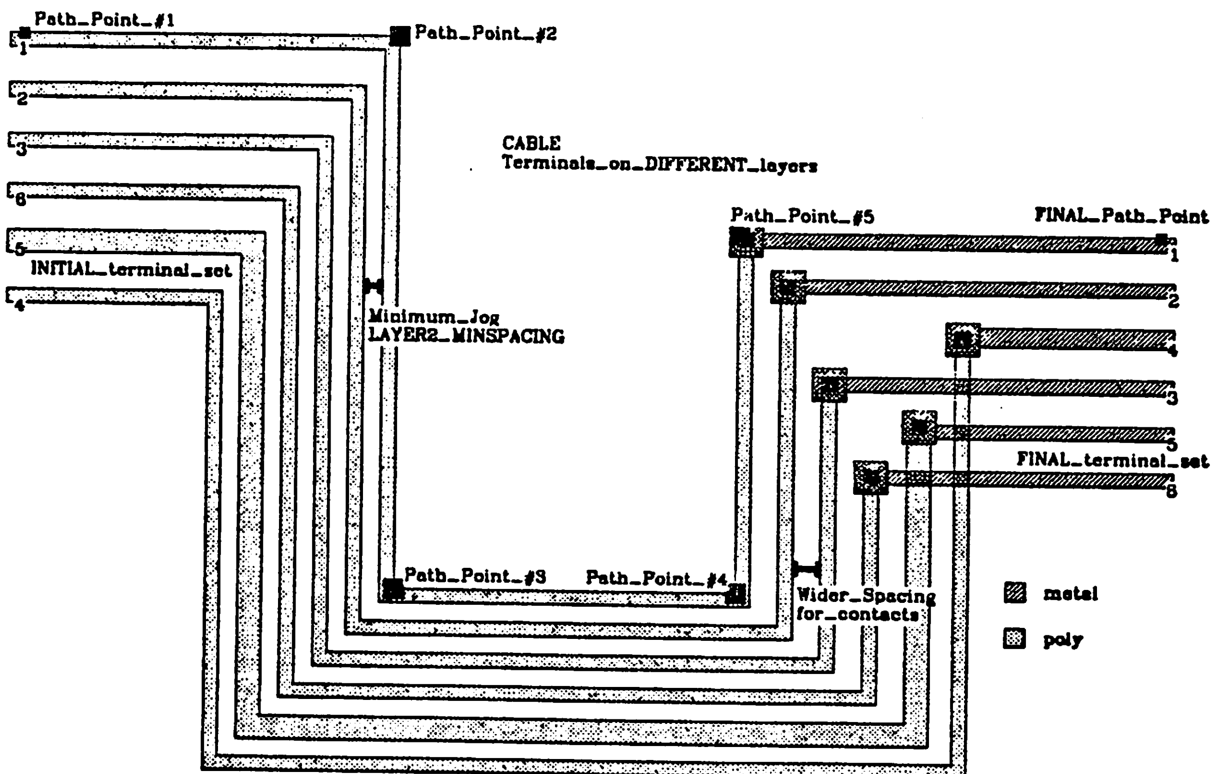

Figure 3.6 Cable Route. Different terminal layers at ends.

terminals, one at a time. Then he points to the final terminals, one at a time. He then specifies the path that he wants the route to take from the initial to the final terminals by pointing to jogs on one side of the cable's path. One side of the path is illustrated in Figure 3.6.

3.2.4.4. Wire Widths.

Wire widths of signals in the cable are determined as follows. There are two different cases to consider.

(1) Case A. When the initial and final terminals sets are on the *same* routing layer, a layer change is performed only where necessary to execute a swap in signal order. When there is no layer change in a wire, as in net 4 of Figure 3.5, the wire width is set to the width of the signal's initial terminal. Net 3 has a layer change, so the initial wire width is determined by the width of the signal's initial terminal: the final wire width is determined by the width of the signal's final terminal. The via layer's width is the minimum width for that layer as specified in the rules file.

(2) Case B. In the case where the initial and final terminals sets are on *different* routing layers, each terminal's width determines its wire's size as illustrated in Figure 3.6.

3.2.4.5. Space conservation.

The route is grown from the user-supplied jog points. The spacing of the wires in the initial and final cable segments is determined by the terminals. Initial terminals in the cable may be placed at minimum spacing. If they are not at at least this minimum spacing, the user is notified. Final terminals are placed at at least geometry to contact spacing, because changes in signal order are made at the last jog. Geometry spacing is widened to accommodate contact vias at a layer change. Spacings are widened from minimum only when a contact dictates. Notice that in Figure 3.5, the spacing between

net 1 and net 2 in segment 4 of the cable is at a minimum spacing because there is no order change in nets 1 and 2 at the last jog. The spacing between net 2 and net 3 is wider in segment 4 to accommodate the layer and order change in net 3.

3.2.4.6. Necessary Supporting Files.

The **contact/physical** Squid file must reside in the current directory at the time Hawk is invoked. This is the physical view of a contact connecting the two routing layers in cable solution. In addition, the two routing layers must be set in the HAWKROUTERS section of user's ".cadrc" file. The parameters necessary to run the cable command are:

```
/* ".cadrc" parameters necessary for the cable routine */
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
 NAME LAYER2_NAME "CM"
 VALUE LAYER1_MINWIDTH 3
 VALUE LAYER2_MINWIDTH 3
 VALUE LAYER1_MINSPACING 3
 VALUE LAYER2_MINSPACING 4
 VALUE CONT_SIZE 7
end
```

The layer definitions are described in the ".cadrc" section of this report.

Execution errors. The cable command flags an error when two initial terminals are not at at least the minimum spacing for that layout layer or when two final terminals are not at at least a geometry-to-contact spacing. It flags an error when there is not a one-to-one match between initial and final terminals.

3.2.5. Channel Routing.

The **Horizontal Channel** command implements a two-layer horizontal channel route by providing an interactive interface between the Hawk graphics editor and the YACR2 channel router. This interface was written in conjunction with Rick Rudell and Jim Reed. A channel is a rectangular routing region. Terminals, or pins, mark signals on the channel periphery that are to be connected. The routing interface is divided into

three parts, the pre-processor, YACR2, and the post-processor. The pre-processor collects the information that YACR2 expects as input from the Hawk cell being edited and calls YACR2. The post-processor translates YACR2's output into Hawk geometries, redisplays the routed solution, and reports useful routing information.

YACR2 was chosen as the channel routing algorithm for several reasons. YACR2 usually routes channels in less area than other channel routing algorithms. It typically uses far fewer routing vias than other algorithms such as Burstein's Hierarchical Channel Router or Deutsch's Dogleg channel router[8]. It can route channels with cyclic constraints, and it maximizes routes on a preferred layer.

The channel routing interface provides fast rip-up and re-route capability on a per-channel basis. Channel routing is effective when routing large amounts of "spaghetti" logic that is prone to error and change. YACR2 and its interface guarantee that all terminals of the same name in the channel are connected. In the CMOS SOAR layout there are seven channels; two of these channels are very large (158 nets and 74 nets). Each were re-routed approximately ten to fifteen times due to changes and fixes in the design and layout. By hand, this would certainly have been a nightmare because changes were usually to more than one or two of the nets in the channel.

3.2.5.1. Channel Definition.

Terminals specify the connectivity or net-list to be routed. The terminals are fixed along the two opposite channel lengths, and are floating on the remaining channel-ends.

(1) **Fixed terminals** along the length of the channel determine exact signal location. Top and bottom terminals must both be colinear.

(2) **Floating terminals** designate signals that must exit the channel from an end. The signal order of the floating terminals in the solution is determined by YACR2, not by the initial floating pin order. Side terminals must be colinear.

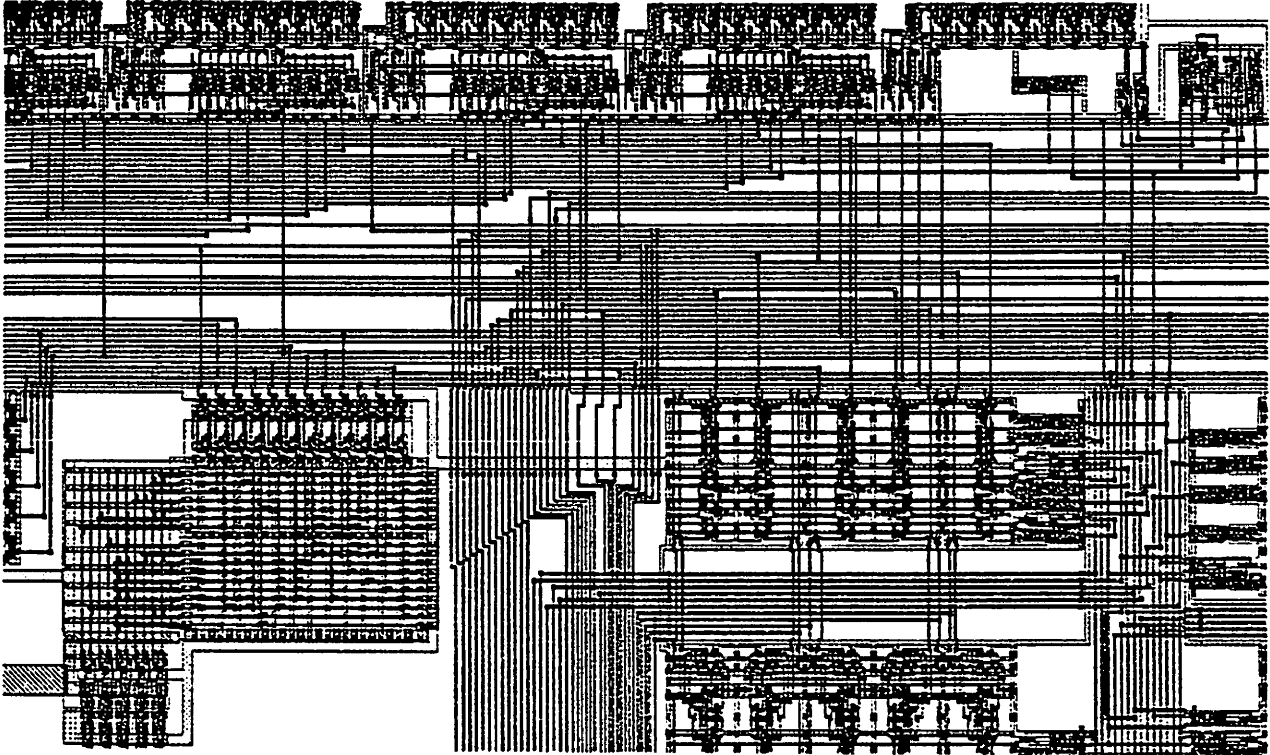

Figure 3.7. SOAR channel route.

Floating and fixed terminals are placed before running the channel router by using the **terminal** command in the Hawk graphics editor. The YACR2 algorithm connects all terminals in the channel of the same name. Terminals must be placed at at least the minimum user-defined spacing of LAYER1_GRIDSPACING and be on one of the two routing layers. Terminals on any other layer in the channel will be ignored by the interface. To avoid layer change in the top and bottom tracks of the channel solution, fixed terminals should be on LAYER1_NAME, the layer that runs vertically across the channel width. This layer is polysilicon the routing example. Figure 3.8 illustrates channel definition.

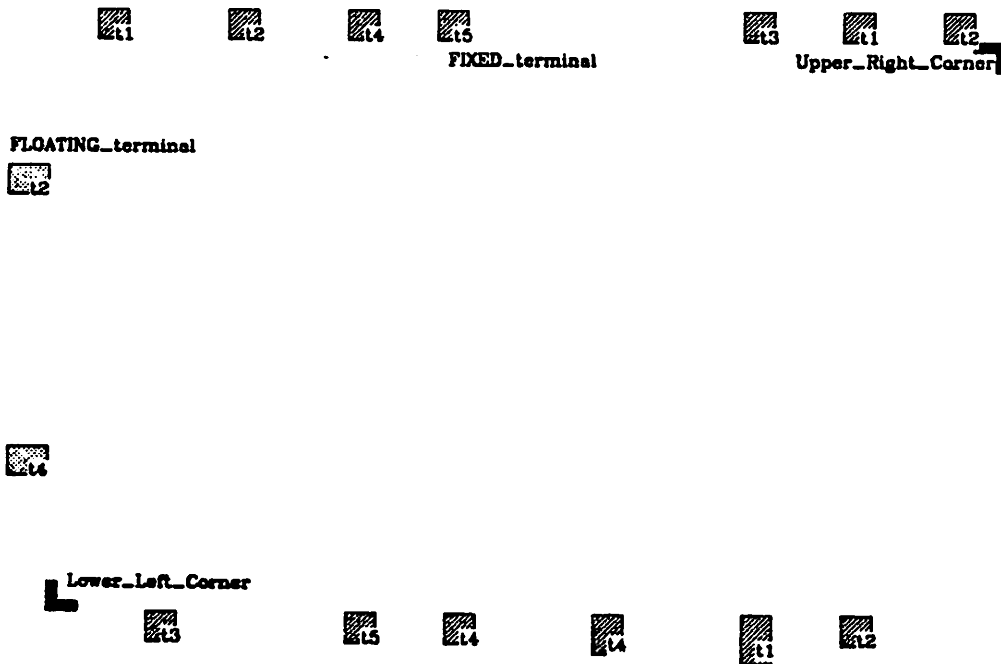

Figure 3.8. Channel definition.

3.2.5.2. Necessary Supporting Files.

There are two necessary files to include in order to run the channel router. A Squid cell named **contact** with the view **physical** must reside in the designer's file system in the same place where the Hawk file *path* is and where Hawk is invoked from the user's project directory. This file is of the form:

```
SQUID
PUT VIEW "contact" "physical" "w" "squidNextObjectID" INT 4
MK RECT 1 ACTIVE LAYER "CP" FILL LB -60 -60 RT 80 80
MK RECT 2 ACTIVE LAYER "CM" FILL LB -60 -60 RT 80 80
MK RECT 3 ACTIVE LAYER "CC" FILL LB -20 -20 RT 40 40
```

This contact master file is placed as an instance at every layer via in the channel routing solution.

In the *.cadrc* file, the user defines the two routing layer names and the associated design rule parameters.

```
/* ".cadrc" parameters necessary to run the channel router */
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
 NAME LAYER2_NAME "CM"
 VALUE LAYER1_MINWIDTH 3
 VALUE LAYER2_MINWIDTH 3
 VALUE LAYER1_GRIDSPACING 8
 VALUE LAYER2_GRIDSPAGING 9
 VALUE CONT_SIZE 7
end
```

The two routing layers are LAYER1_NAME and LAYER2_NAME, "CP" and "CM". LAYER1_NAME is the "slower" routing layer (higher RC time contstant per unit length). This layer is used for routing signals on the shorter vertical channel runs across the width of the channel. LAYER2_NAME is the preferred routing layer; it makes the longer channel track runs and has lower RC per unit length and is *maximized* in the channel by YACR2, as illustrated in Figure 3.9; when there is not another signal's route blocking the LAYER2 continuation of a net into a column, the signal stays on the preferred layer rather than switching to the column layer, LAYER1.

The LAYER1_MINWIDTH and LAYER2_MINWIDTH parameters determine the width of signal runs in the channel. In the above rule specification, both metal and polysilicon runs are drawn at a width of 3 lambda. The CONT_SIZE parameter specifies the size of the contact that connects the two routing layers at signal vias.

The *grid spacing* parameters determine row and column channel pitch.

(1) *Column*. Placement of column geometries is pre-determined by the fixed signal terminals along the top and bottom of the channel. LAYER1_GRIDSPACING is the minimum pitch of two terminals in adjacent columns of the channel. Typically, this parameter is set to a contact-to-contact pitch to accommodate instances in the channel where contacts are in neighboring columns. As described later in the

rules file section of this report. LAYER1_GRIDSPACING may be tightened to a smaller pitch.

(2) *Row*. The rows of the channel solution are grown from the bottom row of terminals. The LAYER2_GRIDSPACING parameter sets the pitch spacing of the solution's tracks. The post-processor minimizes horizontal track placement. Track grid pitch, LAYER2_GRIDSPACING, should be set to contact-to-geometry pitch in the ".cadrc" rule file. When there is an instance of a contact directly above another contact in the channel solution, the post-processor automatically widens from the tighter row spacing to contact-to-contact spacing to accommodate the adjacent contacts.

The channel routing routines minimize channel width in two ways, with YACR2's algorithm and with the post-processor's track placement. Refer to the ".cadrc" section of this document for more specific details about all these parameters.

3.2.5.3. Program Details.

After selecting **HorzChannel** on the routing sub-menu, the user identifies the channel interactively by pointing to two opposite corners of the rectangular channel. Horizontally, channel pins must be colinear. Vertically, the channel definition is gridless: top pins do not have to line up directly over bottom pins. The pre-processor marches across the channel and creates the net list grid that YACR2 is expecting as input. This YACR2 input file is called **.chan.route.in**. It is created in the directory where Hawk is invoked. The file contains the top and the bottom fixed net lists from left to right, and the left and the right floating net lists from bottom to top. Each net name is translated into an integer in this file because YACR2 expects integer net names as input. Here is **.chan.route.in** for the channel shown in Figure 3.8:

```
5                      #five nets in channel
10                     #ten columns in channel
5 0 4 2 1 0 0 3 5 4    #top net list
0 3 0 1 2 0 2 5 4 0    #bottom net list
2                      #two left nets
2 4                    #the two left nets
```

The pre-processor calls YACR2 with this input file and YACR2 returns the output file called **.chan.route.out**. Here is the **.chan.route.out**. solution for the above input:

```
5                      #there are 5 tracks in the solution
10                     #there are 10 columns in the solution
0 3 3 3 3 3 3 3 5 4    #metal row 1
2 2 2 2 2 2 2 0 5 4    #metal row 2
5 0 0 1 1 1 2 0 5 4    #metal row 3
5 5 5 5 5 5 5 5 5 4    #metal row 4
4 4 4 4 4 4 4 4 4 4    #metal row 5
5 3 4 2 1 1 0 3 5 4    #poly row 1
5 3 4 2 2 1 0 0 0 0    #poly row 2
5 3 4 1 2 1 2 0 0 0    #poly row 3
0 3 4 1 2 0 2 5 0 0    #poly row 4
0 3 4 1 2 0 2 5 4 0    #poly row 5
4 15 7                 #longest net is net 4, 15 metal units, 7 poly units
```

The post-processor maps the solution back into the particular Hawk channel. The routed geometries are stored in their own cell. The user is prompted to name the cell for identification and placement in the design hierarchy; an instance of the channel solution is placed in the Hawk physical layout being edited. The routing solution to the channel defined in Figure 3.8 is shown in Figure 3.9.

3.2.5.4. Diagnostics.

Like all the routers in the routing toolbox, the channel router reports information and diagnostics to the user. When the allocated channel width is too narrow for the actual solution, the user is told how much larger he must make the channel. If too much track space has been allocated in the layout, the user is notified how much extra space he has allocated. Information on the channel's longest net may be used to hand-calculate signal delays. The routine reports the name of any terminal that does not have at least one partner connection in the channel. This error often is the result

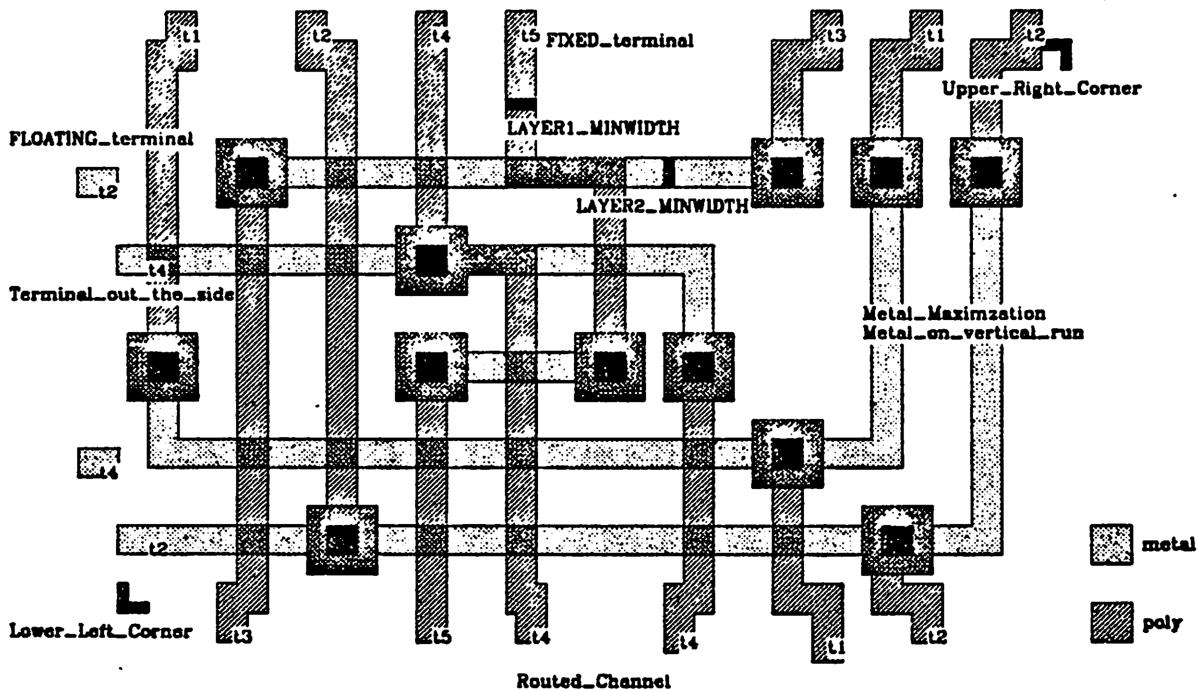

Figure 3.9. Channel Solution.

of a error when typing the terminal name. If the ".cadrc" file is not present or any of the design rule parameters are uninitialized in that file, the program will not run; the user is notified, however, which parameters are missing. Here is the listing of the route_error file for the channel route is illustrated in Figure 3.9:

```
longest_net_index: 4  net_name: t2
CM net length: 135
CP net length: 63
Allocated channel width: 59
The channel is 7 lambda wider than necessary.
Routed channel width = 59
```

3.2.5.5. Vertical Channel Routing.

The **VertChannel** command is identical to the HorzChannel command except that it implements a vertical channel route. In the vertical case, the long channel tracks are vertical; the preferred routing layer is used to route tracks.

3.3. Design Rule Independence.

The spacings used by the routing routines have been parameterized. This gives the user the design rule independence to route in any technology between any two routing layers. The user tunes the routers to his particular routing policy by assigning values to the parameters. This section describes the routing parameters, where they are stored, and where they are used by the routing routines. In addition, an example of their usage is described.

The ".cadrc" file is the standard file that holds parameters specific to CAD tools at UC Berkeley. This file was chosen to also hold layout design rules, enabling the user to set his own spacing rules for the Hawk routers. In the ".cadrc" file there are sections for particular CAD applications. A Hawk-specific section sets key aliases for Hawk menu selections. This section is read when Hawk is first invoked. The routers are dynamically linked into Hawk upon their invocation, so another section called HAWKROUTERS was added into the ".cadrc" file. The HAWKROUTERS section is read when a router is invoked. The section contains the assignment of the design rule parameters needed by the routers.

The HAWKROUTERS section is as follows for the CMOS-PW rules:

```
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
 NAME LAYER2_NAME "CM"
 VALUE CONT_SIZE 7
 VALUE LAYER1_MINSPACING 3
 VALUE LAYER2_MINSPACING 4
 VALUE LAYER1_MINWIDTH 3
 VALUE LAYER2_MINWIDTH 3
 VALUE LAYER1_GRIDSPACING 8
 VALUE LAYER2_GRIDSPACING 9
end
```

The NAME lines specify the two routing layers. In the above example, the layers are CM and CP, p-well CMOS metal and p-well CMOS polysilicon. LAYER1_NAME is the less desirable layer given whatever metric with which the designer judges a layer desirable. LAYER2_NAME is the more desirable layer and will be maximized in the cable and channel routing commands.

One dimension of the square contact is specified in lambda with the CONT_SIZE parameter. In the CMOS-PW case, a contact via between metal and polysilicon is 7 lambda by 7 lambda. The two routing layers' spacing rules are then specified.

The _MINSPACING parameter is the minimum distance that a geometry must be from another geometry on the same layer. In the CMOS-PW technology, the minimum spacing for CM is 4 lambda; the minimum spacing for CP is 3 lambda. This parameter is used by the pitch change and cable routers to perform minimally spaced jogs.

The _MINWIDTH parameter is the minimum allowable drawn width of a geometry. In the CMOS-PW case, this parameter is 3 lambdas for CM and CP. This is used by the channel router to draw minimum width geometries in the routing solution. It is also used by the cable command to draw minimum width geometries on the via layer when a layer change is made to switch the order of the cable signals.

The _GRIDSPACING parameter is used exclusively by the channel router to specify the virtual grid that the channel router observes when searching for terminals and

placing tracks. In a channel route, the faster layer (low RC per unit length) makes the runs along the length of the channel while the slower layer brings the signals into the channel, running the channel width. Connections between the two layers are made with contacts. In a channel solution, a contact may be adjacent to a contact in another column in the worst-case grid pitch.

LAYER1_GRIDSPACING, the pitch of the LAYER1 terminals at the top and bottom of the channel, should be at contact to contact spacing. For the CMOS-PW case above:

```
LAYER1_GRIDSPACING = CONT_SIZE -
              MAX (LAYER1_MINSPACING, LAYER2_MINSPACING)
         = 7 + MAX (3,4)
         = 11.
```

In a YACR2 channel solution, there are sufficiently few places where there is a contact next to another contact on a horizontal track. Signal terminals may then be spaced closer than at contact to contact spacing. LAYER1_GRIDSPACING may be tightened to a smaller number; it is set to 9 lambda for the CMOS-PW case:

```
LAYER1_GRIDSPACING = LAYER1_MINWIDTH
            + MAX (LAYER1_MINSPACING, LAYER2_MINSPACING)
            + (CONT_SIZE - LAYER1_MINWIDTH)/2
         = 3 + MAX(3,4) + (7-3)/2
         = 9.
```

The resulting routed solution contains design rule errors only where there is a contact next to another contact on a channel track. Reducing the pitch of the terminals saves considerable channel length, at the expense of having to edit the solution where the few contact spacing errors occur. For example, there are 72 nets and 169 columns in Deutsch's difficult channel routing example [4]. Of the 287 contact vias in YACR2's channel solution, there are only six instances where a contact is adjacent to another contact on a channel track. Routing at a tighter pitch reduces each column width by two lambda for a total length reduction of 2 lambda/column x 169 columns = 338 lambda. The user may choose whether to route at wide spacing or tight spacing.

While reducing LAYER1_GRIDSPACING may require hand editing of a channel solution, it can reduce channel length significantly.

LAYER2_GRIDSPACING is the minimum distance that two horizontal tracks are spaced. This parameter is minimized to a pitch of geometry-to-contact spacing. The channel router places tracks at this tight GRIDSPACING unless there are two vertically adjacent contacts. It then *automatically* widens the track spacing from this contact-to-geometry spacing to a contact-to-contact spacing. In the CMOS PW case, this parameter is 9 lambda:

```
LAYER2_GRIDSPACING = LAYER2_MINWIDTH
           + MAX (LAYER1_MINSPACING, LAYER2_MINSPACING)
           + (CONT_SIZE - LAYER2_MINWIDTH)/2
        = 3 + MAX(3,4) + (7-3)/2
        = 9.
```

The design rule file must be included to run any router. If the HAWKROUTERS file is not included, or a routing parameter is missing, the user is notified and the route is aborted.

CHAPTER 4

Programming in the Hawk/Squid Environment

In Hawk and Squid, a skeleton framework is provided for client programmers to add features to the package. The package was designed to make the process of adding a client tool as easy as possible, by providing clean database access routines such as those used in the pitch change command. A client programmer, therefore, does not need to worry about parsing a Squid file to stage it into memory or explicitly walking through a linked list of structures. The interface routines mask such details. Pointers are passed to the data access routines, returning handles to Squid objects such as cells, geometries, and terminals. This "open system" approach enables client programmers to interface other CAD tools like the channel router YACR2 into Hawk and Squid without having to learn the internal details of both packages.

In this chapter, several aspects of programming in the Hawk/Squid environment are presented. First, the pitch.c program is explained in detail. This program may be used as a template for adding client commands to Hawk. The Hawk and Squid routines are summarized and then hints for debugging in the Hawk/Squid environment are given. The pitch code is included in Appendix A for reference. Then, the YACR2/Hawk interface is reviewed. Many of the descriptions in this chapter are intentionally quite specific to the UC Berkeley environment since this chapter is intended as a detailed guide for Berkeley students. In Appendix C, a glossary of programs and files is included to help users locate files and programs described in this section.

4.1. Flow of the router. Pitch Change.

The general flow of a router is:

(1) Invoke command.
(2) Read design rules.
(3) Input user points.
(4) Extract information from Squid database.
(5) Solve routing problem.
(6) Report diagnostics.
(7) Convert solution to Squid and redisplay Hawk.

4.1.1. Invoking a command.

Menu Selection. When a routing command is selected from the routing menu, a symbolic view of the routing sub-menu identifies the code associated with the menu-selection. This menu view is named "symbolic". In Figure 4.1, the portion of that file pertaining to the pitch change command is illustrated.

It is created using the "src" view of the routing menu as input and running the "menuview" program. If the selected routine is not currently linked to the Hawk process upon invocation, the appropriate object code is linked dynamically to the Hawk process. In the case of the **pitchChange** menu selection, the pitchChange() routine in pitch object file is invoked. In execution, **HACurrentWindo()** returns the current window selected in Hawk. The structure HAWindo identifies the current cell being edited in Hawk.

```
PUT LABEL 5 "hawkHelp" STRING " Invoke a pitch change."
MK LABEL 4 FRAME LAYER "hawk" JU "" FT "std" LABEL "pitchChange"
        PS 100 ANGLE 0 POS 0 -400
PUT LABEL 4 "hawk.o" STRING "Pitch"
PUT LABEL 4 "hawkRoutine" STRING "pitchChange"
PUT LABEL 4 "hawkHelp" STRING " Invoke a pitch change."
```

Figure 4.1. Menu view of Pitch change command.

4.1.2. Reading Design Rules.

As described in the ".cadrc" section of this report, the design rules are specified in the HAWKROUTERS section of the user's ".cadrc" file. This enables the user to invoke the routers with the appropriate technology. In the pitch code, **RouterCadrcRead()** calls the routines InitCadrc(), NameCadrc(), and KeyCadrc() with the parameters NAME and VALUE to input the LAYER1_NAME, LAYER2_NAME, LAYER1_MINSPACING and LAYER2_MINSPACING parameters. In these routines, the user's ".cadrc" file is opened, the HAWKROUTERS section is found and the design rules parameters are read. If any of these parameters are missing from the ".cadrc" file, the missing parameter will be written to the "route_error" file and reported to the user.

In the ".cadrc" routine, design rule values are obtained from in two places, first from the general "/cad/.cadrc" file, then from the user's ".cadrc" file. Any default values in the "/cad/.cadrc" file are overwritten by the user's personal "~/.cadrc". If there are errors in the .cadrc file because routing parameters have not been properly specified or have been left out, the route is aborted. If there are no errors, the route proceeds by prompting the user to input control points.

4.1.3. Get User Control Points.

For interactive routines, it is important to capture the designer's intent. In the pitch change command, it is assumed that the designer has laid out a bus of geometries that he wants to extend on the same layer. These geometries are colinear and terminate roughly in the same region. The user is prompted to point to the bottom of the geometries that he wants to extend and point to the top. Essentially, he is prompted to draw a line that "cuts" the bus. The third point selection sets where the bottom geometry should extend to. In detail, the user is prompted through the **HATypescript()** routine that prints a string in the Hawk typescript window. Hawk listens through **HAListen()** to find out if the user hit escape, HAESC, to exit from the

routine, or selected another menu item, **HAMenuSelectionP()**, instead of pointing to that third and final point. If the user did point, then the point is retrieved through **HACursorPositionL()**. Figure 4.2 contains a code fragment of inputing a user point.

Checking for another menu selection enables the user to invoke another Hawk command, then return to the routing routine. For example, the user may invoke **HAZoomin** to zoom into a routing control point in a Hawk window, then return to the routing routing by escaping from the zooming routine and pointing to the routing control point.

The pitch change command extends geometries at their current widths. After pointing to the third point, the user types the new uniform spacing between geometries. The pitch solution is determined from the final point, the widths of the geometries in the cable, and the new geometry pitch.

4.1.4. Squid data retrieval.

After the control points have been successfully specified, it is necessary to retrieve the geometries from the Squid database that the user wants to extend. A database query routine is called to find geometries on either the LAYER1_NAME or LAYER2_NAME layers. The region is the area underneath the line that the user cuts across the cable of geometries. Details of Squid's private database are invisible to the user; communication is performed through the query routines.

```
HATypescript(sqTrue,"POINT to the bottom (left) of the cable.");
HAListen();
if (HAKeyTyped() == HAESC || HAMenuSelectionP())
   return(sqFalse);
bottom = *HACursorPositionL();
```

Figure 4.2 Input Control Point.

Squid's special geometry generator[9] retrieves pertinent geometries from the Squid database. In SQSpecialBeginGen(), the user specifies the area and depth of search as well as the type of geometries to be included in the search. The geometries are returned one by one in the SQSpecialGen() routine. Figure 4.3 contains an excerpt from the pitch code.

The SQSpecialBeginGen() routine initializes the geometry query. The necessary arguments are *area*, *mask*, *depth*, and generatorID pointer.

```
area = initLineBB;
mask[0][0] = SQLayerNameToNumber(LAYER1_NAME);
mask[0][1] = (int)sqActiveArea;
mask[1][0] = SQLayerNameToNumber(LAYER2_NAME);
mask[1][1] = (int)sqActiveArea;
mask[2][0] = SQLayerNameToNumber(LAYER1_NAME);
mask[2][1] = (int)sqInterconnect;
mask[3][0] = SQLayerNameToNumber(LAYER2_NAME);
mask[3][1] = (int)sqInterconnect;
mask[4][0] = -1;

if((int)SQSpecialBeginGen(area,mask,SQMAXDEPTH,&genID)<0) {
 HATypescript(sqTrue,SQDiagnostic());
 return(-1);
}
for(;;) {
 status = SQSpecialGen(genID, &geo, integerPath, nIntegerPath,
          NULL,0,instIDs,nInstIDs);
 if (status == sqEndGen) break;
 if ((int)status < 0) {
  HATypescript(sqTrue,SQDiagnostic());
  return(-1);
 }
 if(geo.geoType == sqRect) {
  extractRectInfo(geo);
 } else if (geo.geoType == sqLine) {
  extractLineInfo(geo);
 }
}
```

Figure 4.3 Squid Geometry Retrieval.

The *area* argument specifies the rectangular area in the Squid circuit view that the user wants to query. In the channel router commands, terminals along the channel perimeter are extracted. The search area is a rectangle. In the pitch change command, the line that the user cuts across the initial geometries serves as the degenerate rectangle area to be searched. All geometries intersecting the cut line and satisfying the mask requirements are extracted from the Squid physical view.

The *mask* parameter specifies what geometries the user is searching for. The mask argument is a two dimensional array. The first dimension increments the different type of geometries that are queried. The second dimension's zero-th element is the layer being searched; its first element specifies the geometry's function. For example, in the channel router commands, necessary geometries are of function sqTermArea (terminals) on layers LAYER1_NAME and LAYER2_NAME. In the pitch change command, pertinent geometries are of function sqActiveArea and sqInterconnect on either LAYER1_NAME or LAYER2_NAME. The mask array is terminated with the -1 as the 0th element of the second dimension of the last mask entry.

The *depth* parameter sets the depth of search in the design hierarchy. A depth of one searches only the current level of hierarchy. A depth of two searches the current level of hierarchy and all instances called in the top level of the current hierarchy. The depth SQMAXDEPTH searches all the way down the design hierarchy, provided that all instances called in the hierarchy have been staged into Squid's virtual memory. If a cell is only represented by its bounding box in the Hawk viewport, then the instance has not been staged into memory and the cell will not be searched in the geometry query. The channel router searches only the top level of the hierarchy. The pitch change command searches all the way down the hierarchy (SQMAXDEPTH).

The generator parameter is a pointer to a SQID (SQuidIDentifier). The routine passes this pointer back to the top of the list of geometries found that match the area.

mask and depth constraints. This parameter is the first one used in the SQSpecialGen() routine.

The SQSpecialGen() routine marches through the list of geometries satisfying the specifications set in the SQSpecialBeginGen() routine. As shown in the routine extractGeos() in the pitch code, the generator resides in a "forever" loop (for(;;)) which terminates at the end of the list of geometries returned by SQSpecialBeginGen(). The special generator returns a status variable. The status value sqEndGen signals the end of the list of geometries found in the geometry query and breaks the "forever" loop.

The first argument in the SQSpecialGen() is the genID returned in the SQSpecialGen(). The second argument is a pointer to a Squid geometry structure, SQGeo. The variable integerPath[] is the array of SQIntegerPoint's in a geometry's path in the case when a geometry is a line or polygon. The generator fills the integerPath[] array with the control path points if the geometry is a line or polygon. The variable nIntegerPath is the length of the array integerPath[]. In the pitch command, the integerPath variable is set to twelve, because that is the estimated upper bound on a line's length in vector points. A polygon may easily have more points that twelve, but the pitch command is only interested in geometries of shape rectangle or line.

The NULL, 0 parameters do nothing but must be included. The instIDs[] argument is an array of SQID's which points down the cell hierarchy to the geometry found. When a geometry is found, this tells the user which instance the geometry is from. The length on the array is given in nInstIDs, which is set to SQMAXDEPTH.

4.1.5. Solving the routing problem.

This section of the code is specific to the pitch change code and is not particularly helpful toward learning about Squid: most of the code was omitted and only one interesting routine will be noted. Geometries are retrieved in the extractGeos() routine of the pitch code. Geometries at the same level of hierarchy and on the same layer

may be superimposed in Squid. That is, there is no merging of coincident geometries. For example, if five rectangles on the metal layer are resident in a physical cell view between the coordinates (0,0) and (100,100), then the geometry generator will return all five geometries. The user must check for geometry coincidence to avoid redundancy and errors in an applications solution.

4.1.6. Reporting Diagnostics.

In the course of the pitch command, errors and diagnostics are written to the route_error file with the **logError()** routine. The route may not be completed if there is not enough room to perform the pitch change. Before the route is redisplayed or aborted, the **route_error** file is printed line by line in the Hawk's typescript window. If the route is completed successfully, then there may be no diagnostics. If unsuccessful, the **route_error** routine will tell the Hawk user what prevented the pitch change. The **route_error** file remains in the user's Hawk directory for later reference.

4.1.7. Convert the solution to Squid.

The final portion of the pitch change routine converts the routing solution to Squid geometries. The pitch-changed geometries are stored in their own master file. An instance of that cell is placed in the Hawk cell being routed. When converting the solution to Squid, it is necessary to:

(a) open routed cell
(b) place geometries in cell
(c) save routed cell
(d) place instance of cell in Hawk view being routed

4.1.7.1. Open routed cell.

The user is prompted to input a name for the routing cell. A new Squid view stack is created to work independently of the Hawk viewport. This view stack is created with the call:

SQPushViewStk (SQCreateViewStk()).

Once this view stack has been created, the cell named by the user is entered in the routine:

SQ(sqPush, sqView, sqCircuit, view, &stream).

This establishes the current Squid view being edited as the routing cell. The cell has now been opened on its own view stack and is ready for cell elements. Figure 4.4 contains a code fragment which illustrates this process.

4.1.7.2. Place geometries in cell.

Lines are created in the pitch change solution. Lines are defined by a line width and a path of jog points. Refer to the structure SQGeo of geoType sqLine. Once the SQGeo structure has been filled in, the geometry is added to the Squid view of the routed cell with the call:

SQ(sqCreate, sqGeo, &currLine).

This routine is called successively to create many lines.

```
SQView          view;

view.cell = instanceCell;
view.view = instanceView;
view.mode = "w";

SQPushViewStk(SQCreateViewStk);
if ((int) SQ(sqPush, sqView, sqCircuit, view, &stream) < 0) {
    HATypescript(sqTrue, SQDiagnostic());
    return (-1);
}
```

Figure 4.4. Pushing into a cell.

```
currLine.geoType = sqLine;
currLine.function = sqActiveArea;
currLine.def.line.nPath = PTSINLINE;
currLine.filledP = sqTrue;
currLine.layer = cableLayer;            /* LAYER1_NAME or LAYER2_NAME */

for (i=0;i<cableWidth;++i) {            /* for each line in solution */

   currLine.def.line.width = initPts[i].width;        /* assign line width */
   currLine.def.line.path = linePts;                  /* assign jog points */

   if ((int) SQ(sqCreate, sqGeo, &currLine) < 0) {   /*create line in squid*/
      HATypescript(sqTrue, SQDiagnostic());
      return(-1);
   }
}
```

Figure 4.5. Creating geometries.

4.1.7.3. Save routed cell.

Once the routed view is filled, it is ready to be saved. This is performed using the Squid call:

SQ(sqSave, sqView, sqFast, view).

View is the structure which designates which view is to be saved. In this case, it is the view of the routed solution. After saving, the view is popped from the view stack and staged into memory. Popping removes the cell from the stack of cells being edited. Staging the view keeps the geometries and solution in current Squid data space. This is done with the command:

SQ(sqPop, sqView, sqStage).

The Hawk view being routed is returned to by popping the created view stack. Figure 4.6 contains a the code excerpt.

```
if ((int) SQ(sqSave. sqView. sqFast. view) <0) {
  HATypescript(sqTrue. SQDiagnostic());
  return(-1);
}
if ((int) SQ(sqPop. sqView. sqStage) <0) {
  HATypescript(sqTrue. SQDiagnostic());
  return(-1);
}
SQPopViewStk().
```

Figure 4.6. Saving and popping view.

4.1.7.4. Place instance of route in Hawk view.

The routed cell has been created. It is necessary to place that cell in the current Hawk view. Back in the Hawk circuit view, an instance of the routed master is placed in the current Hawk view with the call in Figure 4.7. The bounding box of this created instance is retrieved with the call:

SQ(sqGet, sqInst, &inst).

To save Hawk redisplay time, only the changed parts of the Hawk screen are

```
inst.masterCell = instanceCell;
inst.masterView = instanceView;
inst.name = "";
sprintf(string,"T 0 0");
inst.cif = string;

if ((int) SQ(sqCreate. sqInst. &inst) <0)
   HATypescript(sqTrue.SQDiagnostic());
   return(-1);
```

Figure 4.7. Creating the routed cell in Squid.

redisplayed. When the routine is first invoked, the routine:

SQEmptyBBSet(HAChangedRect())

is called to empty out the set of bounding box areas that have been altered in the current view. In the pitch change case, the only changed area is the area enclosed by the bounding box of the routed instance. This area is added to the bounding box set of changed areas in the Hawk view with the routine:

SQAddToBBSet(&bb, HAChangedRect()).

The cell's Hawk view is redisplayed as shown in Figure 4.8. The pitch change is now complete.

4.2. Debugging Hawk/Squid Routines.

It is much easier to write and debug Hawk routines at an ASCII terminal than at a graphics terminal. Running a code debugger like dbx is impossible in the middle of graphics routines. Consequently, when debugging new client code, it is advisable to "fake" Hawk routines at an ASCII terminal. In the pitch template, any code that is within **#ifdef** BATCH . . . **#endif** delimiters is the code necessary to simulate Hawk

```
SQPushViewStk(windo.editStk);      /* push into current Hawk window */
i = rePitch(&bb);                  /* call pitch routines, return instance's bb */
SQPopViewStk();                    /* pop Hawk view stack */
if (i<0) {
  HATypescript(sqTrue, "Unable to do pitch change");
} else {
  SQAddToBBSet(&bb, HAChangedRect());
          /* add bounding box to list of changed geometries */
}
/* redisplay Hawk */
HADisplayView(windo.windoID, *HAChangedRect(), sqFalse, sqFalse)
```

Figure 4.8 Updating Hawk.

routines. Code within **#ifndef** BATCH . . . **#endif** delimiters is the code unique to Hawk.

The **main()** routine in the batch version of the pitch code simulates routines called in Hawk initialization and termination. When Hawk is first invoked, **SQBegin()** is called to initialize Squid. Then, a Hawk view stack is created and the current Hawk cell being edited is pushed into. By selecting the **defined** menu selection in the WINDOWS sub-menu of Hawk, the user is opening and staging all instances in the current Hawk view. The batch version of the code simulates the staging by calling **SQOpenMasters()**, which opens and stages the entire Hawk cell. Then client packages may be invoked. The client package in the pitch code is **rePitch()**. After successful completion, **rePitch()** returns, and the current window is saved and popped from the Hawk view stack. The view stack is then popped and **SQEnd()** is called to power down Squid. A fragment from the pitch code is contained in Figure 4.9.

Several Hawk routines are called within the body of the client pitch change code that must be faked when debugging at a terminal. Example routines are **HACursorPositionL()**, **HAKeyTyped()**, and **HATypescript()**. In normal Hawk mode, these routines are linked with the client code when the routing routine is dynamically linked to Hawk. These Hawk routines must be faked at an ASCII terminal because only Squid routines are linked to the terminal version of the code. Refer to the pitch code for example substitutions for these Hawk routines. For example, **HACursorPositionL()**, which gets coordinates that the user points to in the Hawk typescript window, is rewritten to *scanf* two integers from terminal keyboard input. **HADisplayView()**, which prints a string in Hawk's typescript window, does nothing at an ASCII terminal. **HATypescript()** performs a *printf* of the string that is passed to it in terminal mode.

A terminal version of the pitch change command may be compiled with BATCH defined. It may be run at an ASCII terminal by typing "termPitch *cellname cellview*" in

```
main(argc, argv)
int argc;
char *argv[];
{
    FILE        *stream;
    SQBB        bb;

    if (argc != 3) {
        printf("usage: re-pitch cellname and cellview0);
        exit(-1);
    }
    if ((int) SQBegin()<0) {
        HATypescript(sqTrue,SQDiagnostic());
        exit(-1);
    }
    changedRect.l = changedRect.b = changedRect.t = changedRect.r = 0;
    bb.l = bb.b = bb.t = bb.r = 0;

    /* Fake out a current window -- push into view given on command line */
    currentWindow.view.view = argv[2];
    currentWindow.view.cell = argv[1];
    currentWindow.view.mode = "w";
    currentWindow.windoID = 3;
    SQPushViewStk(currentWindow.editStk = currentWindow.displayStk =
        SQCreateViewStk());
    if (SQ(sqPush, sqView, sqCircuit, currentWindow.view, &stream) != sqOK) {
        HATypescript(sqTrue,SQDiagnostic());
        exit(-1);
    }
    if (rePitch(&bb) <0) {
      exit(-1);
    }
    /* Fake out popping current window */
    if ((int) SQ(sqSave,sqView,sqFast,currentWindow.view) <0) {
        HATypescript(sqTrue,SQDiagnostic());
    }
    if ((int) SQ(sqPop, sqView, sqUnstage) <0) {
        HATypescript(sqTrue,SQDiagnostic());
    }
    SQPopViewStk();
    if ((int) SQEnd() <0) {
        HATypescript(sqTrue,SQDiagnostic());
    }

}
```

Figure 4.9. Faking Hawk initialization and termination.

the directory holding the cellname directory and the terminal version of pitch executable code. termPitch. Control points are entered as (x,y) coordinates. Squid creates the pitch change master cell. and calls an instance of that master in the physical cell.

Providing "dummy" Hawk routines is valuable when debugging new code. Later in the code's development. when Hawk is introduced. other debugging tricks are useful. Debugging information may be recorded by using the **HAError()** or **HATypescript()** routine. For example. **HAError()** prints a string in the user's hawklog file and **HATypescript()** prints a string in Hawk's typescript window.

4.3. YACR2/Hawk Interface.

4.3.1. Operation and Description.

Currently. the channel routing interface in Hawk is set up to interface to any channel router that can:

(1) accept .chan.route.in as the input created by the interface engine and.

(2) output .chan.route.out as needed by the engine.

The contents of these two files are described in the channel routing section of Chapter 3. The default channel router is YACR2 as set in the interface with the assignment chan_router = "yacr -H". The "-H" flag alerts YACR2's input routines that the Hawk YACR2 input is being used. not the default YACR2 input. YACR2 is called by the interface with a system call. Overwriting YACR2 as the default channel router may be done by creating a menu selection routine in the routing sub-menu that overwrites the interface's static variable "chan_router". Here is the routine (invoked by selecting the menu selection Foo) that overwrites "yacr -H" with the channel router foo:

```
void Foo()
{
  chan_router = "foo";
}
```

Reselecting YACR2 as interface's channel router is done with the menu selection. **Yacr()**:

```
void Yacr()
{
  chan_router = "yacr -H";
}
```

If other channel routers were added to Hawk, the user could try several of the algorithms and choose the most efficient route. The only restriction to interfacing other routing algorithms is that they must be able to accept .chan.route.in and .chan.route.out as input and output, respectively.

4.3.2. Suggestions for interface to YACR2.

Here are some suggestions for the Hawk/YACR2 interface:

(1) Extend YACR2 and its interface to avoid obstacles. This would be useful when pre-placing power, ground, and critical nets. YACR2 and its interface would have to be rewritten to accommodate this. The preprocessor would report preplaced nets as input to the obstacle avoiding channel router.

(2) Turn off metal maximization in YACR2/Hawk interface. If YACR2 were used as the initial router, but signals were to be added to the channel later, then it would be helpful to be able to turn off metal maximization. Metal maximization can block later routes because the row layer bends around onto the column layer. The absence of metal maximization mandates that the channel observe the routing policy that column geometries are on a different layer from the layer of the row geometries. The absence of metal maximization and the presence of extra tracks in a channel makes it easy to add routes to a channel.

With the use of the interface routines, any number of CAD tools may be integrated into the Hawk/Squid design framework. This "open system" CAD approach enables the framework to grow with the changing developments and needs of IC design.

CHAPTER 5

Evaluation of the Hawk/Squid Programming Environment

The Hawk/Squid system is designed to provide a unified framework for IC design and development. The Squid database maintains and manipulates hierarchical circuit designs, supporting multiple circuit views, such as the simulation, schematic, documentation, and physical views of a design. The Hawk graphics editor supports overlapping multiple windows for editing either multiple circuits or multiple views of a circuit. In this project, interactive routers were developed to enhance the Hawk graphics editor. The development of the routers proceeded in close conjunction with an IC designer working in the framework. While the focus of the routers was primarily on physical circuit views, their development provided insights into the package in general.

5.1. Documentation.

Unfortunately, some useful Hawk and Squid functions are not documented in the package literature and may be found only by looking through the program source. This problem is being solved incrementally. As client programmers learn about useful routines, they are added into the Squid and Hawk documentation. One such routine is **SQOpenMasters()** which stages the masters of all instances called in a design hierarchy. This routine is necessary when using Squid *without* Hawk; Hawk menu selections such as **defined** currently handle the staging of a design that **SQOpenMasters()** performs without Hawk. A useful, but undocumented Hawk routine is **HALastCursorPositionL()** which returns the last point that Hawk user pointed to. This is helpful in Hawk applications, but only if the client programmer knows that it is available.

5.2. Learning to program in Hawk and Squid.

A good way to start programming in Hawk and Squid is to read the package documentation. While the documentation serves as a good reference, it is not always apparent how the Hawk and Squid routines should be used. An efficient way to get a feel for programming in Hawk and Squid is to study a working client program. In Chapter 4 of this report, a routing routine is used as a template for Hawk and Squid programming and the Hawk and Squid calls made by a specific router are described.

5.3. Debugging Hawk and Squid routines.

Another difficulty in programming in Hawk and Squid is the program debugging process. Often programs fail due to a misunderstanding of the Hawk or Squid routine that is called. Here again, looking at a working template helps the user understand the required parameters. Hints for debugging Hawk and Squid routines were described in Chapter 4. Hawk client programs are developed most easily when they are debugged initially at an ASCII terminal.

5.4. The Path Mechanism.

The *path* mechanism enables the user to work in a current project directory and access Squid files relative to that directory. This mechanism has some difficulties when working on a chip design that is resident in a number of user's directories. It is not always clear, as a Hawk user, where a file will be created when rooted in a remote directory. There should be a way to override the relative path mechanism and access files with a "~ user" root, or call a routine "openDirectory" with a desired file name passed to it. While this is possible with some Squid "sqPush" and "sqCreate" routines, a "~ " is not always expanded as anticipated, resulting in files being created in unexpected places.

5.5. Exploiting Properties and Parameters.

Squid objects may have properties attached to them. While properties are not used in the routing routines, this capability could be exploited in client routines to ease object identification and manipulation. For example, a circuit compacting routine could check to see if a cell instance were eligible for compaction before trying to perform compaction. As another example, a cell instance could have a property associated with hierarchical design rule checking: if a cell had been design rule checked previously, then the instance would have the property "designRuleChecked" equal to "TRUE". Only peripheral geometries would have to be checked in the design rule checking routine and the property, "peripheralGeo", could identify those geometries in the instance. If an instance had been altered, then the "designRuleChecked" property could be changed to "FALSE". The next time that a design rule checker was run on the circuit, that instance would be re-checked for design rule errors, and the instance's new peripheral geometries could be identified.

5.6. Wish list of features for Hawk and Squid.

Other features might be useful in the package. For example, it is useful to be able to place an instance of a master and then "flatten" that instance into the hierarchy. This would enable a designer to quickly place template instances of a cell for altering and customizing the design. Additionally, to speed up the Hawk initial display and re-display of a design, coincident Squid geometries at the same level of the hierarchy could be merged at some step in Squid's power-up or power-down. Currently, merging of coincident geometries may be done explicitly in Hawk with the **merge** command, but the geometries must be added to Hawk's selected set of geometries. Of course, this process could be turned off for design styles which depend on particular geometries places by the user.

There is a mechanism in place to "frame" the physical cells of a design. The framer replaces a detailed physical representation of a cell with a symbolic representation of that cell, called *protection frames*[10]. The framer can be programmed to return the peripheral extents of the cell's metal and polysilicon layers, or it alternatively can be programmed to return the extents of all layers in a cell along with the terminal information. The "programmability" of the framer needs to be extended. The framer currently does not enable the user to flag the inclusion or exclusion of terminal information; terminal information is retained from all levels of hierarchy. When used by designers, the terminal information may be of no interest; only the extents of the physical geometries may be of interest. It would help to be able to eliminate the terminal search in this case. For routing purposes, it is useful to request that only peripheral or highest level terminals be included in the framed representation of a cell for use in a global router.

From a design perspective, the Hawk and Squid package would strengthen considerably if the package's schematic capture capabilities could be enhanced. Currently, Hawk is geared toward the physical view of circuit design. Many design simulation and verification tools require as input the circuit description derived from the physical view. It is more expedient to verify a circuit's functionality from a schematic description, because the designer is not yet committed to a physical description; the schematic description can change more easily than a physical description. Later, when functionality is verified, synthesis tools can assemble the physical mask description, assuming that adequate synthesis tools exist. With a sufficient schematic capture mechanism and net list generator, IC designers could verify their designs earlier in the design process and change circuit descriptions quickly without altering the physical mask description. This introduces the need for a consistency checker between the the schematic and physical representations of a circuit. Both physical and schematic circuit representations could build the net list of a circuit for cross check.

A useful connectivity tool to add to the routing toolbox for physical views would be one that enabled a user to find out what was connected to a geometry by pointing to a geometry and selecting a "connected?" menu entry. The "connected?" routine could highlight all of the geometries connected to that geometry even those connected through through layer contacts.

Overall, programming in the Hawk and Squid environment is easy once the system is understood; the initial familiarization, however, is difficult. Efforts are being made by all of the users of the package to ease the problem of learning about Hawk and Squid by improving the documentation and fixing bugs. From the user's perspective, the continued addition and integration of new tools to the framework insures that the package will continue to improve.

CHAPTER 6

Conclusion

In this project report, the steps taken to install an incremental routing system together for custom IC layout are described. The tools were developed to solve the immediate needs of the local IC designers. By providing interactive assistance, the guided routing toolbox eases the bottleneck that routing imposes on the design verification loop. The routines help the designer complete his layout so that the circuit may be simulated and verified. The Hawk/Squid package was designed to enable CAD tool builders to integrate tools into the package easily. Any problems encountered in tool integration are addressed and solved by the users of the package. As other tools are tied into Hawk/Squid, the IC design framework will improve.

In this project, the effects of a close interaction between the CAD tool developer and the tool users was explored. The most successful tools in the routing toolbox resulted from the dedicated cooperation and communication between the CAD tool designers and their "customers," the IC designers. The pitch and LTurn commands solve a simple, recurrent problem. These tools were created in direct response to the SOAR project's bus routing problem and were examples of the IC designer driving the CAD tool designer. The channel router, on the other hand, is an example of the CAD tool designer driving the IC designer. The users of the toolbox did not appreciate the utility of a channel router for custom design until the YACR2 interface was tried and proven effective.

As VLSI designs continue to become larger and more dense, designers will continue to need the guided routing capabilities of the routing toolbox. The routers have been designed to accommodate the changing specifications of IC technologies while satisfying

routing needs common to both structured-custom and semi-custom design. The routers provide the flexibility to quickly alter routing at the floorplan level.

REFERENCES

[1] Keller, K., "An Electric Circuit CAD Framework," Electronics Research Laboratory, University of California, Berkeley, June, 1984, Memo# UCB/ERL M84/54.

[2] Marino, Chris, "Smalltalk on a RISC - CMOS Implementation", MS Report, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, May, 1985.

[3] Patterson, David, "Proceedings of CS290R, SmallTalk on a RISC - Architectural Investigations", Univ. of Cal., Berkeley, Dept. of CS, April, 1983.

[4] Reed, James, "YACR2. Yet Another Channel Router 2", Electronics Research Laboratory Memo# UCB/ERL M85/12, Univ. of Cal., Berkeley, Feb., 1985.

[5] Keller, K., "Tutorial For Hawk Users," appended to [1].

[6] Beyers, J. W., "A 32-bit VLSI CPU Chip," 1981 IEEE ISSCC Digest of Technical Papers", New York, NY, Feb., 1981, P. 104-105.

[7] Mead, C., and L. Conway, "Introduction to VLSI Systems, New York, Addison-Wesley, 1980.

[8] Sangiovanni-Vincentelli, A., Santomauro, M., and Reed, J., "A New Gridless Channel Router: Yet Another Channel Router the Second (YACR-II)", Proceedings of ICCAD, Santa Clara, CA., Nov. 1984., p. 72-75.

[9] Keller, K., "Manual for the Squid Package," Internal Memorandum, CAD Group, University of California, Berkeley.

[10] Keller, K. and A. Newton, "A Symbolic Design System for Integrated Circuits", Proceedings of 19th Design Automation Conference, Las Vegas, Nev., June, 1982, p. 460-466.

[11] Mayo, R., Ousterhout, J., and Scott, W., ed., "1983 VLSI Tools. Selected Works by the Original Artists.", Univ. of Cal., Berkeley, Dept. of C.S., Report #UCB/CSD 83/115, March, 1983.

APPENDIX A

Code For The Pitch Change.

```
#include <stdio.h>
#include <cad/sq.h>
#include <cad/ha.h>

#ifdef BATCH

#include "tech.h"

#else

#include "extern.h"
FILE      *fperr;

#endif BATCH

#define MAX(dragon,eagle) ((dragon) > (eagle) ? (dragon) : (eagle))
#define MIN(dragon,eagle) ((dragon) < (eagle) ? (dragon) : (eagle))
#define HORIZONTAL 'H'
#define VERTICAL 'V'
#define MAXNUMINCABLE 100
#define PTSINLINE 4

struct initialPt{
  int             topOrRight;
  int             bottomOrLeft;
  int             edge;
  int             width;
  char            *geoLayer;
};
static    struct initialPt          initPts[MAXNUMINCABLE];
static  char              HorV;
static  int               minSpace;
static  int               minWidth;
static  int               cableWidth = 0;
static  char              *cableLayer;

static void openMaster(s)
char *s;
{
HATypescript(sqTrue,s);
}
```

getInitPts

```
SQBool
  getInitPts(initLineBB,finalLinePts,pitch)
SQBB              *initLineBB;
SQIntegerPoint    finalLinePts[];
int               *pitch;
{
 HAWindo          windo;
 SQIntegerPoint top,bottom;
 SQIntegerPoint doublePoint[2];
 int              dx,dy;


 windo = HACurrentWindo();

 HATypescript(sqFalse,"POINT to the bottom(left) of the cable ");
 HAListen();
 if(HAKeyTyped() == HAESC || HAMenuSelectionP())
      return(sqFalse);
 bottom = *(HACursorPositionL());


 HATypescript(sqFalse,"POINT to the top(right) of the cable to be re-pitched");
```

```
  HAListen();
  if(HAKeyTyped() == HAESC || HAMenuSelectionP())
       return(sqFalse);
  top = *(HACursorPositionL());


  dx = ABS(top.x - bottom.x);
  dy = ABS(top.y - bottom.y);

  if (dx < dy) { /* draw vertical line from bottom to top */
    if (bottom.y < top.y) {
      doublePoint[0] = bottom;
      doublePoint[1].x = bottom.x;
      doublePoint[1].y = top.y;
    } else {
      doublePoint[0] = top;
      doublePoint[1].x = top.x;
      doublePoint[1].y = bottom.y;
    }
    HorV = VERTICAL;
  } else /* (dx < dy) */ { /* draw horizontal line from left to right */
    if (bottom.x < top.x) {
      doublePoint[0] = bottom;
      doublePoint[1].x = top.x;
      doublePoint[1].y = bottom.y;
    } else {
      doublePoint[0] = top;
      doublePoint[1].x = bottom.x;
      doublePoint[1].y = top.y;
    }
    HorV = HORIZONTAL;
  }
  initLineBB->l = doublePoint[0].x;
  initLineBB->b = doublePoint[0].y;
  initLineBB->r = doublePoint[1].x;
  initLineBB->t = doublePoint[1].y;

  HATypescript(sqFalse,"POINT to where the bottom(left) of the cable is to go");
  HAListen();
  if(HAKeyTyped() == HAESC || HAMenuSelectionP())
       return(sqFalse);
  finalLinePts[0] = *(HACursorPositionL());

  HATypescript(sqFalse,"Please TYPE in the new pitch of the cable in LAMBDAS.");
  sscanf(HAKeyboard(),"%d",pitch);
  return(sqTrue);
}


/*******************************************************************/
/*  extractRectInfo gets the pertinent edge coordinates of a rectangle
    and loads them into the initPts array.  Geometry width and edge
    coordinates are stored for later use when extending the cable
    element in the pitch change.

    Called By: extractGeos.
    Calls: none.
    Returns: filled initPts array element at the index count.
 */
/*******************************************************************/
int
  extractRectInfo(geo,count,finalPt,initLineBB)
```

extractRectInfo

```
SQGeo              geo;              /* geometry in initial cable */
int                count;            /* index in the initPts array */
SQIntegerPoint     finalPt;          /* point to determine cable direction */
SQBB               initLineBB;       /* line cutting initial cable */
{
 if (HorV == HORIZONTAL){
     initPts[count-1].topOrRight = geo.bb.r;
     initPts[count-1].bottomOrLeft = geo.bb.l;
     initPts[count-1].width = ABS(geo.bb.r - geo.bb.l);
     if (finalPt.y > initLineBB.t) {
         initPts[count-1].edge = geo.bb.t;
     } else {
         initPts[count-1].edge = geo.bb.b;
     }
     return(0);
 } else if (HorV == VERTICAL) {
     initPts[count-1].topOrRight = geo.bb.t;
     initPts[count-1].bottomOrLeft = geo.bb.b;
     initPts[count-1].width = ABS(geo.bb.t - geo.bb.b);
     if (finalPt.x > initLineBB.r) {
         initPts[count-1].edge = geo.bb.r;
     } else {
         initPts[count-1].edge = geo.bb.l;
     }
     return(0);
 }
 return(-1);
}

/********************************************************************/
/*  extractLineInfo gets the pertinent edge coordinates of a line
    and loads them into the initPts array.  Geometry width and edge
    coordinates are stored for later use when extending the cable
    element in the pitch change.
        Initial and final line segments in the line are tested to
    see which segment cuts the initial cable cut-line, and line
    information is extracted for the proper segment.

    Called By: extractGeos.
    Calls: none.
    Returns: filled initPts array element at the index count.
 */
/********************************************************************/
int
 extractLineInfo(geo,nIntegerPath,integerPath,count,initLineBB)
```

extractLineInfo

```
SQGeo              *geo;             /* geometry in the initial cable */
int                nIntegerPath;     /* number of points in the line geo */
SQIntegerPoint   integerPath[];      /* path of points in the line geo */
int                count;            /* index in the initPts array */
SQBB               initLineBB;       /* line cutting initial cable */
{
 SQIntegerPoint                segment[2][2];  /* array holding the 1st and last
                                                  line segments in the line */
 int                           i;

 integerPath = geo->def.line.path;
 segment[0][0] =  integerPath[0]; /* endpoint in line */
 segment[1][0] =  integerPath[1];
 segment[0][1] =  integerPath[geo->def.line.nPath - 1];
 segment[1][1] =  integerPath[geo->def.line.nPath - 2];
 initPts[count-1].width = geo->def.line.width;
for(i=0;i<=1;++i){
 if (HorV == VERTICAL) {
     if ((segment[0][i].y == segment[1][i].y)&&(segment[0][i].y >= initLineBB.b)
                                         && (segment[0][i].y <= initLineBB.t)){
```

```
        if(((segment[0][i].x  <=initLineBB.l)&&(segment[1][i].x  >= initLineBB.r))
              ||((segment[1][i].x  <= initLineBB.l)
                    && (segment[0][i].x  >= initLineBB.r))){
           initPts[count-1].bottomOrLeft = segment[0][i].y - geo->def.line.width/2;
           initPts[count-1].topOrRight    = segment[0][i].y + geo->def.line.width/2;
           initPts[count-1].edge          = segment[0][i].x;
           return(0);
        }
      }
  } else /* HorV = HORIZONTAL */ {
      if ((segment[0][i].x == segment[1][i].x)&&(segment[0][i].x >= initLineBB.l)
                                               && (segment[0][i].x <= initLineBB.r)){
       if(((segment[0][i].y  <= initLineBB.b)&&(segment[1][i].y  >= initLineBB.t))
             ||((segment[1][i].y  <= initLineBB.b)
                   &&(segment[0][i].y  >= initLineBB.t))){
           initPts[count-1].bottomOrLeft = segment[0][i].x - geo->def.line.width/2;
           initPts[count-1].topOrRight    = segment[0][i].x + geo->def.line.width/2;
           initPts[count-1].edge          = segment[0][i].y;
           return(0);
        }
      }
  }
}
return(-1);
}

/******************************************************************/
/*  eliminateGeos eliminates the elements of the initPts array that
    are on the wrong layer.

    Called By: checkLayer.
    Calls: none.
    Returns: initPts array with geometries of the wrong layer eliminated.
 */
/******************************************************************/
void
  eliminateGeos(elimLayer,totalCt,elimCt)                              eliminateGeos
char      *elimLayer;                 /* layer to eliminate from initPts array */
int       totalCt;                    /* total array count */
int       elimCt;                     /* number of array elements to eliminate */
{ }

/******************************************************************/
/*  checkLayer  calculates the layer count for each geometry layer
    represented in the cable.  The layer with the majority count
    becomes the cable layer.  eliminateGeos is called to eliminate
    the elements in the initPts array that are of the minority layer.
    Cable layer is established and design spacing rules may be set.
    Cable width is also set.

    Called By: extractGeos.
    Calls: eliminateGeos.
    Returns: cable Width, cableLayer, layer spacing, layer width,
             trimmed initPts array.
 */
/******************************************************************/
void
  checkLayer(count)                                                     checkLayer
int       count;            /* elements in the initPts array */
{ }

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

```
/*  sortInitPts  sorts the initPts array in ascending order by the
    bottomOrLeft element in the initialPt structure.
    Occasionally, 2 geometries may be superimposed.  These geometries
    must be merged into one geometry that the pitch change may be
    performed on.

    Called By: extractGeos.
    Calls: none.
    Returns: sorted initPts array  with merged geometries.
 */
/****************************************************************/
void
  sortInitPts()                                                   sortInitPts
{ }

/****************************************************************/
/*  extractGeos performs a squid database search to find all geometries
    intersecting the init line that the user specifies in setInitPts.
    This line cuts the initial cable section that is to be extanded with
    a pitch change.  Open Masters opens all instances and children of
    instances in the current view.  NOTE: openMasters opens only physical views.
    Geometries of Layer1 and Layer2 rectangles and lines are found and
    increment the "geos found" count.  extractRectInfo or extractLineInfo
    are called.  If geos are found, checkLayer is called to choose the
    layer that the pitch change will use.  SortInitPts is called to
    sort the cable element array. Cable width is returned.

    Called By: rePitch.
    Calls: extractRectInfo,extractLineInfo,checkLayer,sortInitPts.
    Returns: sorted initPts array  , cableWidth,cableLayer.
 */
/****************************************************************/
int
  extractGeos(initLineBB,finalPt)                                 extractGeos
SQBB                initLineBB;
SQIntegerPoint      finalPt;
{
 SQBB               area;
 SQID               genID;
 SQID               *instIDs[SQMAXDEPTH];
 HAWindo            windo;
 int                mask[5][2];
 int                nInstIDs = 100;
 int                count = 0;
 int                nIntegerPath = 12;
 SQStatus           status;
 SQIntegerPoint integerPath[12];
 SQGeo              geo;

 windo = HACurrentWindo();

#ifdef BATCH
 SQOpenMasters("physical",openMaster);
#endif
 area = initLineBB;
 mask[0][0] = SQLayerNameToNumber(LAYER1_NAME);
 mask[0][1] = (int)sqActiveArea;
 mask[1][0] = SQLayerNameToNumber(LAYER2_NAME);
 mask[1][1] = (int)sqActiveArea;
 mask[2][0] = SQLayerNameToNumber(LAYER1_NAME);
 mask[2][1] = (int)sqInterconnect;
 mask[3][0] = SQLayerNameToNumber(LAYER2_NAME);
 mask[3][1] = (int)sqInterconnect;
 mask[4][0] = -1;
```

```
  if((int)SQSpecialBeginGen(area,mask,SQMAXDEPTH,&genID)<0) {
        HATypescript(sqTrue,SQDiagnostic());
        return(-1);
  }
  for(;;) {
     status = SQSpecialGen(genID,&geo,integerPath,nIntegerPath,
                                        NULL,0,instIDs,&nInstIDs);
     if (status == sqEndGen) break;
     if ((int)status < 0) {
        HATypescript(sqTrue,SQDiagnostic());
        return(-1);
     }
     if (geo.geoType == sqRect) {
        ++count;
        if (extractRectInfo(geo,count,finalPt,initLineBB)<0){
             --count;
        } else{
             initPts[count - 1].geoLayer = geo.layer;
        }
     } else if (geo.geoType == sqLine) {
        ++count;
        if(extractLineInfo(&geo,nIntegerPath,integerPath,count,initLineBB)
                                                  <0){
             --count;
        } else{
             initPts[count - 1].geoLayer = geo.layer;
        }
     }
   }

  if (count == 0) {
     return(-1);
  } else {
    checkLayer(count);
    sortInitPts();
    return(cableWidth);
  }
}


/********************************************************************/
/*  setFinalLine sets the finalLinePts end points.  It starts at the
    user inputted point and calculates the endpoint of that final line
    in the pitch-changed bus.

    Called By: rePitch.
    Calls: none
    Returns: endPoints of cut line of final cable points, and the cable
                 sum of widths of the lines in the cable
 */
/********************************************************************/
int
  setFinalLine(finalLinePts,pitch)
SQIntegerPoint          finalLinePts[];
int                     pitch;
{
 int                    sumOfWidths = 0;
 int                    i;

 for (i=0;i<cableWidth;++i) {
   sumOfWidths += initPts[i].width;
 }
 if (HorV == HORIZONTAL) {
   finalLinePts[1].y = finalLinePts[0].y;
```

setFinalLine

...setFinalLine

```
    finalLinePts[1].x = finalLinePts[0].x + sumOfWidths
                              + ((cableWidth-1) * pitch);
  } else /* HorV = VERTICAL */ {
    finalLinePts[1].x = finalLinePts[0].x;
    finalLinePts[1].y = finalLinePts[0].y + sumOfWidths
                              + ((cableWidth-1) * pitch);
  }
  return(sumOfWidths);
}

/*****************************************************************/
/*  findInflectionPoints determines the areas in the pitch change where
    the problem may be split into sub-problems. Inflections in the
    end points of the cable solution are detected when an ith element's
    two endpoints are both under or both above the (i+1)th element's two
    pitch change endpoints. If such an inflection point is found, the
    initPts array element index is stored in the inflection array and
    the inflCnt is incremented.

    Called By: solve.
    Calls: none
    Returns: inflCnt,filled inflection array.
 */
/*****************************************************************/
int
 findInflectionPoints(lines,inflection)
SQIntegerPoint        lines[][MAXNUMINCABLE];
int                   inflection[];
{
return(0);
}
```

findInflectionPoints

```
/*****************************************************************/
/*  solveSubProblem is called for each sub-problem inflection area of
    the cable to be re-pitched. It makes the pitch change by hugging
    the larger line segment - initial sub-segment or final sub-segment.
    Jogs are offset by one min spacing from the selection end points
    and are placed at min spacing. Cable element width is determined
    by the width if the initial geometry.

    Called By: solve.
    Calls: none
    Returns: jog coordinates in the lines array
 */
/*****************************************************************/
void
 solveSubProb(lower,upper,lines)
int                   lower,upper;
SQIntegerPoint        lines[][MAXNUMINCABLE];
{ }
```

solveSubProb

```
/*****************************************************************/
/*  checkForFit finds the pitch change can fit in the channel area
    provided. The routine searches for the widest sub problem of the
    cable pitch change. It calculates what width is necessary to perform
    the appropriate jogs of the pitch change. It checks to make sure
    that the pitch change fits. Ifit fits the routine returns a 0 value.
    If the cable cannot fit, the user is notified what cable width is
    needed and how much wider the channel must be made. If the cable
    cannot fit, a -1 value is returned and the program does not continue.

    Called By: solve.
    Calls: none
    Returns: status 0 okay. -1 no fit.
*/
```

checkForFit

```
/*************************************************************/
int
  checkForFit(inflCnt,lines,inflection,cableWidth,sumOfWidths)
int                     inflCnt;
SQIntegerPoint          lines[][MAXNUMINCABLE];
int                     inflection[];
int                     cableWidth;
int                     sumOfWidths;
{ }

/*************************************************************/
/*  solve sets up the lines array and loads the initial and final points
    of each lines in the pitch changed cable.  FindInflectionPoints
    returns the array values where the solution may be broken up into
    similar sub-solutions.  It returns the inflection count which specifies
    how many sub solutions there are to solve.  CheckFotFit is then called
    make sure that the channel area is wide enough to proceed with a
    pitch change solution.  Then solve sub problem is called successively
    on the broken down problem.

    Called By: rePitch.
    Calls: findInflectionPoints,checkForFit,solveSubProblem.
    Returns: filled lines array to convert to squid solution.
*/
/*************************************************************/
```

solve

```
SQBool
 solve(finalLinePts,pitch,lines,sumOfWidths,initLineBB)
SQIntegerPoint          finalLinePts[];
int                     pitch;
SQIntegerPoint          lines[][MAXNUMINCABLE];
int                     sumOfWidths;
SQBB                    initLineBB;
{
 int                    i;
 int                    inflection[MAXNUMINCABLE];
 int                    inflCnt;
 int                    lower,upper;
 int                    partSum = 0;

 if (HorV == VERTICAL) {
   for (i=0;i<cableWidth;++i) {
        lines[0][i].x = initLineBB.r;
        lines[0][i].y = (initPts[i].topOrRight + initPts[i].bottomOrLeft)/2;
        lines[1][i].y = lines[0][i].y;
        lines[3][i].x = finalLinePts[0].x;
        lines[3][i].y = finalLinePts[0].y + partSum + (0.5 * initPts[i].width) +
                                                  (i*pitch);
        lines[2][i].y = lines[3][i].y;
        partSum += initPts[i].width;
   }
 } else /* HorV == HORIZONTAL */ {
   for (i=0;i<cableWidth;++i) {
        lines[0][i].y = initLineBB.t;
        lines[0][i].x = (initPts[i].topOrRight + initPts[i].bottomOrLeft)/2;
        lines[1][i].x = lines[0][i].x;
        lines[3][i].y = finalLinePts[0].y;
        lines[3][i].x = finalLinePts[0].x + partSum + (0.5 * initPts[i].width) +
                                                  (i*pitch);
        lines[2][i].x = lines[3][i].x;
        partSum += initPts[i].width;
   }
 }
 for (i=0;i<cableWidth;++i) inflection[i] = 0;
 inflCnt = findInflectionPoints(lines,inflection);
 if (checkForFit(inflCnt,lines,inflection,cableWidth,sumOfWidths) < 0) {
```

```
        return(sqFalse);
    }
    if (inflCnt == 0) {
            lower = 0;
            upper = cableWidth - 1;
            solveSubProb(lower,upper,lines);
    } else { /* at least one inflection point in the cable */
        for (i=0;i<=inflCnt;++i) {
            if (i==0) {
                lower = 0;
                upper = inflection[i];
            } else if ((i > 0) && (i < inflCnt)) {
                lower = inflection[i-1] + 1;
                upper = inflection[i];
            } else if (i==inflCnt) {
                lower = inflection[i-1] + 1;
                upper = cableWidth - 1;
            }
            solveSubProb(lower,upper,lines);
        }
    }
    return(sqTrue);
}


/*****************************************************************/
/*  convertSolnToSquid is called to convert the pitch change solution to
    squid.  The pitch change is performed with a cableWidth number of
    lines.  These lines are deposited in an instance.  The instance's
    view is pushed into and cableWidth number of lines is created in
    that view.  the view is then saved and then an instance of that
    view is placed in the current view on the screen.

    Called By: rePitch.
    Calls: none.
    Returns: bounding box of the created instance.
    Squid Interaction:  Push into an instance and place lines in that
                instance.  create that instance.
*/
/*****************************************************************/
int
 convertSolnToSquid(lines,bb)
SQIntegerPoint      lines[][MAXNUMINCABLE];
SQBB                *bb;
{
 HAWindo            windo;
 SQView             view;
 SQInst             inst;
 FILE               *stream;
 char               string[256];
 char               instanceCell[256];
 char               instanceView[256];
 SQGeo              currLine;
 SQIntegerPoint     linePts[PTSINLINE];
 int                i;
 int                j;

 HATypescript(sqFalse,
            "Type re-pitched area's view's name as \"cellsName viewsName\"");

#ifdef BATCH
 sscanf(HAKeyboard(),"%s",instanceCell);
 sscanf(HAKeyboard(),"%s",instanceView);
#else
 sscanf(HAKeyboard(),"%s%s",instanceCell,instanceView);
```

convertSolnToSquid

```
#endif

 sprintf(string, "Saving re-pitched area as cell: \"%s\" view \"%s\"",
                instanceCell,instanceView);
 HATypescript(sqFalse,string);

 view.cell = instanceCell;
 view.view = instanceView;
 view.mode = "w";

 SQPushViewStk(SQCreateViewStk());
 if (((int) SQ(sqPush,sqView,sqCircuit, view, &stream)) < 0) {
      HATypescript(sqTrue,SQDiagnostic());
      return(-1);
 }

 currLine.geoType = sqLine;
 currLine.function = sqActiveArea;
 currLine.def.line.nPath = PTSINLINE;
 currLine.filledP = sqTrue;
 currLine.layer = cableLayer;

 for (i=0;i<cableWidth;++i){
     for(j=0;j<PTSINLINE;++j) {
          linePts[j] = lines[j][i];
     }
     currLine.def.line.width = initPts[i].width;
     currLine.def.line.path = linePts;
     if ((int) SQ(sqCreate,sqGeo,&currLine) < 0) {
         HATypescript(sqTrue, SQDiagnostic());
         return(-1);
     }
 }

 if ((int) SQ(sqSave,sqView,sqFast, view) < 0) {
     HATypescript(sqTrue, SQDiagnostic());
     return(-1);
 }

 if ((int) SQ(sqPop,sqView,sqStage) < 0) {
     HATypescript(sqTrue, SQDiagnostic());
     return(-1);
 }
 SQPopViewStk();

 inst.masterCell = instanceCell;
 inst.masterView = instanceView;
 inst.name = "";
 sprintf(string,"T %d %d", 0, 0);
 inst.cif = string;

 if ((int) SQ(sqCreate,sqInst, &inst) < 0) {
     HATypescript(sqTrue,SQDiagnostic());
     return(-1);
 }
#ifndef BATCH
 switch(SQPushMaster(&inst,"physical")) {
       case sqViewAlreadyStaged:
       case sqOK:
             if((int)SQ(sqPop,sqView,sqStage) < 0) {
                     HATypescript(sqTrue,SQDiagnostic());
             }
             break;
       default:
             sprintf(string,"Can't push master %s/%s.  %s",inst.masterCell,
```

```
                inst.masterView,SQDiagnostic());
            HATypescript(sqTrue,string);
            break;
  }
  if ((int) SQ(sqGet,sqInst,&inst) < 0) {
      HATypescript(sqTrue,SQDiagnostic());
      return(-1);
  }

  *bb = inst.bb;
#endif

return(0);
}



/*****************************************************************/
/*  rePitch calls the main sections of the pitch change code.

    Called By: pitchChange.
    Calls: getInitPts,extractGeos,setFinalLine,solve,convertSolnToSquid.
    Returns: bounding box of the created instance.
*/
/*****************************************************************/
int
   rePitch(bb)
SQBB                *bb;
{
    SQBB                    initLineBB;
    SQIntegerPoint          finalLinePts[2];
    SQIntegerPoint          lines[PTSINLINE][MAXNUMINCABLE];
    int                     pitch;
    int                     sumOfWidths;

    if (! getInitPts(&initLineBB,finalLinePts,&pitch))
        return(-1);

    if(extractGeos(initLineBB,finalLinePts[0])<0) {
        HATypescript(sqTrue,"No geometries found for cable. ");
        return(-1);
    }
    sumOfWidths = setFinalLine(finalLinePts, pitch *= HASQUNITSPERLAMBDA);
    if (! solve(finalLinePts,pitch,lines,sumOfWidths,initLineBB)) {
        return(-1);
    }
    if (convertSolnToSquid(lines,bb)<0) {
        HATypescript(sqTrue,"problem in convert soln to squid routine.");
        return(-1);
    }

    return(0);
}

#ifndef BATCH

/*****************************************************************/
/*  pitchChange checks that the hawk window is selected.  It pushes
    into it for editing, emptys the boundig box set, calls rePitch
    and redisplays if the pitch change is performed successively.

    Called By: PitchChange.
    Calls: rePitch
*/
```

rePitch

```
/*****************************************************************/
void
  pitchChange()
{
        HAWindo windo;
        int        i;
        SQBB       bb;

        bb.l = 0;
        bb.r = 0;
        bb.b = 0;
        bb.t = 0;

        windo = HACurrentWindo();
        if (windo.windoID == NULL) {
                HATypescript(sqTrue,"Current window not selected.");
                return;
        }
        fperr = fopen("route_error","w");
        if (fperr== NULL) {
                HATypescript(sqTrue, "Unable to open error file");
                return;
        }
        if (!RouterCadrcRead()) {
                HATypescript(sqTrue,
                   "Problem reading .cadrc rules, check route_error file.");
                fclose(fperr);
                showfile("route_error");
                return;
        }

        SQPushViewStk(windo.editStk);
        SQEmptyBBSet(HAChangedRect());
        i=rePitch(&bb);
        SQPopViewStk();

        if (i < 0) {
            HATypescript(sqTrue,"UNABLE to do pitch change.");
        } else {
            SQAddToBBSet(&bb, HAChangedRect());
        }

        fclose(fperr);
        showfile("route_error");

        HADisplayView(windo.windoID,*HAChangedRect(), sqFalse, sqFalse);

}
```

pitchChange

```
/*****************************************************************/
/*  PitchChange invokes the pitch change from the hawk layout editor.
    Called By: symbolic hawk menu selection /lib/hawk/menus/userCommand/
                                                          routing.
    Calls: pitchChange
*/
/*****************************************************************/
void
  PitchChange()
{
        pitchChange();
}
#endif BATCH

#ifdef BATCH
```

PitchChange

```
SQIntegerPoint
    *HACursorPositionL()                          HACursorPositionL
{
int xx,yy;
SQIntegerPoint readVals;

    scanf("%d %d",&xx,&yy);
    readVals.x = xx;
    readVals.y = yy;
    return(&readVals);
}

static void
   HAListen()                                     HAListen
{}

SQBool
 HAMenuSelectionP()                               HAMenuSelectionP
{
 return(sqFalse);
}

char
  HAKeyTyped()                                    HAKeyTyped
{
 return(2);
}


static char
    *HAKeyboard()                                 HAKeyboard
{
     static char s[256];
     scanf("%s", s);
     return(s);
}

static void
    HATypescript(moreP, s)                        HATypescript
SQBool moreP;
char *s;
{printf("%s\n",s);}


SQBB changedRect;

SQBB
  *HAChangedRect()                                HAChangedRect
{
 return(&changedRect);
}

static SQStatus
    HADisplayView()                               HADisplayView
{ }

HAWindo currentWindow;

HAWindo
   HACurrentWindo()                               HACurrentWindo
{ return(currentWindow); }
```

```
main(argc, argv)
int argc;
char *argv[];
{
	FILE			*stream;
	SQBB			bb;

	if (argc != 3) {
		printf("usage: re-pitch cellname and cellview\n");
		exit(-1);
	}

	if ((int) SQBegin()<0) {
		HATypescript(sqTrue,SQDiagnostic());
		exit(-1);
	}

	changedRect.l = 0;
	changedRect.b = 0;
	changedRect.t = 0;
	changedRect.r = 0;
	bb.l = 0;
	bb.b = 0;
	bb.t = 0;
	bb.r = 0;

	/* Fake out a current window -- push into view given on command line */
	currentWindow.view.view = argv[2];
	currentWindow.view.cell = argv[1];
	currentWindow.view.mode = "w";
	currentWindow.windoID = 3;
	SQPushViewStk(currentWindow.editStk = currentWindow.displayStk =
		SQCreateViewStk());

	if (SQ(sqPush, sqView, sqCircuit, currentWindow.view, &stream) != sqOK) {
		HATypescript(sqTrue,SQDiagnostic());
		exit(-1);
	}

	if (rePitch(&bb) < 0) {
		exit(-1);
	}

	/* Fake out popping current window */
	if ((int) SQ(sqSave,sqView,sqFast,currentWindow.view) < 0) {
		HATypescript(sqTrue,SQDiagnostic());
	}
	if ((int) SQ(sqPop, sqView, sqUnstage) < 0) {
		HATypescript(sqTrue,SQDiagnostic());
	}
	SQPopViewStk();

	if ((int) SQEnd() < 0) {
		HATypescript(sqTrue,SQDiagnostic());
	}

exit(0);
}
#endif BATCH
```

APPENDIX B

Routing Toolbox Users Guide.

1. Introduction

The Hawk graphics editor provides interactive and incremental routing routines that automate the routing of integrated circuits. These routers were developed in close conjunction with a designer laying out a custom IC. Therefore, the facilities they provide fulfill the requirements of custom layout, easing incremental changes to layout. Hawk's routers are invoked via menu selection. The commands are highly-interactive; the editor prompts the designer through the route. Upon completion, the solution is usually the route that the designer would have drawn himself had he placed every rectangle. The tools' help produce an efficient custom layout by automating repetitive steps in the layout process. The designer guides routes by specifying control points when prompted.

Five routers are described in this user's guide. The routers may be invoked from the ROUTING sub-menu of Hawk. The first three routers are bus oriented routers, the pitch change, "l"-turn, and cable routers. The pitch change and lturn routers are simple river routers that enable incremental bus routing when there is no swap in signal order. The cable command enables the routing of a bus of signals when there is a switch in signal order from source to destination. The last two routers are channel routers for the routing of horizontal and vertical channels. The two channel routers interface Hawk to the YACR2 channel router.

The routers report diagnostics and routing information to the user. For example, the channel router reports such information as the channel's longest net-name, any pin that does not have a matching pin in the channel, and whether the channel width be grown or shrunk based on the routing density result. Pertinent information is reported during the course of the route and saved in an error file for later reference. The diagnostics help keep the designer on the right track.

Because the routers were developed for custom design and layout, the routers emulate the designer's routing strategies. The cable and channel routers have a preferred routing layer, and maximize runs on that preferred layer. While making jogs, the pitch change and the cable commands route wires at the minimum design rule spacing. The routing routines essentially return the route that the designer would have constructed had he routed the region rectangle by rectangle.

Each routing routine creates a cell filled with the routing solution. A pitch change, LTurn, cable route, or channel route instance of the master cell filled with routing geometries is placed in the cell being edited in Hawk. Placing the geometries as a cell instance, rather than placing flat geometries saves the context of the route. The entire routing cell may then easily be manipulated or deleted as routes are re-run.

Hawk may be invoked under various layout technologies. Technologies used at UC Berkeley are CMOS-PW and NMOS fabrication technologies. Because each technology has its own layout design spacing rules and routing layers, the routing routines are parameterized and not constrained to a particular technology-dependent set of rules and layers. They route between any two layers of a particular technology as specified by the user. The ".cadrc" file resides in the user's home directory and stores tool-specific information in the UC Berkeley CAD environment. Each ".cadrc" file is divided into several sections. The **HAWKROUTERS** section of this file contains routing information that tunes the routers to a particular routing policy. The routing parameters are described at the end of this document.

Routing Toolbox User's Guide

Abstract.

The routers in Hawk's routing sub-menu are described in this document. It is assumed that the reader is familiar with the Hawk/Squid package. The routers enable the user to interactively guide the routing process. There are six sections in this user's guide:

(1) Introduction to the routers.

(2) **Pitch Change Router.** This is a bus oriented router. It enables the user to change the pitch in a bus of geometries.

(3) **LTurn Router.** Enables the user to guide a bus of geometries around one turn.

(4) **Cable Router.** Enables the user to guide a cable of geometries around several jogs and with a permutation in signal order.

(5) **Channel Router.** This router enables the user to interactively invoke the channel router YACR2.

(6) **Rules File.** The file in which the layout design rules that are used by the routers are specified.

2. Routers.

2.1. Bus Routing.

The pitch change and L Turn commands automate bus routing when there is no switch of signal order in the bus.

2.2. Pitch Change.

The pitch change command enables the user to direct a change in the pitch of a *bus* of nets from any regular or irregular spacing to a user-specified regular spacing. Here a **bus** of nets is defined as a parallel set geometries on the **same** layer ending roughly at the same edge as illustrated in Figure 1. The pitch change command may be used to fan out a minimally-spaced group of geometries to a contact to contact spacing or squeeze an irregular spacing down to a minimum spacing. In a pitch change, there is no order change in the bus's nets. The bus elements are simply river-routed to a new pitch.

2.2.1. Invoking the Pitch Change.

The pitch change command is found in Hawk's Routing sub-menu. Upon **pitchChange** menu selection, the user is prompted to cut across the bus of nets *where* he wants to start the pitch change, by pointing to the bottom and top of the bus. These points are endpoints of a **cut line** across the pitch geometries. (see Figure 1). The user is then prompted to mark the point where he wants the bottom bus element to route to (final pitch point), and type the new *pitch* or separation of the wires of the bus in lambda units. Each geometry is extended in the pitch change at its original width. Jogs in the pitch change are performed at the routing layer's minimum spacing. A pitch change of the initial geometries is shown in Figure 2 with control points marked.

FINAL_POINT._Point_where_the_pitch_change_starts

metal

poly

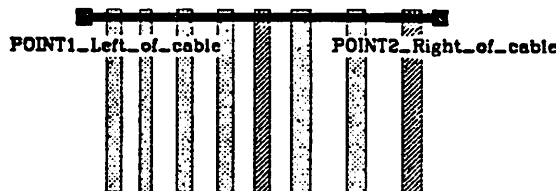

Figure 1. Initial Bus Geometries.

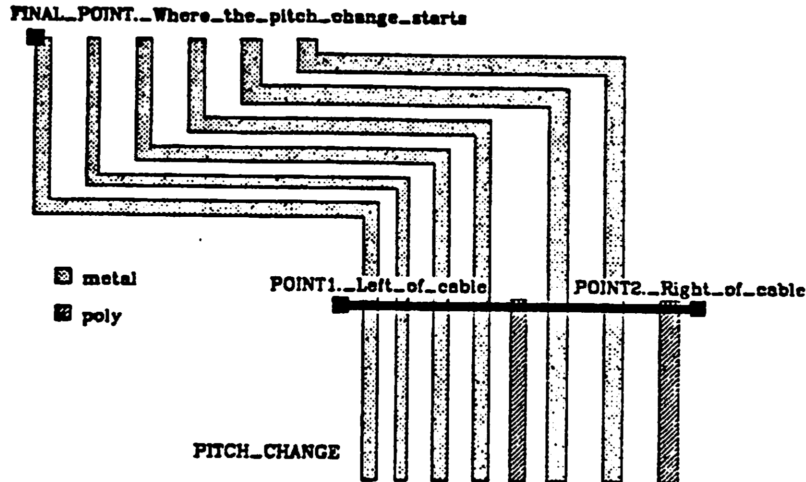

Figure 2. Pitch change with control points marked.

2.2.2. Squid data storage.

The pitch change is stored in its own Squid file; a copy or instance of it is placed in the Hawk view being edited. When the route is complete, the user is prompted to type a Squid cell-name and cell-view identifying that routed cell. The Hawk screen is redisplayed showing the new pitch change. If unhappy with the route, the user may select the **instance** layer on the Hawk layer window, **addPtSel** or put the instance in the selected set, and delete the routed instance.

2.2.3. Errors.

Any of several errors prohibit the completion of a pitch change. In case on an error, the user is notified what is preventing the pitch change; that error information is stored in the **route_error** file. The pitch change is not performed when no geometries are found under the cut line. The routine aborts when the required ".cadrc" parameters are absent from the HAWKROUTERS section of the ".cadrc" file. The pitch change is not performed when there is not enough room in the layout region to squeeze the jogs of the pitch change; i.e., there is not enough room between the initial cut-line and the final pitch point. The user is told how much space is needed in lambdas to perform the jogs of the pitch change.

2.2.4. Rule File Parameters.

The pitch change may be performed on one of the two routing layers specified in that HAWKROUTERS section of the user's ".cadrc" file. Four ".cadrc" design rule parameters are necessary to perform a pitch change. The two possible routing layers' names and their minimum spacing. Here is an example for the mosis CMOS_PW technology:

```
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
```

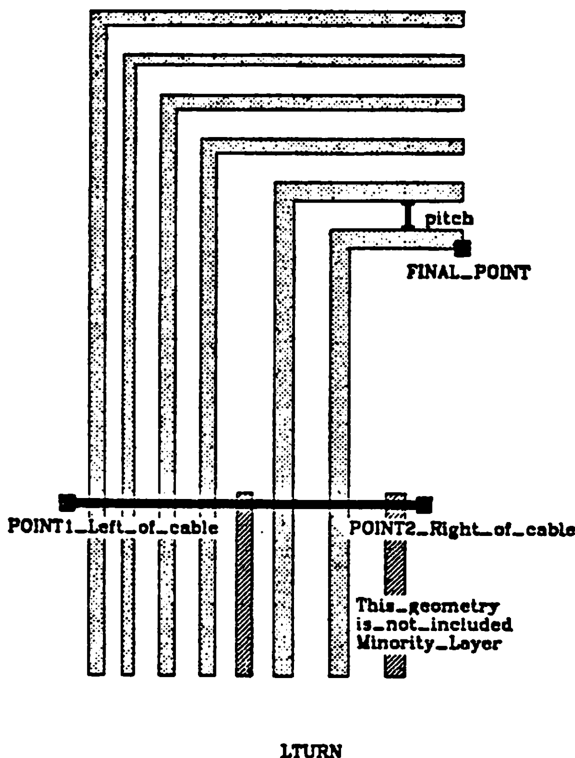

Figure 3. L Turn with control points marked.

switch physical order from the source to the destination. If a switch in signal order is needed, the command performs the order change at the last jog in the bus's path. In the interactive cable command, bus specification is simple.

2.4.1. Signal Connectivity.

Because the physical signal order may be permuted from the initial to the final terminals, signal connectivity is specified with terminals at the beginning and end of the cable. Matched terminal names imply connectivity. Each signal in the cable has a terminal in the initial terminal set and in the final terminal set. It is necessary to place these terminals before running the cable command. Terminals are shown in Figure 4. Initial terminals may be at a minimum spacing; final terminals must be at a geometry-to-contact spacing to accommodate any change in signal order.

Note that when there is no swap in signal order from the initial to the final terminals, it is easier to use the **pitch** and **LTurn** commands, because in those commands it is not necessary to place signal terminals.

2.4.2. Routing Layer.

The routing layer is selected by the terminal layer. All of the initial terminals must be on the same layer and all of the final terminals must be on the same layer. Cable layer is determined as follows:

```
NAME LAYER2_NAME "CM"
VALUE LAYER1_MINSPACING 3
VALUE LAYER2_MINSPACING 4
end
```

The two routing layers are "CM", CMOS metal, and "CP", CMOS polysilicon. A CMOS poly geometry must be spaced at least 3 lambda from another poly geometry, while a CMOS metal geometry must be at lease 4 lambda from another metal geometry.

2.2.5. Majority Selection.

A pitch change is performed on only one routing layer. If geometries under the cut line are on two different routing layers, a pitch change is performed on the layer with the majority count in the bus. For example, the pitch change region in Figure 2 has six polysilicon and two metal elements. The pitch change is performed on the six poly elements. Majority rule is used because the objects on the other layer are usually not intended to be in the bus; they are unavoidably in the selection region.

2.3. LTurn.

The **LTurn** router is similar to the **pitchChange** router. It enables the user to perform an *l-turn* on a bus of nets from any regular or irregular spacing to a user-specified regular spacing. Once again, a **bus** of nets is defined as a set of parallel geometries on the **same** layer ending roughly at the same edge. (see Figure 1). The **LTurn** command may be invoked when there is no order swap in the geometries of the bus; no layer change is made and no terminals are needed to imply routing connectivity. Geometries are extended around one turn in their original order with their original widths.

An "l"-turn may be performed on either of the two layers specified in the HAWKROUTERS section of the ".cadrc" file, but, as with the pitch change command, it is only performed on one of those layers.

```
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
 NAME LAYER2_NAME "CM"
end
```

An LTurn routine is aborted if the two routing layers are not included in the ".cadrc" file.

Upon **LTurn** menu selection, the user is prompted to cut the bus of nets *where* he wants to start the l-turn by pointing to the top and bottom of the bus. The user would typically point to the top and bottom **edges** of the bus; he may alternatively point before the edge to do an l-turn on geometries whose edges do not line up. The user is then prompted to point to where he wants the most negative point of the l-turn to begin and type the new spacing of the wires in lambda units. Each geometry's width is extended in the l-turn. The user is prompted to type a Squid cell name and view name for the l turn. That cell is created and placed in the current Hawk cell from which the l-turn was called. An LTurn is illustrated in Figure 3 with the appropriate control points marked.

2.4. Cable Router.

The **cable** command routes a bus of nets around obstacles. It is not a general river router; it routes each signal in a bus between one source terminal and one destination terminal. **Cable** enables the user to guide a bus of nets around obstacles and through a variable number of jogs. Like the **pitchChange** and **LTurn** routers, there must be a one to one correspondence between connections in the bus. Unlike those commands, the cable routine allows the signals to

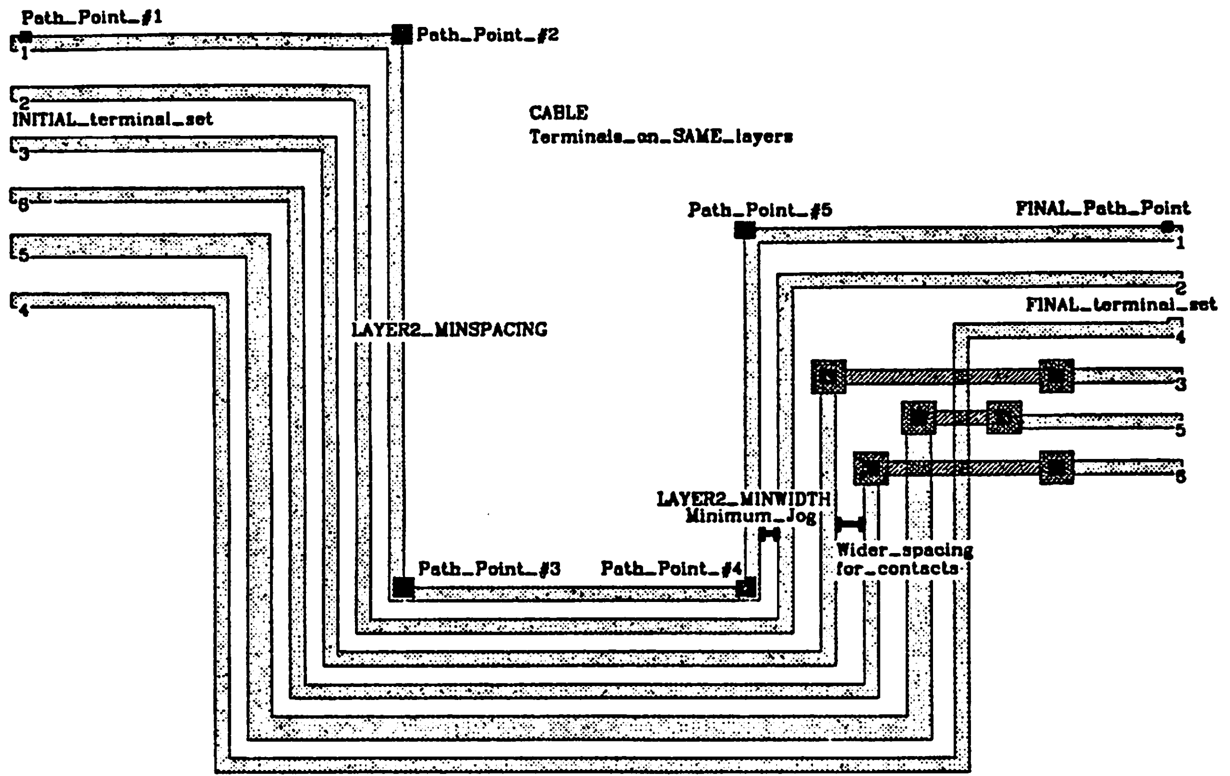

Figure 5. Cable Route. Same terminal layers at ends.

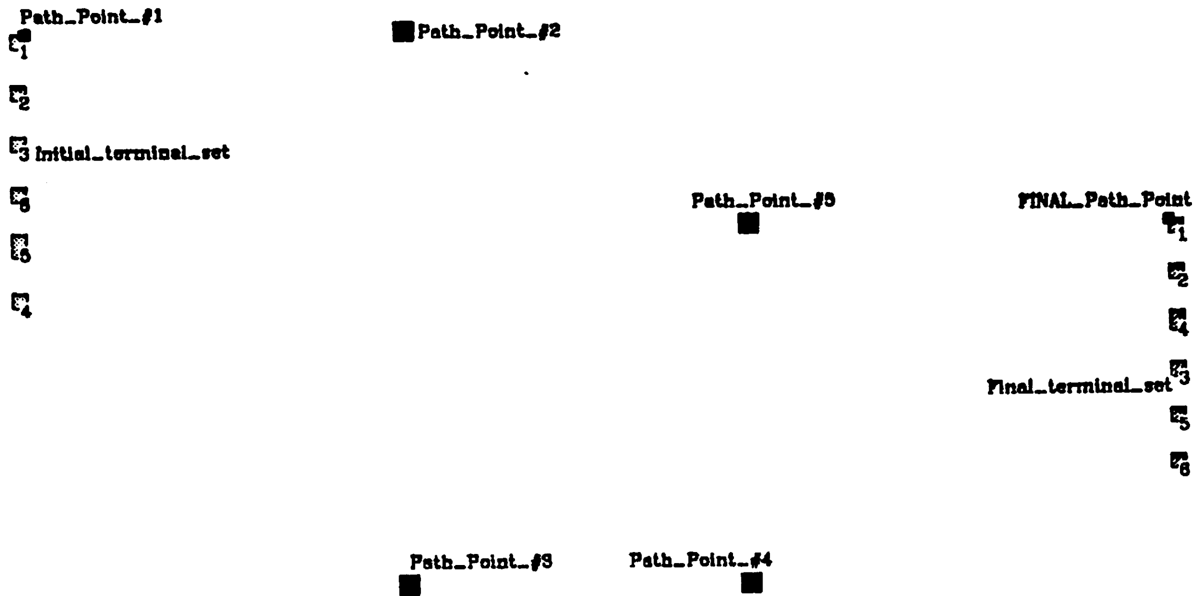

Figure 4. Cable Set-Up.

(1) Case A. The initial terminal and the final terminal sets are on the *same* layer. At the last jog in the cable, a via route onto the alternate routing layer is made to accommodate a change in signal order. The layer shared by the initial and final terminals is maximized as shown in Figure 5.

(2) Case B. The initial and final terminal sets are on the two *different* layers specified in the rules file. In this case, the initial terminal set's layer is maximized as shown in Figure 6. The layer change from the initial terminal set's layer to the final terminal set's layer is made at the **final** jog.

2.4.3. Cable Path Specification.

The cable command allows the user to guide the bus's route around obstacles through a variable number of jogs. First the user points to the initial terminals, one at a time. Hits ESC. Then he points to the final terminals, one at a time. Hits ESC. He then specifies the path that he wants the route to take from the initial to the final terminals by pointing to jogs on one side of the cable's path, and terminates by hitting ESC. Jog points are illustrated in Figure 6.

2.4.4. Signal Wire Widths.

Wire widths are determined as follows. There are two different cases to consider. Jogs are performed in the cable at minimum spacing unless a contact necessitates the widening of the spacing.

(1) Case A. When the initial and final terminals sets are on the *same* routing layer, a layer change is performed only where necessary to execute a swap

```
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
 NAME LAYER2_NAME "CM"
 VALUE LAYER1_MINWIDTH 3
 VALUE LAYER2_MINWIDTH 3
 VALUE LAYER1_MINSPACING 3
 VALUE LAYER2_MINSPACING 4
 VALUE CONT_SIZE 7
end
```

The layer definitions are described in Section 2 of this document, the Design Rule File.

2.4.6. Execution errors.

The cable command flags an error when two initial terminals are not at at least the minimum spacing for that layout layer. It flags an error when two final terminals are not at at least a geometry-to-contact spacing. It flags an error when there is not a one-to-one match between initial and final terminals. Errors are stored in the **route_error** file for later reference.

2.5. Channel Routing.

The **Horizontal Channel** command implements a two-layer horizontal channel route by providing an interactive interface between the Hawk graphics editor and the YACR2 channel router. A channel is a rectangular routing region. Terminals, or pins, mark signals on the channel periphery that are to be connected. The interface collects the information that YACR2 expects as input from the Hawk cell being edited and calls YACR2. It then translates YACR2's output into Hawk geometries, redisplays the routed solution, and reports useful routing information.

The channel router provides fast rip-up and re-route capability. It is useful when routing large amounts of "spaghetti" logic that are prone to error and change. Re-running the channel router is easy. YACR2 and its interface guarantee that all terminals of the same name in the channel are connected.

2.5.1. Channel Definition.

Terminals specify the connectivity problem. The terminals are fixed along the two opposite channel lengths, and are non-fixed on the remaining channel-ends.

(1) **Fixed terminals** along the top and bottom lengths of the channel determine exact signal location. Top terminals must be colinear. Bottom terminals must be colinear.

(2) **Floating terminals** designate signals that must exit the channel from an end. The signal's order in YACR2's solution may not be the same as the order of the floating pins.

Floating and fixed terminals are placed before running the channel router by using the **terminal** command in the Hawk. Fixed terminals must be placed at at least the minimum user-defined spacing of LAYER1_GRIDSPACING and be on one of the two routing layers defined in the rules file. Terminals on any other layer in the channel will be ignored by the interface. To avoid layer change in the top and bottom tracks of the channel solution, fixed terminals should be on LAYER1_NAME, the layer that runs vertically across the channel width. This layer is polysilicon the routing example. Top pins do not have to line up directly over bottom pins. Figure 7 shows a channel.

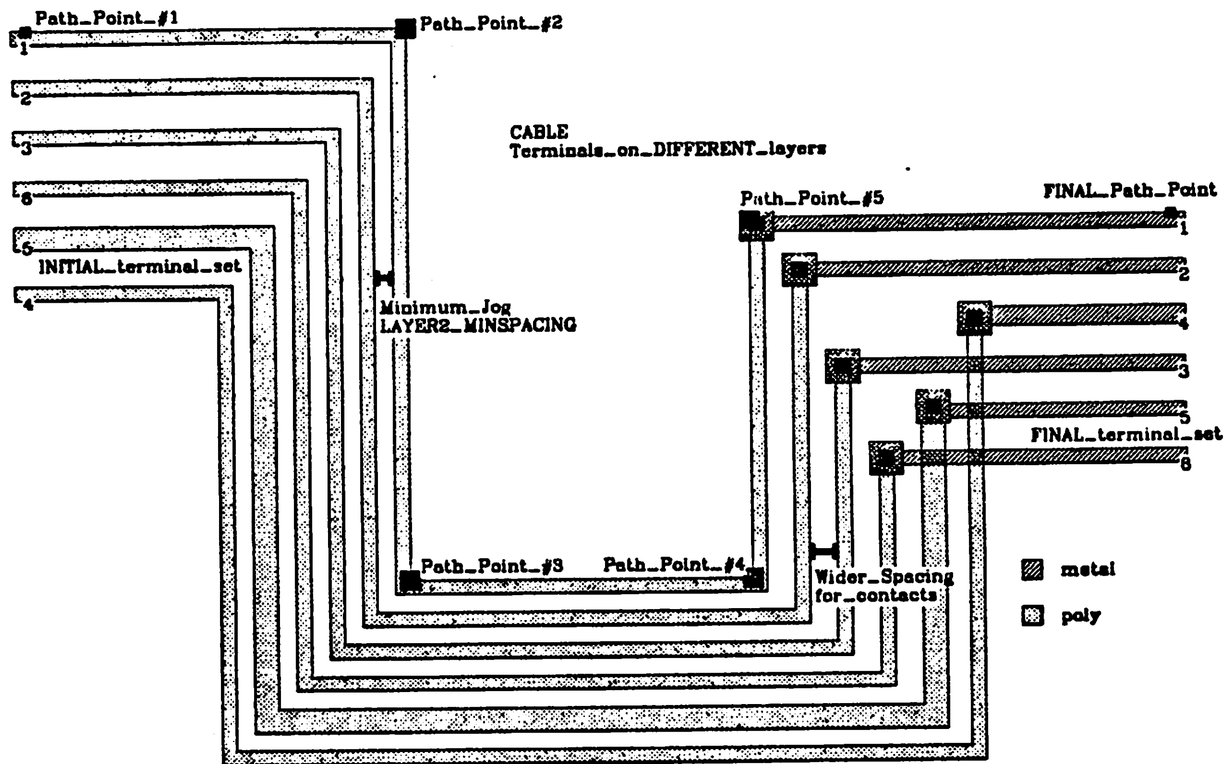

Figure 6 Cable Route. Different terminal layers at ends.

in signal order. When there is no layer change in a wire, as in net 4 of Figure 5, the wire width is set to the width of the signal's initial terminal. Net 3 has a layer change, so the initial wire width is determined by the width of the signal's initial terminal; the final wire width is determined by the width of the signal's final terminal. The via layer's width is the minimum width for that layer as specified in the routing rules file.

(2) Case B. In the case where the initial and final terminals sets are on *different* routing layers, each terminal's width determines its wire's size as illustrated in Figure 6.

2.4.5. Necessary Supporting Files.

(1) The **contact/physical** Squid file must reside where Hawk is invoked. This is the physical view of a contact connecting the two routing layers in cable solution. See Figure 9 for the contact between metal and polysilicon in the CMOS-PW technology.

(2) The HAWKROUTERS section of ".cadrc" file specifies the two routing layers. The necessary routing parameters associated with these layers are:

called **.chan.route.out**. Here is the .chan.route.out solution for the above input:

```
5                    #there are 5 tracks in the soln
10                   #there are 10 columns in the soln
0 3 3 3 3 3 3 3 5 4  #metal row 1
2 2 2 2 2 2 2 0 5 4  #metal row 2
5 0 0 1 1 1 2 0 5 4  #metal row 3
5 5 5 5 5 5 5 5 5 4  #metal row 4
4 4 4 4 4 4 4 4 4 4  #metal row 5
5 3 4 2 1 1 0 3 5 4  #poly row 1
5 3 4 2 2 1 0 0 0 0  #poly row 2
5 3 4 1 2 1 2 0 0 0  #poly row 3
0 3 4 1 2 0 2 5 0 0  #poly row 4
0 3 4 1 2 0 2 5 4 0  #poly row 5
4 15 7               #longest net is net 4, 15 metal units, 7 poly units
```

The post-processor maps the symbolic solution back into the particular Hawk channel. The routed geometries are stored in their own cell. The user is prompted to name the cell for identification and placement in the design hierarchy; an instance of the channel solution is placed in the Hawk physical layout being edited. An example name would be "route6" "physical". The routing solution to the channel defined in Figure 7 is shown in Figure 8.

2.5.3. Necessary Supporting Files.

There are two files necessary to run the channel router. A Squid cell named **contact** with the view **physical** must reside in the designer's file system in the same place where the Hawk file *path* is and where Hawk is invoked from the user's project directory. This contact master file is placed as an instance at every layer via in the channel routing solution. This file is of the form:

```
SQUID
PUT VIEW "contact" "physical" "w" "squidNextObjectID" INT 4
MK RECT 1 ACTIVE LAYER "CP" FILL LB -60 -60 RT 80 80
MK RECT 2 ACTIVE LAYER "CM" FILL LB -60 -60 RT 80 80
MK RECT 3 ACTIVE LAYER "CC" FILL LB -20 -20 RT 40 40
```

Note by the Squid coordinates of the contact rectangles that the center of the

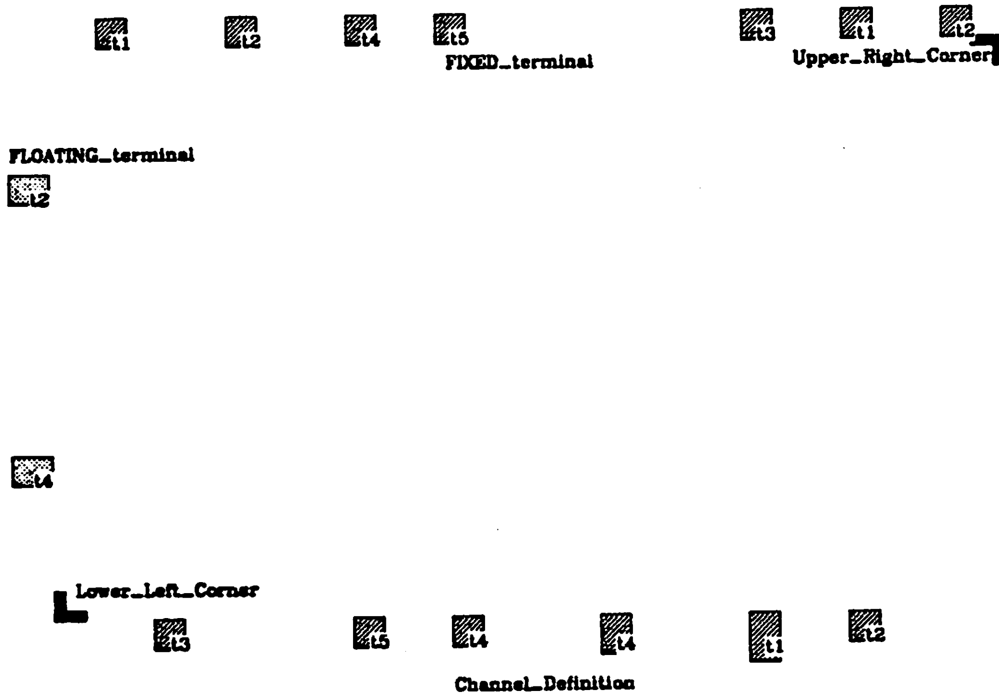

Figure 7. Channel definition.

2.5.2. Program Details.

After selecting **HorzChannel** on the routing sub-menu, the user identifies the channel interactively by pointing to two opposite corners of the rectangular channel. Figure 7 marks these points. The interface requires that the top and bottom terminals line up on the same lambda grid and that the user point to that particular lambda when specifying the channel periphery. The pre-processor marches across the channel and creates the net list grid that YACR2 is expecting as input. This YACR2 input file is called **.chan.route.in**. It is created in the directory where Hawk is invoked. The file contains the top and the bottom fixed net lists from left to right, and the left and the right floating net lists from bottom to top. Each net name is translated into an integer in this file because YACR2 expects integer net names as input. Here is an example of .chan.route.in:

```
5                      #five nets in channel
10                     #ten columns in channel
5 0 4 2 1 0 0 3 5 4    #top net list
0 3 0 1 2 0 2 5 4 0    #bottom net list
2                      #two left nets
2 4                    #the two left nets
```

The pre-processor calls YACR2 with this input file and YACR2 returns the output file

In the **.cadrc** file, the user defines the two routing layer names and the associated design rule parameters.

```
begin HAWKROUTERS
 NAME LAYER1_NAME "CP"
 NAME LAYER2_NAME "CM"
 VALUE LAYER1_MINWIDTH 3
 VALUE LAYER2_MINWIDTH 3
 VALUE LAYER1_GRIDSPACING 8
 VALUE LAYER2_GRIDSPAGING 9
 VALUE CONT_SIZE 7
end
```

The two routing layers are LAYER1_NAME and LAYER2_NAME, "CP" and "CM". LAYER1_NAME is the slower routing layer. This layer is used for routing signals on the shorter vertical channel runs across the width of the channel. LAYER2_NAME is the preferred routing layer; it makes the longer channel track runs and is the faster layer. LAYER2 is maximized in the channel by YACR2, as illustrated in Figure 8; when there is not another signal's route blocking the LAYER2 continuation of a net into a column, the signal stays on the preferred layer rather than switching to the column layer, LAYER1.

The LAYER1_MINWIDTH and LAYER2_MINWIDTH parameters determine the width of signal runs in the channel. In the above rule specification, both metal and polysilicon runs are drawn at a width of 3 lambda. The CONT_SIZE parameter specifies the size of the contact that connects the two routing layers at signal vias. The CMOS-PW contact is seven lambda by seven lambda.

2.5.4. Channel-grid parameters.

The grid spacing parameters determine row and column channel pitch.

(1) Column Pitch. Placement of column geometries is pre-determined by the fixed signal terminals along the top and bottom of the channel. LAYER1_GRIDSPACING is the minimum spacing between two terminals in adjacent columns of the channel. Typically, this parameter is set to a contact to contact pitch to accommodate instances in the channel where contacts are in neighboring columns.

(2) Row Pitch. Before the router is run there is no way to tell how many channel rows, or tracks, will be in the solution. After YACR2 is run, the rows of the channel solution are grown from the bottom row of terminals. The LAYER2_GRIDSPACING parameter sets the pitch spacing of the solution's tracks. The post-processor minimizes horizontal track placement. Track grid pitch, LAYER2_GRIDSPACING, should be set to contact-to-geometry pitch in the ".cadrc" rule file. When there is an instance of a contact directly above another contact in the channel solution, the post-processor automatically widens from the tighter row spacing to contact-to-contact spacing to accommodate the adjacent contacts.

2.5.5. Errors and Diagnostics.

Like all the routers in the routing toolbox, the channel router reports information and diagnostics to the user. When the allocated channel width is too narrow for the actual solution, the user is told how much larger he must make the channel. If too much space has been allocated, the user is notified how much extra space he has allocated. Information on the channel's longest net may be used to hand-calculate signal delays. The routine reports the name of any terminal that does not have at least one partner connection in the channel. This error often is the result of a error when typing the terminal name. If the

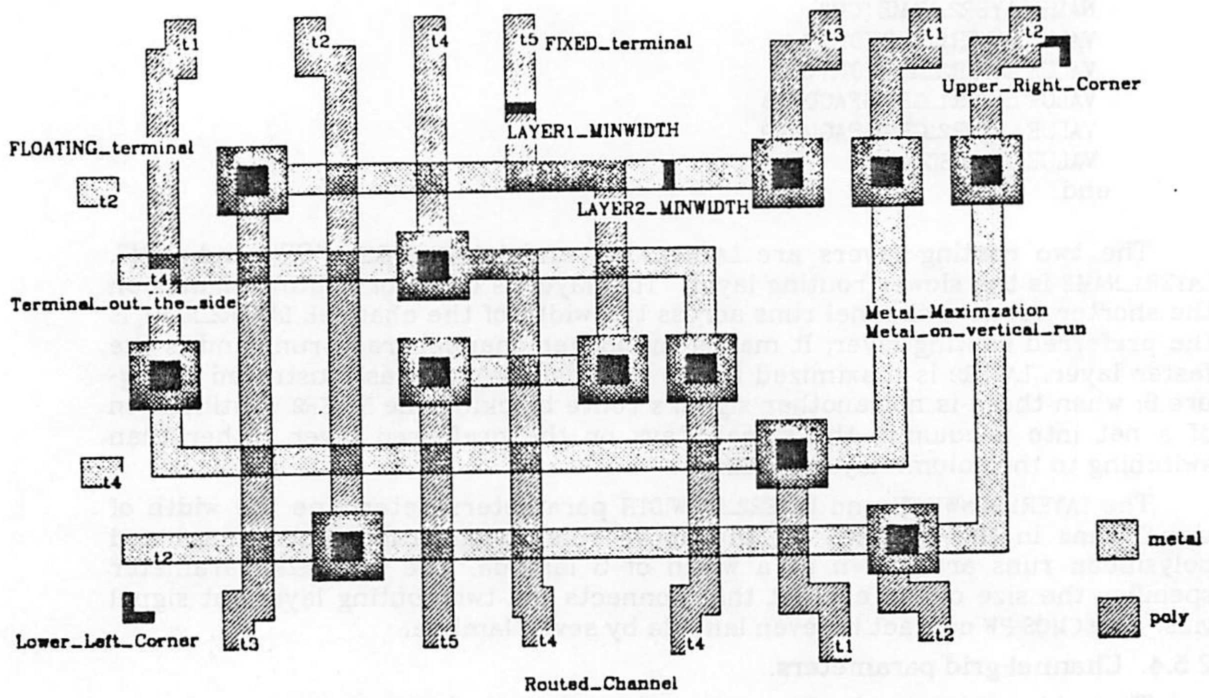

Figure 8. Channel Solution.

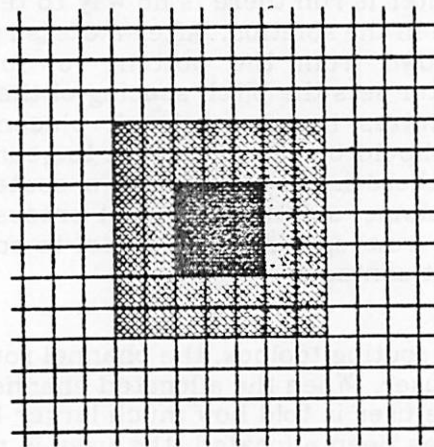

Figure 9. Contact for the CMOS-PW technology.

contact is roughly at the origin of the master. A contact is shown in Figure 9.

minimum spacing for CM is 4 lambda; the minimum spacing for CP is 3 lambda. This parameter is used by the pitch change and cable routers to perform minimally spaced jogs.

The **_MINWIDTH** parameter is the minimum allowable drawn width of a geometry. In the CMOS-PW case, this parameter is 3 lambdas for CM and CP. This is used by the channel router to draw minimum width geometries in the routing solution. It is also used by the cable command to draw minimum width geometries on the via layer when a layer change is made to switch the order of the cable signals.

The **_GRIDSPACING** parameter is used exclusively by the channel router to specify the virtual grid that the channel router observes when searching for terminals and placing tracks. In a channel route, the faster layer makes the runs along the length of the channel while the slower layer brings the signals into the channel, running along the channel width. Connections between the two layers are made with contacts. In a channel solution, a contact may be adjacent to a contact in another column in the worst-case grid pitch. **LAYER1_GRIDSPACING**, the pitch of the LAYER1 terminals at the top and bottom of the channel, should be at contact to contact spacing. For the CMOS-PW case above:

```
LAYER1_GRIDSPACING = contact size + max ( layer spacings );
          = CONT_SIZE +
             MAX (LAYER1_MINSPACING, LAYER2_MINSPACING)
          = 7 + MAX (3,4)
          = 11.
```

In a YACR2 channel solution, there are sufficiently few places where there is a contact next to another contact on a horizontal track. Signal terminals may then be spaced closer than at contact to contact spacing. LAYER1_GRIDSPACING may be tightened to a smaller number; it is set to 9 lambda for the CMOS-PW case:

```
LAYER1_GRIDSPACING = vertical track geometry width
               + max ( routing layer's min spacing )
               + ( contact overlap ) / 2;
          = LAYER1_MINWIDTH
            + MAX ( LAYER1_MINSPACING, LAYER2_MINSPACING )
            + ( CONT_SIZE - LAYER1_MINWIDTH ) / 2;
          = 3 + MAX(3,4) + (7-3)/2;
          = 9.
```

The resulting routed solution has design rule errors only where there is a contact next to another contact on a channel track. Reducing the pitch of the terminals saves considerable channel length, at the expense of having to edit the solution where the few contact spacing errors occur. For example, there are 72 nets and 169 columns in Deutsch's difficult channel routing example. Of the 287 contact vias in the channel solution, there are only six instances where a contact is adjacent to another contact on a channel track. Routing at a tighter pitch reduces each column width by two lambda for a total length reduction of 2lambda/column x 169 columns = 338 lambda. The user may choose whether to route at wide spacing or tight spacing. While reducing LAYER1_GRIDSPACING may require hand editing of a channel solution, it can significantly reduce channel length.

LAYER2_GRIDSPACING is the minimum distance that two horizontal tracks may be spaced. This parameter is minimized to a pitch of geometry-to-contact spacing. The channel router places tracks at this tight GRIDSPACING unless there are two vertically adjacent contacts. It then **automatically** widens the track spacing

minimum spacing for CM is 4 lambda; the minimum spacing for CP is 3 lambda. This parameter is used by the pitch change and cable routers to perform minimally spaced jogs.

The _MINWIDTH parameter is the minimum allowable drawn width of a geometry. In the CMOS-PW case, this parameter is 3 lambdas for CM and CP. This is used by the channel router to draw minimum width geometries in the routing solution. It is also used by the cable command to draw minimum width geometries on the via layer when a layer change is made to switch the order of the cable signals.

The _GRIDSPACING parameter is used exclusively by the channel router to specify the virtual grid that the channel router observes when searching for terminals and placing tracks. In a channel route, the faster layer makes the runs along the length of the channel while the slower layer brings the signals into the channel, running along the channel width. Connections between the two layers are made with contacts. In a channel solution, a contact may be adjacent to a contact in another column in the worst-case grid pitch. **LAYER1_GRIDSPACING**, the pitch of the LAYER1 terminals at the top and bottom of the channel, should be at contact to contact spacing. For the CMOS-PW case above:

```
LAYER1_GRIDSPACING = contact size + max ( layer spacings );
                   = CONT_SIZE +
                     MAX (LAYER1_MINSPACING, LAYER2_MINSPACING)
                 = 7 + MAX (3,4)
                 = 11.
```

In a YACR2 channel solution, there are sufficiently few places where there is a contact next to another contact on a horizontal track. Signal terminals may then be spaced closer than at contact to contact spacing. LAYER1_GRIDSPACING may be tightened to a smaller number; it is set to 9 lambda for the CMOS-PW case:

```
LAYER1_GRIDSPACING = vertical track geometry width
                       + max ( routing layer's min spacing )
                       + ( contact overlap ) / 2;
                   = LAYER1_MINWIDTH
                     + MAX ( LAYER1_MINSPACING, LAYER2_MINSPACING )
                     + ( CONT_SIZE - LAYER1_MINWIDTH ) / 2;
                 = 3 + MAX(3,4) + (7-3)/2;
                 = 9.
```

The resulting routed solution has design rule errors only where there is a contact next to another contact on a channel track. Reducing the pitch of the terminals saves considerable channel length, at the expense of having to edit the solution where the few contact spacing errors occur. For example, there are 72 nets and 169 columns in Deutsch's difficult channel routing example. Of the 287 contact vias in the channel solution, there are only six instances where a contact is adjacent to another contact on a channel track. Routing at a tighter pitch reduces each column width by two lambda for a total length reduction of 2lambda/column x 169 columns = 338 lambda. The user may choose whether to route at wide spacing or tight spacing. While reducing LAYER1_GRIDSPACING may require hand editing of a channel solution, it can significantly reduce channel length.

LAYER2_GRIDSPACING is the minimum distance that two horizontal tracks may be spaced. This parameter is minimized to a pitch of geometry-to-contact spacing. The channel router places tracks at this tight GRIDSPACING unless there are two vertically adjacent contacts. It then **automatically** widens the track spacing

from this contact-to-geometry spacing to a contact-to-contact spacing. In the CMOS-PW case, this parameter is 9 lambda:

```
LAYER2_GRIDSPACING = routing geometry's min width
                     + MAX ( layers' minimum spacing )
                     + ( contact overlap ) / 2;
         = LAYER2_MINWIDTH
           + MAX ( LAYER1_MINSPACING, LAYER2_MINSPACING )
           + ( CONT_SIZE - LAYER2_MINWIDTH ) / 2;
         = 3 + MAX( 3,4 ) + ( 7-3 ) / 2;
         = 9.
```

The design rule file must be included to run any router. If the HAWKROUTERS file is not included, or a routing parameter is missing, the user is notified and the route is aborted.