BEAR-FP Manual

Distribution 1.0

by

Massoud Pedram
Wei-Ming Dai
Margaret Marek-Sadowska
George Carvalho, Jnr.
Deborah Wang
Benjamin Chen

Memorandum No. UCB/ERL M90/118

December 20, 1990

BEAR-FP Manual

Distribution 1.0

by

Massoud Pedram
Wei-Ming Dai
Margaret Marek-Sadowska
George Carvalho, Jnr.
Deborah Wang
Benjamin Chen

Memorandum No. UCB/ERL M90/118

December 20, 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BEAR-FP Manual

Distribution 1.0

by

Massoud Pedram
Wei-Ming Dai
Margaret Marek-Sadowska
George Carvalho, Jnr.
Deborah Wang
Benjamin Chen

Memorandum No. UCB/ERL M90/118

December 20, 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BEAR-FP Manual

Table of Contents

Introduction

BEAR-FP is a macrocell-based layout system which builds on its predecessor, the BEAR system [1]. It uses BEAR's dynamic and efficient data representation, which unifies topological and geometrical information, and BEAR's routing system. BEAR-FP, however, has a new floorplanning procedure with integrated global routing and hierarchical pin assignment, a new Steiner-tree global router, a detailed channel pin arrangement procedure, and a timing-driven clustering and placement capability. BEAR-FP supports traditional macro-cell layout with routing channels as well as channel-free layout style.

The guiding principle of BEAR-FP is that of stepwise refinement, which implies that the geometrical positions and interface characteristics (shape and pin locations) of cells are determined gradually and in a top-down fashion. For example, the positions of floating pins on cells are initially determined based on minimizing total wire length in the chip. These positions are later modified to honor channel capacity constraints at a minimal increase in wire length. Before detailed routing, the pin positions within a channel can be changed to minimize the routing density. As another example, the shapes of flexible cells are determined during the top-down floorplanning. These shapes can be further optimized after global routing (when more detailed information about connection paths exist).

BEAR-FP provides a complete pipeline from a net list specification of a circuit to a finished layout. The user specifies various physical, timing and clustering constraints, controls the order that various optimization tools are invoked and sets parameters to control the outcomes of these procedures. BEAR-FP addresses the issue of performance optimization by including a scheme for timing-driven layout that spans the entire layout process, i.e., the net-based timing constraints influence the clustering, floorplanning, pin assignment and global routing steps.

BEAR-FP can be used for "early" floorplanning as well. In this mode, it does a quick floorplanning whose results can be used to guide the logic synthesis procedure by giving the logic design tools information about the interface characteristics (shape and pin distribution) of the blocks and about how the physical design process can impact the timing on certain critical paths of the logic.

This manual describes the BEAR-FP system in three sections. In section 1, installation instructions and a detailed explanation of the user interface are given, as well as a step-by-step example to help users get started. Clustering, floorplanning, pin assignment, and shape optimization are described in Section II. Although this part of the system is fully automated, the set of parameters and graphic display features provided allow the user to tailor the results to suit their individual needs. In Section III, routing and spacing processes are discussed. While we provide a set of operations (or mechanisms), users have the freedom to choose when and how to apply them. The appendices contain descriptions of the various data files BEAR-FP relies on.

BEAR-FP runs on workstations which support the X-window system (currently X11, release 4). Users may customize the colors, font styles, and other parameters by setting the *X defaults* file (Appendix 3). The BEAR-FP system has been integrated into the OCT framework (Appendix 2).

Acknowledgements

We are indebted to Professor Ernest Kuh for his guidance and support of our work on this project. In addition to those mentioned in the references, we wish to thank Tim Collins, Sangjin Hong, Denis Lee, and June Wang, who helped write and debug the C code. We are also grateful to Tahani Sticpewich for compiling this manual and for accommodating our frequent changes and last-minute corrections.

I. Installation and User Interface

A. General Information

1. Hardware requirements.

- Disk Space: about 40 megabytes of disk space is required to compile the system on Digital's DECstation 3100.

2. Software requirements.

- X-Window System, version 11, release 4.
- UNIX operating system. The system has been tested on Ultrix Worksystem V2.1, Rev. 14.
- C compiler: cc. The system has not been tested using gcc.
- Optional: OCT library.

B. Getting Started

This section of the manual gives step-by-step instructions to help you get BEAR-FP up and running on your system.

1. Reading from the tape.

BEAR-FP is stored in the tar format so it can be read by any UNIX-like system. The blocking factor of the tape record is 20, which should be the default on most systems. At any rate, this parameter is already determined when reading in the tapes. To get BEAR-FP onto your system, you should do the following:

a. Load the tape into your tape drive so that it is ready for reading. Be sure the write-protect mechanism on the tape is activated to avoid accidental erasure of media.

b. Make a directory where BEAR-FP will reside. A typical command might be:

% mkdir /users/bear

This directory will be the one used for the rest of this guide.

c. Now you are ready to read from the tape.

% tar x /users/bear

This process will take a while depending on how fast your system is, so be patient.

d. When the prompt returns, the taping has been finished. Type

% cd /users/bear/BearFP_1.0_Tape/src

and then

% ls

to list the source directories that should be present:

Include	globalSpacer	placer
Installed	grEditor	polarBear
Lint	gtGraph	polarBearTest
Object	iv	rectSlice
bblInterface	list	ringRouter
bearGrUtils	localRouter	routeDB
bearMM	localSpacer	st
bearMisc	localVia	textio
bfpInterface	mac	tileMapping
cluster	magInterface	tileProp
cpa	misII	tiles

errtrap	newC	topChDecomposer
ezPlot	newG	ts
fpg	newP	uprintf
geoChDecomposer	octInterface	utility
globalRouter	pgRoute	utils

2. *Compiling* BEAR-FP.

To compile the system, some variables must be set describing the environment in which BEAR-FP resides. Also, a few local directories must be installed before proceeding to compile BEAR-FP.

a. The main Makefile must be edited to match the system. This file resides in /users/bear/BearFP_1.0_Tape. You can use your favorite editor to do accomplish this task. Find the place in the Makefile where you see:

BEAR_DIR = /users/bear/BearFP_1.0_Tape

This specifies where the BEAR-FP system resides. In this case, the default directory matches the directory previously recreated. If the default differed then */users/bear* would be replaced by the directory specified by the initial **mkdir** command.

b. Indicate where the X11 libraries are located by filling in the makefile variable X11R4_LIB_DIR.

c. Indicate where the X11 header files are located by filling in the makefile variable X11R4_IFLAGS.

d. Quit the editor. Be sure you are in the BEAR-FP directory. In this case, it is */users/bear/BearFP_1.0_Tape*. Now type:

% make install

This directive installs the directories necessary to support the BEAR-FP compilation and coding environment, and then compiles all of the libraries, support modules, and BEAR-FP modules and links them together. Note that the default configuration will contain the new placement package, and that the runnable program called *polarBear* will reside in */users/bear/BearFP_1.0_Tape/bin*. If changes are made to the BEAR-FP code, the BEAR-FP modules can be recompiled and linked by typing **make debugbear**

C. User Interface

BEAR-FP's user interface is based on the X-Window System, and employs various types of windows to converse with the user and display the status of the program. The console window is the main window from which commands are issued. Chip windows graphically display the status and characteristics of the current layout example. And interactive variable windows (IV) allow the user to modify various parameters of the program.

1. The console window.

The console window of BEAR-FP is the main area from which the user directs program action. Commands are specified in the UNIX tradition. Each command consists of a command name followed by an optional list of arguments usually preceded by a '-'. Each command that has optional arguments recognizes the **-help** option which causes a small summary of all the command options to be printed out in the console window. (See appendix listing of available commands.) For primitive editing, a few simple key strokes have been defined. The last word of text on a command line can be deleted by typing **control-w** (^W), while the entire line may be deleted with a **control-u** (^U). The console window may be closed by typing **control-d** (^D).

Unless a default geometry has been specified in the user's *~/.Xdefaults* (See XDefaults), the program prompts the user to create the console window upon invoking the program. Once the console window has been placed, a prompt is displayed when the program is ready to accept commands. Holding down the middle mouse button on the window will display a menu of commands directly related to the console.

Along the right side of the console window is a scroll bar window. Scrolling is controlled by pushing mouse buttons in the scroll bar. The middle button scrolls to a particular spot in the window. This operation causes the screen to scroll so that the center of the scroll bar indicator moves to the current position of the mouse. The other two mouse buttons are used for scrolling down or up some proportion of the screen. The left button causes the screen to scroll so that the line adjacent to the mouse position becomes the top line of the screen. Thus, clicking near the top of the scroll bar scrolls only a couple of lines while a click near the bottom will scroll almost an entire screen. The right button causes the top line of the screen to scroll down to the current position of the mouse.

2. The chip window.

The chip window of BEAR-FP is the main area in which the user can view the results of their commands. It displays the current layout of the chip using graphical abstractions. Cells are represented by rectilinear shapes. Pins are fixed-size squares usually placed along the inner boundary of cells. Regions of the chip window can be magnified for closer inspection. Other abstractions displayed in the manual are explained in detail below.

3. Interactive variable windows (IV).

This form of dialog is specifically used to view and edit program variables. The appearance of these dialogs is very distinct. A title describing the current operation is displayed at the top of the dialog. All the interactive variables are shown on a window, one on each row. Each variable is displayed with its description and a region containing its current value.

At any one time, the IV window maintains at most one active edit region where the variable may be changed. All keyboard input anywhere in the IV window will be directed to this region. Edit regions are activated by placing the mouse cursor over an edit region, and either clicking a mouse button or pressing a key. This action is indicated by a cursor (a pointer under a line of text) inside the active edit-region. The user is not allowed to enter more text than there is space in the edit region component. *Changes are accepted only by a carriage return or end-of-file.* The original value of the variable can be restored by typing **control-u (^U)** before accepting any changes. Edit regions are usually denoted by a different color background where the value of the variable is displayed. Edit regions whose background color matches the background color of the entire window are read-only, unless buttons are present.

For integer or floating-point variables, two buttons are provided to change the value of the variable. The "+" button has the following effect:
If the *LEFT* mouse button is pressed, the value of the variable is incremented by 1%, or by one for integer variables.
If the *MIDDLE* mouse button is pressed, the value of the variable is incremented by 10%
If the *RIGHT* mouse button is pressed, the value of the variable is doubled.

The "-" button has similar behavior, but the value of the variable is decremented. Integer variables can be distinguished from floating point variables by the presence of a decimal point. For variables with strings as values (except booleans), the plus and minus buttons advances or reviews through a list of values that the user can choose. For boolean variables, one button is provided for easy toggling of its state. Single buttons are also provided for directing actions such as aborting the dialog.

D. Sample Layout

This section of the manual describes a typical set of instructions that is used to complete the routing of a chip. The data used here are located in */users/bear/BearFP_1.0_Tape/src/polarBear/bearData*, along with a number of other examples.

1. Starting the program.

The first step is to start the program:

% polarBear

Be sure your DISPLAY environment variable is set. If it is not, the user may specify the display on which to run BEAR-FP by typing **polarBear -display cowabunga:0.0**, where **cowabunga** is the hostname of the destination machine.

2. Start log file.

At this point the BEAR-FP console window will be created and a prompt displayed. (See Fig. 1.)

Fig. 1.

Now the user should type:

bear > **log sample.log**

This creates a file in your current working directory called *sample.log*. It will contain all messages echoed to the console window from the time the user invoked the command.

3. Read in the placement file for the chip.

Next, type:

bear > **ow -bbl testData/Iccad1.r**

The command **ow** stands for open window and in this case has been used to read in an example described in the BBL format which is located in the directory named testData. This command causes a window to be created in which the cell layout will be displayed (see Fig. 2).

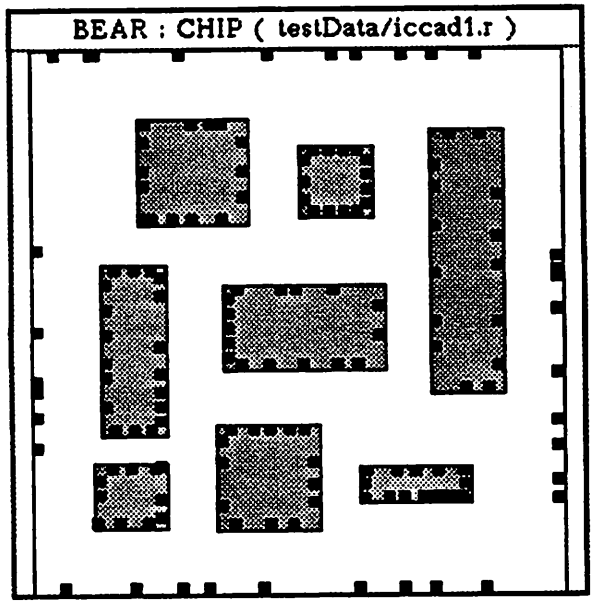

Fig. 2.

Note that the *testData* directory mentioned above is in your current working directory in this case. Although not specified directly, one other file must exist in the same directory as *iccad1.r*. This file is *iccad1.tech*. The format and usage of these files are described in greater detail in a separate section of the manual.

4. Create a clustering tree with the new version of the cluster package.

Now that the initial placement file has been loaded, BEAR-FP is ready to optimize the placement. This step is optional and the user may proceed directly to routing. Usually, the first step in placement optimization is the creation of a *clustering tree*. This task is accomplished by the **ncl** command. Remember that the mouse cursor must be in the chip window for this command to be valid.

bear > **ncl**

After this command is entered, an IV (interactive variable) window will be created since more information is required. (See Fig. 3a.) The user must specify the type of clustering algorithm. The matching algorithm usually gives good results. Select this algorithm by placing the mouse cursor over the box adjacent to the name and clicking the button. Then click the button over the *Okay* box. *Abort* will close the dialog box and return input to the console window. After the choice has been made, a message will be echoed to the console:

ncl -alg m

This message is an abbreviation for the command specified by the dialog box. In the future, the user may wish to enter the abbreviation instead of working with the dialog box.

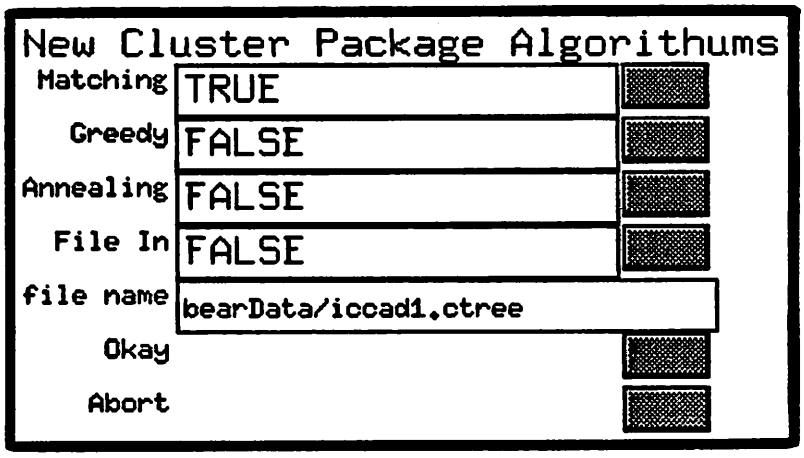

Fig. 3a.

Next, an IV window is created centered around the current mouse position. (See Fig. 3b.)

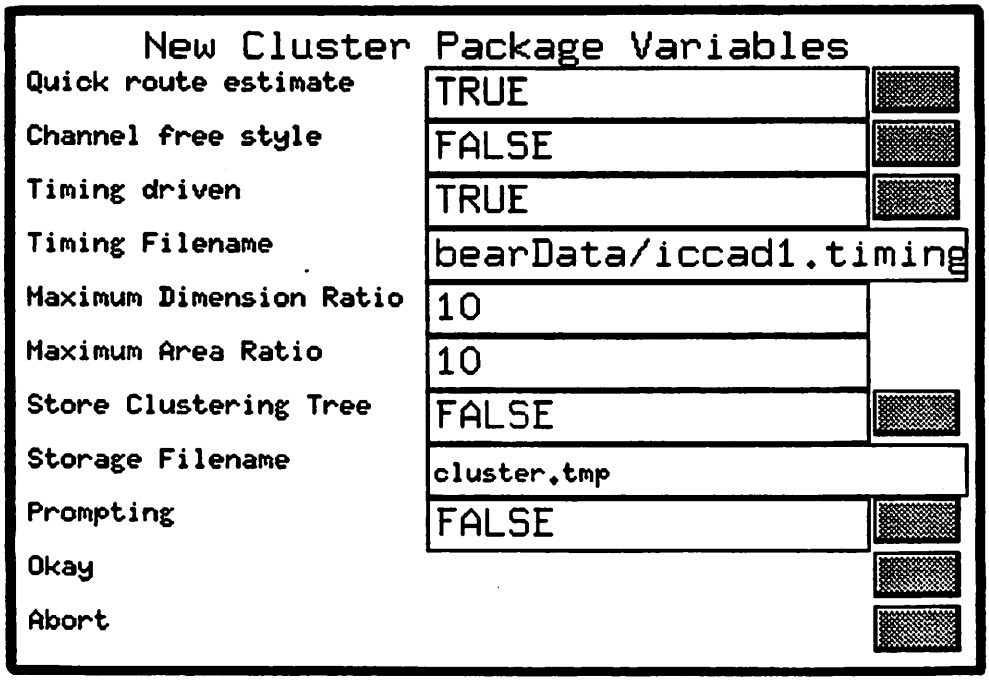

Fig. 3b.

Variables are displayed to allow the user to change the default values for the clustering parameters (see section on Clustering and Placement). In general, the user accepts the default values within IV windows. In this case, a click on the button adjacent to the *Cluster cells* label will accept the defaults and begin generating the clustering tree. When the tree has been generated, this message will be displayed in the console window:

Clustering is done.

And the prompt will return.

5. Placement of cells with the new version of the placement package.

Now the placement of the cells can be computed. This task is done with the **npl** command.

bear > **npl**

An IV window will appear with default values in its fields. (See Fig. 4.) If the user chooses a fixed target shape (fixed x or fixed y), the dimension value will be used and the aspect ratio value will be ignored. Similarly, if the user wants to specify the aspect ratio of the target shape, the aspect ratio value will be used and the dimension value will be ignored. In this case, all of the defaults will be accepted by clicking the mouse over *Okay*. As with the cluster command, the equivalent command line sequence is echoed to the console window:

npl -ar 1.00

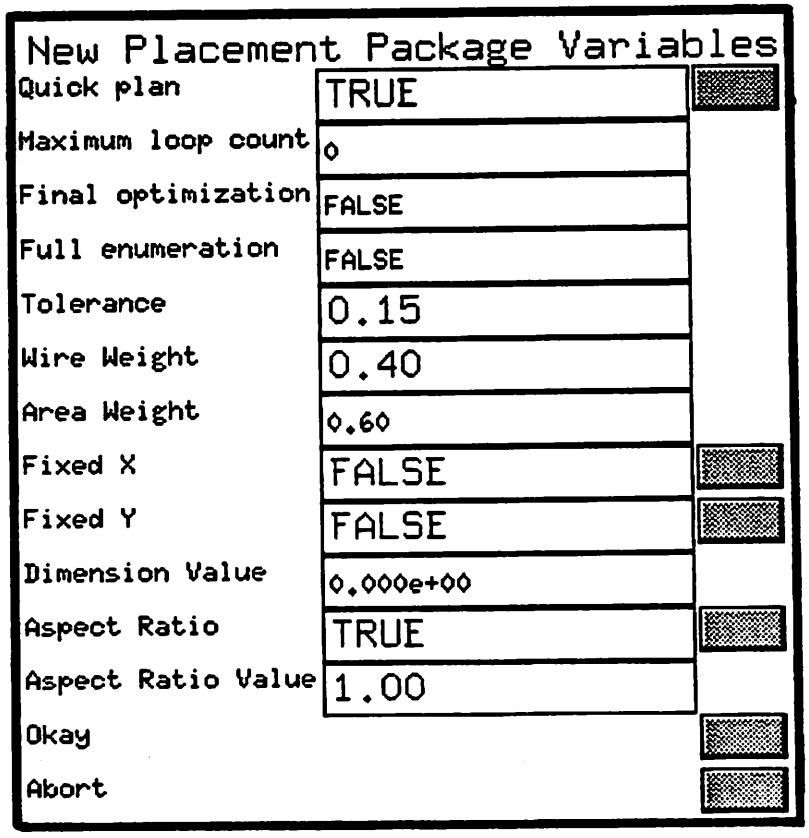

Fig. 4.

See the section on Clustering and Placement for details. The default value is printed in the brackets and is accepted by a carriage return. This portion of the program usually takes some time to finish. The conclusion of placement is signaled by a message:

Placement is done.

and the command line returns.

6. Global routing.

Now that the placement is completed, the example is ready for global routing:

bear > **gr**

The congestion factor and timing mode must then be specified (See section on Global Routing, III B):

Congestion Factor [0]:

An IV window similar to the cluster IV window will appear showing the default values. When the global routing is done, the prompt will return.

7. Channel Pin Arrangement.

Now you may rearrange pins within some of the bottleneck tiles. Note that this is optional.

bear > **cpa**

An IV window will appear to assist you. Fig. 5 gives an example of a channel before and after channel pin arrangement.

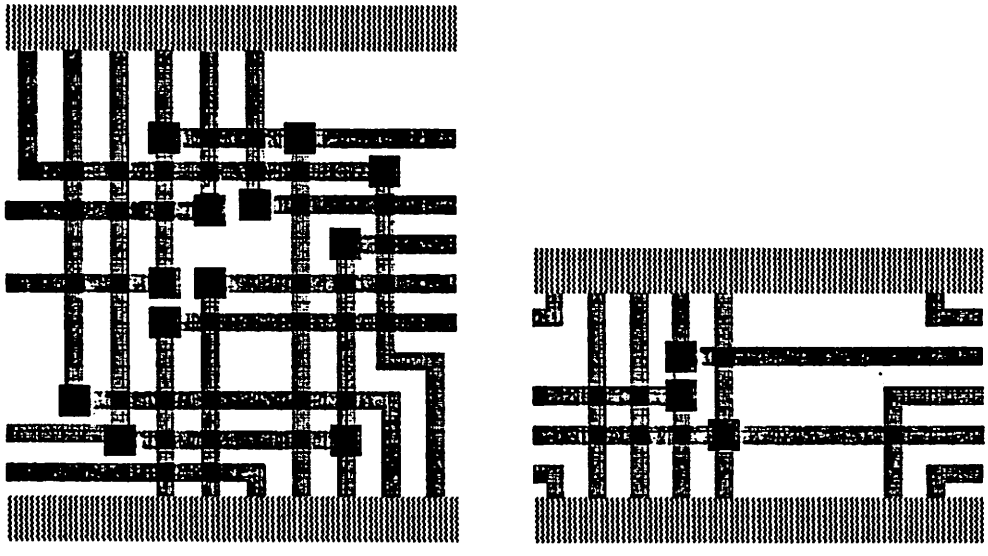

Fig. 5.

8. Shape optimization.

For further space minimization of the floorplan, the user may want to alter the shape of the cell blocks but still stay within the technology constraints. This task is accomplished by the shape optimizer command:

bear > **so**

Another IV window will emerge. (See Fig. 6.)

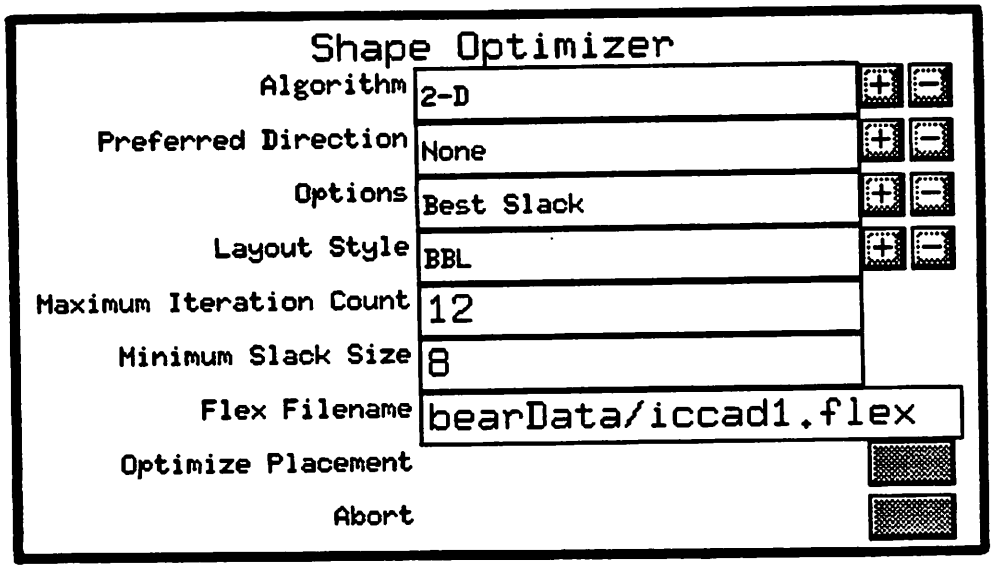

Fig. 6.

See the section on Shape Optimization (Sec. II E) for explanation of variables. Clicking the mouse button next to *Optimize Placement* will begin the algorithm. When the algorithm is finished:

Shape Optimization finished

will be echoed, and the prompt will return to the console.

9. Global spacing.

Now the example is ready for further global spacing. First, horizontal compaction will be invoked:

bear > **hcm**

Horizontal compaction is done.

Similarly, vertical compaction is invoked:

bear > **vcm**

Vertical compaction is done.

10. Detailed routing.

The next step is the detail routing:

bear > **dch**

This command stands for "define channel." The chip window will be redrawn showing the floorplan graph. (See Fig. 7.)

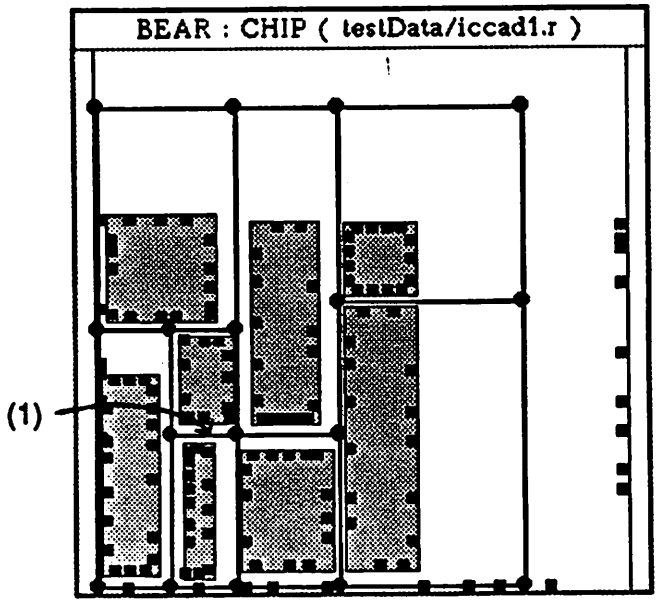

Fig. 7.

Independent channels will be highlighted. These channels are the valid channels that may currently be routed. A channel is picked by clicking the mouse over two junctions on the floorplan graph. At this point, this cursor will have changed from its normal cross to a circle that is to be placed over a junction. The user will then be prompted:

Pick junction number one.

The user should then move the circle over an end junction of a highlighted channel and click the left mouse button. If the user did not select a valid end junction, the error:

Picking junction failed.

will be echoed to the console window, and the console prompt will return. Choose the channel marked (1) on Fig. 7. The second junction is specified in a similar manner.

Pick junction number two.

When this junction is successfully chosen, an IV window is displayed showing routing parameters. (See Fig. 8.)

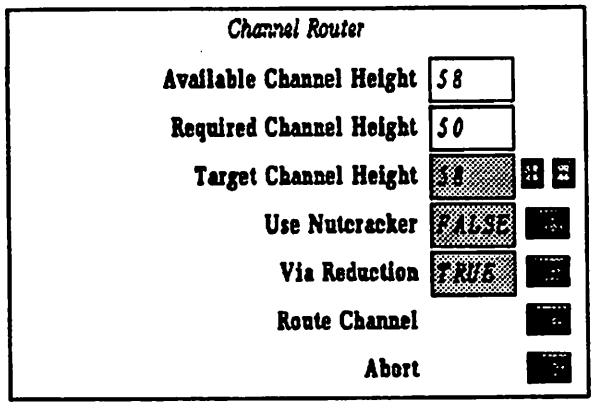

Fig. 8.

Nutcracker is a local spacing routine which attempts to compact the routing to fit the available channel height. (See manual section on Detailed Routing.) All of the parameters are described in the Detailed Routing section of the manual, however, two particular parameters should be noted: available channel height and required channel height. If the required height is greater than the available height, then the channel should not be routed and the command should be aborted. Otherwise, there is enough space to accommodate the routing, and the user should click on *Route Channel.* Since sufficient space exists in this case, the channel can be routed.

When the routing is completed, the route cell will be drawn. The user can closely examine the route cell by defining a region to magnify:

bear > **ow**

Opening a window without any arguments allows the user to specify a region of a chip window to display. The user will be prompted to:

pick a rectangle

to define the region. The mouse button should be clicked and held down while moving the mouse itself to define the region. Once the button is released the window is created focused on that region. (See Fig. 9.)

In any window except the console window, the user can zoom in, zoom out, magnify an area, or pan the view. In this case, moving the mouse to the newly created region and typing:

bear > **Z**

will show more detail of the routing. To destroy this window, type:

bear > **<control-d>**

Now the user can define another channel (2):

bear > **dch**

Pick junction number one.

Pick junction number two.

At this point if the user does not have enough space in the channel to fit the routing, the command must be aborted and decompaction must be done. If the channel to be routed is a vertical channel, horizontal decompaction is necessary. If the channel to be routed is a horizontal channel, vertical decompaction must be done as in this case:

bear > **vdcm**

Vertical decompaction is done.

The program is aware of the amount of space that the channel lacks for routing. The decompaction routines tries to adjust channel height accordingly so in the next attempt to define a channel:

bear > **dch**

Pick junction number one.

Pick junction number two.

the available height is as close to the required height as possible. If the channel is not on a critical path, however, the estimate may not be that close.

After routing a channel, it is always a good idea to run a decompaction routine to minimize the number of attempts to define a channel. In this case it is:

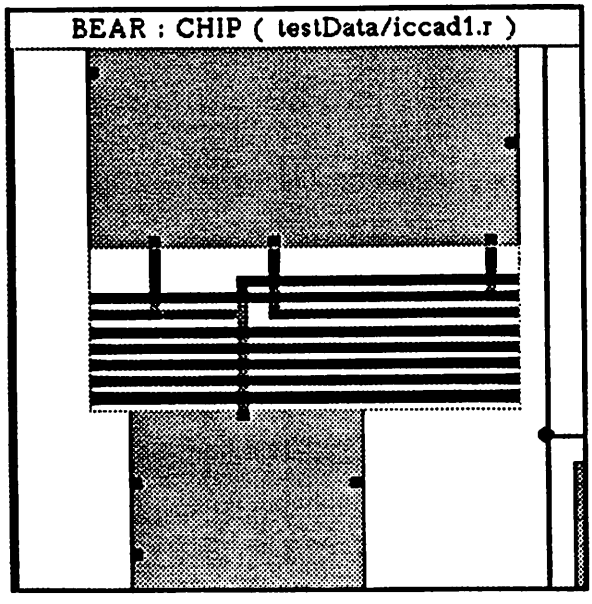

Fig. 9.

bear > **hdcm**

Horizontal decompaction is done.

The rest of the example is routed similarly. When all of the channels are routed, the floorplan will appear as in Fig. 10.

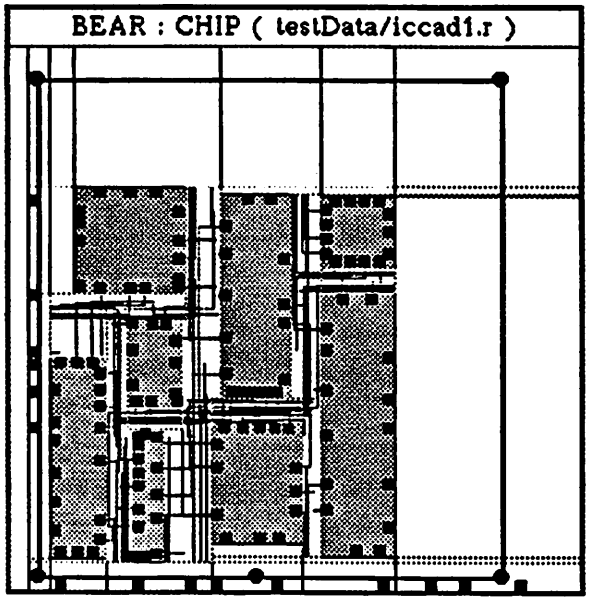

Fig. 10.

11. Resizing the parent cell.

Due to the compaction of cells, much space is left over in the parent cell, as can be seen in Fig. 10. This parent cell can be resized by:

bear > **rp**

At this point the mouse cursor changes to a circle and the user must select a corner or edge of the parent cell to drag to reduce the cell boundaries. In this case, the upper right corner of the cell is appropriate to begin dragging. As in the open window command, a rectangular outline is rubber-banded indicating the target size of the parent cell. When the resizing is finished, the console window will display:

Resizing...done.

(See Fig. 11.)

If the cell blocks with routing are not centered in the parent cell, they can be moved all at once using the transform cell command:

bear > **tc -lm**

The **-lm** option specifies an interactive mode. When the circular cursor appears, the user must hold down the mouse button on any of the cell blocks and move the mouse. The outline of

all the cells will be displayed giving the user an idea of where the cells will be oriented after releasing the button.

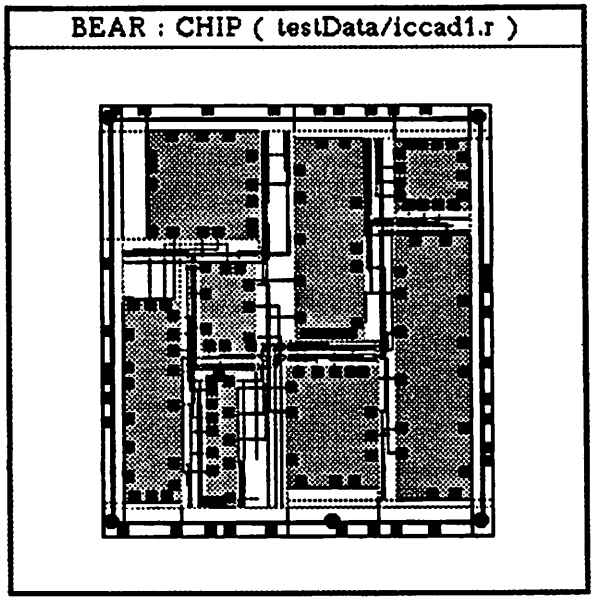

Fig. 11.

12. Ring routing.

The final step in the routing process is the ring routing:

bear > **rr**

An IV window will be opened displaying the default filename for the output of the ring router. When the ring router is finished, the layout is final. (See Fig. 12.)

13. Saving the example.

The user may want to save the example in CIF format. The command is:

bear > **w -clf iccad1.clf**

where **iccad1.clf** is the name of the file to be written. A particularly useful feature of the save command is the scaling option. The user could have typed:

bear > **w -clf iccad1.clf -scale 5**

All geometrical specifications in the CIF file will then be scaled by a factor of 5 in the horizontal and vertical directions.

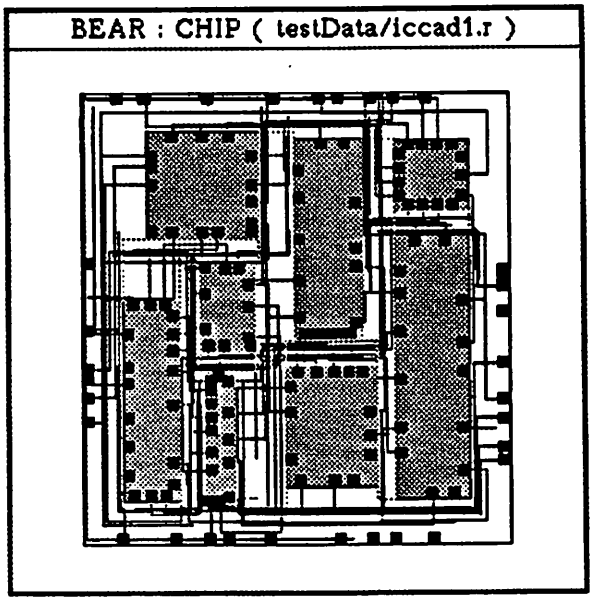

Fig. 12.

14. Leaving the program.

To leave the program, the user must move the mouse to the console window and type:

bear > **cw**

II. Hierarchical Floorplanning with Integrated Pin Assignment and Global Routing

During the early stages in the design of electronic systems, decisions are made which have a dramatic effect on the quality (performance, density or area) of the resulting design. Choices must be made in partitioning functions into physical cells and in choosing interface characteristics of the cells such as size, shape and pin positions. These choices are difficult because their effects on the final layout are hard to predict and may not become apparent until much later in the design process. Floorplanning helps solve this problem. It is a procedure for allocating adequate area and assigning shapes and pin locations to system modules minimizing the layout area. Extensions are obtained if performance criteria, such as critical nets, interconnection length, and power dissipation are considered.

Our floorplanner is based on bottom-up clustering, shape function calculation, and top down floorplan computation with integrated global routing and pin assignment. It performs shape function estimation and assignment of macro-cell shapes, positions and pin locations, satisfying various constraints as it does so.

Besides unifying several existing ideas, the floorplanner introduces the following new components:

(1) Accurate and dynamic routing area estimation during floorplan computation in order to avoid an increase in the chip area after global routing.

(2) A systematic optimization procedure during the selection of suitable floorplan patterns that integrates floorplanning, global routing and pin assignment.

(3) Extensions to incorporate timing issues by a novel timing-driven clustering technique, followed by bottom-up estimation of interconnection length for critical nets and multi-function optimization procedure which includes a critical net length violation cost term.

(4) A new pin assignment technique based on linear assignment and driven by the global routing solution.

(5) Accommodation of channel-free style layout which is especially useful when placing mixed macro-cell and standard-cell assemblies.

A. Clustering

We initially generate a hierarchical representation of the circuit in the form of a multi-way cluster tree. This tree is generated bottom-up and is obtained by minimizing connections among various cells [2]. However, to avoid a cell shape mismatch in the clusters that makes it difficult to find a good placement for cells, shapes of the cells are also considered. Each leaf in the tree corresponds to an actual cell and each internal node (which we call a cluster node) represents a collection of highly connected cells (or clusters of cells). The maximum branching factor in the tree is restricted to a small value (e.g., 4). This restriction is necessary because the number of multi-way floorplan patterns increases dramatically as the branching factor is increased. In addition, if the branching factor is too large, the problem of finding a floorplan solution for a node in the tree

becomes as complex as the general floorplanning problem. In the timing mode, the upper bound net length constraints are used to modify the natural connectivities among blocks, and a cluster tree reflecting the timing requirements is generated. (See [3].) Note that the current implementation does not handle lower bound net length constraints.

The following description of the user interface of the BEAR-FP placement program assumes that a chip window has been opened. In order to obtain a new placement, a hierarchical cluster tree must be built. Placement is then performed by traversing the tree top-down and placing the elements of each node optimally [4, 5]. The clustering algorithm can be invoked by typing **ncl** while the cursor is in the chip window. A small pop-up window will ask the user to specify the type of clustering that is desired. Four algorithms are available:

- *Matching:* generates a cluster tree by optimal pairwise matching of modules and clusters [2].
- *Greedy:* does clustering based on a greedy heuristic.
- *Annealing:* generates a cluster tree using simulated annealing.
- *Input from File:* reads the clustering tree directly from a file (see Appendix 1).

The best clustering results are often obtained by the matching algorithm. After the desired algorithm is selected, a *Cluster Parameter* window will pop up. We shall describe each parameter, its effect on the clustering procedure, a typical range of values for it, and its default value. In particular, the matching algorithm parameter set is described. Parameters for other algorithms have similar meanings and ranges of values (see following table):

Clustering Parameter Set		
parameter	*range*	*default*
Quick Routing Area Estimation Flag	TRUE, FALSE	FALSE
Channel Free Flag	TRUE, FALSE	FALSE
Timing Flag	TRUE, FALSE	FALSE
Maximum Dimension Ratio	1.0 — 10.0	10.0
Maximum Area Ratio	1.0 — 10.0	10.0
Prompting Flag	TRUE, FALSE	FALSE

The user may specify default values for most of these parameters as indicated below with a file using the **-cf** option of the **ncl** command (see Appendix 1).

1. Timing flag (newC.TimingFlag).

If this boolean flag is TRUE, the clustering algorithm will look for the timing file and will use the net length constraints specified in that file to do timing-driven clustering. The floorplanning step which follows will do timing-driven placement. At the end timing statistics regarding number of

satisfied constraints will be printed on the standard output.

2. Quick routing area estimation flag (newC.QuickRouteEstFlag).

If this boolean flag is TRUE, the required routing area around a given cell is estimated based on the total number of pins attached to the cell. Otherwise, routing area is estimated based on the topology in which the cell is placed and the probability that a connection passes through a particular channel [6]. The first option is about 2-3 times faster and often produces reasonable routing area estimates.

3. Channel free style flag (newC.ChannelFreeFlag).

If this boolean flag is TRUE, the routing area around all cells will be a minimum fixed value (e.g., four track-track spacing). This option is useful to support channel free layout styles and mixed standard-cell (grouped into virtual flexible macro-cells) and macro-cell layout style.

4. Maximum dimension and maximum area ratios.

A clustering based only on connectivity information can result in a block shape mismatch that makes it impossible for the placement algorithm to avoid big *dead space* areas. It is our conjecture that two blocks do not match if (a) their areas and (b) the length of their longer sides are sufficiently different. A simple implementation is to prohibit the merging of block pairs whose areas or lengths differ by more than some ratio. The maximum dimension and maximum area ratios are computed based on the distribution of block sizes and areas available for clustering at each level of the hierarchy. Note that these parameters only appear in the dialog window for the matching and greedy clustering algorithms.

5. Prompting flag.

If this flag is set to *FALSE*, the clustering algorithm will use the internally computed values for the maximum dimension and maximum area ratios. If the flag is set to *TRUE*, it will stop and ask the user to enter new values (or accept the defaults) at each level of the hierarchy. The user may want to experiment with these ratios, however, the default values are often good.

B. Floorplanning

Each cell has a shape function that defines its height as a function of its width. (Fixed shape cells are a subset of flexible shape cells. A fixed shape cell has a one- or two-point shape function representing the cell and its rotation.) During the clustering phase, we generate the composite

shape function for every cluster node by extending the procedure in [7, 8] to multi-way cluster nodes with no pre-specified cut direction. (See [9].) These functions are used during the top down floorplanning to guide the search for a good floorplanning solution.

In the top down phase we start from the root of the cluster tree and floorplan the nodes in a breadth first manner. As a result of floorplanning a cluster node, we assign shapes and positions to its child nodes and update the current partial floorplan solution. Next, we enter the pin assignment and global routing phase for the node. We do an initial pin assignment which produces a solution minimizing the total interconnection length. We then perform the global routing which produces shortest connection paths for all nets. After global routing, if capacity constraints for some channels are violated, we perform a new pin assignment which re-positions the floating pins to reduce congestions in the over-subscribed channels. (The user controls the number of iterations between global routing and linear-sum assignment. We have observed that one iteration will suffice.) The process of the top down traversal of the cluster tree continues until the leaf level is reached [10].

In the timing mode, we sum the cost of violating the given constraints on each critical net. This sum is then linearly added to other cost terms. The cost has the form of a penalty function. That is, if the bound for a critical net is violated the cost rises sharply, else it is a linear function of wire length.

The placer can be invoked by typing **npl** while the cursor is in the chip window. To start the placer, a few input parameters are requested from the user (see following table):

Placement Parameter Set		
parameter	*range*	*default*
Chip Goal Shape	Fixed-X, Fixed-Y, Aspect-Ratio	Aspect-Ratio
Fixed-X Dimension	integer	computed
Fixed-Y Dimension	integer	computed
Aspect-Ratio	0.2 — 5.0	1.0
Border Size	greater than 1.25	1.5
Quick Plan Flag	TRUE, FALSE	FALSE
Max Loop Count	0 — 2	1
Final Optimization Flag	TRUE, FALSE	TRUE
Full Enumeration Flag	TRUE, FALSE	TRUE
Tolerance	positive real number	0.15
Area Weight	positive real number	0.6
Wire Weight	positive real number	0.4
Critical Weight	positive real number	1.0

The user may specify default values for most of these parameters as indicated below with a file using the **-cf** option of the **npl** command (see Appendix 1).

1. Determination of the chip goal shape.

The desired shape of the final layout can be specified in either of two ways: as goal aspect ratio (ratio of width to height) or as a fixed width or height of one dimension of the layout. The default shape in either case is a square. Although it cannot be guaranteed that the specified goal can be achieved exactly, the results are never far away from the desired value. Because the goal shape plays an important role in the computation of the objective function, it is often helpful to play around with this number to get the best result (for example, a change from 1.1 to 1.15 may have a big impact).

2. Border size (newP.BorderSize).

This parameter specifies spacing between the edges of the core bounding box (rectangle enclosing all placed blocks) and the edges of chip bounding box (rectangle defined by the off-chip IO pads). This parameter is especially useful because of the following situation. After we floorplan a cluster node, we delete the block representing the node and insert its child blocks into the tile plane. We then do global spacing on the entire tile plane. It is assumed that if the core bounding box is expanded to accommodate the child blocks, the core boundary will still lie within the chip boundary. However, if the value of the BorderSize parameter is small, this assumption may be violated and the floorplanning will abort. The solution is to increase the value of the parameter and repeat the process. (This is a temporary hack.)

3. Full enumeration flag (newP.FullEnumFlag).

During the clustering phase, we compute the shape functions for each internal node in the cluster tree. During the top-down phase, we use the user-specified aspect ratio (or chip dimensions) at the root of cluster tree and then recursively compute the best shapes and positions for the internal nodes.

If this boolean flag is set to TRUE, we do a full enumeration of all floorplan templates, template orientations and cluster-to-room assignments and use a linear cost function (consisting of area penalty, wire cost, and critical net bound violation cost) to find the best floorplanning solution. (We use the bottom-up shape functions only to estimate areas of subclusters.) Otherwise, a point on the shape function for the root node which satisfies the user specified aspect ratio is chosen and, using hints saved during the bottom-up process, floorplan templates are assigned to internal nodes of the cluster tree.

This step is then followed by an optimization procedure aimed at minimizing the total interconnection length or satisfying the timing constraints by switching cells across cuts. During this procedure, we only enumerate various template orientations and those cluster-to-room assignments which lead to little or no increase in floorplan area.

The objective function to be minimized is then a linear combination of wire cost and critical net bound violation cost. The partial enumeration option is on average about three times faster than the full enumeration.

4. Maximum loop count (newP.MaxLoopCnt).

The pin assignment step can be interwoven with the global router in order to assign pin positions minimizing interconnection length and satisfying channel capacity constraints. The loop count states the number of iterations between linear-sum assignment step and the global routing step. In particular, a value of zero indicates pin assignment without any global routing iteration.

5. Final orientation optimization flag (newP.FinalOptFlag).

We always perform a greedy orientation optimization on the floorplan solution in order to reduce wire length by mirroring or flipping fixed-shape cells. However, it is possible to perform an exhaustive orientation optimization as well. If this boolean flag is set to TRUE, such optimization will be performed on all clusters which contain at least one fixed-shape child cell. Since this step is exhaustive enumeration of all possible reflections and uses the global net list, it tends to consume CPU time. The result of greedy optimization is usually as good as those of exhaustive optimization.

6. Quick plan flag (newP.QuickPlanFlag).

This option has been provided for convenience. It is equivalent to setting newP.FullEnumFlag to FALSE, newP.MaxLoopCount to zero and newP.FinalOptFlag to FALSE. Once set, this flag overrides the settings of the three parameters mentioned above.

7. Cost function parameters.

Tolerance factor specifies the maximum amount by which the cost of a candidate floorplan solution can exceed that of the best known floorplan solution in one of the components of the cost function and still be considered for further evaluation (in terms of other cost components). Weight factors give the relative weight of various cost components in the linear objective function. Internally, the sum of cost coefficients is normalized to 1. Note that when newP.FullEnumFlag is set to TRUE (FALSE), newP.FullEnum (newP.PartialEnum) parameters are read.

C. Global Routing

This global router operates on the floorplan graph and builds a minimal Steiner tree approximation for each net. The algorithm is presented in [11] and has the following steps.

INPUT: the floorplan graph which is an undirected distance graph $G = (V, E, d)$ and a net, S.

OUTPUT: a Steiner tree, T_H, for G and S.

(1) Insert the net pins into the floorplan graph. Let the undirected distance graph $G = (V, E, d)$ denote the augmented floorplan graph and S denote the newly created vertices representing the net pins.

(2) Construct the complete undirected distance graph $G_1 = (V_1, E_1, d_1)$ from G and S.

(3) Find the minimal spanning tree, T_1, of G_1.

(4) Construct the subgraph, G_s, of G by replacing each edge in T_1 by its corresponding shortest path in G.

(5) Find the minimal spanning tree, T_s, of G_s.

(6) Construct a Steiner tree, T_H, from T_s by deleting edges in T_s, if necessary, so that all the leaves in T_H are Steiner points. It is shown that T_H produced by this algorithm has a total length no more than $2(1-\frac{1}{p})$ times that of the optimal tree where p is the number of net pins.

D. Channel Pin Arrangement

The global pin assignment procedure based on linear sum assignment does not find the optimal pin locations within each routing channel, and therefore, must be followed by a channel pin arrangement procedure. Our channel pin arrangement extends the algorithm for basic channels developed in [12] to non-basic channels. In particular, it considers each bottleneck tile to be a channel and if the pins on both sides are floating (i.e., there are no position or order constraints), the given pin arrangement will be optimized.

E. Shape Optimization

Floorplanning procedure described above, chooses the best shape for a flexible cell from a finite set of discrete shapes. The shape optimization procedure, however, assigns a shape to the flexible cell by varying its aspect ratio (while keeping its area fixed) between a minimum and maximum aspect ratio. The pin assignment performed during this procedure is a simple scaling of pins on the sides of cell such that the relative spacings between pins are intact. The global router *and* compactor must have been run on the chip prior to invoking the shape optimization procedure. After resizing a cell, the global routing information around the cell is incrementally updated. The shape constraints specified in .flex file are independent of the shape functions given in .bfp file. Note that the current shape of the cell on the tile plane must be within the given aspect ratio bounds for the cell.

This procedure is not robust, i.e., during dynamic updating of global route information some nets may get disconnected which will cause the procedure to abort itself. In addition, since the topology may drastically change after a cell is resized and repositioned, it is possible that the procedure will run for a long time before it terminates. (See subsection 6.)

To run the shape optimizer, the user should type **so** in the console window. The user can specify a set of parameters to guide the shape optimizer. These parameters are as follows (see following table):

Shape Optimization Parameter Set		
parameter	*range*	*default*
Design Style	BBL, SC, GA	BBL
Algorithm	1-D, 2-D	2-D
Preferred Direction	HORZ, VERT, NONE	NONE
Options	BEST-SLK, HALF-SLK, FULL-SLK	BEST-SLK
Minimum Slack Size	1-128	8

1. Design style.

This parameter specifies the type of modules on the chip. This is required since the shape optimizer must know which resizings are legal. In the current BEAR-FP release, only the *General Cell* or *Building Block* (BBL) design style (which says that the module dimensions may be continuously changed in either direction subject to aspect ratio constraints) is supported, and other styles (*Standard Cell* (SC) and *Gate Array* (GA)) are not.

2. Algorithm.

This parameter specifies whether one-dimensional or two-dimensional shape optimization algorithms should be used. The 2-D algorithm performs simultaneous X- and Y- axis optimization. After the global routing both the block sizes and the estimated routing densities around the blocks are known. The longest or critical paths in either X- or Y- direction, which determine the extent of the layout, can thus be computed. The 2-D algorithm iteratively reduces the layout area by picking up a module with the largest resize capacity lying on a critical path (in either direction) and resizing it so that the module dimension along the critical path is reduced. The process terminates when no improvement in the layout area has been achieved after a certain number of previous iterations. For small macrocell circuits (less than 15-20 blocks), this algorithm is very efficient and gives excellent results. For larger circuits, because of the unpredictability of the changes made to the underlying topology, the algorithm may have a long run-time. Therefore, we allow for a two-pass 1-D shape optimization option (an X-direction pass followed by a Y-direction pass).

3. Preferred direction.

This parameter must be set to NONE when the 2-D algorithm is selected. With the 1-D algorithm, however, it specifies the direction in which the chip dimension will be reduced. The other direction often remains unchanged. For large chips (> 15 modules), it is suggested that the 1-D algorithm be used because it keeps the circuit topology relatively unchanged and therefore is more likely to quickly converge to good solution.

4. Options.

This parameter determines the amount by which a given flexible module is resized on each iteration of the algorithm. The horizontal slack of a module is the amount by which the X dimension of the module can be increased without increasing the chip X dimension. The vertical slack is defined similarly. Consider a block which lies on the longest path through the horizontal *space tile adjacency graph* [13, 14]. The *Best-Slack* resizes this block by an amount such that this block will at worst be placed on the second longest path through the vertical adjacency graph. The *Half-Slack* resizes the same block by half its slack in the vertical direction. The *Full-Slack* resizes this block by all the available slack in the vertical direction. The Full-Slack terminates much faster but often leads to poor optimization results. The program may even be trapped in a cycle whereby a block is resized in opposite directions alternatively until the phenomenon is detected and the program is automatically terminated. The Half-Slack takes more conservative resizing steps and is therefore slower. The Best-Slack has proven to be a good compromise between the speed of the Full-Slack approach and the quality of the Half-Slack approach .

5. Minimum slack size.

This parameter specifies the stopping parameter for the shape optimizer. Whenever the horizontal or vertical slack for every module drops below the value of this parameter, the shape optimizer terminates. Hence, the user may initially set the value of this parameter high (e.g. 32), and do the shape optimization. If the rough shape optimization result is acceptable, the user reruns the shape optimizer on the partially optimized chip, this time with a smaller value of Minimum Slack Size (e.g. 2). It is recommended that the user compact the chip before doing shape optimization so that the circuit topology becomes more representative of the final routed layout. (See section on the global spacer, III C).

When the shape optimizer is run, information about the block being resized and repositioned at each iteration will be printed as standard output. The block being resized will be highlighted. At the end of the shape optimization phase the percentage of layout area reduction is printed to the standard output.

6. Maximum iteration count.

This parameter specifies a different way for terminating the shape optimization procedure. The procedure will terminate when the number of cell resizing — global route updating cycle equals the value of the parameter. This is the preferred termination criterion.

III. Routing

A. Overview of the Routing Process

Routing in a building block environment is a complicated task. Not only is the routing region irregular, but we also want to be able to move blocks during the routing process. The freedom to move blocks is a mixed blessing. It enables us to achieve more compact layouts than in the static case, but it complicates the problem tremendously. BEAR-FP can handle block movement during the routing process and calls for a different routing data representation from that of conventional systems.

In the routing process, we assume that the placement of blocks has been predetermined. This placement is not completely rigid, but serves as a starting point. As we will see later, the initial placement can be deformed during routing, when the amount of distortion and the method of change depend on the user's actions. The starting placement can be arbitrary as long as blocks do not overlap, their sides are parallel to the x and y coordinates, and all cells are contained inside a chip area specified by the user.

After the placement has been read into BEAR-FP, two *tile planes* are built. The horizontal tile plane consists of horizontal *tiles* [15]; the vertical tile plane consists of vertical tiles. Each of the tile planes has two kind of tile: *solid tiles,* which represent blocks; and *space tiles,* which cover the routing area. In the horizontal plane, the routing area is dissected horizontally into maximal horizontal stripes. In the vertical tile plane, the routing area is dissected vertically. The command *show tile property* can be used to view the tile planes. Invoking **stp -h** displays the horizontal tile plane; **stp -v** displays the vertical tile plane.

During the routing process, the *bottleneck* tiles play a key role. Intuitively speaking, these are tiles between the parallel edges of two neighboring modules, in the critical regions where congestion is most likely. Invoking **stp -b** displays the bottleneck tiles. For a more formal classification of tiles, please refer to [13].

The first step in the routing process is the determination of topologies and rough placement of all the nets. The nets' topologies and relative positions with respect to the blocks can be determined automatically by using the *global route* command (invoked by **gr**). The information about net routes is stored in the bottleneck tiles. The topology of a net may be viewed using the *show net property* command (invoked by **snp**).

Since it is very difficult to determine *a priori* how much space is needed to accommodate all the wires, block positions are adjusted after the global routing step. The adjustment is made by using the *compaction* (invoked by **cm**) or *decompaction* (invoked by **dcm**) commands. The user may wish to perform one-dimensional spacing (invoked by **hcm** for horizontal compaction, **vcm** for vertical compaction, **hdcm** for horizontal decompaction and **vdcm** for vertical decompaction) in order to move blocks and the global routes of nets so that the space between cells matches the estimate provided by the global router. Compaction removes extra space, while decompaction

provides more space in congested areas of the chip.

Now we enter the detailed routing phase. The unrouted region is divided into straight channels and/or L-channels. First subregions are ordered, then one of those which can be routed at this time is selected. If we wish to select a *straight* channel to be routed now, the **dch** command is used. If we want to choose an L-channel, the **dlch** command is used. When **dch** or **dlch** commands are invoked, the legal channels are highlighted. When the detailed router completes its task, two previously disjoint chunks of layout are merged into one larger block. The nets exiting from the routed channel are fixed pins of the combined block. Such a combined block is called a *route block*. The results of partial detailed routing are now used to adjust the blocks' placement by performing the decompaction/compaction process again. After the adjustments, the remaining routing region is split and ordered again, the next channel is chosen, and the loop is executed until all the blocks are merged into one.

The last step is to connect this merged block to the pads on the periphery of the chip. This task is performed by the *ring route* command (invoked by **rr**).

There are also many useful commands for showing the nets' topologies, checking connectivity etc., or creating or modifying examples.

B. The Global Router

1. Automatic global routing.

The global router is invoked by typing the **gr** command. The purpose of the global router is to determine the topologies and rough placement of wires on all unconnected nets. For each net, the global router determines through which bottleneck tiles the net will pass. This information is stored by means of *pseudo pins* which are attached to the open sides of bottleneck tiles. Each pseudo pin has an *internal* and an *external id*. Two pseudo pins connected inside a bottleneck tile have matching internal *ids*. Similarly, when the external *ids* of pseudo pins are the same, the pseudo pins are connected between different bottleneck tiles. Pins of cells, I/O pins and pins of route cells also have external and internal *ids*. Their internal *ids* are always 0. If a pin is inside a bottleneck, the external *id* matches the internal *ids* of corresponding pseudo pins; if a pin is outside a bottleneck, the external *id* matches the external *ids* of appropriate pseudo pins.

The global router used in BEAR-FP takes a subregion in which routing has not yet been completed and determines a *cut* which separates it into two smaller subregions. When a net has pins or pseudo pins in both sides of the partition, the net crosses the cut line. For each such net, pseudo pins are inserted along the cut line in appropriate bottleneck tiles. This cutting process continues until each subregion on the list is free of bottlenecks. At each partitioning step, the linear assignment algorithm is used to determine where the net will be placed. The number of nets which can pass through a bottleneck tile is determined by the initial placement (geometrical dimensions of the tiles) and design rules (wire widths and spacings between them). The size of a bottleneck is measured by its *capacity*. The amount of available space inside bottleneck tiles can

be decreased further by manually prerouting nets. The occupied space inside a bottleneck is measured by its *density.* The global router tries to place nets in bottleneck tiles so that the density (used space) does not exceed the capacity (available space) of each tile. If there is enough space for all the nets to cross the current cut line, then the assignment is performed. When there is not enough space, the global router increases the bottlenecks' capacities proportionally to their initial capacities until there is enough space for all the nets to pass through. Thus if the initial placement is too compact, the global router may produce many nets which do not take their shortest paths because it will try to utilize the existing area.

The global route command is controlled by a floating-point type parameter called the *congestion factor.* Its legal range is 0.0 to 1.0; its typical value is 0.0. This parameter controls the tradeoff between modifying placement and taking longer paths for nets. Small values will cause small modifications of the initial placement and, for placements with an underestimated routing area, may result in nets taking detours. Large values (close to 1) will result in nets taking shorter paths and more modified placement. A congestion factor larger than 0 artificially increases the capacities of the bottleneck tiles, but only for the purpose of the global router. The bottleneck tiles which are adjacent to the external bounding box are treated somewhat differently from the other bottleneck tiles: nets are assigned to pass through them only when necessary. This is because it is usually quite difficult to determine the dimensions of the bounding box, so even if it is overestimated, it will not cause nets to take detours through these regions.

The global router can take into account spatial constraints imposed on some nets. To execute this option the user has to set the *timing mode* to **true** in the dialog box, and then enter the name of a file which contains the following information:

number_of_nets [int],
vertical_layer_multiplication_factor [float],
net_name [string], max_net_length [float].

There must be as many *net_name, max_net_length* pairs as the value of *number_of_nets. Vertical_layer_multiplication_factor* is a parameter by which the lengths of vertical wires are multiplied in net length calculations; it is useful when wires on different layers have different conductances. If this distinction is not needed then the value of *vertical_layer_multiplication_factor* should be specified 1.0.

A detailed description of the global router can be found in [16].

Limitations: The global router requires the capacity of every bottleneck tile to be at least wide enough that one wire can pass through it. If a bottleneck tile does not fulfill this requirement a warning message is printed and the command aborts.

2. Show commands which display results of global routing.

a. *Show net properties* (invoked by **snp**.) The user is prompted to choose a net by selecting one of its pins. A net may also be selected by indicating its name on the command line (**snp** *net-Name*). The specified net is then highlighted on the screen. **snp -off** turns off the net highlighting.

b. *Show tile properties.* **stp -h** displays horizontal tiles on the screen, **stp -v** displays vertical tiles, **stp -bp** displays both planes, **stp -b** displays bottleneck tiles, and **stp -co** displays cells only. When the **stp -lp** command is invoked with the mouse on a bottleneck tile in a chip window display, the list of pseudo pins attached to this bottleneck tile is printed.

c. *Show pin properties.* When the **spp** command is invoked, the user is prompted to choose a pin. The coordinate positions, type, external *ids*, etc., of the selected pin are printed.

d. *Show cell properties.* The cell under the mouse is highlighted and information pertaining to it is printed.

C. Global Spacing

After the global router completes its job, the topologies and positions of all nets with respect to blocks are determined. Since it is quite difficult to estimate the routing area precisely, some bottleneck tiles will have more nets passing through them than their sizes permit. Similarly, some tiles will have less. The purpose of global spacing is to match the capacity of each bottleneck tile with its density as much as possible while preserving the existing nets' topologies.

There are two steps in global spacing. The first, global decompaction, is invoked by **dcm.** Its goal is to increase the size of the chip as little as possible so that negative mismatches (i.e. more nets than are allowed pass through a bottleneck) are eliminated. The second step, global compaction, is invoked by **cm.** Its goal is to reduce the size of the chip as much as possible without creating negative mismatches. The global spacer, working in either mode, selects a *ridge* which is a path through space tiles from one side of chip to the other. For horizontal compaction/decompaction the ridge goes from the top to the bottom of the chip; for vertical compaction/decompaction it goes from left to right. Ridges are chosen through bottlenecks with mismatches and then all objects on the top or the right side of the ridge are moved to increase or decrease the size of the ridge. For decompaction, ridges are selected from the smallest to the largest mismatch. For compaction, ridges are selected from the largest to the smallest mismatch. In addition, the ridges are selected alternately in the horizontal and vertical direction to preserve the topology of the placement.

The **cm** command compacts all mismatched ridges. **cm -l** allows the user to compact one ridge at a time.

The *push* command allows the user to manually select a ridge. First the user chooses a set of adjacent tiles from one side of chip to the other. Then he or she is prompted to give the amount that the ridge should be moved. The push command is invoked by **pu** for a horizontal ridge and **pu -v** for a vertical ridge.

Detailed description of the global spacing algorithm can be found in [14]. Each time blocks are moved, some bottleneck tiles may be destroyed and/or new ones created. Since we want to preserve the nets' topologies, after block movement the net connectivities are updated in the background. This process is invisible to the user and is not controlled by any external parameters. Details of the updating algorithm can be found in [13].

D. Iteractive Detailed Routing

1. Detailed routing iterative loop.

Detailed routing repeatedly executes the following steps:

a. The unrouted region is broken into straight and L-channels which are ordered appropriately. Please see [17] for details of the routing regions ordering algorithm. Horizontal channels can change their vertical dimensions, vertical channels can change their horizontal dimensions and L-channels can change both dimensions without affecting previously routed regions.

b. A straight channel or an L-channel is selected from those currently feasible. The channel is defined by invoking the **dch** (straight channel), or the **dlch** (L-channel) command. These commands call detailed routers which perform routing but do not enter results into the data base. The available channel height (current size of channel) and required channel height are displayed on the screen. The user is prompted to either:

   - Route the channel (store the results in the data base). This option is used when the available height is not less than the required height and detailed compaction is not used.
   - Attempt to decrease the channel height using *detailed compaction* by checking the **Nutcracker** option on the screen and specifying the target height for the channel.
   - Abort the command. This option is used when the available channel height is less than the required height.

c. If the previous *channel route* command was aborted due to a mismatch between available and required space, then placement must be adjusted by the *local decompaction/compaction, manual move blocks* (invoked by **pu** or **pu -v**), or *transform cell* commands.

2. Detailed routers.

The channel router *Glitter* does the detailed routing. It is a gridless, variable width router. The **dch** command invokes *Glitter* on a straight channel; the **dlch** command invokes it on two straight subchannels created by dividing the L-channel. Details of the detailed routing algorithms can be found in [18] and [19].

When the **dch** command is invoked on the chosen channel, *Glitter* runs and displays the number of tracks it needs to connect all the wires of the selected channel. In addition to *Glitter's* results, the available height of the channel and the *target* height are displayed. Initially the target height is set equal to the available height. The user can compact routing produced by *Glitter* by decreasing the target height, setting the *Nutcracker* option to **true**, and checking **route**. This will invoke the channel compactor *Nutcracker*. *Nutcracker* will attempt to compact the initial detailed routing by inserting jogs. It will compact the channel as much as possible, but no more than the specified target height. After completing its job, *Nutcracker* displays its results and the user is prompted to either:

- Continue, if required and target heights match.
- Abort, if results are not satisfactory.

Glitter places horizontal wires on one layer and vertical wires on the other layer. This strategy may lead to many more vias than necessary. By default the detailed routing is followed by a *via reducer* which slides and removes unnecessary contacts. To turn this feature off, change the *via reduce* option from *true* to **false** in the box displayed by the **dch** command.

Detailed discussion of the algorithms used by the *channel compactor* and the *via reducer* can be found in [20] and [21], respectively.

Glitter routes L-channels and displays the results. At this point, results are not yet entered into the data base and the user is prompted to either:

- *Route channel* (store in the data base) if horizontal and vertical adjustments displayed are 0.
- *Abort* if router asks for horizontal or vertical adjustments.

By default, unnecessary vias produced by the detailed router are removed by the *via reducer.* To turn this feature off, change the *via reduce* option from *true* to **false. Nutcracker** cannot be invoked on an L-channel.

3. Placement adjustments.

This step is used to adjust a channel region to match the requirement specified by the detailed router. As we have seen in the section on detailed routers, if the detailed routing does not match the available area, the detailed routing command is aborted and the results are not entered into the data base. *Placement adjustment* is performed to correct this situation. There are two cases: either the available channel height was larger than required or it was smaller. In the first case, *local compaction* and in the second case, *local decompaction* may be used to modify the placement.

The local compaction step is used to compact the channel region to the number of tracks required by the previous detailed route. The orientation of the channel determines which compaction command is needed. A horizontal channel requires vertical compaction (invoked by **vcm**); a vertical channel requires horizontal compaction (invoked by **hcm**). The compaction command finds ridges across the chip and moves all blocks to the right or top of the ridge. If executed immediately after **dch,** the compactor will find the ridge which passes through the recently aborted channel. The amount the blocks are moved depends on the mismatch between the number of tracks needed by the router and the actual height of the channel. The compactor will not select ridges unless it results in a smaller chip area.

Local decompaction is used to add the number of tracks required to the channel that was previously detail routed and aborted. The orientation of the channel determines which decompaction command is needed. A horizontal channel requires vertical decompaction (the **vdcm** command); a vertical channel requires horizontal decompaction (the **hdcm** command). Local compaction and decompaction are based on similar principles.

The user may manually select a ridge for compaction or decompaction by invoking **pu** (horizontal ridge) or **pu -v** (vertical ridge).

Another method of adjusting placement is to manually move the blocks. Moving blocks maintains the global routing information and previously routed channels. To move a block manually, the *transform cell* command is used. First **tc** is invoked. The user is then prompted to choose a transformation: for manual cell moving, check the *move* option. Next, the user is prompted to choose the type of move: *delta x-y* or *interactive*. **dlch** gives x,y measurements for the delta x-y move. Finally the user is prompted to specify the cell on the screen.

E. Ring Route

The last step in detailed routing is ring route (invoked by **rr**), which connects the core of a chip to the I/O pads at the periphery. The ring router expects all signal wires to be the same width. Wires specified as power/ground can be of arbitrary widths.

Limitations: Ring route cannot handle power pads at the corners of the bounding box or vertical constraints between power nets.

F. Wire Widths Sizing

We are developing code capable of determining the widths of power and ground nets so that the area of wire segments is minimized while fulfilling electromigration and voltage drop constraints. Details of the wire sizing algorithm can be found in [22]. A wire technology file will be required in which the parameters needed will be listed. These parameters are as follows:

grid [float]: specifies how many microns correspond to one grid line;

conductance [float] (in A/V): specifies conductance of wires;

curPerMicron [float] (in A/micron): electromigration constant (i.e. the max branch current ≤ curPerMicron * width);

minWidth [int] (in grid lines): specifies the minimum acceptable wire width;

timeSteps [int]: specifies in how many time steps the calculations are to be performed (usually 1);

flag [int]: used to set appropriate parameters for **wn**.

There are three parameters which need to be set in *wire net: elect_flag, volt_flag* and *feasible_flag*. When *elect_flag* is 1, electromigration constraints are included in calculations; if it is 0 then they are not. When *volt_flag* is set to 1, voltage constraints are taken into account; if it is set to 0 then they are not. When *feasible_flag* is set to 1, only a feasible solution is sought; if it is 0

then an optimal solution is calculated.

These parameters are calculated as results of the following bitwise operations:

elect_flag = PGR_ELECT_FLAG & *flag*
volt_flag = PGR_VOLT_FLAG & *flag*
feasible_flag = PGR_FEASIBLE_FLAG & *flag*

where PGR_ELECT_FLAG = 01, PGR_VOLT_FLAG = 02, PGR_FEASIBLE_FLAG = 04. Thus for example, *flag* set to 05 causes elect_flag = 1, volt_flag = 0 and feasible_flag = 1.

Appendix 1. Input/Output Format Specifications

A. BBL Format for a Routing File

Input data in BBL format are specified in two files. The first file contains block dimensions and pin-net information. It is called a *routing file* and its name must be *something.r*. The second file contains descriptions of design rules and is called a *technology file*. Its name must be *something.tech*. The first portion (up to the dot) of both files must be the same.

We encourage users to check the validity of their BBL files by using the **-c** option when opening a window with the **ow** command.

1. Rules for describing a module and terminals.

a. Only rectilinear modules are allowed.

b. If a placement is given then the spacing between any two adjacent modules must be wide enough to allow one signal wire to pass between them.

c. The coordinates of the modules corners must each be on a separate line of the input file in counter-clockwise order.

d. Every terminal must lie on a boundary of a module.

e. No terminal may lie on a corner of a module.

f. No two terminals may have the same coordinates.

2. The overall format for a BBL file.

Note that the smallest routing file allowed is a root module with no terminals.

```
SN <number of nets> <nl>
< root module >
[ < one or more child modules > ]
$
```

3. The format for a single module of the BBL file.

```
MOD <nl>
<ox> <oy> [ <mt> ] <nl>
<module name> <nl>
<module type> <nl>
<cx1> <cy1> <nl>
<cx2> <cy2> <nl>
<cx3> <cy3> <nl>
<cx4> <cy4> <nl>
$ <nl>
T <nl>
[ <tx> <ty> <net name> <routing direction> <terminal type> <nl> ]
   .       .
   .       .
$ <nl>
```

4. Detailed description of a BBL file's syntax.

Please note that information enclosed in braces [ ] above is optional. Also all numbers are expected to be integers, and finally no blank lines are permitted.

MOD	This is a marker that indicates the beginning of a module.
SN	This marker must be the first thing in the file.
T	This is a marker that indicates the beginning of a terminal list. It must always be present, i.e. even if there aren't any terminals.
$	This is a marker that indicates the end of a particular group of data. Two of them in a row indicates the end of the file.
<nl>	The new line character is used to terminate each line. Simply type a carriage return when inputing the file.
<ox> <oy>	This is the module's origin, which all of the coordinates in the module are relative to. The origin of a child module is relative to the root module.
<cxi> <cyi>	A pair of coordinates indicating a corner of the module.
<tx> <ty>	A pair of coordinates relative to the module's origin indicating the position of the terminal.
<net name>	The name of the net that the terminal belongs to. It may be up to 40 characters.
<number of nets>	The total number of nets.

<mt>	Optional module transformation which is applied to all of the modules coordinates. 0, 3, 4, and 5 are legal values as follows: 0 for Identity Transform; 3 for Transform module by 90 degrees; 4 for Transform module by 180 degrees; 5 for Transform module by 270 degrees.
<module type>	The type of the module. 1 and 0 are legal values as follows: 1 for the root module; 0 for the other modules.
<terminal type>	The type of terminal. At the moment 2 is the only legal value, and it indicates that the terminal is fixed in place.
<routing direction>	The routing direction of the terminal. 0, 1, 2, and 3 are legal values as follows: 0 if Pin is on WEST side of module; 1 if Pin is on SOUTH side of module; 2 if Pin is on EAST side of module; 3 if Pin is on NORTH side of module.

5. Sample BBL file.

A copy of this file may be found in */users/bear/BearFP_1.0_Tape/src/polarBear/bearData/bblSample.r.*

```
SN 2
MOD
0 0
bound
1
1000 1000
0 1000
0 0
1000 0
$
T
133 1000 net1 1 2
$
MOD
133 542
Mickey
0
411 306
0 306
0 0
411 0
$
T
411 190 net2 2 2
0 128 net2 0 2
```

```
411 54 net1 2 2
$
MOD
703 145
Donald
0
146 510
0 510
0 0
146 0
$
T
76 0 net1 1 2
0 91 net2 0 2
$
MOD
183 202
Goofy
0
240 247
0 247
0 0
240 0
$
T
0 143 net1 0 2
202 0 net2 1 2
$
$
```

B. BFP Format for a Routing File

Input data in the BFP (bear floor planning) format is specified in a number of text files. There is a file with information about the cells and nets, a file with information about the design rules, and additionally a file with a list of critical nets.

Again we encourage users to check the validity of there BFP files by using the -C option when opening a window with the **ow** command.

1. Rules for describing cells and pins.

a. Only four sided rectangular modules are allowed.

b. If a placement is given then the spacing between any two adjacent modules must be wide enough to allow one signal wire to pass between them.

c. Every pin must lie on a boundary of a cell.

d. No pin may lie on a corner of a cell.

e. No two pins may have the same coordinates.

2. The overall format for a BFP file.

```
# comments <nl>

Version 1.0 <nl>

TechFileName <technology file name> <nl>

BorderCellName <border cell name> <nl>
BorderCellOrigin <x> <y> <nl>
[ BorderPin <border pin name> <net name> <side> <fraction> <nl> ]
BorderCellShape <width> <height> <nl>

[ <one or more child cells> ]
```

3. The format for a single child cell.

```
CellName <cell name> <nl>
CellOrigin <x> <y> <nl>
[ Pin <pin name> <net name> <pin type> <x> <y> <nl> ]
```

```
InitCellShape <width> <height> <rotation flag> <nl>
[ CellShape <width> <height> <rotation flag> <nl> ]
```

4. Detailed description of a routing file's syntax.

Please note that information enclosed in braces [ ] above is optional. As far as coordinate systems go, the border cell is considered to be absolute; all cells are relative to it. Pins are relative to the lower left hand corner of the cell they belong to. (See Fig. 13.)

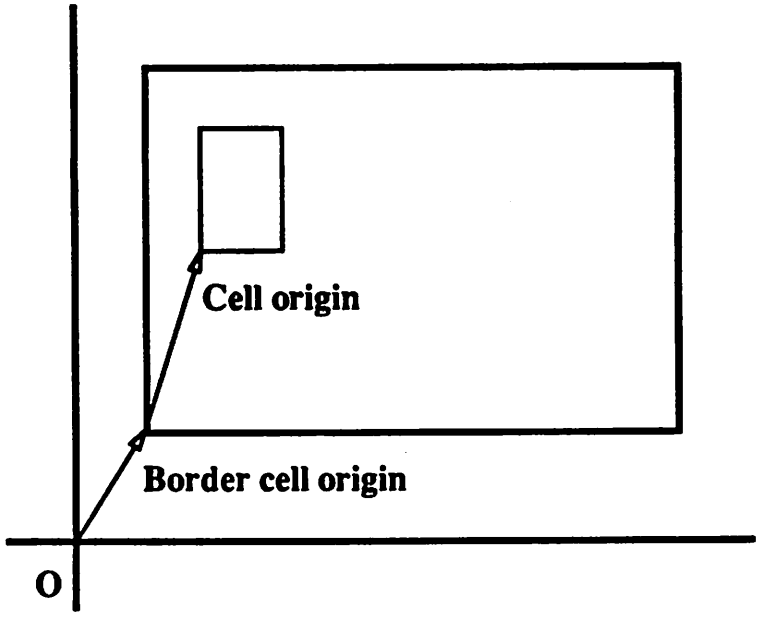

Fig. 13.

`<fraction>` This field must be a real number between 0.0 and 1.0. It is used to determine the position of the border pin.

`<height>` This field must be an integer greater than 0.

`<pin type>` This field must be either "fixed" or "floating". Note that all of the pins in a given cell must have the same value.

`<rotation flag>` This field must be either TRUE or FALSE. A value of FALSE indicates that one coordinate pair (width, height) will be added to the shape function, where as a value of TRUE indicates that two coordinate pairs will be added (width, height) and (height, width). (See Fig. 14.)

`<side>` This field must be either N, E, S, or W and is used to indicate which side of the border the given pin is on.

`<width>` This field must be an integer greater than 0.

<x> <y> A pair of integers.

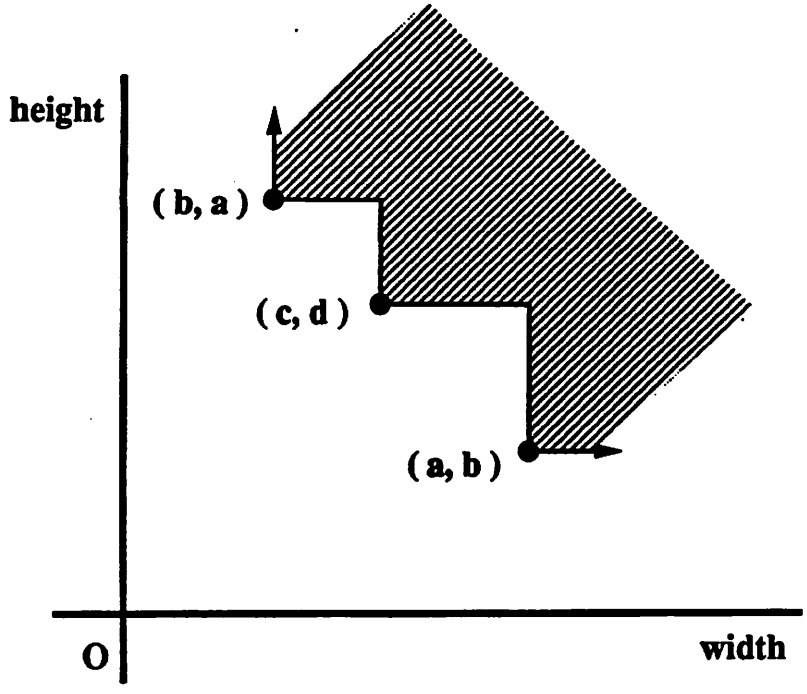

Fig. 14.

5. Sample BFP file.

A copy of this file may be found in */users/bear/BearFP_1.0_Tape/src/polarBear/bearData/bfpSample.*

```
#
# Sample bearFP data file.
#

Version 1.0

TechFileName    bearData/bfpSample.tech

#
# Pins on the border cell are always fixed.
#
BorderCellName Apple
BorderCellOrigin 0 0
BorderPin pin1 net1 N 0.3
BorderPin pin2 net2 N 0.5
```

```
BorderPin pin3 net1 E 0.6
BorderCellShape 650 650

#
# This cell has a fixed shape and no pins
#
CellName Bannana
CellOrigin 50 500
InitCellShape 150 100 TRUE

#
# This cell has a fixed shape and all fixed pins
#
CellName Kiwi
CellOrigin 50 350
Pin pin4  net1 fixed       75   100
Pin pin5  net2 fixed      150    50
Pin pin6  net3 fixed        0    50
InitCellShape 150 100 FALSE

#
# This cell has a fixed shape and all floating pins
#
CellName Mango
CellOrigin 50 200
Pin pin7  net1 floating    75   100
Pin pin8  net2 floating undef    50
Pin pin9  net3 floating undef undef
Pin pin10 net4 floating undef undef
InitCellShape 150 100 TRUE

#
# This cell has a flexible shape and all floating pins
#
CellName Peach
CellOrigin 50 50
Pin pin11 net1 floating undef undef
Pin pin12 net2 floating undef undef
Pin pin13 net3 floating undef undef
Pin pin14 net4 floating undef undef
InitCellShape 150 100 TRUE
CellShape     100  75 TRUE
CellShape     100  50 TRUE
CellShape     100  25 FALSE
```

C. Input Format for a Technology File

Below is a complete description of the format for a technology file.

1. The format for a technology file.

M1_MinWireWidth <number> <nl>

M2_MinWireWidth <number> <nl>

M1_MinWireSpacing <number> <nl>

M2_MinWireSpacing <number> <nl>

ViaSize <number> <nl>

MinOverLap <number> <nl>

2. Detailed description of a technology file's syntax.

Please note that all numbers are expected to be integers.

M1_MinWireWidth This keyword is followed by an even number bigger than or equal to two that indicates the minimum width of a wire on metal one's layer. (See Fig. 15.)

M2_MinWireWidth This keyword is followed by an even number bigger than or equal to two that indicates the minimum width of a wire on metal two's layer. (See Fig. 15.)

M1_MinWireSpacing This keyword is followed by an even number that indicates the minimum spacing between two wires on metal one's layer. (See Fig. 17.)

M2_MinWireSpacing This keyword is followed by an even number that indicates the minimum spacing between two wires on metal two's layer. (See Fig. 17.)

ViaSize This keyword is followed by a number bigger than or equal to two that indicates the size of a via. (See Fig. 16.)

MinOverLap This keyword is followed by a number that indicates the size of the overlap. (See Fig. 18.)

<number> An integer.

<nl> The new line character is used to terminate each line. Simply type a carriage return when inputting the file.

3. Sample technology file.

A copy of this file may be found in .
/users/bear/BearFP_1.0_Tape/src/polarBear/bearData/bblSample.tech.

```
M1_MinWireWidth 20

M2_MinWireWidth 20

M1_MinWireSpacing 10

M2_MinWireSpacing 10

ViaSize 25

MinOverLap 5
```

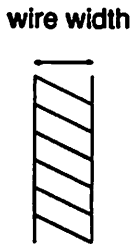

Fig. 15.

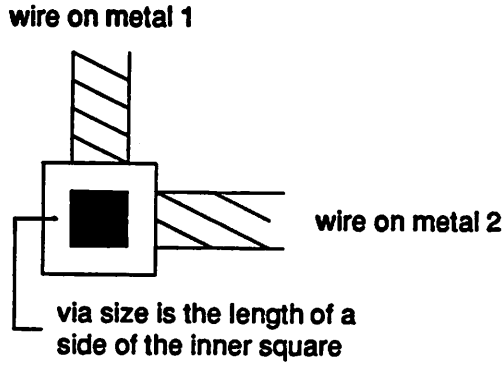

Fig. 16.

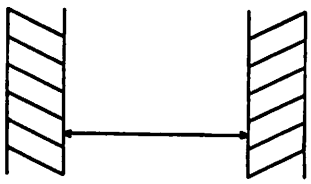

Fig. 17.

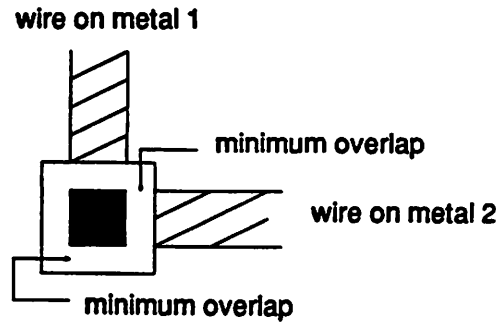

Fig. 18.

D. Format For a Timing File

1. The overall format for a timing file.

```
# comments <nl>

Version 1.0 <nl>

CriticalNetList <nl>
<net name> <type> <length> <nl>
   .       .
   .       .
; <nl>
```

2. Detailed description of a timing file's syntax.

<net name>	This field indicates the name of the critical net and must match one of the nets in the BFP file.
<type>	This field indicates the type of constraint the critical net is to have. At the moment the only acceptable value is MAX, which means that the indicated length should not be exceeded during the floorplanning/placement portion of the program.
<length>	This field contains an integer value.

3. Sample timing file.

A copy of this file which was generated with the timing option of the write command may be found in */users/bear/BearFP_1.0_Tape/src/polarBear/bearData/bfpSample.timing.*

The timing option is intended as a means of generating test data for the timing code of the new cluster package. It looks at ten percent (or at least three) of all the nets with four or fewer pins. For each one it calculates the steiner tree and then takes eighty percent of the length of this tree as the proposed maximum length of the net.

```
Version 1.0

#
# Critical nets for bearData/bfpSample
#

CriticalNetList
net2 MAX 627
net3 MAX 848
net4 MAX 216
;
```

E. Format for a Flex File

Input data for the **so** (shape optimizer) command is specified with a text file as described below.

1. The overall format for a flex file.

```
# comments <nl>

Version 1.0 <nl>

FlexibleCellList <nl>
<cell name> <minimum aspect ratio> <maximum aspect ratio> <nl>
   .        .
   .        .
; <nl>
```

2. Detailed description of a flex file's syntax.

<cell name>	This field indicates the name of a cell and must match one of the cells in the BFP file.
<minimum aspect ratio>	A real number greater than 0 cooresponding to the minimum aspect ratio (which is height / width) that the given cell may have.
<maximum aspect ratio>	A real number greater than 0 cooresponding to the maximum aspect ratio (which is height / width) that the given cell may have.

Note that the current shape of the cell (when the command is used) must satisfy the minimum and maximum aspect ratio constraints.

3. Sample flex file.

A copy of this file may be found in */users/bear/BearFP_1.0_Tape/src/polarBear/bearData/iccad1.flex.*

```
Version 1.0

#
# Input data for the so (shape optimizer) command that goes
# with the cell data in bearData/iccad1.bfp
```

```
#

FlexibleCellList
B1 0.5 2.0
B2 0.25 2.0
B3 0.33 3.0
;
```

F. Format for a Cluster Tree File

The text file described below is used by the **ncl** (new cluster) command to input and output a cluster tree. Such a tree may have upto four branches per node, and leaf nodes are cells that must all have the same level within the tree.

1. The overall format for a cluster tree file.

```
# comments <nl>

Version 1.0 <nl>


# One root cluster. <nl>

RootCluster <cluster name> <nl>
<cluster or cell name> <nl>
   .       .
   .       .
; <nl>


# One or more clusters. <nl>

Cluster <cluster name> <nl>
<cluster or cell name> <nl>
   .       .
   .       .
; <nl>
```

2. Detailed description of a cluster tree file's syntax.

`<cell name>`	This is a string indicating the name of a cell and must match one of the cells in the BFP file.
`<cluster name>`	This is a string indicating the name of a cluster.

3. Sample cluster tree file.

A copy of this file may be found in
/users/bear/BearFP_1.0_Tape/src/polarBear/bearData/iccad1.ctree

```
Version 1.0

#
# A posible cluster tree based on the data in bearData/iccad1.bfp
# that may be used as input to the ncl (new cluster) command.
#

RootCluster R
E
F
G
;

Cluster E
A
B
;

Cluster F
C
;

Cluster G
D
;

Cluster A
B1
;

Cluster B
B2
B3
;

Cluster C
B4
B5
;

Cluster D
```

```
B6
B7
B8
;
```

G. Sample Control File

A copy of this file may be found in
/users/bear/BearFP_1.0_Tape/src/polarBear/bearData/controlData.

```
#
# This file contains default values for parameters used by the
# new cluster and new placement modules.
#

newC.QuickRouteEstFlag          FALSE
newC.ChannelFreeFlag            FALSE

#
# If newC.TimingFlag is FALSE that implies the following:
#
#     newP.FullEnum.CritWeight  undef
#     newP.PartEnum.CritWeight  undef
#
newC.TimingFlag                 FALSE

#
# newP.TmpDir location of temporary files (can have '~')
#
#newP.TmpDir                    ~lolita/tmp

newP.BorderSize                 1.5

#
# If newP.QuickPlanFlag is TRUE that implies the following:
#
#     newP.MaxLoopCnt           0
#     newP.FinalOptFlag         FALSE
#     newP.FullEnumFlag         FALSE
#
newP.QuickPlanFlag              FALSE

newP.MaxLoopCnt                 1
newP.FinalOptFlag               TRUE
newP.FullEnumFlag               TRUE

newP.FullEnum.Tolerance         0.15
newP.FullEnum.AreaWeight        0.6
newP.FullEnum.WireWeight        0.4
newP.FullEnum.CritWeight        1.0
```

```
newP.PartialEnum.Tolerance      0.15
newP.PartialEnum.WireWeight     1.0
newP.PartialEnum.CritWeight     1.0
```

Appendix 2. OCT Interface

The following is a description of how to read from and write to OCT in BEAR-FP.

First, BEAR-FP must be executed before the read from or write to OCT commands can be performed. To load a chip from the OCT database once BEAR-FP is running, use the open window command:

ow -oct cellname viewname [output_cellname [output_viewname]]

Where **cellname** is the name of the OCT cell to be read, **viewname** is the name of the OCT cell's view, and **output_cellname** is the name of the OCT cell used temporarily in the reading process. Also, **output_cellname** is the name of the OCT cell to be written back out when the *write* command without any optional information, **w -oct**, is used later. (The default for the optional **output_cellname** is *macout*.) **output_viewname** is the view name of the OCT cell which is temporarily used and is to be written back out. (The default for the optional **output_viewname** is the same name as the **viewname**.) This temporarily used OCT cell is an exact copy of the OCT cell described by **cellname** and **viewname**, but with two additional bags to facilitate reading and writing back.

When writing to OCT in BEAR-FP, issue the following write command,

w -oct [[-r] output_cellname output_viewname]

where the **-r** option is for saving routing information to OCT and **output_cellname** is the name of the OCT cell to be written out to OCT. This name is only optional when saving an OCT cell that has been read in by the *open window* command described above, has never been saved after being read in, and has not been changed other than the placement of its cells. The default for the optional **output_cellname** is the same name as the **output_cellname** used in the *open window* command (above) when reading in from OCT. **output_viewname** is the name of the OCT cell's view to be written out and is also only optional under the same conditions as the ones described just above in **output_cellname** of this command. The default for the optional **output_viewname** is the same name as the **output_viewname** used in the *open window* command (above) when reading in from OCT.

The *write* (back) command, **w -oct** , is a much faster write mechanism than the full *write* command with options. It uses knowledge from the *open window* (read) command and only updates the placement of the cells.

Example 1: *Modify placement of an OCT cell (chip) and save using defaults.*

(Open window commands are typed with the cursor in the BEAR-FP console window and all other commands, *write* and *close window*, are typed with the cursor in the window displaying the cell read in using *open window*.)

ow -oct foocell fooview

(Opens a window displaying the information read in from OCT cell, foocell, and view, fooview.

[placement modifications]

.

.

w -oct

(Saves the placement information in the window by writing the modifications to OCT cell default, macout, and view default, fooview.)

cw

(Closes the window.)

Example 2: *Modify placement of default OCT cell just written, save the resulting placement as the new OCT cell's name while reading, and save placement and routing modifications in a newly-created OCT cell.*

ow -oct macout fooview foocell1 fooview1

(Opens a new window containing the previously saved OCT cell, *macout*, and view, *fooview.*)

[more placement modifications]

.

.

w -oct

(Saves the placement information in the window by writing the modifications to OCT cell, *foocell1*, and view, *fooview1.*)

.

.

[more placement modifications, routing, and/or creating, deleting, modifying new cells, pins, and nets]

.

.

w -oct -r foocell2 fooview2

(Saves the placement and routing information in the window by creating and writing to OCT cell, *foocell2*, and view, *fooview2*.)

cw

(Closes the window.)

Example 3: *Read in a BBL file and save in the OCT database afterwards.*

ow -bbl foobbl.r

(Opens a new window displaying the information read in from the BBL file, *foobbl.r*)

.

.

[placement modifications and/or creating, deleting, modifying new cells, pins, and nets]

.

.

w -oct foocell3 fooview3

(Saves the placement information in the window by creating and writing to OCT cell, *foocell3*, and view, *fooview3*.)

cw

(Closes the window.)

Unless otherwise specified during the *open window* command with the OCT option, the default output OCT cell name is *macout* and the view name is the same as the view name read in. Any type of modifications can be done on the data that has been just read into BEAR-FP from OCT with the *open window* command. After all the necessary changes have been made on the data, the *write* command, **w -oct**, should be issued to write back only the placement data to the default *OCT cell* and *view*. This *write* command can be used to write back placement data only

once after every read from OCT. Thus, after a write back, the window must be closed and a new window must be opened to read in the changed data for any new modifications made thereafter to be saved properly. Alternatively, the full *write* command with all the options can be issued. This action will write the placement and routing information to the newly created OCT cell and view named in the options. If only placement data is to be written out to a new *OCT cell* and *view* then the full *write* command with all the options minus the -r option should be issued. Other types of data such as *BBL read into BEAR-FP by the open window* command or new information created inside BEAR-FP can be written to OCT by using the full *write* command with the options.

Appendix 3. X Defaults

bearFP.BlackAndWhite	If **on**, a black and white color scheme will be used even on a color display so that programs that dump windows to printers will work.
bearFP.ReverseVideo	If **on**, reverse the definition of foreground and background colors on black and white displays.
bearFP.Background	Determines the background color for all windows other than the console window.
bearFP.Border	Determines the border color for all windows other than the console window.
bearFP.BorderWidth	Determines the border width for all windows other than the console window.
bearFP.Foreground	Determines the foreground color for all windows other than the console window.
bearFP.Font	Determines the font for text in all windows other than the console window.
bearFP.Highlight	Determines the highlight color for all windows other than the console window.
bearFP.Mouse	Determines the mouse cursor color for all windows.
bearFP.Text	Determines the color of prose printed in a window.
bearFP.Chip.Cell	Determines the color of cells on the chip.
bearFP.Chip.RouteCell	Determines the color of route cells on the chip.
bearFP.Chip.DummyCell	Determines the color of dummy cells on the chip.
bearFP.Chip.CellBorder	Determines the border color of all cells on the chip, the pin color, as well as the color of the floor plan graph.
bearFP.Chip.Pin	Determines the color of the pins for the chip.
bearFP.Chip.Background	Determines the background color of the chip.
bearFP.Chip.hchannel	Determines the horizontal channel color.
bearFP.Chip.vchannel	Determines the vertical channel color.
bearFP.Chip.lchannel	Determines the L-shaped channel color.
bearFP.Chip.Net1	Determines the color of net one on the chip.
bearFP.Chip.HorzBottleNeckTile	Determines the color of horizontal bottleneck tiles.
bearFP.Chip.VertBottleNeckTile	Determines the color of vertical bottleneck tiles.
bearFP.Chip.HorzDominantTile	Determines the color of horizontal dominant tiles.

bearFP.Chip.VertDominantTile	Determines the color of vertical dominant tiles.
bearFP.TF.leaf1	Determines the color of leaf number one in the tree and floorplan windows.
bearFP.TF.leaf2	Determines the color of leaf number two in the tree and floorplan windows.
bearFP.TF.leaf3	Determines the color of leaf number three in the tree and floorplan windows.
bearFP.TF.leaf4	Determines the color of leaf number four in the tree and floorplan windows.
bearFP.TF.leaf5	Determines the color of leaf number five in the tree and floorplan windows.
bearFP.TF.node	Determines the color of the nodes in the tree windows.
bearFP.TF.edge	Determines the color of the edges in the tree windows.
bearFP.Clf.BND0	Determines the color of clf boundary zero.
bearFP.Clf.BND1	Determines the color of clf boundary one.
bearFP.Clf.BND2	Determines the color of clf boundary two.
bearFP.Clf.NC	Determines the color of clf nMos contact cut color.
bearFP.Clf.NM	Determines the color of clf nMos metal color.
bearFP.Clf.NP	Determines the color of clf nMos polysilicon color.
bearFP.Clf.TRM	Determines the color of clf text.

The following defaults are for the IV windows:

bearFP.Iv.Background	Set the background color. Default is light grey on color displays, black on monochrome.
bearFP.Iv.BorderColor	Set the border color. Default is black on color displays, white on monochrome.
bearFP.Iv.BorderWidth	Set the border width of the main IV window, and the border around the edit region windows. Default is 1.
bearFP.Iv.ButtonColor	Set the color of the buttons. Default is yellow on color displays, white on monochrome. For best results, choose a non-dark color.
bearFP.Iv.CursorColor	Set the color of the mouse cursor. Default is green on color displays, white for monochrome.
bearFP.Iv.EditBackground	Set the background color of the edit region. Default is light blue on color displays, black for monochrome.

bearFP.Iv.EditFont	Specify the font to print the edit region. Default is 6x10.
bearFP.Iv.EditFontColor	Set the font color of the edit region. Default is red for color displays, white for monochrome.
bearFP.Iv.EraseValue	If **on** clear the edit region upon editing the variable. The default is **off**. Note that data can still be recovered by CONTROL_U.
bearFP.Iv.Padding	Specifies the extra padding above and below each IV row (text and variable). The default is 2.
bearFP.Iv.TextFont	Specify the font to print the documentation field. Default is 6x10.
bearFP.Iv.TextFontColor	Set the font color of the documentation field. Default is blue for color displays, white for monochrome.
bearFP.Iv.TitleFont	Specify the font to print the title. Default is 9x15.
bearFP.Iv.TitleFontColor	Set the font color of the title Default is dark slate blue for color displays, white for monochrome.

Appendix 4. Partial Summary of Commands

1. General Commands

The following commands may be used in any window:

cw The *close window* command is used to close windows. Note that closing a window does not save changes that may have been made. You must use the *write* command if you want the changes to be saved.

help The *help* command lists all of the commands in alphabetical order along with some information about each one. In this way you may determine which commands can be included in a script, as well as which windows a command may be used in.

log [logFile]

The *log* command causes everything that is printed in the console window to be logged in a file, until you enter another log command.

r [-options ...]

The *read* command may be given the following options:

-bfp filename	cell data in bearFP format
-bbl filename	cell data in BBL format
-oct cellname viewname [[output_cellname] [output_viewname]]	
-tech filename	technology data (bbl and oct only)
-np	cells aren't placed
-C	attempt to verify the validity of the data

ow [-options ...]

The *open window* command may be given the following options:

-s	duplicate window with same view
-bfp filename	cell data in bearFP format
-bbl filename	cell data in BBL format
-oct cellname viewname [[output_cellname] [output_viewname]]	
-tech filename	technology data (bbl and oct only)
-cif filename	cell data in CIF format
-np	cells aren't placed
-C	attempt to verify the validity of the data

v The *version* command prints the number of the current version of *polarBear* in the console window.

2. Viewing Commands

The following commands may be used in cif and chip windows:

c The *center* command finds the boundary of the object being viewed (a cif file for instance) and centers it within the window.

mag The *magnify* command allows you to enlarge a portion of a window to get a better view. You are asked to pick a rectangle from within the window the mouse is in that defines the area you want enlarged.

pan The *pan* command centers the window around the mouses current position.

z [amount]
The *zoom out* command enlarges the current view.

Z [amount]
The *zoom in* command shrinks the current view.

3. Floor Planning Commands

ncl [-alg < m || g || a || f filename >] [-sf fileName] [-tf fileName] [-cf fileName] [-NDW]

The *new clustering* command may be given the following options:

-alg algorithms:
- matching
- greedy
- annealing
- file-in

-sf filename	storage data file
-tf filename	timing data file
-cf filename	control data file
-NDW	no dialog windows

npl [-m < x || y || r > <float>] [-cf fileName] [-NDW]

The *new placement* command may be given the following options:

-m	indicate mode and value (The allowed modes are x, y, and r followed by a fixed x dimension, a fixed y dimension or an X to Y aspect ratio.)
-cf filename	control data file
-NDW	no dialog windows

cpa [-options...]

The *channel pin arrangement* command may be given the following option:

-NDW no dialog windows

4. Routing Commands

gr [-C] [-CW] [-NDW]

The *global-route* command may be given the following options:

-C check the connectivity of all the nets after the global routing has been completed

-CW check the connectivity of all the nets after the global routing has been completed and print warning messages

-NDW no dialog windows

rr [-via] [-NDW]

The *ring-route* command may be given the following options:

-via reduce via's

-NDW no dialog windows

5. Other Commands

bg [-C] [-CW] [-n [netName]]

The *build graph* command may be given the following options:

-C check the connectivity of all the nets

-CW check the connectivity of all the nets and print warning messages

-n [netName] check the connectivity of only one net
(If the name is not specified and no net is currently being displayed then the user will be asked to indicate a net with the mouse.)

Please note the graph of a net that goes through an L-channel will not be connected properly.

rs scriptFile [-log] [-i]

The *run script* command may be given the following options:

-log the script file is a previously generated log file

-i use interactive mode

This command enables the user to have a sequence of commands automatically executed. Two types of scripts are reconized. The default is a user generated file in which comments begin with a '#' symbol and end with a new line. Alternatively a previously generated log file may be used, by specifying the **-log** option. Note that in either case only commands that return status may be used. To determine whether or not a command returns status you may type its name followed by the -help option. Below is a sample script in the default format.

```
#
# Sample script
#
# To use this script to give a demonstration start polarBear,
# open a chip window from the console window, and then run this
# script with the interactive option.
#

 log bear.log

#
# This is the sequence of commands to run the new placement package.
#

ncl -cf bearData/controlData
npl -cf bearData/controlData

# ----- Below this line are a few things you can do after placement. -----

w -rst2 bear.rst2

cm -C

log
```

scp [cellName] [-off]

The *show cell properties* command may be given the following option:

-off turn off cell high-lighting
-names toggle cell names (only)

When the off option is not indicated information about the cell is listed and the cell is high-lighted for your perusal. When the name of a cell is not given the cell directly under the mouse is chosen.

sjp [-fp] [-mr]

The *show junction properties* command may be given the following options:

-fp toggle floor plan graph (only)

-mr main room (only)

snp [netName] [-off]

The *show net properties* command may be given the following option:

-off erase all of the nets that are currently being displayed

The command lists some information about the net and then high lights the net for your perusal. There are two ways to indicate the net you want. When the name of a net is not given you will be asked to select a net by picking one of its pins with the mouse. Alternatively you may specify the name of the net yourself.

spp [-p] [-off] [-mps]

The *show pin properties* command may be given the following options:

-p show pins (on/off)

-off turn off pin high-lighting

-mps print the minimum pin spacing

The command allows you to select a pin with the mouse. It then lists some information about the pin and high lights the pin for your perusal.

stp [-co] [[-bp -h -v] [-b] [-d]] [-lp] [-bnl] [-ap] [-off]

The *show tile properties* command may be given the following options:

-co show cells only

-bp show both tile planes

-h show horizontal tile plane only

-v show vertical tile plane only

-b show bottleneck lines and tiles (on/off)

-d show dominant tiles (on/off)

-lp list the pseudo pins of the tile under the mouse

-bnl pick a bottleneck line and then list its pseudo pins

-ap arrange pins of the bottleneck tile the mouse is in

-off turn off bottleneck line high-lighting

When no options are given various properties of the tile directly under the mouse are limned within the console window.

tc [[-dx #] [-dy #]] [-lm] [-mx] [-my] [-rt <X90>] [-rf <45,135>] [[-rsx #] [-rsy #] [<UL,LL,UR,LL>]] [-lr]

The *transform cell* command may be given the following options:

-dx # move cell in x direction

-dy #	move cell in y direction
-im	interactive move
-mx	mirror cell across x-axis
-my	mirror cell across y-axis
-rt #	rotate cell by multiples of 90 degrees
-rf #	reflect cell across the line y = x or the line y = -x
-rsx # #	resize cell in x direction
-rsy # #	resize cell in y direction
-ir	interactive resize

w [-options ...]

The *write* command may be given the following options:

-bfp filename	use BearFP format
-bbl filename	use BBL format
-oct [[-r] output_cellname output_viewname]	
-oct	writes back to OCT only if the data was previously entered as follows: "ow -oct cellname viewname"
-oct output_cellname output_viewname	
-r	write the routing data back to OCT.
-err	any errors that occur while saving the chip will be listed here. (bfp, bbl and oct only)
-tech filename	write out technology data
-clf filename	use CIF format
-rst0 filename	write net list
-rst1 filename	write routing stats based on bounding boxes
-rst2 filename	write routing stats based on steiner trees
-rrst filename	write statistics after ring router
-mag filename	use magic format (wiring only)
-timing filename	write out timing data
-xform filename	write out transform
-np	no pin information will be printed
-s	scale (float)
-my	mirror about y axis (bbl only)
-ps	output window to PostScript

References

1. Dai, W., Marek-Sadowska, M., Chen, B., Pedram, M., and Solden, S., "BEAR Manual," Electronics Research Laboratory Memorandum No. UCB/ERL M89/36, October 1989.
2. M. Khellaf "On the Partitioning of Graphs and Hypergraphs," Ph.D. Diss., *Dept. IEOR, Univ. of California, Berkeley*, 1987.
3. Y. Ogawa, M. Pedram and E.S. Kuh, "Timing-Driven Placement for General Cell Layout," *Proc. Int'l Symposium on Circuits And Systems*, vol. 2, pp. 872-875, May 1990.
4. W. Dai, E.S. Kuh, "Simultaneous Floorplanning and Global Routing for Hierarchical Building-Block Layout," *Proc. Int. Conf. on Computer-Aided Design*, pp. 828-837, 1986.
5. B. Eschermann, W. Dai, E.S. Kuh, and M. Pedram, "Hierarchical Placement for Macrocells," *Proc. Int. Conf. on Computer-Aided Design*, pp. 460-463, 1988.
6. W.M. Dai, B. Eschermann, E.S. Kuh and M. Pedram, "Hierarchical Placement and Floorplanning for BEAR," *IEEE Trans. on Computer Aided Design*, vol. CAD-8, no. 12, pp. 1335-1349, 1989.
7. R.H.J.M. Otten, "Efficient floorplan optimization," *Proc. of the International Conference on Computer Design*, pp. 499-502, 1983.
8. L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Information and Control*, vol. 57, pp. 91-101, 1983.
9. M. Pedram and B.T. Preas, "A Hierarchical Floorplanning Approach," *Proc. Int'l. Conf. on Computer Design*, pp. 332-337, 1990.
10. M. Pedram, M. Marek-Sadowska and E. S. Kuh, "Floorplanning with Pin Assignment," *Proc. Int. Conf. Computer-Aided Design*, pp. 98-101, 1990.
11. L. Kou, G. Markowsky and L. Berman, "A fast algorithm for Steiner trees," *Acta Informatica*, 15, pp. 141-145, 1981.
12. J. Cong, "Pin Assignment with Global Routing," *Proc. Int'l Conf. on Computer-Aided Design*, pp. 302-305, 1989.
13. W. Dai, M. Sato, and E.S. Kuh, "A Dynamic and Efficient Representation of Building Block Layout," *Proc. 24th Design Automation Conf.*, pp. 376-384, 1987.
14. W. Dai and E.S. Kuh, "Global Spacing of Building Block Layout," *Proc. VLSI Conf.*, pp. 161-173, 1987.
15. J.K. Ousterhout, "Corner Stitching: A Data Structure Technique for VLSI Layout Tools," *IEEE Trans. on Computer-Aided Design*, vol. CAD-3, no. 1, 1984.
16. M. Marek-Sadowska, "Route Planner for Custom Chip Design," *Dig. Tech. Papers, IEEE Int. Conf. on Computer-aided Design*, pp.246-249, 1986.
17. W.M. Dai, T. Asano and E.S. Kuh, "Routing Region Definition and Ordering Scheme for Building Block Layout," *IEEE Trans. on Computer-Aided Design*, vol. CAD-4, no. 3, pp.189-197, 1985.
18. H.H. Chen and E.S. Kuh, "Glitter: A Gridless Variable-Width Channel Router," *IEEE Trans. on Computer-Aided Design*, vol. CAD-5, no. 4, pp. 459-465, 1986.
19. H.H. Chen, "Routing L-Shaped Channels in Nonslicing Structure Placement," *Proc. of 24th Design Automation Conf.*, pp. 152-158, 1987.
20. X.M. Xiong and E.S. Kuh, "Nutcracker: An Efficient and Intelligent Channel Spacer," *Proc. of 24th Design Automation Conf.*, pp. 298-304, 1987.
21. X.M. Xiong and E.S. Kuh, "The Constraint Via Minimization Problem for PCB and VLSI Design," *Proc. 25th Design Automation Conf.*, pp. 573-578, 1988.
22. R. Dutta and M. Marek-Sadowska, "Automatic Sizing of Power/Ground (P/G) Networks in VLSI," to appear in *Proc. 26th Design Automation Conf.*, 1989.

Appendix 5. Bugs

1. Where to Report a Bug

Please send electronic mail to:

bear@nowhere.Berkeley.EDU

2. How to Report a Bug

When you report a bug, we will need a copy of all the text files that were used as input to the program. Please be sure to include the BFP description of the cells and the technology file. If you used a timing file, a cluster tree file, a flex file, or a control data file, please include it as well. We will also need a complete list of all the commands that were called. The simplest way to record them is to make a log of your session with the **log** command.