



Itools Expert Documentation

Version v2.0.0pre38q

Wed Jul 25 16:38:53 CDT 2007

© 2005 InternetCAD.com, Inc. All Rights Reserved

Differences between EZ and PDF / Postscript Documentation

Copyright (c) 1998–2005. InternetCad, Inc. All rights reserved.

First and foremost is PDF or Postscript Documentation is not dynamic which means that none of the buttons or entry forms are active. They are instead static pictures. Fortunately, recent versions of PDF support links and the conversion program **htmldoc** retains the links in the output document. Where possible, text links to the output have been added to the documentation wherever the EZ document has navigation buttons as shown in the picture below. For example, click on the blue arrow to the right of the blue button below in order to traverse to the *Getting Started* page. Since the PDF and Postscript documents are static documents, you can not capture all of the information available – only a snapshot. However, PDF and Postscript are portable and can be converted readily into hardcopy. This will allow you to study iTools away from your computer anywhere.

- ✿ Flexible pad placement algorithm.
- ✿ Ability to generate placements close to that obtained from a previous run.
- ✿ CPU time control via fast/slow option.
- ✿ Automatic design flow control.
- ✿ Support for Bourne and Korn shell execution.



Blue arrow on right of button traverses link.

Welcome to EZCad

Copyright (c) 1998–2005. InternetCad, Inc. All rights reserved.



Welcome to **InternetCAD's** Placement and Routing Package or *iTools*. Our software is a consistent [performance leader](#) in the [computer aided design of integrated circuits](#). We believe that industry–leading software should not be difficult to use.

Ittools is a complete timing–driven placement and global routing package applicable to row based and building block design styles. **Ittools** is capable of handling any of the row–based design styles, namely: standard cell circuits, gate arrays, and sea–of–gates circuits. In addition, **ittools** is applicable to circuits containing building blocks or macro cells of any rectilinear shape. Furthermore, the cells may have fixed geometry including pin locations (hard macro cells) or the cells may have an estimated area with a specified aspect–ratio range, and with pins that need to be placed (soft macro cells). **Ittools** is also applicable to floorplanning problems and may be used to completely place and global route mixed macro/standard cell circuits.

EZ is the easy to use interface to **ittools** applications. It uses a new concept in software delivery called [Dynamic Documentation](#). Applications are started and results are displayed within the documentation. Documentation does not become outdated as applications change. There is less context switching and faster learning. Yet the EZ documentation system can now [generate PDF and Postscript documents](#) so you can learn about iTools when you are away from your computer.

In addition, the system is open and all of the Tcl/Tk sources are provided. Individual tools are programable thru the use of Tk/Tcl scripts known as *do* scripts. While knowledge of Tcl/Tk is not necessary, it does allow the user to extend and customize the tools. Learning Tcl is easy and we provide a list of [recommended books for Tcl](#).

Ittools Features

- Standard cell, macro cell, and mixed macro/standard cell design styles.
- Fast state–of–the–art timing driven placement algorithm.

Ittools Documentation

- No constraints on the number or orientation of rows.
- Floating point data supported.
- Macro cells of any rectilinear shape.
- Hard and soft macro cells.
- Gate array designs.
- X11 (X11R2 – X11R6 inclusive) Tk-based graphics interface.
- Based on new simulated annealing algorithm.
- Signal path-based timing driven.
- Upper and lower bounds on path lengths and time constraints.
- Wire length calculations are based on actual pin locations.
- Flexible pad placement algorithm.
- Ability to generate placements close to that obtained from a previous run.
- CPU time control via fast/slow option.
- Automatic design flow control.
- Support for Bourne and Korn shell execution.

[Next Page: Getting Started](#)

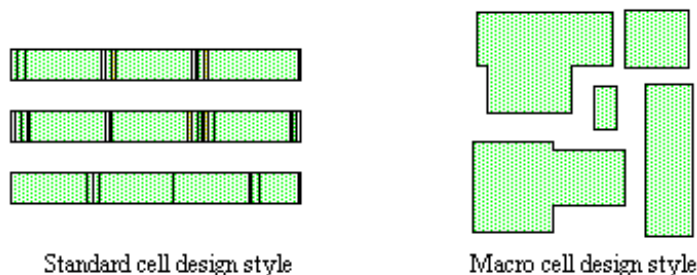


Introduction

Today's rapidly increasing technological advances are due partly to the design and production of low-cost, highly reliable integrated semiconductor circuits. Indeed, integrated circuits have been incorporated into a plethora of common consumer goods ranging from computer controlled automobile ignition systems to video cassette recorders. The complexity of integrated circuits (ICs) in consumer and computer applications has increased exponentially. As the demand increases for integrated circuits, *time-to-market* becomes critical; technology is evolving so quickly that design cycle time has now become a major consideration.

To reduce the design cycle time, computer programs have been applied. Such computer programs are known as computer aided design (CAD) tools which increase the productivity of the integrated circuit designer. In fact, integrated circuits have become so complex that it is now impractical to design without using the computer. Computer aided design has successfully reduced the time of the physical design (layout) phase, the placement and interconnection of the transistors which constitute the integrated circuit. Today's CAD tools primarily focus strictly on digital circuits. One common approach is to use a reusable library of predefined functions or *standard cells* whose specifications have been fully characterized as a basis for the implementation of the design. These standard cells are arranged in rows; the goal of the CAD tool is to place and interconnect these cells in such a way as to minimize the size of the integrated circuit. The size of an integrated circuit directly affects its cost. A smaller integrated circuit yields two major benefits. First, should a defect arise on the silicon wafer during processing, it is more likely that the defect will affect a smaller number of chips on the wafer. Second, a smaller IC will increase the number of chips for a given wafer size [\[142\]](#).

Another method is to design using hand-crafted collections of interconnected transistors known as *macro cells* as the basis of the design. In this case, the requirement for the macro cells to be arranged in rows is not necessary. Hence, the density of the transistors (number of interconnected transistors per unit area) for the macro cell design methodology is greater than the standard cell methodology, but since they are generally not reusable from one design to another, the time necessary to build each of the macro cells is costly. Figure 1.1 shows the two design methodologies.



Standard cell and macro cell methodologies.

Current CAD tools are tailored for one methodology or the other. None have been developed for the mixed macro/standard cell topologies. With the recent introduction of module (cell) generator programs, the design time bottleneck for large regular array structures such as random access memory (*RAM*), read-only memory (*ROM*), and programmable logic arrays (*PLAs*), have been removed. Therefore, it now is advantageous to pursue a mixed approach.

The current physical design CAD tools do not understand *analog* circuits. Analog circuits process continuously varying signals (real world signals) as opposed to the discrete binary levels of digital circuits. Analog circuits have the

additional problems of noise, thermal differences in transistors, and resistive and capacitive parasitic effects, which affect circuit performance. Automatic IC design has existed for only digital circuits. A method for handling the many analog constraints does not exist. This deficiency becomes increasingly important as whole systems are integrated on a chip. The interface to the outside analog world will need to be accommodated.

Further, the assurance that the circuit will function after placement and interconnection is not guaranteed using the current CAD tools. Placement and interconnection influence the time constraints of the signals of the circuit. Current tools ignore the timing ramifications during the layout process. The designer often has to make many alterations in order for the circuit to meet specifications. It is essential that tomorrow's tools understand timing constraints if the design cycle time is to shorten. These problems will be addressed in this thesis.

Background

In order to put the physical design stage into its proper framework, the entire design process for a typical integrated circuit is shown in Figure 1.2. Backtracking and iteration are performed until design goals are achieved for each individual stage. The input to the physical design stage is a structural representation describing the interconnection of *physical* components. The output of the physical design stage contains the geometric information to perform fabrication of the integrated circuit. Physical components may be defined at any of three levels: the *device* level, the *cell* level, and the *module* level. The device level is the lowest level; it describes the physical devices such as transistors, resistors, and capacitors. The cell level describes a small collection of devices previously interconnected, and with geometric data determined at lower stages of the hierarchy. The cell has previously completed the physical design stage at a lower level of the hierarchy. Examples of cell level descriptions are logic gates such as AND gates, or flip-flops, as well as analog subcomponents such as single stage op-amps. The highest level or module level contains large collections of devices which too have been physically defined earlier. Examples are microprocessors, PLAs, RAMs, and ALUs.

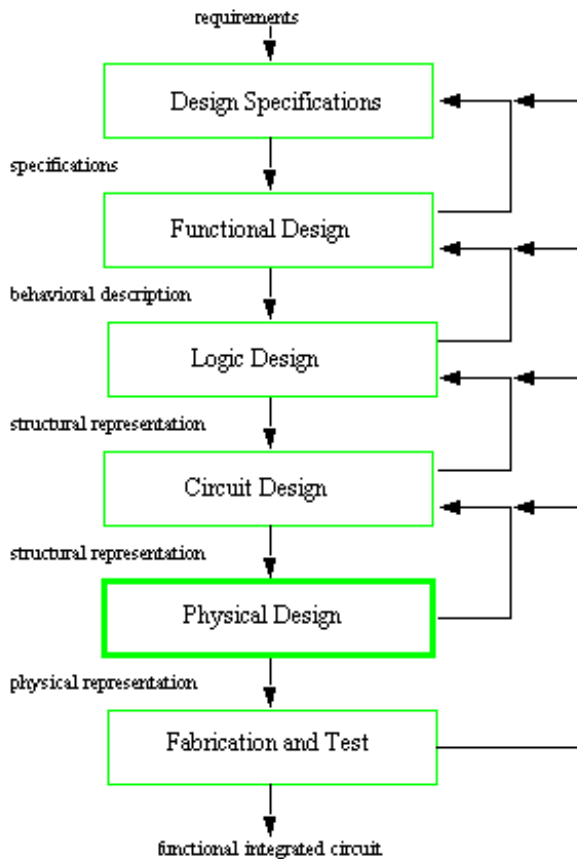


Figure 1.2 Phases of electronic system design. Adapted from [170].

The structural description of the physical components and how they are interconnected is known as a *netlist*. The netlist from the logic or circuit phase contains references to physical components known as *cell instances* and physical interconnections known as *network signals*. Network signals are also known as *nets* or *signals* for short. Throughout, we will use the three terms interchangeably. Figure 1.3 shows an example of the transformation from the circuit level to the physical level. At the circuit level, the netlist may be either a schematic or a textual representation. A schematic consists of symbols denoting the physical components and lines denoting the signals. The point where a signal connects to a component is known as a *terminal pin*. Other names for terminal pins include *I/O*, *pin*, and *terminal*. Figure 1.3a shows the schematic for a one-bit ripple counter. In this figure, "Flipflop" is an instance, "Feedback" is a signal, and "CLK" is a pin. In the textual representations, a cell instance is listed followed by its signals. The signals are ordered according to their position at a lower level of the hierarchy. For example, in Figure 1.3b the signal "Feedback" of instance "1" is connected to pin "D" of cell type "FlipFlop". In the example shown, both the schematic and the text describe the same netlist.

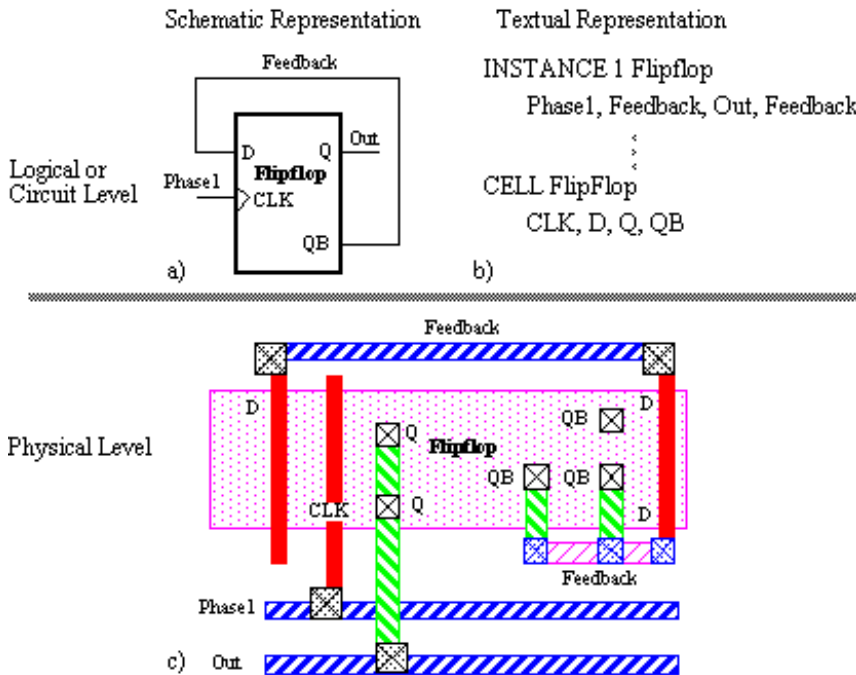


Figure 1.3 Mapping from logical or circuit level to physical level. In c) the labels D, CLK, Q, and QB denote ports.

Figure 1.3c shows the transformation to the physical level. In the figure, hatched regions are *interconnection wires* or *interconnect*, wires connecting the physical components. With the absence of transients and the neglect of resistance effects, wires maintain a constant voltage. Wires are fabricated by depositing various materials on the silicon wafer or *substrate*, usually polysilicon or a metal such as aluminum.

Materials are deposited sequentially on the substrate using photolithography [148]. Photolithography uses *photomasks* to define areas on the silicon substrate where the material will be deposited. Each deposition is known as a *layer*. Layers may loosely be divided into two groups: device layers and routing layers. Device layers such as diffusion are used to create physical devices. Routing layers such as metal are used to interconnect the devices. Some layers, such as polysilicon, may be used for both purposes. The number and types of layers are defined by the fabrication technology. Routing layers are electrically isolated and may cross freely without shorting.

The electrical characteristics of the layer determine the current carrying capacity and the length the interconnection can extend before the signal degrades. For example, polysilicon has a high sheet resistance and can only be used for short interconnection distances whereas aluminum has a lower sheet resistance and is suitable for longer interconnection distances. Early technologies only had two layers of wires, typically one polysilicon layer and one aluminum layer. Today's technologies use up to four metal layers for interconnection. The CAD tools must comprehend the material characteristics of each interconnect layer in order to produce a functional design.

The point where the wire connects to a component is known as a *port*. A circuit pin may have many physical ports, some of which may be electrically equivalent. In Figure 1.3c, there are nine ports for the four pins described in Figure 1.3a or Figure 1.3b. For example, pin QB has three ports, only two of which need to be interconnected. The other port is electrically equivalent and may be connected based on area considerations. The point where two interconnection layers join is known as a *via* or *contact*.

Each layer in the fabrication technology has a set of guidelines known as *design rules*. The design rules specify fabrication constraints on each of the layers. Generally, each layer has a minimum required width and spacing. There also may be rules between two layers. Design rules are changing frequently as fabrication technology is advancing. It is therefore necessary for the physical layout process to be design-rule independent since time-to-market is critical.

Algorithmic Complexity

The transformation from circuit to physical is a difficult task, and one that is normally subdivided into simpler steps as shown in Figure 1.4. In fact, the subproblems are extremely difficult and nearly impractical. For example, suppose we want to solve the placement stage optimally. The placement stage determines the position of the physical components within the IC. If we place n equal size cell instances using a brute force technique that tries every possible placement permutation, we will examine $n!$ different placements. The function $n!$ grows extremely fast. For example, for $n=69$, $n! > 10^{100}$. Even using a computer that is a trillion times faster than the fastest computer on earth, it would take longer than the age of the universe to compute the result! Modern circuits have tens of thousands of components. Clearly, brute force techniques will not suffice. We must use algorithms which run in reasonable execution time. We must look for *efficient algorithms* or algorithms whose worst-case running time is bounded by a polynomial function of the input size [\[42\]](#) [\[134\]](#) [\[222\]](#).

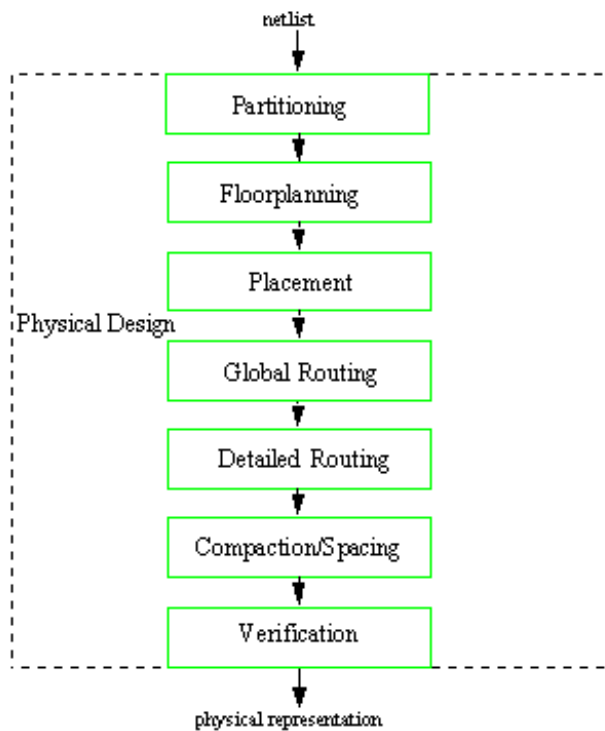


Figure 1.4 Physical design stages.

<i>complexity</i>	n=20	n=50	n=100	n=200	n=500	n=1000
$1000n^*$	0.02 sec	0.05 sec	0.1 sec	0.2 sec	0.5 sec	1 sec
$1000n \lg n^*$	0.09 sec	0.3 sec	0.6 sec	1.5 sec	4.5 sec	10 sec
$100n^2^*$	0.04 sec	0.25 sec	1 sec	4 sec	25 sec	2 min
$10n^3^*$	0.02 sec	1 sec	10 sec	1 min	21 min	2.7 hr
$n^{\lg n}$	0.04 sec	1.1 hr	220 days	125 cent	5×10^8 cent	
$2^{n/3}$	0.0001 sec	0.1 sec	2.7 hr	3×10^4 cent		
2^n	1 sec	35 yr	3×10^4 cent			
3^n	58 min	2×10^9 cent				

Running times as a function of input size. One step takes one microsecond. Asterisks denote polynomial time algorithms [\[222\]](#)

<i>time</i>		10^2 sec	10^4 sec	10^6 sec	10^8 sec	10^{10} sec
complexity	1 sec	(1.7 min)	(2.7 hr)	(12 days)	(3 yrs)	(3 cent)
$1000n^*$	10^3	10^5	10^7	10^7	10^{11}	10^{13}
$1000n \lg n^*$	1.4×10^2	7.7×10^3	5.2×10^5	3.9×10^7	3.1×10^9	2.6×10^{11}
$100n^2^*$	10^2	10^3	10^4	10^5	10^6	10^7
$10n^3^*$	46	2.1×10^2	10^3	4.6×10^3	2.1×10^4	10^5
$n^{\lg n}$	22	36	54	79	112	156
$2^{n/3}$	59	79	99	119	139	159
2^n	19	26	33	39	46	53
3^n	12	16	20	25	29	33

Maximum size of a solvable problem. Asterisks denote polynomial time algorithms. [\[222\]](#)

In order to quantify program execution times, *asymptotic time complexity* has been developed. The constants of the function describing the program's running time are ignored. This simplifies the analysis and ignores the differences in particular machine implementations. For large problem sizes, the relative merit of two algorithms can be determined from the asymptotic growth of the execution time as a function of input size, independent of any constants. Table 1.1 shows the running times for various time complexities. Notice as the problem size increases, polynomial-time algorithms gradually become unusable whereas nonpolynomial-time algorithms abruptly degenerate. Table 1.2 shows the maximum size of a solvable problem for a given algorithm. None of the nonpolynomial algorithms can solve a problem larger than 160 if we allocate three centuries to solve the problem. Even higher order polynomials such as $10n^3$ are not efficient for solving the large problems associated with very large scale integrated circuits (VLSI). Hence, the goal of the CAD tool developer is to design low order polynomial algorithms for solving the physical layout problem.

In addition to time resources, programs require memory resources to store intermediate results as well as the final answer. The space complexity of a program is the rate at which the memory resources grow as a function of the input size of the problem. We seek algorithms that are linear in space complexity.

We will use "big- O " notation to define the asymptotic upper bound for time complexity functions. O -notation gives an upper bound of a function within a constant factor. More formally: For a given function $g(n)$ we denote by $O(g(n))$ the set of functions

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$

$0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0\}$.

For example, $1000n^3 = O(n^3)$ and $2^n + 1000n = O(2^n)$. Figure 1.5 shows the intuition behind O -notation. All algorithms will be described using O -notation.

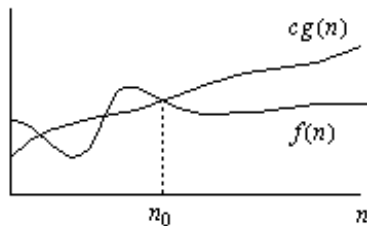


Figure 1.5 O -notation gives an upper bound for a function within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that to the right of n_0 , the value of $f(n)$ always lies on or below $g(n)$ [42].

Most of the layout problems are extremely difficult to solve exactly; they have been shown to belong to the class of *nondeterministic polynomial* (NP) time complexity. NP problems have the characteristic that given a proposed solution to a problem, the solution can be verified in polynomial time [1]. However, the problems seem to have an exponential number of possible solutions and therefore seemingly run in exponential time. If we could build a nondeterministic computer that could guess the correct path to the solution at every decision in the algorithm in polynomial time, all NP problems could be solved efficiently. Unfortunately, such a computer does not exist. We say "seems to run in exponential deterministic time" since no one has been able to prove a superpolynomial-time lower bound. Whether NP belongs to P (the class of polynomial algorithms) is one of the great unanswered questions of computer science [73].

An important subset of NP problems is the class of NP-complete problems. Any NP problem can be transformed into another NP-complete problem in polynomial time. [2] If any NP-complete problem is polynomial-time solvable, then all NP problems can be solved in polynomial time. This would mean that $P=NP$. On the other hand, if it can be shown that any NP problem is not polynomial time solvable, then all NP-complete problems are not polynomial time solvable [46]. Most theoretical computer scientists believe that $P \neq NP$ because no one has found a polynomial time solution for any of the many NP problems. If a problem has been determined to be NP-complete, it is likely that this problem is *intractable* or that no polynomial-time algorithm exists. In this case, it is important to develop fast approximation algorithms and heuristics to solve the problem rather than to try to find a fast exact algorithm.

Graph Theory

Many of the physical design subproblems can be transformed into *graph* problems in which a solution is known. Graphs are one of the fundamental structures of discrete mathematics. A graph G has two ingredients: a set of nodes or *vertices* V , and a set of arcs or *edges* E that connect the nodes. A graph $G = (V, E)$ may either be directed or undirected. In a directed graph, the edge set E consists of ordered pairs of vertices (u, v) where $u, v \in V$. For

directed graphs, self-loops (edges from a vertex to itself) are possible. An undirected graph has unordered pairs of vertices for its edges. Graphs may be represented symbolically. Vertices are denoted by circles or dots. Directed edges are represented by lines with arrows whereas undirected edges are simply drawn as lines. Figure 1.6 shows examples of directed and undirected graphs. We now will present some graph definitions. If (u, v) is an edge in a graph $G = (V, E)$, we say that vertex v is *adjacent* to vertex u . In an undirected graph, the *degree* of a vertex is the number of edges incident on it. In a directed graph, the *out-degree* of a vertex is the number of edges leaving the node, and the *in-degree* is the number of edges entering it. A *path* of length k from a vertex v_1 to vertex v_k is a sequence of vertices (v_1, v_2, \dots, v_k) such that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, \dots, k-1$. The path contains the vertices v_1, v_2, \dots, v_k and the edges $(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k)$. The length of the path is the number of edges in the path. A path is *simple* if all of its vertices are distinct. If there is a path from a vertex u to a vertex v , then v is *reachable* from u . An undirected graph is *connected* if every vertex is reachable from every other vertex and disconnected otherwise. In a directed graph, a path forms a *cycle* if $v_1 = v_k$ and the path contains at least one edge. The cycle is *simple* if v_1, v_2, \dots, v_{k-1} are all distinct. In an undirected graph, a path forms a *cycle* if $v_1 = v_k$ and v_1, v_2, \dots, v_{k-1} are distinct. A graph with no cycles is *acyclic*. Each edge of a graph may be given a property known as a *weight*.

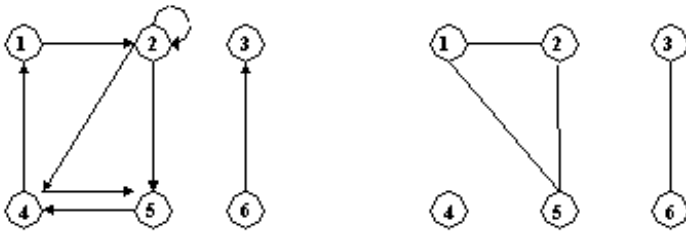


Figure 1.6 Examples of directed and undirected graphs. (a) A directed graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\}$. Edge $(2, 2)$ is a self-loop. (b) An undirected graph $G = (V, E)$, where $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$. Vertex 4 is isolated. From [43].

Several kinds of graphs are given special names. A *complete* graph is an undirected graph in which every pair of vertices is adjacent. A *bipartite* graph is an undirected graph in which V can be partitioned into two sets V_1 and V_2 such that every edge has one end in V_1 and one end in V_2 . An acyclic, undirected graph is a *forest*, and a connected, acyclic, undirected graph is a *tree*. A directed acyclic graph is called a *dag* for short. $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. All of the edges of a *planar* graph can be drawn in the two dimensional plane without crossing. Two other types of graphs are *multigraphs* and *hypergraphs*. A *multigraph* is similar to an undirected graph but may have both multiple edges between vertices and self-loops. A *hypergraph* is like an undirected graph, but each *hyperedge*, rather than connecting two vertices, connects an arbitrary subset of vertices. Two graphs $G = (V, E)$ and $G' = (V', E')$ are *isomorphic* if there exists a *bijection* (one-to-one correspondence) $f: V \rightarrow V'$ such that $(u, v) \in E$ if and only if $(f(u), f(v)) \in E'$. In other words, the vertices of G can be relabeled to be the vertices of G' while maintaining the corresponding edges in G and G' .

When describing graph algorithms, we shall use n to denote the number of vertices and m to denote the number of edges. A graph is *dense* if m is large compared to n and *sparse* otherwise.

Phases of Physical Design

Design styles

Before the start of the physical design stage, the designer must choose the technology and layout methodology for the design. A technology is a particular integrated circuit fabrication process. The layout methodology or *design style* determines the construction of photomasks. Technologies are normally defined by the minimum feature size (typically the smallest allowable layer width), the number and type of routing layers, and the types of devices possible for the

process. For example, in a $1\mu\text{m}$ double level metal CMOS technology, the feature size is $1\mu\text{m}$, two metal layers are available for interconnection, and CMOS (Complementary Metal Oxide Semiconductor) transistors are the semiconductor devices available.

The design methodology is the physical design synthesis process. The physical layout of a design may be handcrafted or constructed using CAD autolayout tools. Layout may be entirely performed by the integrated circuit designer or divided between the IC designer and the system's designer. Figure 1.7 shows the spectrum of integrated circuit design methodologies. Generally, designs at the extremes of the spectrum are created solely by the integrated circuit manufacturer, whereas the semicustom or *application specific integrated circuit* (ASIC) designs in the center of spectrum are created in two stages. In the first stage, the entire design processes for the device and cell levels are performed by the integrated circuit manufacturer. In the second stage, a systems designer or end-user completes the design. The completed design is then returned back to the IC manufacturer for fabrication. Since the lower levels of the physical hierarchy are predesigned by the IC manufacturer, the systems designer is able to design entire systems quickly, and thereby reduce time-to-market. Standard cell, gate array, and sea-of-gates arrays are all ASIC design methodologies.

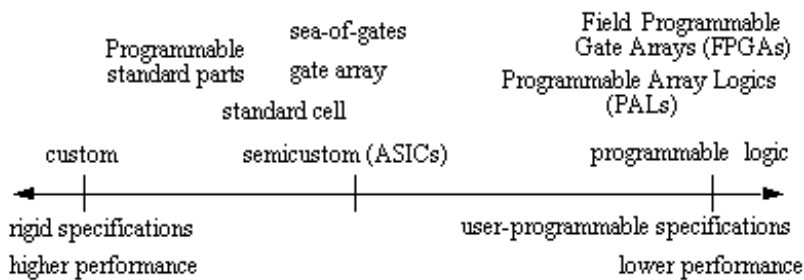


Figure 1.7 Spectrum of design methodologies for integrated circuits.

All physical designs, from custom to automatic layout, may be characterized by three basic geometric styles. The three basic geometric styles are: *row-based* (standard cell), *building-block* (macro cell), and *mixed* (standard/macro cell).

Each style may be implemented with different methodologies. The row-based style may use the *standard cell*, *gate array*, *sea-of-gates* (SOG), or row-based field programmable gate array (FPGA) design methodologies. The building block style may be implemented with *island-style* gate-arrays or macro cells. By abutment, the cell instances in the row-based style can share power and ground signals implicitly yielding area savings. Generally, the height of an individual cell is fixed by the largest cell height of the library leading to area inefficiency³. The building block style needs to have power and ground signals routed explicitly but each of the cells may be individually optimized. Individual cell optimization gives the largest area savings; hence, the building block style is generally more area efficient than the row-based style.

The most flexible row-based methodology, the standard cell methodology, programs all layers at fabrication. Since all layers including diffusion layers are fabricated, the location, size, and the number of transistors of the cells are not fixed at the second stage of the ASIC physical design process. If desired, the system's designer can customize the standard cells for a particular design. In the past, such customization would slow the design cycle reducing the advantage of the predefined library. Recently, procedural standard cell libraries have been developed which automatically optimize the layout of standard cells [176] [177]. These procedural library CAD tools generate customized physical standard cells from symbolic topological descriptions enabling systems engineers to complete the physical design for the device and cell level instantly.

Another advantage of programming all layers is the ability to add more area-efficient macrocells. Automatic module generators exist for building many useful logic functions including ALUs, PLAs, RAMs, and ROMs [219]. Using procedural standard cells and module generators increases the standard cell design performance substantially with a minimal increase in design time. The increased performance and area savings make mixed standard/macro cell designs very desirable.

Figure 1.8 shows an example of a standard cell design. The integrated circuit can be divided into regions, the core region (area inside dotted square) and the I/O region (area outside). The connections to the outside world (the package terminal pin) to the integrated circuit are made using a bond pad cell. Generally, the bond pad is connected to the package pin ultrasonically using fine gold wire [148]. The regions between the standard cell rows are routing regions known as *channels*. In the standard cell methodology, the heights of these channels are not fixed but rather determined by the necessary routing area.

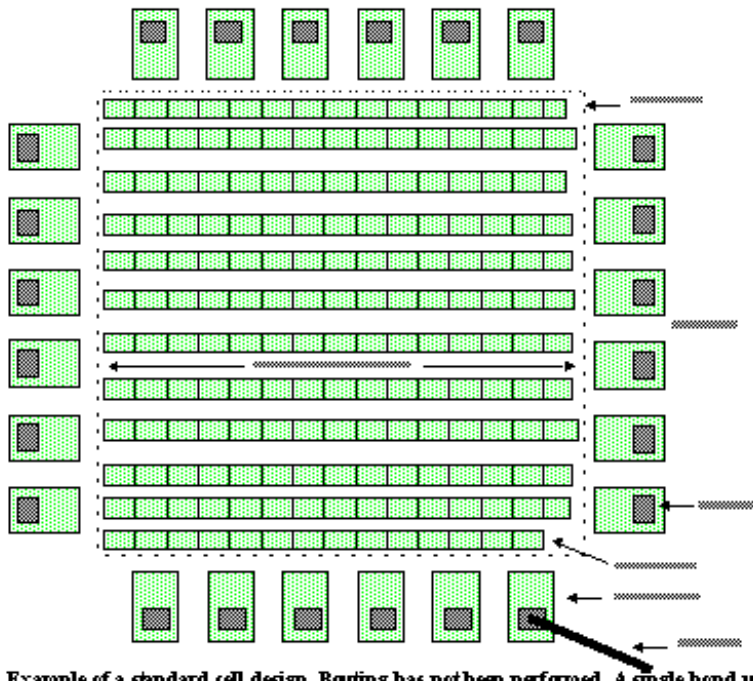


Figure 1.8 Example of a standard cell design. Routing has not been performed. A single bond wire is shown.

One disadvantage of the standard cell methodology is cost. Performing all of the photolithography steps costs time and money. While programming all layers leads to design flexibility, it requires unique photomasks for each design. None of these photomasks can be used for any other future designs. In addition, any equipment developed to test the IC cannot be shared over designs. The gate array methodology tries to alleviate this problem by allowing the systems designer to only customize the interconnect layers. All transistor and device levels are prefabricated by the IC manufacturer. Instead of the ten to twenty photomasks required by the standard cell methodology, gate arrays need only three to seven interconnect photomasks to program the design. After the systems designer completes the second phase of the design, the IC manufacturer needs only to process the interconnect layers, saving up to two weeks of fabrication time. In addition, the IC manufacturer can mass produce the unprogrammed gate arrays known as *base arrays* and take advantage of the economies of scale. This reduces the lead time and cost for manufacturing the completed design. Furthermore, the test apparatus may be shared over all designs.

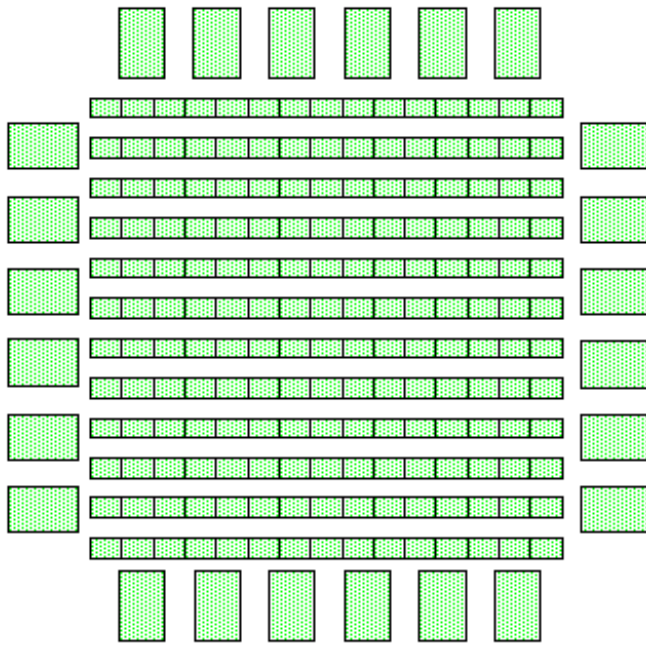


Figure 1.9 Example of a gate array design. The routing is not shown.

However, the decision not to program all layers has drawbacks. Figure 1.9 shows a gate array design. Since the device and cell level are already prefabricated, the heights of the channels are fixed. The IC manufacturer must decide how much area should be reserved for each of the channels in the first phase. Too much reserved routing area leads to low utilization of the gate array resources while too little area leads to designs that are unrouteable in the second stage. For this reason, IC manufacturers offer a line of various sized gate arrays; the systems designer seeks to find the one that fits the best.

An extension of the gate array methodology is the sea-of-gates style. The sea-of-gates array consists of many rows of transistors as shown in Figure 1.10. In the sea-of-gates style, there are no areas reserved strictly for interconnect as with the traditional gate array. Instead, routing is performed in the same area as the rows of transistors. If there is not enough area to complete the routing in a region, an entire row of transistors may be left unconnected. Since the cell level routing is eliminated when the transistors are left unused, more resources are available to complete the routing. The region is expanded by eliminating rows of transistors until the routing can be completed. Many layers of interconnect must be available to make this scheme practical. However, if enough routing resources are available, this scheme is very area efficient.

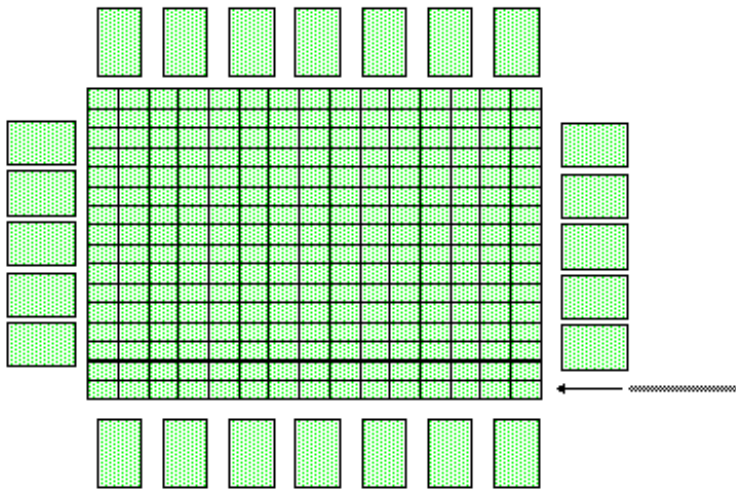


Figure 1.10 Example of a sea-of-gates design. The routing is omitted.

All types of gate arrays suffer from the inability to implement large regular arrays such as RAMs and PLAs efficiently. Either the array is defined as a prerouted macro cell in the base array or as a collection of adjacent row cells which then need to be routed. If the array is defined as a macro cell, then the size and placement of the array is fixed. Because it is unlikely that the defined size exactly meets the needs of the systems designer, parts of the array would be unused and result in wasted space. If the array is created by wiring cells together, the routing area will greatly exceed that of the macro cell because the individual row cells cannot be optimized for both array cells and the normal gate array cells. Therefore, gate arrays are not ideal candidates for the mixed approach.

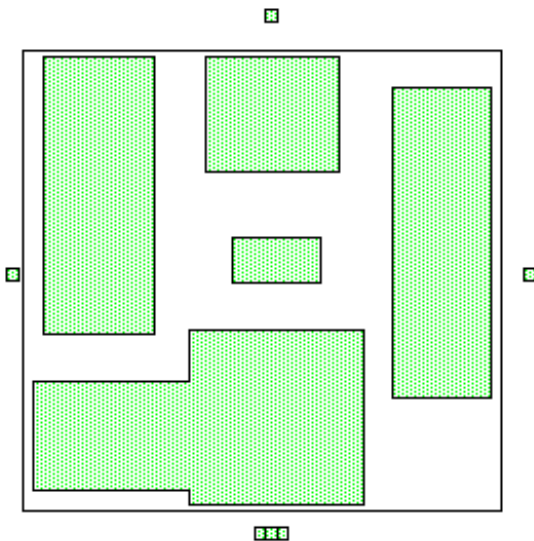


Figure 1.11 Example of a macro cell design.

At the other extreme of the spectrum is the building block approach which is the typical style for fully handcrafted designs. Figure 1.11 shows an example of a macro cell design. In general, macro cells can take arbitrary shapes. Most floorplanning and placement algorithms, however, limit the shapes to rectangles or rectilinear figures. Macro cells are optimized for area and performance, but routing using this approach becomes more difficult. Characteristically, the routing regions become complex and some area is wasted. However, the area efficiency of the macro cell style is still better than strict row-based methodologies.

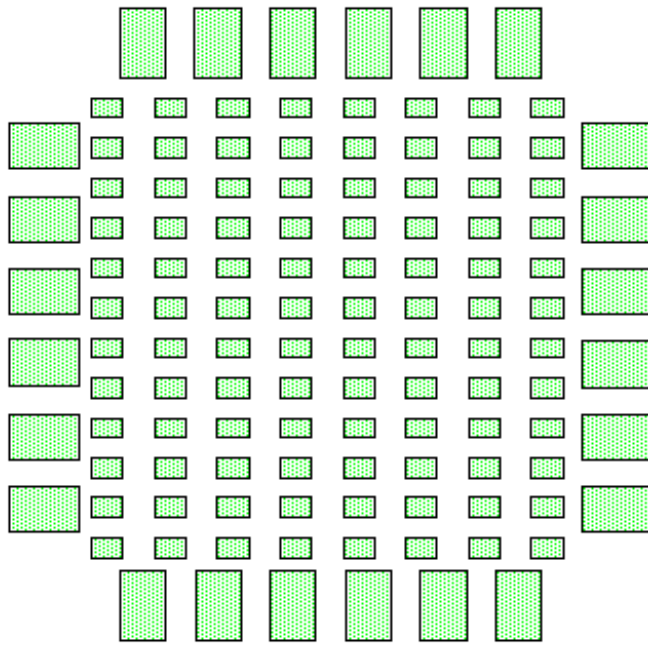


Figure 1.12 Example of an island-style gate array.

Another methodology is the island-style gate array. It shares aspects with both the row-based styles and the building block styles. It is related to the building block style in that signals may not be routed implicitly through abutment. It is similar to row-based styles where all of the cell instances are arranged in rows and columns. Since it is a gate array, it again suffers from the inability to handle large arrays efficiently. Unlike row-based gate arrays, routing needs to be performed in two dimensions. Its usefulness is therefore limited, and is not very common today except for some FPGA implementations where the cell instances are logically more complex. An example of an island-style gate array is shown in Figure 1.12.

Field programmable gate arrays (FPGAs) differ from traditional ASIC methodologies in that the system designer completes the programming of the integrated circuit but does not need to return the design to the IC manufacturer for fabrication. FPGAs are completely processed integrated circuits. The systems designer executes software which programs the integrated circuit. The system's designer can program an FPGA instantly (neglecting software execution time). This allows rapid prototyping of large systems. There is, however, a compromise between performance and programmability. FPGAs do not have the performance of their less programmable counterparts. FPGAs have been proposed in all three geometric styles.

Partitioning

Once the technology and design style have been chosen, physical layout can begin. Partitioning is the first stage in the physical design process. The goal of this stage is to break the netlist into one or more manageable pieces. There are two levels of partitioning: partitioning a system among multiple integrated circuits and partitioning a single integrated circuit design into multiple components. Today's designers are faced with a dearth of automatic tools which they use to partition systems into multiple chips. Within a single integrated circuit, partitioning is performed whenever macro cells exist, or when the number of row-based cells is extremely large. Each piece of the netlist will become a component in the floorplanning stage. Often, the partitioning stage is implicit; the designer has created the logic to mimic the physical components available using a library of predefined components. If all the components are standard cells or gate array cells, and the number of standard cells is sufficiently small, partitioning need not be performed. In the event that both macro blocks and standard cells are present in the design, the standard cells will be partitioned into one or more *softcells* while completed macro blocks will be partitioned into *hardcells*. A soft macro is a macro cell in which some geometric quantity is unknown such as aspect ratio, pin locations, or shape. A hard macro has all information determined. After partitioning, each integrated circuit will have a reduced netlist ready for floorplanning.

Partitioning a netlist into two pieces by minimizing the number of nets interconnecting the two pieces can be computed in $O(n^3)$ using a variant of the Ford–Fulkerson algorithm [66]. When size restrictions are imposed on the two pieces, partitioning becomes NP-complete.

Floorplanning

Floorplanning is the second stage in the physical design process. As its name suggests, the purpose of floorplanning is to plan the overall physical structure of the integrated circuit. The floorplanner's input consists of a partitioned netlist of macro cells modeling the physical components. Some components in this netlist may not be completely characterized in terms of area, aspect ratio, timing, or I/O terminal positions. The floorplanner will determine the uncharacterized aspects of any soft macro cells.

The goal of the floorplanner is to place the hard and soft macro cells at the position within the integrated circuit which minimizes total cell area and maximizes circuit performance. In order to estimate total chip area and performance accurately, the floorplanner must also estimate the wiring area between components.

Some floorplanning algorithms only allow *sliceable* floorplans. A sliceable floorplan is derived by repeated bipartitions of the core as shown in Figure 1.13a. This floorplan can be subdivided into two pieces at each step if the core is divided using the order specified in the figure. Sliceable floorplans have the desirable property that they can be routed using only a channel router in the reverse order of the bipartitioning cut lines; however, this method does not yield the smallest area. Other floorplanning algorithms permit *nonsliceable* floorplans as shown in Figure 1.13b. In the nonsliceable floorplan, there is no way to bipartition the area into two complete pieces at each step. Routing the nonsliceable floorplan is a complex task requiring specialized routers known as *switchbox* routers or *area* routers. However, this floorplan yields the smallest area, the primary objective of floorplanning.

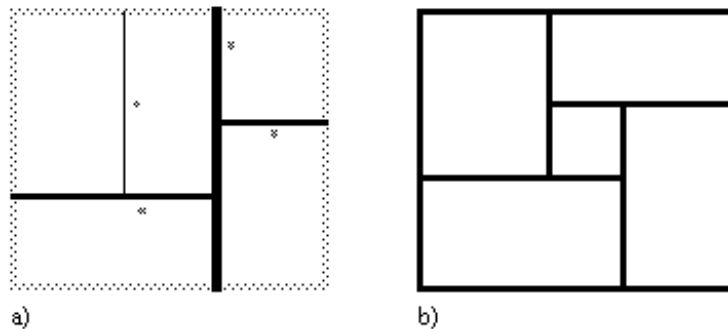


Figure 1.13 Examples of a) sliceable and b) nonsliceable floorplans.

Placement

Placement positions the cell instances within the integrated circuit. The ideal goal of placement is to position the cells to yield maximum performance using minimum area. Unfortunately, this is an incredibly tough task because the number of possible placements grows exponentially with the number of placeable objects. In order to calculate the exact area required for a placement, we must complete the remaining steps of the physical design process. However, all of these subtasks are NP-complete problems, and there is not an efficient way to calculate the exact area of a placement. Therefore, at the placement stage, we must work with an estimate of the area or use a heuristic.

There have been numerous algorithms proposed to solve the placement problem. Placement algorithms may be broadly divided into four categories: constructive, iterative, analytical, and esoteric algorithms. Constructive algorithms selectively add one cell at a time to the placement based on an evaluation function. Iterative algorithms take an initial placement and improve it by modifying the configuration. Analytical algorithms mathematically

calculate the positions of the cells from the network description. Esoteric algorithms are derived from recent advances in other related fields. Examples of these methods will be discussed in Chapter 3.

Global Routing

Global routing is the decomposition of an integrated circuit interconnection network into *net segments*, and the assignment of these net segments to regions or channels. The global routing results will be fed to a detailed router on a channel by channel basis. The detailed router will create the physical geometries necessary to manufacture the photomasks. This divide-and-conquer strategy produces global view solutions while managing the complexity of large circuit designs. It is assumed that the positions of the pins of a net have been determined in the placement phase.

There are two types of global routing: graph-based global routing and plane-based global routing. In graph-based global routing, a graph is built which models the topology of the placement. Each edge of the graph is associated with a routing region. Every edge is assigned a weight equal to the width or *capacity* of the routing region. The pins of the nets are then projected onto the graph. The task of the graph-based global router is to connect the pins of all the nets without violating any capacity constraints. Each net that passes through an edge adds a *track*, or a net spacing requirement, to that routing region. The total cost of an edge is the maximum density of the tracks passing through the edge. For a feasible routing, the cost for every edge must be less than or equal to the capacity for that edge. In addition, the resulting subgraph should be a tree (free from any cycles). The minimum interconnection trees are known as Steiner trees. [4](#) Figure 1.14 shows an example of graph-based global routing. The graph surrounds the cell instances which are the shaded rectilinear regions. The minimum Steiner tree for five pins is shown.

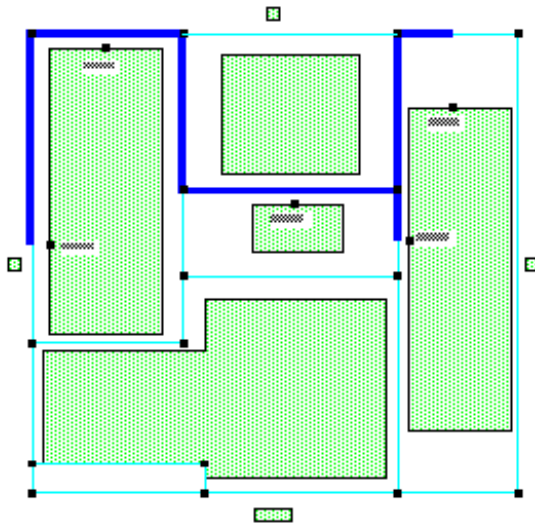


Figure 1.14 Example of graph-based global routing for a 5 pin net. The thick line shows the Steiner tree for this net.

The second type of global routing is performed on the plane as shown in Figure 1.15. In this case, the global router may have to add *feedthroughs* which are cells that allow a wire to cross through a cell region. There are five feedthroughs in Figure 1.15. There are two types of feedthroughs: *explicit* and *implicit*. An *explicit* feedthrough is a special cell instance that only contains interconnect to cross the row. It adds to the length of a row. An *implicit* feedthrough is an uncommitted crossing point built into a library cell. Its addition does not change the length of a row. When necessary, it is advantageous to use implicit feedthroughs because they do not add to the width of the chip.

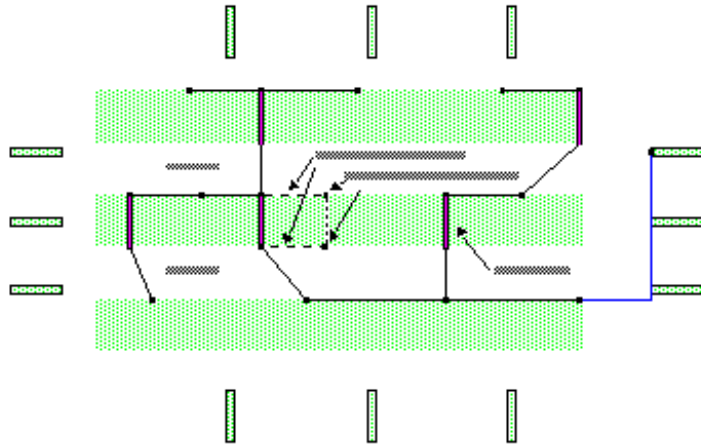


Figure 1.15 Example of plane-based global routing.

Plane-based global routers also need to determine the position of switchable net segments. Switchable net segments are net segments which may be placed in any of several routing regions. The global router must determine the routing regions for all switchable segments such that the total area of the chip is minimized. Figure 1.15 shows a switchable net segment which may be placed in either region 1 or region 2.

Most global router algorithms route one net at a time finding the shortest route for each net. Often it is not possible to meet the capacity constraints using the shortest routes for all nets because they compete with each other for the available routing space. The route for one net will often block another from completing its connection. The order that the nets are routed becomes a critical factor in the final result. In this thesis, we will propose a method which looks at all nets simultaneously and seeks to avoid the *routing-order dependence problem*.

Detailed Routing

After global routing has determined the topology of the interconnections, the detailed routing phase begins. Detailed routing creates the geometries for fabrication including the size, position, and layer for each net segment, and the placement of the vias which join the segments of a net. The many detailed routers that have been proposed can be broken into two main groups: general purpose and restricted routers. *Maze* and *line probe* are examples of general purpose routers. The restricted category includes *channel*, *switchbox*, *power and ground*, and *river* routers.

A *maze* router (Lee router) operates on a gridded model of the routing region and routes a single net at a time [129] [156]. The width of the grid is set to the *pitch* design rule for the routing layer. A maze router starts from a source port and expands in a *breadth-first* manner labeling each grid with the current length until it hits the target port as shown in Figure 1.16. A maze router will always find the shortest path between the source and the target if such a path exists. A maze router can always find its way around obstacles and can be extended to handle any number of layers. It may be implemented in $O(n \log n)$ time where n is the number of grid points in the maze [130].

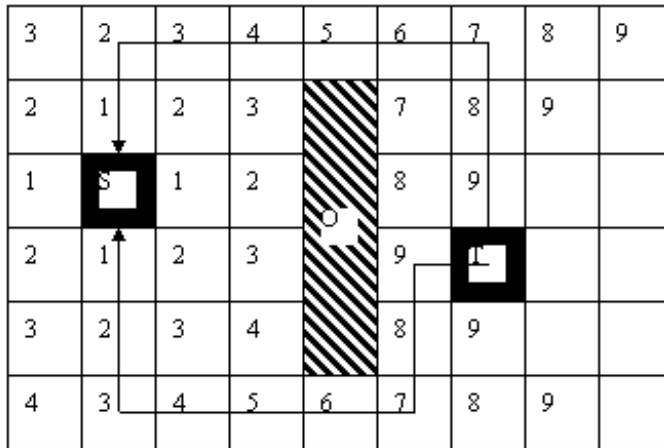


Figure 1.16 An example of one layer maze routing. The width or height of each square equals the grid size. The algorithm labels the grids in a breadth-first manner. Each grid was visited at the step given by the label. There are several paths from the source *S* to the target *T* avoiding obstacle *O*. Each path has the same length (9) but the number of bends may vary. Two such paths are shown.

However, maze routing is not without disadvantages. The most severe problem is the large amount of memory required for large designs. The space complexity of maze routing is $O(n^2)$ where n is the number of grid points in the maze. For large circuits with long interconnects, the number of grids to visit in the search becomes enormous, and the run time becomes prohibitive. In addition, maze routers create connections sequentially and, therefore, suffer from routing order dependence. Additional problems arise when routing layers have different design rules. It is difficult to match the grids between layers except in the special case where all layers have a non-trivial greatest common divisor. Otherwise, it is best to route with a simple grid for each layer and have the compaction/spacing phase correct design rule violations.

In order to reduce the memory requirements of detailed routing, line probe routers have been developed [89] [152]. Instead of storing the entire routing area in terms of grids, a line probe router stores only the features of the routing boundary, obstacles, and previously routed nets. Each feature is stored as a set of line segments. The algorithm starts by projecting *line probes*, or lines from both the source and target ports, as far as possible in horizontal and vertical directions. If two probes intersect, the route is complete, but if blocked by an obstacle or already placed wiring segments, an escape point is generated and new probes are projected from that point. This process continues until two lines intersect yielding a route, or all escape points are exhausted. Figure 1.17 shows a line probe router in action. Although line probe routers drastically reduce the memory requirement for detailed routing, they may not find the shortest length route and they may not find a route even if one exists. In addition, they also suffer from the routing-order dependence problem.

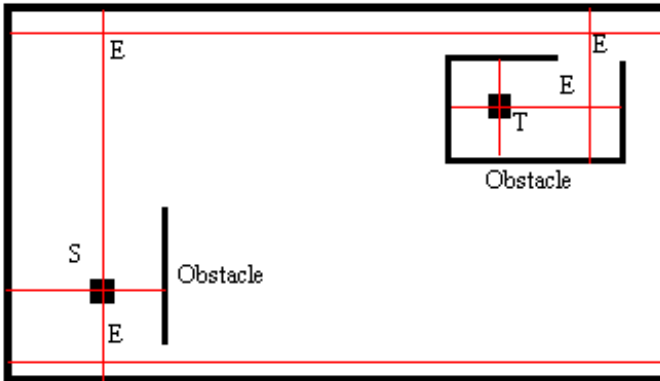


Figure 1.17 Example of a line-probe router [89]. Escape points are labeled E.

The most prevalent integrated circuit router is the channel router. The channel router restricts pins to the top and bottom sides of a rectangular routing region known as a *channel*. The width of a channel is fixed but not the height. In exchange for this restriction, a channel router is able to route all nets in parallel avoiding the routing–order dependence problem.

The first channel router was based on the left-edge algorithm (LEA) [86]. This algorithm restricts each net's horizontal span to a single wire segment. The algorithm proceeds as follows: First, all of the horizontal spans of the nets are sorted by their left endpoint. Each track is processed in turn starting at the left edge of the channel. The first unplaced segment in the sorted list is placed into the bottom track. Next, the algorithm searches the sorted list to find the next segment which will fit in the bottom track. The scanning is repeated until no other segment can fit in this track. The algorithm then repeats for the next track, trying to add as many segments as possible to the current track. The scanning terminates when all horizontal spans are placed. The connections to the pins in the vertical directions are then added completing the route.

The above algorithm works correctly except in the case of *cyclic vertical constraints*. *Vertical constraints* are formed wherever different vertical wire segments attach to the terminal pins at the top and bottom of the channel at the same x coordinate. If a vertical wire segment connects to a terminal at the top of the channel, its connection to the horizontal wire segment must be above any connection to a pin of another net at the bottom of the channel at the same x coordinate. Otherwise, a short circuit would develop between the two signals. This can be represented using a vertical constraint graph. The vertical constraint graph is a directed graph where the nodes are the signal names of the terminal pins and directed edges are formed from pins at the top of the channel to pins at the bottom of the channel at the same x coordinate. Figure 1.18 shows an example of a cycle in the vertical constraint graph. In this case, nets A and B cannot be routed in two layers using the left-edge algorithm without creating a short circuit. In order to complete the route, a *dogleg* must be introduced. A dogleg is a vertical wire segment which occurs at a nonterminal position (for the net). A dogleg will break the horizontal span into two pieces which will be placed on different tracks. With the single horizontal wire segment restriction removed, the channel can now be routed as shown in Figure 1.19.

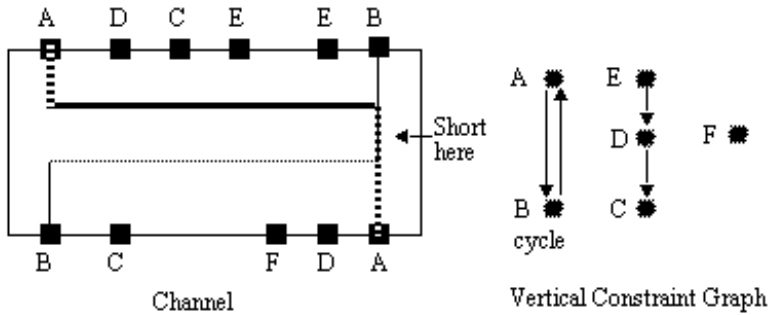


Figure 1.18 Example channel and corresponding vertical constraint graph.

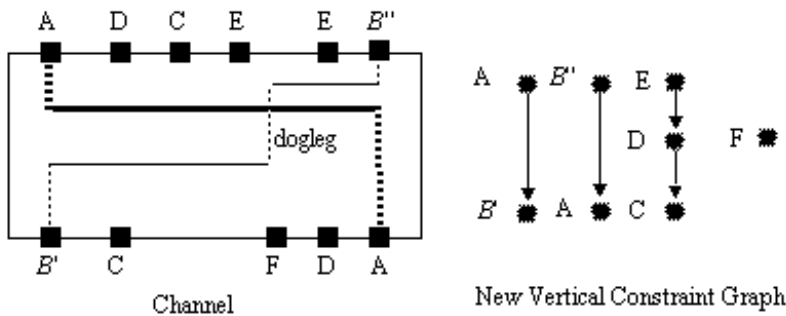


Figure 1.19 Addition of a dogleg to the example channel. The new vertical constraint graph is shown

Many channel routing algorithms have been proposed. The dogleg channel router breaks vertical constraints by splitting tracks into sections and connecting them with doglegs [48]. YACR2 uses a pattern router (special maze router) to fix the vertical constraints [179]. Other channel routers avoid the left edge algorithm entirely: The *greedy* approach wires the channel column by column [182]. The hierarchical channel router routes the channel recursively [22]. Regardless of the algorithm, the channel router guarantees a 100% completion rate since it has the freedom to increase the height of the channel, but such a freedom adds area to the chip. Figure 1.20 shows the output of a channel router.

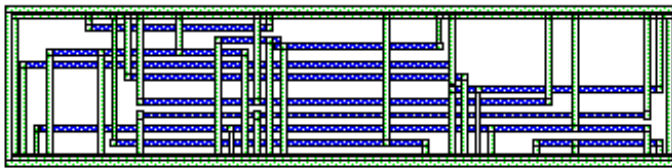


Figure 1.20 A channel routing example using two layers. This is not a left-edge algorithm[234].

Another type of router is the area or switchbox router [37] [80] [102] [207]. An area router makes interconnections within a fixed boundary. Pins may occur anywhere within the fixed boundary. Figure 1.21 shows an example of an area router.

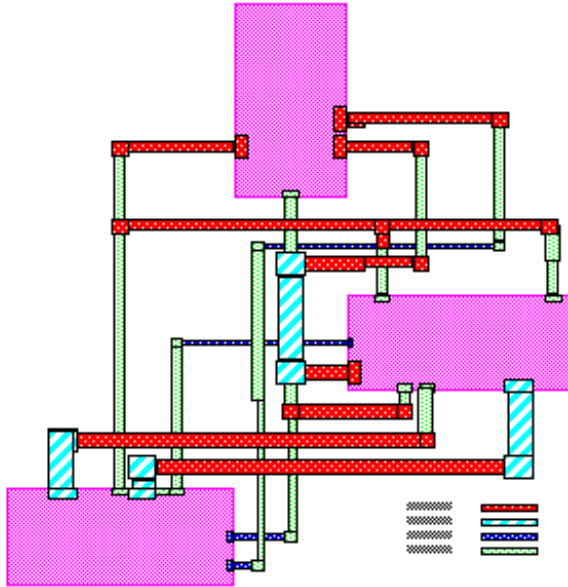


Figure 1.21 An area route for a macro cell example using four layers.

Another specialty router is the *river* or planar router. A route is planar if it can be described using a planar graph. In the planar graph, the nodes of the graph are the pins and the edges of the graph are the nets. Planar graphs can be implemented using a single layer. This style of routing is useful for buses and data flow architectures.

Power and ground interconnections are normally made with a specialized router. Power and ground connections need to have different widths due to *electromigration* problems and voltage drop constraints. *Electromigration* is the redistribution of metal molecules of a wire when the current density in a wire exceeds $5 \times 10^5 \text{ A} \cdot \text{cm}^{-2}$ [149]. Since the redistribution removes metal molecules from the region of highest current density, open circuits are created at these points. The problem can be rectified by increasing the width of the conductor and thus reducing the current density. In addition, a circuit may not function if the voltage drop due to the resistance in the power and ground wires is large. The resistance R of a wire segment is given by,

$$R = n \cdot R_s,$$

where n is the number of squares of a conductor and R_s is the sheet resistance of the conductor in ohms per square [6].

A power and ground router must size the width of the power nets to meet the electromigration and voltage drop constraints yet route in a minimum amount of area [33] [58].

Compaction/Spacing

Compaction or spacing is an optional step to reduce integrated circuit area while eliminating design rule violations. If a design has initial design rule violations, spacing could actually increase the size of the chip. Spacing minimizes the area of an integrated circuit without changing its topology. It is important to keep the topology constant to preserve the timing and performance optimization of the previous phases. Spacing is mandatory whenever the detailed routing step is performed in the symbolic domain. Spacing is also used to transform design rule independent layouts into properly spaced designs for a given technology.

Spacing algorithms can be divided into two types: *virtual grid* compactors and *constraint-graph* compactors. They can be further classified as one-dimensional (1D) or two-dimensional (2D) compactors. The virtual grid method finds

component positions by grouping all components at the same grid line together such that adjacent virtual grid lines are as close as possible, and design rules are maintained between any two components [\[184\]](#) [\[236\]](#) . Experiments have shown that virtual grid compaction is not as effective as constraint–graph compaction [\[171\]](#) .

The most widely used algorithm for spacing is constraint–graph compaction [\[137\]](#) . The required spacing and connections for physical components are modeled using a directed weighted graph. The nodes of the graph represent the positions of the components, and the edges of the graph denote constraints between components. The constraint graph algorithm is usually executed in one dimension, where compaction is attained optimally. Two dimensional compaction is performed by compacting in one direction and then in the other. The direction is alternated until no further consolidation of area is possible.

The 1D compaction algorithm proceeds as follows: First, the longest path in the constraint graph is found. Next, components along the longest or *critical* path are placed at their minimum positions. Finally, the remaining components which are not on the critical path are placed. For noncritical components, there is some freedom in placement. Many move strategies have been proposed including left–edge, right–edge and centering strategies [\[164\]](#) . The move strategy employed in the current compaction direction affects the outcome of the next compaction direction. Figure 1.22 shows an example of 2D compaction using successive applications of 1D compaction. In all steps, the topology in the compaction direction is preserved. However, the topology in the orthogonal direction is changed and the resulting 2D topology is different for the two orders. This is not surprising since 2D compaction has been shown to NP–complete [\[191\]](#) . Large changes in the topologies of cell instances can create large changes in the routing resources leading to an increase in chip area. In this thesis, we will present an algorithm which preserves the 2D topology using the 1D compaction algorithm.

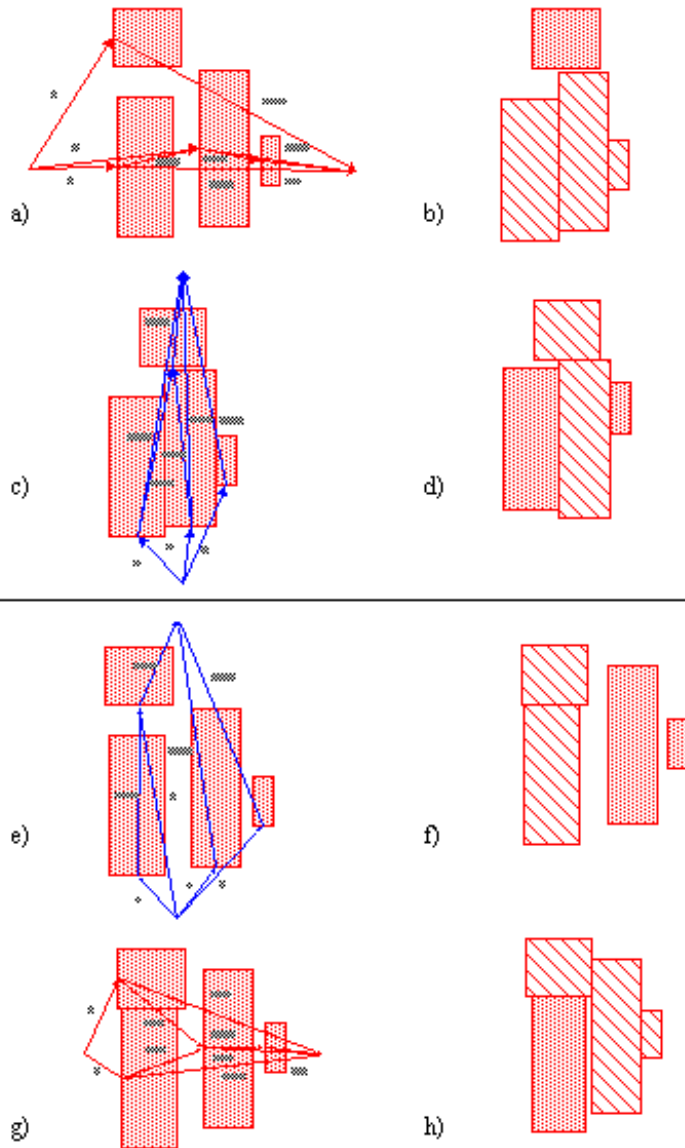


Figure 1.22 Order differences in 1D compaction. Steps a-d show x-compaction before y-compaction while steps e-h show y-compaction before x-compaction. The final topology is different in the two cases. The hatched areas are on the critical path. The labels denote the constraints. A centering move strategy has been employed.

Design Verification

The final step in the layout process is the verification step. The design must be verified to make sure it is design rule correct, functionally correct, and meets all of the performance criteria [172]. Verification CAD tools have been developed to insure that the design does not contain any design rule violations. A design rule checker verifies that each mask geometry meets all spacing and width constraints [141].

Functional correctness is insured by comparing a physical netlist against the system designer's circuit netlist [59]. The physical netlist is created by *extracting* all devices from the physical layout. The *extractor* program recognizes devices by the composition of layers. For example, an N -channel transistor is formed whenever the polysilicon layer overlaps an n -type diffusion layer. The devices are then interconnected by extracting the routing layers. The resulting physical netlist is matched against the user's input and any discrepancy is noted. Automatic layout systems are correct-by-construction and should not need this test. However, a flaw in an algorithm or data entry mistake could

render the entire IC nonfunctional. This step seeks to avoid such costly problems.

Performance criteria may be validated by resimulating the extracted physical netlist. In this case, the routing interconnect parasitics are extracted. The parasitics may be modeled as a lumped capacitance placed at output of transistors or modeled as resistor–capacitor (RC) trees if more accuracy is needed [\[94\]](#).

If design verification does not detect any errors, the IC is sent for fabrication. Otherwise, the systems engineer must determine the problem and correct it. Such troubleshooting is time consuming. Therefore, the layout process must be flawless as time–to–market is critical in today's world. In this thesis, we will present algorithms which meet this need.

Organization of the Thesis

The remaining part of this thesis will deal with specific topics in automatic placement and routing. A chapter will be devoted to each subject. Each chapter will describe its subject, outline previous work done on the topic, and present new algorithms for solving the problem.

1. More formally, an algorithm or language L belongs to NP if and only if there exists a two–input polynomial–time algorithm A and a constant c such that

$L = \{x \in \{0, 1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}$.

2. An algorithm or language L is NP complete if

a) $L \in NP$, and

b) $L \leq_P L'$ for every $L' \in NP$ ($L \leq_P L'$ means there is a polynomial time transformation from L to L').

3. This problem can be reduced somewhat by using a channel router which can handle a variable height channel.

4. Formally, the Steiner tree problem in graphs is defined as follows: Given a graph $G = (V, E)$, a weight $w(e) \in \mathbb{Z}_{\geq 0}$ for each $e \in E$, a subset $R \subseteq V$ and a positive integer bound B , is there a subtree of G that includes all the vertices of R and such that the sum of the edge weights in the subtree is no more than B ? [\[73\]](#)

5. Pitch is defined as the sum of the minimum spacing plus minimum width for a layer. The center–to–center distance between two adjacent routing tracks on the same layer must be greater or equal to the minimum pitch for the layer.

6. A square is a square piece of conductor. The number of squares for a wire segment may be obtained by dividing the length of the conductor by its width.

Dynamic Documentation

Dynamic Documentation is the first *active* hypertext system for Integrated Circuit Place and Route Systems. Unlike other passive help systems where documentation is added as an afterthought, **Dynamic Documentation** **IS** an active documentation wizard which does the work for you! But **Dynamic Documentation** is much more than just a wizard; it is a system where the software and documentation are one.

Dynamic Documentation is a revolutionary new graphical user interface (GUI) which allows users of all levels, from beginners to experts, to productively interact with a program. **Dynamic Documentation** supplies precisely the right amount of documentation detail based on the user's experience. Users will be led through the necessary documentation, and actually execute the program from within the documentation itself! **Dynamic Documentation** incorporates the latest advances in usability engineering, passing knowledge directly to the user.

How it works

This section describes a user's perspective of Dynamic Documentation. For a technical discussion please [click here](#).

Dynamic Documentation is written in HTML (Hypertext Markup Language). Two extensions to standard HTML are used, JavaScript and plugins. Both of these extensions are available from **itools'** native browser and Netscape Navigator (c) version 3.0 and above. You must use either Netscape or the native browser in order to unlock the power of Dynamic Documentation.

Both the **itools** plugin to Netscape Navigator and the **itools** native browser are based on the general purpose [Tcl/Tk](#) interpreter. Tcl/Tk is an interpreted scripting language for creating textual and graphical interfaces. **itools** applications use Tcl/Tk user interfaces and they can be invoked and communicate between themselves using Tcl/Tk. Learning Tcl is easy and we provide a list of [recommended books for Tcl](#).

When the **itools** plugin is first loaded into Netscape Navigator, it creates a Tcl interpreter. Subsequent invocations of the plugin can be used to create windows on the Netscape page. The Tcl interpreter can be called from JavaScript to manipulate this window and start **itools** applications.

The native **itools** browser's architecture is quite different from Netscape. **Ittools'** browser is written completely in Tcl/Tk augmented with a Javascript interpreter. It is generally more stable than the Netscape's browser but supports only features needed to implement Dynamic Documentation.

The other major function of Dynamic Documentation is to save and restore the user's state. This allows users to start up EZ where it was last exited. The state of design and display are stored in a number of Tcl variables. These variables are written out to the "~/itools/eztcl.ini" file whenever they change. This file is read on startup.

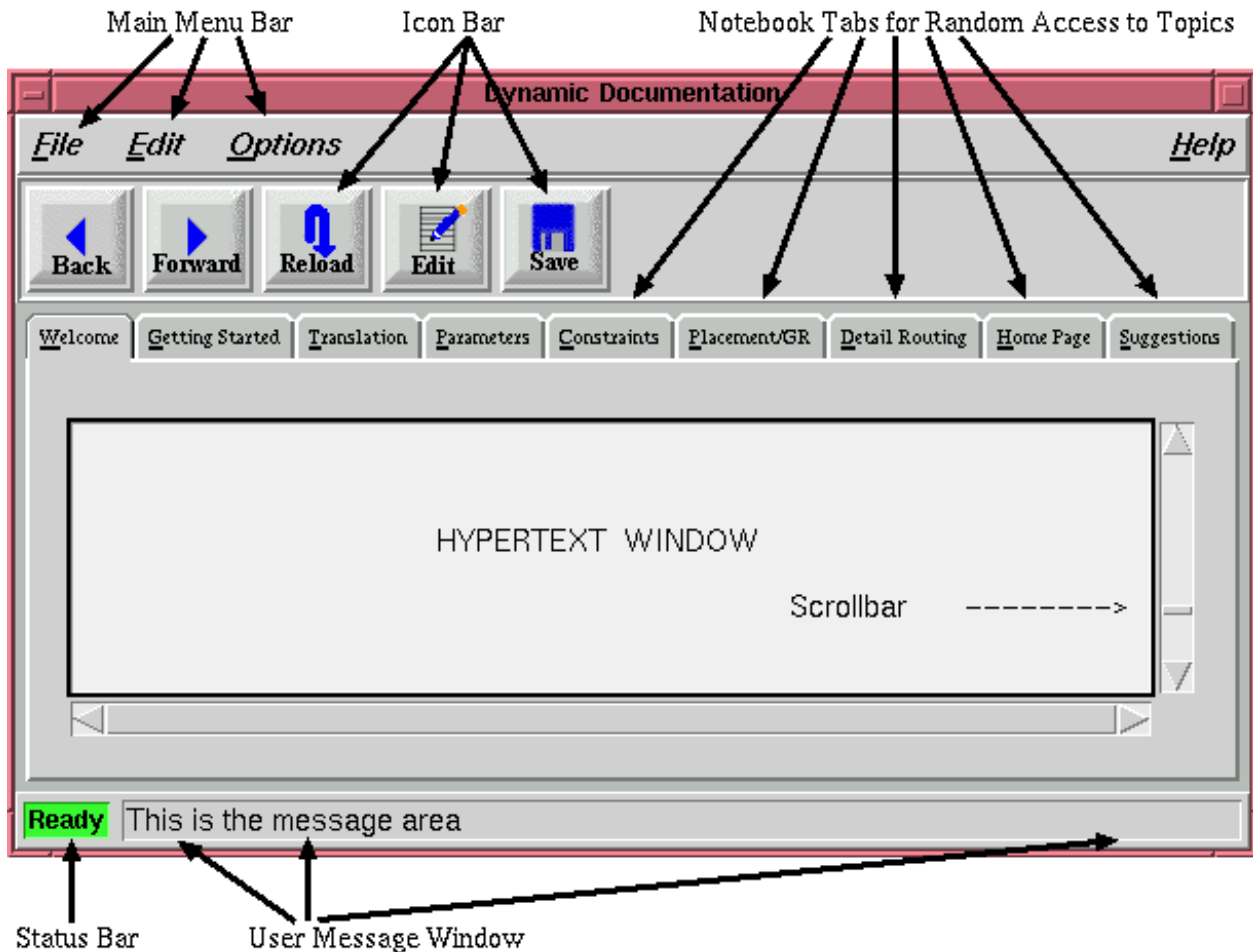
Native Browser Graphical Interface

*This section only applies to **itools** native browser*

EZCAD graphical user interface is composed of seven major parts namely, the main menu bar, the icon bar, the hypertext notebook, the hypertext display window, the hypertext scrollbar, the current status bar, and the user message area as shown in the picture below

.

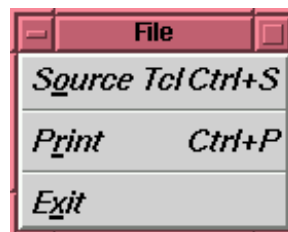
Ittools Documentation



The Main Menu Bar

The main menu bar contains pull down submenus which allow the user to execute commands. There are currently four pulldown menus in the menubar: **File**, **Edit**, **Options**, and **Help**.

Here we unroll the **File** pulldown menu:

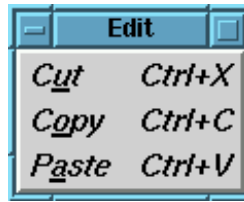


The **Source Tcl** command is only available in *Developer* mode. It allows the developer to source a Tcl file without leaving EZ CAD. This is a productivity enhancement that allows the developer to write the Tk procedures incrementally without having to exit the entire program.

The **Print** command is available in all user modes and prints the contents of the currently displayed hypertext to a PostScript printer.

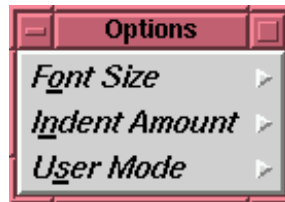
The **Exit** command terminates the EZ CAD program.

Now we present the **Edit** pulldown menu:



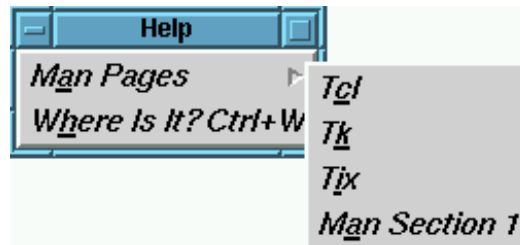
The **Cut**, **Copy**, and **Paste** commands under the **Edit** menu are only available in the *Developer* user mode during a hypertext edit. Otherwise, these commands are inactive.

Now we unroll the **Options** pulldown menu:



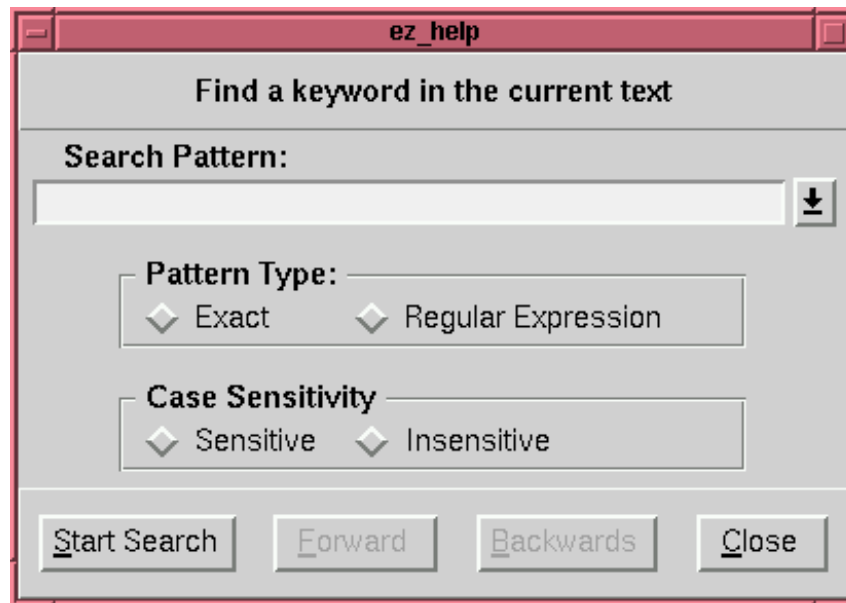
The **Font Size** command changes the font size of the hypertext. The cascade menu which is displayed offers *small*, *medium*, and *large* fonts. The **Indent Amount** command modifies the indentation distance for the display of ordered lists. Here too, the available choices are *small*, *medium*, and *large*. The **User Mode** command controls the current user knowledge level. The supported modes are **Novice**, **Intermediate**, **Expert**, and **Developer**. Each successive knowledge level allows the user access to more of the tools' capabilities.

The final submenu, the **Help** pulldown menu:



The **Man Pages** command displays the Unix **man**ual pages from *Tcl*, *Tk*, *Tix*, and section 1 of the Unix system. A file selection dialog is presented and allows the user to determine the man page to display. By changing the directory in the file selection dialog, this command can display any man page on the system. Compressed man pages are supported.

The **Where Is It?** command performs a search of the hypertext or editing window. The following window is presented to the user:



The search string is entered on text entry area. A carriage command should be entered in order to add the pattern to the history list. The pattern entered may be either a regular expression or an exact search using glob style matching. In addition, the search may be made case sensitive or insensitive. The search is started by depressing the "**Start Search**" button. The search may continue either forward or backward by depressing the appropriate buttons. If no match is found you will be warned on the user message line. The **Close** button removes the search engine from the screen.

The Icon Bar



The Icon Bar presents the often used commands to the user. The hypertext traversal buttons – back and forward are prominently displayed in the icon bar. Back revisits the previously visited hypertext links. If the user, traverses backwards, they may return to their previous location by depressing the forward icon. The **Reload** icon is used to redisplay the current hypertext document. If the user was viewing or editing the source of the hypertext, source is replaced with the rendering of it. The **Home** icon redisplay the entry point HTML for the current topic denoted by the raised tab. It allows the easy traversal to the top of the current topic. The **Search** mode renders the search hypertext system in order to find a topic. The **View** or **Edit** icon allows the user to view and edit the hypertext source respectively. The **Edit** icon is available only in the "*Developer*" user mode; otherwise, the **View** icon is presented. The **Save** icon is only available in "*Developer*" mode and saves the current copy of the hypertext source. The original hypertext source is renamed to *filename.orig*. The **Stop** icon stops the rendering of the current HTML.

The Hypertext Display

The hypertext display consists of the **Tix** notebook, the main hypertext window, and its accompanying scrollbar. The key feature to the hypertext display is the ability to access the documentation both sequentially and randomly. The tabs on the notebook choose the general topic of interest whereas the scrollbar permits sequential access of the current document. It is this power to access documentation both sequentially and randomly that makes **Dynamic Documentation** a leader in usability engineering.

User Feedback

The current status bar, and the user message area are designed to give the user feedback during interaction session. The current status bar has two messages: **Ready** and **Busy**. The **busy** message indicates that the program is executing and that the user should wait before entering a new command. It should be noted that the system periodically displays the incompletely drawn hypertext in order to inform the user of its progress. The **ready** message indicates that EZ CAD is waiting for its next command.

The user message area displays messages from the execution of the hypertext. Normal messages are displayed with a foreground color of black, warning messages are displayed with a foreground color of blue, and error messages are displayed in red. The message area is *persistent*, that is, the message remains displayed until another message occurs to overwrite it.

Summary

We have presented a revolutionary new active hypertext documentation system and graphical user interface. This system understands different user knowledge levels, actively integrates the program into the documentation, and incorporates the latest research in usability engineering.

[\(Back\)](#) [\(Next\)](#) [Technical Discussion](#)

Recommended Reading List

Tcl/Tk

● *Practical Programming in Tcl and Tk* by Brent B. Welch, Prentice Hall Publishing, 650 pages, July 1997, ISBN: 0136168302

● *Tcl and the Tk Toolkit* by John K. Ousterhout, Addison–Wesley Publishing Company, 458 pages, May 1994, ISBN: 020163337X

● *Exploring Expect* by Don Libes, O'Reilly and Associates, Inc., 602 pages, December 1994, ISBN: 1565920902

ICAD Benchmarks

The following results were achieved from the internal development version of **itools**

These results are for the MCNC suite of standard placement benchmarks, which can be obtained from the [University of Washington](#). Each result is the best of a number of runs of **itools**, for a particular setup of the program. The released version will use the same program setup.

Benchmark	Number of Cells	Wire Length TimberWolf Version 7 *	Wire Length itools Version 1.4.0
Golem3	99932	90.39	79.9
Avqlarge	25114	5.65	4.78
Avqsmall	21854	5.08	4.48
Industry 3	15059	41.53	39.6
Industry 2	12142	13.30	11.4
Biomed	6417	3.22	2.90
Primary 2	2907	3.53	3.37
Struct **	1888	N/A	0.272
Primary 1	752	0.83	0.799

* TimberWolf Version 7 produces the best results previously published.

** Struct was run without the normal pad constraints used in the benchmark

Autoloading Function

Copyright (c) 1996–2005. InternetCAD, Inc. All right reserved.

One of the most useful functions performed by the `unknown` procedure is *autoloading*. Autoloading allows you to write collections of Tcl procedures and place them in script files in library directories. You can then use these procedures in your Tcl applications without having to `source` the files that define them explicitly. You simply invoke the procedures. The first time that you invoke a library procedure it won't exist, so `unknown` will be called. `unknown` will find the file that defines the procedure, `source` the file to define the procedure, and then reinvoke the original command. The next time the procedure is invoked it will exist, so the autoloading mechanism won't be triggered.

Autoloading provides two benefits. First, it makes it easy to build large libraries of useful procedures and use them in Tcl scripts. You need not know exactly which files to `source` to define which procedures, since the autoloader takes care of that for you. The second benefit of autoloading is efficiency. Without autoloading an application must `source` all of its scripts when it starts. Autoloading allows an application to start up without loading any script files at all; the files will be loaded later when their procedures are needed, and some files may never be loaded at all. Thus autoloading reduces the startup time and saves memory.

Ousterhout, John K., "Tcl and the Tk Toolkit", pp. 137–138.

Convenience Definitions

The table below lists the current predefined variables in the EZ system.

VARIABLE	FUNCTION	TCL VARIABLE
<i>ICUSER</i>	Current User type: {Novice,Intermediate,Expert,Developer}	<i>ICvarsG(S_user)</i>
<i>ICMODE</i>	Design or Tutorial Name	<i>ICvarsG(S_mode)</i>
<i>ICDESIGN</i>	Design Name	<i>ICvarsG(S_design)</i>
<i>ICFLOW</i>	Execution Flow: {flow.noflrplan,flow.route,...}	<i>ICvarsG(S_flow)</i>

Design Modes

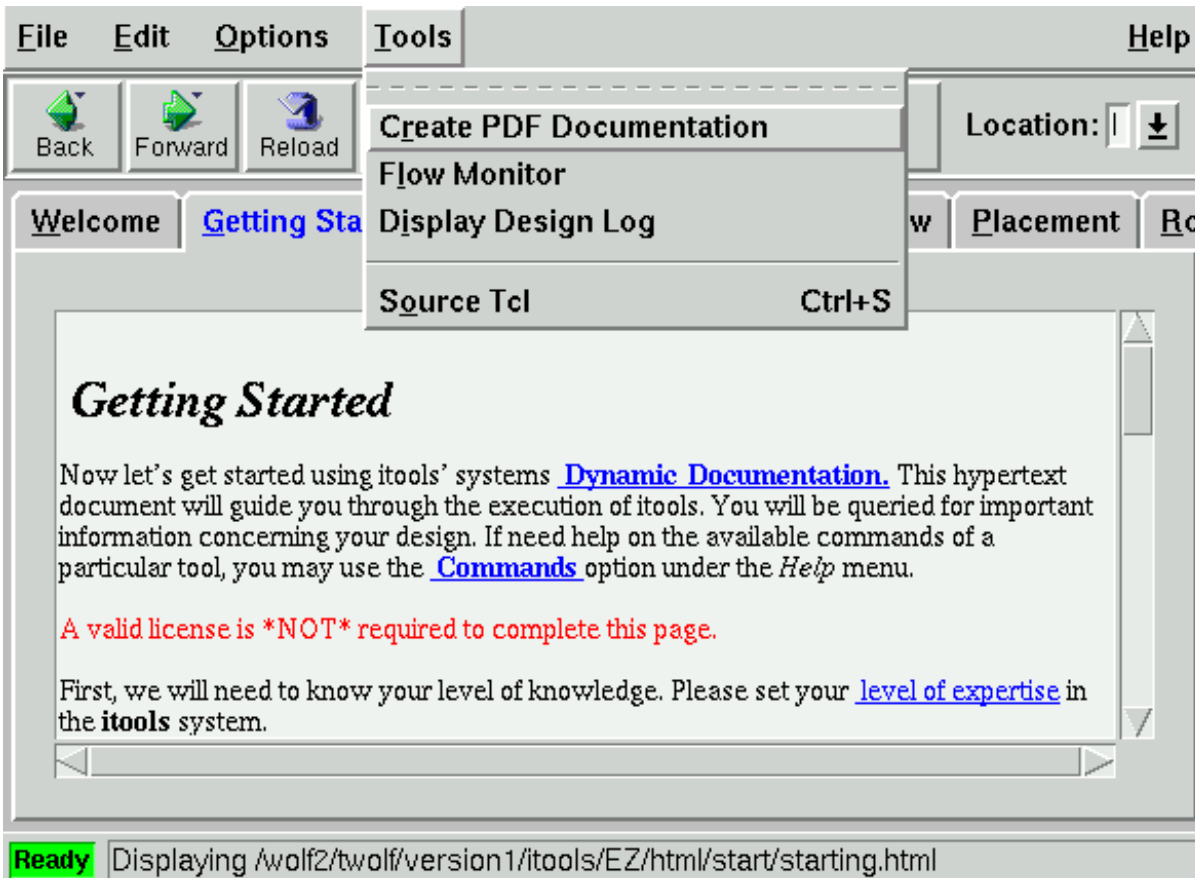
Copyright (c) 1996–99. InternetCAD, Inc. All right reserved.

EZ CAD allows users to switch modes for easy learning. The normal or default mode is the design mode which allows you to execute **itools** on your design. The other modes are tutorial modes which show you how to run **itools** for various situations.

Generating PDF / Postscript

Copyright (c) 1998–2005. InternetCad, Inc. All rights reserved.

Now the EZ documentation system can generate a PDF or Postscript document from the Dynamic Documentation system. You will find the document generation tool under the **Tools** menu.



You will be presented with the graphical user interface as shown below. You have the option of generating either PDF or PostScript output. PDF is the default and recommended output format. In addition, you may select the chapter which you wish to output. It is recommended that you output all chapters so that all of the links will work properly. The user also has the choice to set the *Level of Expertise* and *Design Mode*. Since the EZ documentation system is dynamic, it will be rendered in one of the four [expertise levels](#). Beginners should normally start with the Novice level. The documentation is also subject to the design mode. While you can capture the documentation in design mode, it might be useful to set the design mode to one of the tutorials that best suits your needs in order to display useful examples and settings. You should be aware that the static nature of PDF / Postscript documents do have some [disadvantages](#).

Output Format: —

Format: ☒ PDF ☐ PostScript

Chapter Selection: —

Chapters: ☒ all ☐ select

<input type="checkbox"/> Welcome	<input type="checkbox"/> Getting Started
<input type="checkbox"/> Inputs/Translate	<input type="checkbox"/> Syntax/Flow
<input type="checkbox"/> Placement	<input type="checkbox"/> Routing
<input type="checkbox"/> Post-Routing	<input type="checkbox"/> Customizing
<input type="checkbox"/> Tips/Hints	

User Model: —

Level of Expertise

Design Mode: —

iTools Mode

Notes: —

This operation will pop up Tk widgets in order to capture them. Please do not move the mouse or move/create any windows as this will influence the capture. This window will remain visible until the entire process is complete. While it is possible to generate documentation in design mode, it is recommended to choose a tutorial mode which corresponds to your design style so example data is present.

How it works

The PDF/Postscript documents are generated using the open source program **htmldoc**. We include this program in the distribution as a convenience but it easy to compile your own version if desired as we will [show below](#).

First the itools HTML renderer renders the HTML of interest at a given expertise and design mode into a temporary directory found at `~/itools/htmldump`. All pictures are linked rather than copied to save space. It is at this time that the renderer will popup various Tk windows and capture them as GIFs to be included in the document. It will also resize any Tk window which is larger than 650 pixels across so that it will fit in width of a PDF document. In addition, any pictures which result in more than 256 colors will be color reduced so they can be represented as a GIF in order to keep things simple. Finally, htmldoc will be called with the following arguments to convert the document:

- `htmldoc -f itools.pdf --left 36 -datadir $ICDIR/htmldoc --webpage --linkcolor file1.html file2.html ...`

The details of the entire process can be found in the file `EZ/tcl/dump.html` under the itools root directory.

How to compile htmldoc

We use htmldoc version 1-8.24 which can be found at <http://www.htmldoc.org> without any modifications. We use the following configurations option in order to make a version which will work on as many machines as possible:

- `./configure --enable-ssl=no --enable-localjpeg --enable-localzlib --enable-localpng`
- `make`

References

1. E. H. L. Aarts, F. M. J. de Bont, J. H. A. Habers, and P. J. M. van Laarhoven, "Parallel Implementation of the Statistical Cooling Algorithm," *Integration* , Vol 4. No. 3, 1986, pp. 209–238.
2. S. K. Abd–El–Hafiz, V. R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proc. IEEE Conference of Software Maintenance* , 1991, pp. 212–219.
3. W. Allen, D. Rosenthan, and K. Fiduk, "The MCC CAD Framework Methodology Management System," *Proc. Design Automation Conference* , 1991, pp. 694–698.
4. K. Aoshima and E. S. Kuh, "Multichannel Optimization in Gate Array LSI Layout," *Proc. ISCAS* , 1983, pp. 1005–1008.
5. T. Asano, and S. Sato, "Long Path Enumeration Algorithms for Timing Verification on Large Digital Systems," in *Graph Theory with Applications to Algorithms and Computer Sciences* , Wiley and Sons, 1985, pp. 25–35.
6. P. Banerjee and M. Jones, "A Parallel Simulated Annealing Algorithm for Standard Cell Placement on a Hypercube Computer," *Proc. of Intl. Conf. on Computer–Aided Design* , 1986, pp. 34–37.
7. J. Benkoski and R. B. Stewart, "TATOO: An Industrial Timing Analyzer with False Path Elimination and Test Pattern Generation," *Proc. European Conf. on Design Automation* , 1991, pp. 256–260.
8. J. Benkoski and A. Strojwas, "The Role of Timing Verification in Layout Synthesis," *Proc. Design Automation Conference* , June, 1991, pp. 612–619.
9. J. Bentley, *Writing Efficient Programs* , (Englewood Cliffs : Pentice–Hall, 1982).
10. J. Bentley, *Programming Pearls* , (Reading : Addison–Wesley Publishing Company, 1982).
11. J. Bentley, *Programming Pearls* , (Reading: Addison–Wesley Publishing Company, 1988).
12. M. W. Bern, "Two Probabilistic Results on Rectilinear Steiner Trees," *Algorithmica* 3, 1988, pp. 191–204.
13. M. W. Bern and M. De Carvalho, "A Greedy Heuristic for the Rectilinear Steiner Tree Problem," *Report No UCB/CSD 87/306* , Computer Science Division, University of California, Berkeley, 1986.
14. J. Blair, S. Kapoor, E. Lloyd, and K. Supowit, "Minimizing Channel Density in Standard Cell Layout," *Algorithmica* 2 1987, pp. 267–282.
15. J. Blanks, "Near–optimal Placement using a Quadratic Objective Function," *Proc. 21st Design Automation Conference* , 1985, pp. 609–615.
16. M. A. Breuer, "Min–Cut Placement", *Journal of Design Automation and Fault Tolerant Computing* , Vol. 1, No. 4 , 1977, pp. 343–362.
17. R. Brouwer and P. Banerjee, "A Parallel Hierarchical Global Router," *IEEE Trans. on Computer–Aided Design* , Vol. CAD–2, No. 4. pp. 223–234.
18. T. Bull, "An Introduction to the WSL Program Transformer," *Proc. IEEE Conference on Software Maintenance* , 1990, pp. 242–250.
19. J. L. Burns and A. R. Newton, "SPARCS: A New Constraint–Based IC Symbolic Layout Spacer," *Proc. of CICC* , 1986, pp. 534–539.
20. J. Burns, A. Casotto, M. Igusa, F. Marron, F. Marron, F. Romeo, A. Sangiovanni–Vincentelli, C. Sechen, H. Shin, G. Srinath, and H. Yaghutiel, "Mosaico: An Integrated Macro–Cell Layout System," *Proc. of VLSI '87* , Vancouver, Canada, August, 1987.
21. M. Burstein and R. Pelavin, "Hierarchical Wire Routing," *IEEE Trans. Computer–Aided Design* , Vol. CAD–2, 1983, pp. 223–234.
22. M. Burstein, and R. Pelavin, "Hierarchical Channel Router", *Proc. 20th Design Automation Conference* , June 1983, pp. 591–597.
23. M. Burstein and M. N. Youseff, "Timing Influenced Layout Design," *Proc. 22th Design Automation Conference* , 1985, pp. 124–130.
24. R. C. Carden and C. Cheng, "A Global Router Using an Efficient Approximate Multicommodity Multiterminal Flow Algorithm," *Proc. Design Automation Conference* , 1991, pp. 316–321.
25. R. Carey and M. Bendick, "The Control of a Software Test Process," *Proc. IEEE Conference on Computer Software and Applications* , 1977, pp. 327–334.

26. A. Casotto, F. Romeo, and A. Sangiovanni–Vincentelli, "A Parallel Simulated Annealing Algorithm for the Placement of Macro–Cells." *IEEE Trans. on Computer–Aided Design of ICs and Systems*, Vol. 6, No. 5, 1987, pp. 838–847.
27. A. Casotto, A. R. Newton, and A. Sangiovanni–Vincentelli, "Design Management Based on Design Traces," *Proc. Design Automation Conference* , 1990, pp. 136–141.
28. E. Charbon, E. Malavasi, U. Choudhury, A. Casotto, and A. Sangiovanni–Vincentelli, "A Constraint–Driven Placement Methodology for Analog Integrated Circuits," *Proc. of Custom Integrated Circuits Conference* , 1992, pp. 28.2.1–28.2.4.
29. D. Chen and C. Sechen, "Mickey: A Macro Cell Global Router," *Proc. European Conf. on Design Automation* , 1991, pp. 248–252.
30. C. K. Cheng and E. S. Kuh, "Module Placement Based on Resistive Network Optimization," *IEEE Trans. Computer–Aided Design*, vol. CAD–3, July 1984, pp. 218–225.
31. M. Chi, "An Automatic Rectilinear Partitioning Procedure for Standard Cell." *Proc. 24th Design Automation Conference*, 1987, pp. 50–55.
32. S. Chopra and E. Rosenberg, "Efficient Method for Custom Integrated Circuit Global Routing", *Proc. IEEE Custom Integrated Circuits Conference* , May 1988, paper 11.3.
33. S. Chowdhury, "Optimum Design of Reliable IC Power Networks Having General Graph Topologies," *Proc. 26th Design Automation Conference* , June 1989, pp.787–790.
34. J. K. Chua and Y. C. Lim, "Fast Vicinity–Upgrade Algorithm for Rectilinear Steiner Trees," *Electronic Letters* , Vol 27. No. 13, June 1991, pp. 1139–1140.
35. J. M. Cohn, D. J. Garrod, R. Rutenbar, and L. R. Carley, "KOAN/ANAGRAM II: New Tools for Device–Level Analog Placement and Routing," *IEEE Journal of Solid–State Circuits* , Vol. 26, No. 3, March 1991, pp. 330–342.
36. J. M. Cohn, D. J. Garrod, R. Rutenbar, and L. R. Carley, "Technique for Simultaneous Placement and Routing of Custom Analog Cells in KOAN/ANAGRAM II," *Proc. of IEEE Intl Conf. on Computer–Aided Design*, 1991, pp 394–397.
37. J. P. Cohoon, S. U. Hegde, W. N. Martin, and D. Richards, "Floorplan Design using Distributed Genetic Algorithms," *Proc. IEEE Intl. Conf. on Computer–Aided Design* , 1988, pp. 452–455.
38. J. P. Cohoon and P. L. Heck, "BEAVER: A Computational–Geometry–Based Tool for Switchbox Routing," *IEEE Trans. on CAD* , Vol. 7, No. 6, pp. 684–697.
39. R. Condamoor and I. G. Tollis, "A New Heuristic for Rectilinear Steiner Trees," *Proc. IEEE Intl. Symposium on Circuits and Systems* , 1990, pp. 1676–1677.
40. J. Cong and B. Preas, "A New Algorithm for Standard Cell Global Routing," *Proc. IEEE Intl. Conf. on Computer–Aided Design* , 1988, pp. 176–179.
41. J. Cong, A. Kahng, G. Robins, M. Sarrafzadeh, and C. K. Wong, "Provably Good Performance–Driven Global Routing," *IEEE Trans. on Computer–Aided Design* , Vol. 11, No. 6, 1992, pp. 739–751.
42. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* , (Cambridge: The MIT Press, 1990), pp. 23–41.
43. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* , (Cambridge: The MIT Press, 1990), p. 87.
44. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* , (Cambridge: The MIT Press, 1990), p. 329–355.
45. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* , (Cambridge: The MIT Press, 1990), p. 463–497.
46. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* , (Cambridge: The MIT Press, 1990), p. 932.
47. W. Dai, H. Chen, R. Dutti, M. Jackson, E. Kuh, M. Marek–Sadowska, M. Sato, D. Wang, and X. Xiong, "BEAR: A New Building–Block Layout System," *Proc. Int. Conf. on Computed–Aided Design*, 1987, pp. 34–37.
48. D. N. Deutsch, "A Dogleg Channel Router", *Proc. 13th Design Automation Conference* , June 1976, pp. 425–433.

49. R. Dietz, D. A. Mlynski, "New Model for Global Routing of Gate-Arrays," *Proc. IEEE International Symposium on Circuits and Systems* , May 1987, pp. 35–38.
50. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik* , Vol. 1, 1959, pp. 269–271.
51. E. W. Dijkstra, "Go To Statement Considered Harmful," *Communications of the ACM* , Vol. 11, No. 3, 1968, pp. 147–148.
52. E. W. Dijkstra, "A Constructive Approach to the Problem of Program Correctness," *BIT* , Vol. 8, No. 3, 1968, pp. 174–186.
53. W. Donath, R. Norman, B. Agrawal, S. Bello, S. Han, J. Kurtzberg, P. Lowy, and R. McMillan, "Timing Driven Placement using Complete Path Delays," *Proc. Design Automation Conference* , June 1990, pp. 84–89.
54. S. E. Dreyfus, "An Appraisal of Some Shortest-Path Algorithms," *Operations Research* , Vol. 17, 1969, pp. 395–412.
55. A. E. Dunlop, V. D. Agrawal, D. N. Deutsch, M. F. Jukl, P. Kozak, and M. Wiesel, "Chip Layout Optimization Using Critical Path Weighting," *Proc. 21st Design Automation Conference* , 1984, pp. 133–136.
56. A. E. Dunlop and B. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *IEEE Trans. on Computer-Aided Design* , Vol. 4, No. 1, 1985, pp. 92–98.
57. A. E. Dunlop, G. F. Gross, C. D. Kimble, M. Y. Luong, K. J. Stern, and E. J. Swanson, "Features in LTX2 for Analog Layout," *Proc. of ISCAS* , 1985, pp. 21–23.
58. R. Dutta and M. Marek-Sadowska, "Automatic Sizing of Power/Ground (P/G) Networks in VLSI," *Proc. 26th Design Automation Conference* , June 1989, pp. 783–786.
59. C. Ebeling and O. Zajicek, "Validating VLSI Circuit Layout by Wirelist Comparison," *Digest Intl. Conf. on Computer-Aided Design* , 1983, pp. 172–173.
60. M. Eda, T. Yoshimura, "New Placement and Global Routing Algorithms for Standard Cell Layouts," *Proc. Design Automation Conference* , 1990, pp. 642–645.
61. S. Even, *Graph Algorithms* , (Potomac: Computer Science Press, 1979), pp. 138–142.
62. R. J. Evans, "SMARTsystem: A CASE Development Environment for Existing C Programs," *Proc. IEEE Conference on Software Maintenance* , 1990, p. 256.
63. P. de Forcrand and H. Zimmermann, "Timing-Driven Auto-Placement," *Proc. Int. Conf on Comp. Design* , 1987, pp. 518–521.
64. C. Fiduccia and R. Mattheyses, "A Linear Time Heuristic for Improving Network Partitions," *Proc. 19th Design Automation Conference* , 1982, pp. 175, 181.
65. R. Fiebrich and C. Wang, "Circuit Placement Based on Simulated Annealing on a Massively Parallel Computer," *Proc. Intl. Conf. on Computer Design* , 1987, pp. 78–82.
66. L. R. Ford and D. R. Fulkerson, *Flows in Networks* , (Princeton: Princeton University Press, 1962).
67. J. Frankle and R. Karp, "Circuit Placements and Cost Bounds by Eigenvector Decomposition," *Digest Intl. Conference on Computer-Aided Design* , 1986, pp. 414–417.
68. K. Fukahori, "Computer Simulation of Integrated Circuits in the Presence of Electrothermal Interaction," *IEEE Journal of Solid-state Circuits* , Vol. 11, No. 6, 1976.
69. K. Fukunaga, S. Yamada, H. Stone, and T. Kasai, "Placement of Circuit Modules Using a Graph Space Approach," *Proc. 20th Design Automation Conference* , 1983, pp. 465–471.
70. J. D. Gannon, R. G. Hamlet, and H. D. Mills, "Theory of Modules," *IEEE Trans. on Software Engineering* , Vol. SE-13, No. 7, July 1987, pp. 820–829.
71. T. Gao, P. M. Vidya, and C. L. Liu, "A New Performance Driven Placement Algorithm," *Proc. ICCAD* , November 1991, pp. 44–47.
72. T. Gao, P. M. Vidya, and C. L. Liu, "A Performance Driven Macro-Cell Placement Algorithm," *Proc. Design Automation Conference* , June 1992, pp. 147–152.
73. M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* , (New York: W. H. Freeman and Company, 1979).
74. D. Garrod, R. Rutenbar, L. Carley, "Automatic Layout of Custom Analog Cells in ANAGRAM," *Proc. of ICCAD* , 1988, pp. 544–547.

75. V. R. Gibson and J. A. Senn, "System Structure and Software Maintenance Performance," *CACM* , Vol. 32, No. 3, 1989, pp. 347–358.
76. S. Goto and E. Kuh, "An Approach to the Two–Dimensional Placement Problem in Circuit Layout," *IEEE Trans. on Circuits and Systems*, Vol. 25, No. 4, 1978, p. 208.
77. S. Goto and T. Matsuda, "Partitioning, Assignment and Placement," in *Layout Design and Verification* , edited by T. Ohtsuki, (Amsterdam: North Holland, 1986), pp. 76–82.
78. P. Groeneveld, "On Global Wire Ordering for Macro–Cell Routing." *Proc. 26th Design Automation Conference*, 1989, pp. 155–160.
79. K. Hall, "An r–Dimensional Quadratic Placement Algorithm, *Management Science* , Vol. 17. No. 3, 1970, pp. 219–229.
80. G. T. Hamachi and J. K. Ousterhout, "A Switchbox Router with Obstacle Avoidance," *Proc. Design Automation Conference* , 1984, pp. 173–179.
81. T. Hamada, C. Cheng, and P. Chau, "A Wire Length Estimation Technique Utilizing Neighborhood Density Equations," *Proc. Design Automation Conference* , 1992, pp. 57–61.
82. M. Hanan, "On Steiner's Problem With Rectilinear Distance", *SIAM J. of Applied Mathematics* , Vol. 14, 1966, pp. 255–265.
83. M. Hanan and J. Kurtzberg, *Design Automation of Digital Systems : Theory and Techniques* , edited by M. Breuer, (Englewood Cliffs: Prentice–Hall, Inc., 1972), pp. 214–224.
84. N. Hasan, G. Vijayan, and C. K. Wong, "A Neighborhood Improvement Algorithm for Rectilinear Steiner Trees," *Proc. IEEE Intl Symp. on Circuits and Systems* , 1990.
85. T. Hasegawa, "A New Placement Algorithm Minimizing Path Delays," *Proc. ICCAD* , 1991, pp. 2052–2055.
86. A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures", *Proc. 8th Design Automation Workshop* , 1971, pp. 155–163.
87. G. D. Hachtel and C. R. Morrison, "Linear Complexity Algorithms for Hierarchical Routing," *IEEE Trans. on Computer–Aided Design of Integrated Circuits and Systems* Vol. 8, No. 1, Jan 1989, pp. 64–80.
88. P. Hauge, R. Nair, and E. Yoffa, "Circuit Placement for Predictable Performance." *Proc. Intl. Conf. on Computed–Aided Design*, 1987, pp. 88–91.
89. D. W. Hightower, "A Solution to the Line Routing Problem on a Continuous Plane", *Proc. 6th Design Automation Workshop* , 1969 pp. 1–24.
90. H. Hillner, B. Weis, D. Mlynski, "The Discrete Placement Problem: a Dynamic Programming Approach," *Proc. Intl. Symposium on Circuits and Systems* , 1986, pp. 315–318.
91. R. B. Hitchcock, G. L. Smith, D. D. Cheng, "Timing analysis of computer hardware," *IBM Journal of Research and Development* , Vol 24, No. 1, Jan. 1983, pp. 100–105.
92. J. M. Ho, G. Vijayan and C. K. Wong, "New Algorithms for the Rectilinear STEiner Tree Problem", *IEEE Trans. on Computer–Aided Design* , Vol 9–2, 1990, pp. 185–193.
93. C. A. R. Hoare, "Proof of Correctness of Data Representations," *Acta Inform .*, Vol. 1, 1972, pp. 271–281.
94. M. Horowitz and R. W. Dutton, "Resistance Extraction from Mask Layout Data," *IEEE Trans. on Computer–Aided Design* , Vol. 2. No. 3, July 1983, pp. 145–150.
95. F. K. Hwang, "On Steiner Minmal Trees with Rectilinear Distance," *SIAM J. Applied Mathematics* , Vol. 30(1), 1976, pp. 104–114.
96. F. K. Hwang, "An O(nlogn) Algorithm for Suboptimal Rectilinear Steiner Trees," *J. ACM* , Vol. 26–2, 1979, pp. 177–182.
97. M. Igusa, M. Beardslee, A. Sangiovanni–Vincentelli, "ORCA: A sea–of–gates place and route system", *Proc. Design Automation Conference* , June 1989, pp. 122–127.
98. M. Jackson and E. Kuh, "Performance–Driven Placment of Cell Based IC's," *Proc. Design Automation Conference* , June 1989, pp. 370–375.
99. M. Jackson, E. Kuh, and M. Marek–Sadowska, "Timing–Driven Routing for Building Block Layout," *Proc. of ISCAS* , 1987, pp. 518–519.
100. M. Jackson, A. Srinivasan, and E. Kuh, "A Fast Algorithm for Performance–Driven Placement," *Proc. Int. Conf. on Computed–Aided Design*, 1990, pp. 328–331.

101. D. Jepsen and C. Gelatt, Jr., "Macro Placement by Monte Carlo Annealing." *Proc. Intl. Conf. on Comp. Design*, 1983, pp. 495–498.
102. R. Joobbani and D. Siewiorek, "WEAVER: A Knowledge–Based Routing Expert," *IEEE Design and Test of Computers* , Vol. 3. No. 1, Feb. 1986, pp. 12–23.
103. A. Kahng and G. Robins, "On Performance Bounds for a Class of Rectilinear Steiner Tree Heuristics in Arbitrary Dimension", *IEEE Trans. on Computer–Aided Design* , Vol. 11, No. 11, 1992, pp. 1462–1465.
104. A. Kahng and G. Robins, "A New Class of Iterative Steiner Tree Heuristics with Good Performance", *IEEE Trans. on Computer–Aided Design*, Vol. 11, No. 7, pp. 893–902.
105. R. M. Karp, "Reducibility Among Combinatorial Problems," *Complexity of Computer Computations*, New Rouk: Plenum, 1972.
106. Y. Kashai, "Flow – A Concurrent Methodology Manager," *European Conference on Design Automation* , 1992, pp. 20–24.
107. D. Keller, "A Guide to Natural Naming," *SIGPLAN Notices* , Vol. 25, No. 5, pp. 95–102.
108. B. W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, Vol. 49, No. 2, Feb. 1970, pp. 219–307.
109. B. W. Kernighan and D. M. Ritchie, *The C Programming Language* , (Englewood Cliffs: Prentice Hall, 1988).
110. K. Keutzer, S. Malik, and A. Saldanha, "Is Redundancy Necessary to Reduce Delay?", *IEEE Trans. on CAD* , CAD–10 No. 4, April 1991, pp. 427–435.
111. C. D. Kimble, A. E. Dunlop, G. F. Gross, V. L. Hein, M. Y. Luong, K. J. Stern, and E. J. Swanson, "Autorouted Analog VLSI," *Proc. Custom Integrated Circuits Conference* , 1985, pp. 72–78.
112. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, Vol. 220, May 1983, pp. 671–680.
113. J. Kleinhans, G. Sigl, and F. Johannes, "Gordian: A New Global Optimization / Rectangle Dissection Method for Cell Placement," *Proc. ICCAD* , 1988, pp. 506–509.
114. R. Kling and P. Banerjee, "ESP: A New Standard Cell Placement Package Using Simulated Evolution," *Proc. Design Automation Conference* , June 1987, pp. 60–66.
115. R. Kling and P. Banerjee, "ESP: Placement by Simulated Evolution," *IEEE Trans. on Computer–Aided Design* , Vol. 7, No. 3, March 1989, pp. 245–256.
116. R. Kling and P. Banerjee, "Empirical and Theoretical Studies of the Simulated Evolution Method Applied to Standard Cell Placement," *IEEE Trans. on Computer–Aided Design* , Vol. 10, No. 10, October 1991, pp. 1303–1315.
117. D. E. Knuth, *The Art of Computer Programming, Volume I, Fundamental Algorithms* , (Reading: Addison–Wesley, 1968).
118. D. E. Knuth, "Structure Programming with GO TO Statements," *Computing Surveys* , Vol. 6, No. 4, 1974, pp. 261–301.
119. D. E. Knuth, *Literate Programming* , (Leland: Center for the Study of Language and Information, 1992), pp. 99–358.
120. H. Koh, C. Sequin, and P. Gray, "Automatic Synthesis of Operational Amplifiers Based on Analytic Circuit Models," *Proc. ICCAD* , 1987, pp. 502–505.
121. H. Koh, C. Sequin, and P. Gray, "Automatic Layout Generation for CMOS Operational Amplifiers," *Proc. ICCAD* , 1988, pp. 548–551.
122. S. A. Kravitz and R. R. Rutenbar, "Placement by Simulated Annealing on a Multiprocessor," *IEEE Trans. on Computer–Aided Design* , CAD–6, No. 4, pp. 534–549, 1987.
123. J. Lam and J. M. Delosme, "An Efficient Simulated Annealing Schedule: Derivation", Yale University Technical Report 8816, Department of Electrical Engineering, Yale University, New Haven, CT, 1988.
124. J. Lam and J. M. Delosme, "An Efficient Simulated Annealing Schedule: Implementation and Evaluation", Yale University Technical Report 8817, Department of Electrical Engineering, Yale University, New Haven, CT, 1988.
125. J. Lam and J. M. Delosme, "An Efficient Simulated Annealing Schedule." Yale University Technical Report 8818, Department of Electrical Engineering, New Haven, CT, 1988.

126. J. Lam and J. M. Delosme, "Performance of a New Annealing Schedule." *Proc. 25th Design Automation Conf.*, 1988, pp. 306–311.
127. U. Lauther, "A Min–Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation," *Proc. 16th Design Automation Conference* , 1979, pp. 1–10.
128. E. Lawler, *Combinatorial Optimization: Networks and Matroids* , (New York: Holt, Rinehart, and Winston, 1976), pp. 98–106.
129. C. Lee, "An Algorithm for Path Connections and its Applications", *IRE Trans. on Electronic Computers* , VEC–10, Sept. 1961, pp. 346–365.
130. K. W. Lee, "Global Routing of Row–Based Integrated Circuits," Ph.D. thesis Yale University, May 1990, p. 109.
131. K. W. Lee and C. Sechen, "A New Global Router for Row–Based Layout," *Proc. IEEE Intl. Conf. on Computer–Aided Design* , 1988, pp. 180–183.
132. K. W. Lee and C. Sechen, "A Global Router for Sea–of–Gates Circuits," *Proc. European Conf. on Design Automation* , 1991, pp. 242–247.
133. J. H. Lee, N. K. Bose and F. K. Hwang, "Use of Steiner's Problem in Sub–Optimal Routing in Rectilinear Metric," *IEEE Trans. on Circuits and Systems* , CAS–23, 1976, pp. 470–476.
134. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout* , (Chichester: John Wiley & Sons, 1990), pp. 31–45.
135. S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software* , May 1986, pp. 41–49.
136. H. Leung and L. White, "A Study of Integration Testing and Software Regression at the Integration Level," *Proc. IEEE Conference on Software Maintenance* , 1990, pp. 290–301.
137. Y. Z. Liao and C. K. Wong, "An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints," *IEEE Trans. on Computer–Aided Design of Integrated Circuits* , Vol. 2, No. 2, pp. 62–69, 1983.
138. J. T. Li and M. Marek–Sadowska, "Global Routing for Gate Arrays," *IEEE Trans. on Computer–Aided Design* , Vol. 3, No. 4, October, 1984, pp. 298–307.
139. R. Libeskind–Hadas and C. L. Liu, "Solutions to the Module Orientation and Rotation Problems by Neural Computation Networks," *Proc. Design Automation Conference* , 1989, pp. 400–405.
140. R. Lin and E. Shragowitz, "Fuzzy Logic Approach to Placement Problem," *Proc. Design Automation Conference* , 1992, pp. 153–158.
141. B. W. Lindsay and B. T. Preas, "Design Rule Checking and Analysis of IC Mask Designs," *Proc. 13th Design Automation Conference* , 1976, pp. 301–308.
142. M. Lorenzetti, "The Effect of Channel Router Algorithms on Chip Yield", paper 4.2 Vol. 1, *Proc. International Workshop on Layout Synthesis* , May 8–11, 1990, MCNC Research Triangle Park, N.C.
143. W. Luk, "A Fast Physical Constraint Generator for Timing Driven Layout," *Proc. Design Automation Conference* , June 1991, pp. 626–631.
144. M. Marek–Sadowska, "Global Router for Gate Array," *Proc. IEEE Int. Conf. on Computer Design* , 1984, pp. 332–337.
145. M. Marek–Sadowska and S. Lin, "Timing Driven Placement." *Proc. Int. Conf. on Computer–Aided Design*, 1989, pp. 94–97.
146. S. Mayrhofer and U. Lauther, "Congestion–Driven Placement Using a New Multi–Partitioning Heuristic," *Proc. Int. Conf. on Computer–Aided Design*, 1990, pp. 332–335.
147. P. C. McGeer and R. K. Brayton, "Efficient Algorithms for Computing the Longest Viable Path in a Combinatorial Network," *Proc. 26th Design Automation Conference* , 1989, pp. 561–567.
148. C. Mead and L. Conway, *Introduction to VLSI Systems* , (Reading: Addison–Wesley, 1980).
149. C. Mead and L. Conway, *Introduction to VLSI Systems* , (Reading: Addison–Wesley, 1980), p. 52.
150. G. Meixner and U. Lauther, "A New Global Router Based on a Flow Model and Linear Assignment," *Proc. International Conference on Computer–Aided Design* , 1990, pp. 44–47.
151. N. Metropolis, A. Rosenbluth, and M. Rosenbluth, A. Teller, and E. Teller, *Journal of Chemical Physics* , Vol. 21, 1953, p. 1087.

152. K. Mikami and K. Tabuchi, "A Computer Program for Optimal Routing of Printed Circuit Connectors," *IFIPS Proc.* , Vol H47 pp. 1475–1478.
153. D. Mitra, R. Romeo, and A. Sangiovanni–Vincentelli, "Convergence and Finite–Time Behavior of Simulated Annealing," *Advances in Applied Probability* , Vol. 18. No. 3, pp. 747–771, 1986.
154. D. A. Mlynski and C. Sung, Layout Design and Verification, T. Ohtsuki, editor, (Amsterdam: Elsevier Science Publishers B. V., 1986), pp. 219.
155. M. Mogaki, C. Miura, and H. Terai, "Algorithm for Block Placement with Size Optimization Technique by the Linear Programming Approach," *Proc. Design Automation Conference* , 1987, pp. 80–83.
156. E. F. Moore, "The Shortest Path through a Maze," *Annals of the Harvard Computation Laboratory* , Vol. 30, Pt. II, 1959, pp. 185–292.
157. J. T. Mowchenko, C. S. Ma, "New Global Routing Algorithm for Standard Cell ICs," *Proc. Intl. Symposium on Circuits and Systems* , May 1987, pp. 27–30.
158. G. J. Myers, *Software Reliability: Principles and Practices* , (New York: Wiley–Interscience, 1976), p. 130.
159. G. J. Myers, *Software Reliability: Principles and Practices* , (New York: Wiley–Interscience, 1976), p. 169.
160. G. J. Myers, *Software Reliability: Principles and Practices* , (New York: Wiley–Interscience, 1976), p. 190.
161. G. J. Myers, *Software Reliability: Principles and Practices* , (New York: Wiley–Interscience, 1976), pp. 169–195.
162. R. Nair, "A Simple Yet Effective Technique for Global Wiring," *IEEE Trans. Computer–Aided Design* , Vol. CAD–6, No. 2, March 1987, pp. 165–172.
163. A. Ng., P. Raghavan, and C. Thompson, "Experimental Results for a Linear Program Global Router," *Computers and Artificial Intelligence* , 1987.
164. T. Ohtsuki, editor, Layout Design and Verification, (North–Holland: Elsevier Science Publishers B. V., 1986), pp. 199–235.
165. H. Onodera, Y. Taniguchi, and K. Tamaru, "Branch–and–Bound Placement for Building Block Layout," *Proc. Design Automation Conference* , 1991, pp. 433–439.
166. J. K. Ousterhout, "Corner Stitching: A Data–Structure Technique for VLSI Layout Tools," *IEEE Trans. Computer–Aided Design* , Vol. CAD–3 pp. 87–100, Jan. 1984.
167. T. Parng and R. Tsay, "New Approach to Sea–of–gates Global Routing," *Proc. IEEE International Conference on Computer–Aided Design* , Nov. 1989, pp. 52–55.
168. M. Pedram and B. Preas, "Interconnection Length Estimation of Optimized Standard Cell Layouts," *Proc. Intl. Conf. on Computer–Aided Design* , Oct. 1989, pp. 100–108.
169. S. Prasitjutrakul and W. J. Kubitz, "A Timing–driven Global Router for Custom Chip Design", *Proc. Intl. Conf. on Computer–Aided Design* , 1990, pp. 48–51.
170. B.T. Preas, and M. J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems* , (Menlo Park: The Benjamin/Cummings Publishing Company, Inc., 1988), p. 5.
171. B.T. Preas, and M. J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems* , (Menlo Park: The Benjamin/Cummings Publishing Company, Inc., 1988), p. 264–265.
172. B.T. Preas, and M. J. Lorenzetti, editors, *Physical Design Automation of VLSI Systems* , (Menlo Park: The Benjamin/Cummings Publishing Company, Inc., 1988), pp. 347–407.
173. W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical Recipes in C – The Art of Scientific Computing* , (New York: Cambridge University Press, 1988), pp. 517–558.
174. R. Putatunda, D. Smith, M. Stebnisky, C. Puschak, and P. Patent, "VITAL: Fully Automatic Placement Strategies for Very Large Semicustom Designs." *Proc. IEEE International Conference on Computer Design: VLSI in Computers & Processors* 1988, pp. 434–439.
175. P. Raghavan, C. D. Thompson, "Multiterminal Global Routing: a Deterministic Approximation Scheme," *Algorithmica* , Vol. 6, No. 1, 1991, pp. 73–82.
176. K. Ramachandran, R. R. Cordell, D. F. Daly, D. N. Deutsch, and A. F. Kwan, "SYMCELL – A Symbolic Standard Cell System", *IEEE Journal of Solid State Circuits*, Vol. 26, No. 3, March 1991.
177. K. Ramachandran, D. G. Boyer, and R. R. Cordell, "SYMCELL II – A Second–Generation Symbolic Standard Cell System", *Proc. International Workshop on Layout Synthesis* , May 18–21, 1992. MCNC Research Triangle Park, N.C. , pp. 105–109.

178. C. P. RaviKumar and L.M. Patnaik, "Parallel Placement by Simulated Annealing," *Proc. Int. Conf on Comp. Design*, 1987. pp. 91–94.
179. J. Reed, A. Sangiovanni–Vincentelli, and M. Santomauro, "A New Symbolic Channel Router: YACR2", *IEEE Trans. on Computer–Aided Design* , Vol. CAD–4, No. 3, July 1985, pp. 208–219.
180. E. Reingold and K. Supowit, "Hierarchy–Driven Amalgamation of Standard and Macro Cells," *IEEE Trans. on CAD* , Vol. CAD–3, No. 1, Jan 1984, pp. 3–11.
181. D. Richards, "Fast Heuristic Algorithms for Rectilinear Steiner Trees," *Algorithmica* 4, 1989, pp. 191–207.
182. R. L. Rivest and C. M. Fiduccia, "A Greedy Channel Router," *Proc. 19th. Design Automation Conference* , June 1982, pp. 418–424.
183. K. Roberts, G. Fredericks, D. Skinner, D. Layman, and D. Harris, "Automatic Layout in the Highland System," *Proc. IEEE Intl. Conf on Computer–Aided Design* , 1984, pp. 224–226.
184. C. D. Rogers, J. B. Rosenberg, and S. W. Daniel, "MCNC's Vertically Integrated Symbolic Design System," *Proc. 22nd. Design Automation Conference* , June 1985, pp. 62–68.
185. J. Rose, "LocusRoute: A Parallel Global Router for Standard Cells," *Proc. Design Automation Conference* , 1988, pp. 189–195.
186. J. Rose and S. Brown, "Flexibility of Interconnection Structures for Field–Programmable Gate Arrays," *IEEE Journal of Solid–State Circuits* , Vol. 26, No. 3, March 1991, pp. 277–282.
187. J. Rose, R. Francis, D. Lewis, and P. Chow, "Architecture of Field–Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency," *IEEE Journal of Solid–State Circuits* , Vol. 25, No. 5, October 1990, pp. 1217–1225.
188. A. Saldanha, R. K. Brayton, and A. Sangiovanni–Vincentelli, "Circuit Structure Relations to Redundancy and Delay: the KMS Algorithm Revisted," *Proc. 29th Design Automation Conference* , 1992, pp. 245–248.
189. J. Sametiger, "A Tool for the Maintenance of C++ Programs," *Proc. IEEE Conference on Software Maintenance* , 1990, pp. 54–59.
190. S. Sahni and T. Gonzales, "P–complete approximation problem," *J. ACM*, vol. 23, no. 3, July 1976, pp. 555–565.
191. S. Sastry and A. Parker, "The Complexity of Two–Dimensional Compaction of VLSI Layouts," *Proc. International Conference on Circuits and Computers* , New York, 1982, pp. 402–406.
192. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, Vol. 5, No. 2, 1986, pp. 79–109.
193. A. T. Schreiner and H. G. Friedman, *Introduction to Compiler Construction with UNIX* , (Englewood Cliffs : Prentice Hall, 1985), pp. 21–81.
194. D. G. Schweikert and B. W. Kernighan, "A Proper Model for the Partitioning of Electrical Circuits," *Proc. 9th Annual Design Automation Workshop*, 1972, pp. 57–62.
195. C. Sechen, "Chip–Planning, Placement, and Global Routing of Macro/Custom Cell Integrated Circuits Using Simulated Annealing." *Proc. 25th Design Automation Conference*, 1988, pp. 73–80.
196. C. Sechen, *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer Academic Publishers, 1988.
197. C. Sechen and D. Chen, "An Improved Objective Function for Mincut Circuit Partitioning." *Proc. IEEE International Conference on Computer–Aided Design*, 1988, pp. 502–505.
198. C. Sechen and A. Sangiovanni–Vincentelli, "The TimberWolf Placement and Routing Package," *Proc. 1984 Custom Integrated Circuits Conference*, Rochester, New York, May 1984.
199. C. Sechen and A. Sangiovanni–Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE J. of Solid–State Circuits* , Vol. 20 No. 2, 1985, pp. 432–439.
200. C. Sechen, D. Braun, and A. Sangiovanni–Vincentelli. "ThunderBird: A Complete Standard Cell Layout Package." *IEEE J. of Solid–State Circuits* Vol. 23 No. 2, 1985, pp. 410–420.
201. C. Sechen and K. W. Lee, "An Improved Simulated Annealing Algorithm for Row–Based Placement," *Proc. of ICCAD*, 1987, pp. 478–481.
202. M. Servit, "Heuristic Algorithms for Rectilinear Steiner Trees," *Digital Process* , Vol. 7, No. 1, 1981, pp. 21–31.

203. L. Sha and R. Dutton, "An Analytical Algorithm For Placement of Arbitrarily Sized Rectangular Blocks," *Proc. 22nd Design Automation Conference* , 1985, pp. 602–608.
204. L. Sha and T. Blank, "ATLAS – A Technique for Layout using Analytic Shapes," *Proc. Intl. Conf. on Computed–Aided Design*, 1987, pp. 84–87.
205. K. Shahookar and P. Mazumder, "A Genetic Approach to Standard Cell Placement Using Meta–Genetic Parameter Optimization," *IEEE Trans. on Computer–Aided Design* , Vol. 9, No. 5, May 1990, pp. 500–513.
206. N. Shahmehri, M. Kamkar, and P. Fritzon, "Semi–automatic Bug Localization in Software Maintenance," *Proc. IEEE Conference of Software Maintenance* , 1990, pp. 30–36.
207. H. Shin and A. Sangiovanni–Vincentelli, "A Detailed Router Based on Incremental Routing Modifications: Mighty," *IEEE Trans. Computer–Aided Design* Vol. 6, No. 6, 1987, pp. 942–955.
208. E. Shragowitz, J. Lee, and S. Sahni, "Placer–Router for 'Sea of Gates' Design Style," *Proc. International Conference on Computer Design*, October 1987, pp. 330–335.
209. P. Siarry, L. Bergonzi, and G. Dreyfus, "Thermodynamic Optimization of Block Placement,," *IEEE Trans. on Computer–Aided Design of ICs and Systems*, Vol. 6, No. 2, 1987, pp. 211–221.
210. G. Sigl, K. Doll, and F. M. Johannes, "Analytical Placement: A Linear or a Quadratic Objective Function?" *Proc. Design Automation Conference* , pp. 427–432.
211. A. Srinivasan, "An Algorithm for Performance–Driven Initial Placement of Small–Cell ICs," *Proc. Design Automation Conference* , June 1991, 636–639.
212. A. Srinivasan, K. Chaudhary, and E. Kuh, "RITUAL: A Performance Driven Placment Algorithm for Small Cell ICs," *Proc. Int. Conf. on Computed–Aided Design*, 1991, pp. 48–51.
213. L. Steinberg, "The Backboard Wiring Problem: A Placement Algorithm," *SIAM Review* , Vol. 3, No. 1, 1961, pp. 37–50.
214. P. Suaris and G. Kedem, "Quadrisection: A New Approach to Standard Cell Layout," *Proc. Intl. Conf. on Computed–Aided Design*, 1987, pp. 474–477.
215. P. Suaris and G. Kedem, "An Algorithm for Quadrisection and its Application to Standard Cell Placement," *IEEE Trans. on CAS* , Vol. 35, No. 3, March 1988, pp. 294–303.
216. P. Suaris and G. Kedem, "A Quadrisection–based Combined Place and Route Scheme for Standard Cells," *IEEE Trans. on CAD* , Vol. 8, No. 3, March 1989, pp. 234–244.
217. S. Sutanthavibul, and E. Shragowitz, "An Adaptive Timing–Driven Layout for High Speed VLSI," *Proc. Design Automation* , June 1990, pp. 90–95.
218. S. Sutanthavibul, and E. Shragowitz, "Dynamic Prediction of Critical Paths and Nets for Constructive Timing–Driven Placement," *Proc. Design Automation* , June 1991, pp. 632–635.
219. W. Swartz, H. Khan, D. A. Thomas, C. R. Giuffre, M. deWit, T. Pavey, C. McIntosh, and W. H. Banzhaf, "CMOS RAM, ROM, and PLA Generators for ASIC Applications," *Proc. Custom Integrated Circuits Conference* , 1986, pp. 334–338.
220. W. Swartz and C. Sechen, "New Algorithms for the Placement and Routing of MacroCells," *Proc. Int. Conf. on Computed–Aided Design*, 1990, pp. 336– 339.
221. T. Tanaka, T. Kobayashi, and O. Karatsu, "HARP: Fortran to Silicon," *IEEE Trans. Computer–Aided Design*, Vol. 8, No. 6, June, 1989, pp. 649–660.
222. R. E. Tarjan, *Data Structures and Network Algorithms* , (Philadelphia: Society for Industrial and Applied Mathematics, 1983), pp. 2–6.
223. K. O. ten Bosch, P. Bingley, and P. van der Wolf, "Design Flow Management in the NELSI CAD Framework," *Proc. Design Automation Conference* , 1991, pp. 711–716.
224. M. Terai, K. Takahashi, and K. Sato, "A New Min–Cut Placement Algorithm for Timing Assurance Layout Design Meeting Net Length Constraint," *Proc. Design Automation Conference* , June 1990, pp. 96–102.
225. H. F. Trickey, "A High–Level Hardware Compiler," *IEEE Trans. Computer–Aided Design*, Vol. CAD–6, No. 2, March, 1987, pp. 259–269.
226. J. Trnka, R. Hedman, G. Koehler, and K. Ladin, "A Device Level Auto Place and Wire Methodology for Analog and Digital Masterslice," *Proc. ISSCC* , 1988, pp. 260–261.
227. R. Tsay, E. Kuh, and C. Hsu, "PROUD: A Fast Sea–Of–Gates Placement Algorithm," *Proc. 25th Design Automation Conference* , 1988, pp. 318–323.

228. R. Tsay and J. Koehl, "An Analytic Net Weighting Approach for Performance Optimization in Circuit Placement," *Proc. Design Automation Conference* , June 1991, pp. 620–625.
229. M. Upton, K. Samii, and S. Sugiyama, "Integrated Placement for Mixed Macro Cell and Standard Cell Designs," *Proc. Design Automation Conference* , 1990, pp. 32–35.
230. P. van den Hammer and M. A. Treffers, "A DataFlow Based Architecture for CAD Frameworks," *Proc. ICCAD*, 1990, pp. 482–485.
231. L. P. van Ginneken and R. H. Otten, "Global Wiring for Custom Layout Design," *Proc. International Symposium on Circuits and Systems* , June 1985, pp. 207–208.
232. A. Vannelli, "Interior Point Method for Solving the Global Routing Problem," *Proc. IEEE 1989 Custom Integrated Circuits Conference* , May 1989, paper 3.4.
233. A. Vannelli, "An adaptation of the interior point method for solving the global routing problem," *IEEE Trans. on Computer-Aided Design* , Vol 10. No. 2, Feb. 1991, pp. 193–203.
234. R. Weier, "TimberWolfAR: An Arbitrary Design Rule Multi-Layer Area Router", paper 2.2 Vol. 1, *Proc. International Workshop on Layout Synthesis* , May 18–21, 1992. MCNC Research Triangle Park, N.C. , pp. 45–46.
235. G. M. Weinberg, *The Psychology of Computer Programming* , (New York: Van Nostrand Reinhold, 1971).
236. N. Weste, "Virtual Grid Symbolic Layout", *Proc. 18th Design Automation Conference* , June 1981, pp. 225–233.
237. N. Wirth, "Program Development by Step-Wise Refinement," *Communications of the ACM* , Vol. 14, No. 4, 1971, pp. 221–227.
238. D.F. Wong and C.L. Liu, "A New Algorithm for Floorplan Design," *Proc. 23rd Design Automation Conference*, 1986, pp. 101–105.
239. D.F. Wong and C.L. Liu, "Floorplan Design for Rectangular and L-Shaped Modules," *Proc. Intl. Conf. on Computed-Aided Design*, 1987, 520–523.
240. G. Wong, "Analog Integrated Circuit Placement Optimization by Simulated Annealing," Master's thesis MIT, 1985.
241. O. Yasushi, T. Ishii, et al., "Efficient Placement Algorithms Optimizing Delay for High-Speed ECL Masterslice LSI's." *Proc. 23rd Design Automation Conference*, 1986, pp. 404–410.
242. J. Y. Yen, "Finding the K Shortest Loopless Paths in a Network," *Management Science* , Vol. 17, July 1971, pp. 712–716.
243. H. Youssef, "Timing Issues in Cell Based VLSI Design," Ph. D. Dissertation, Computer Science Department, University of Minnesota, January, 1990.

Technical Discussion of Dynamic Documentation

Dynamic documentation uses two Javascript (tm) functions to control and create the hypertext document, namely the Tk function **JS_TK()** and the conditional function **JS_COND()**. Both of these functions are invoked from within the **<SCRIPT>** tag.

The Tk Function

The **JS_TK** Javascript function creates active embedded windows within the hypertext. It is similar in nature to the JAVA *"applet"* tag. In fact, the **JS_TK** function is an alternative implementation of the applet tag using Tk/Tcl rather than JAVA. The major reasons for using Tk/Tcl rather than JAVA is that **1)** EZ CAD is a Tk/Tcl application, **2)** Tk/Tcl is freely available **NOW** for all Unix workstations, as well as MacIntosh, Windows, and WindowsNT, **3)** the **JS_TK** function does not allow access over the network for increased security, and **4)** the instability the of Java Virtual Machines on the platforms that we support.

The definition of the Tk function is:

```
JS_TK(PROCEDURE="...", TKWINDOW [,ARGS="..."] [,WIDTH="..."] [,HEIGHT="..."] [,ALIGN="..."]
    [,PADX="..."] [,PADY="..."] [,ALT="..."] [,ALTWIDTH="..."] [,ALTHEIGHT]);
```

The keyword **JS_TK** starts a Tk embedded window. It must be followed with the **PROCEDURE** and **TKWINDOW** phrases. All other hypertext phrases are optional. Unlike JAVA, the width and height of the window is optional; without these phrases, Tk determines the size of the window based on the content of the windows in the native browser. Unfortunately, Netscape has the same shortcomings as JAVA. In order to support Netscape, the **ALTWIDTH** and **ALTHEIGHT** parameters are available. The string following the **PROCEDURE** phrase must be a valid Tk procedure. The procedure will be [autoloaded](#) by adding the procedure to the tclIndex in the EZ tcl directory. Only procedures which are in the autoload index will be available for execution, thereby, increasing the security of the system. The Tk procedure's first argument must be the [Tk window hierarchy](#) where the embedded window is to be created. The string following the **TKWINDOW** phrase names the tkwindow for Netscape embedding. It must be a unique [Tk window identifier](#).

The optional keyword **ARGS** allows the user to pass additional arguments from the hypertext to the Tk/Tcl procedure. The optional **WIDTH** and **HEIGHT** phrases allow the user to control the size of the embedded window. The width and height are specified in screen pixels. Both **WIDTH** and **HEIGHT** must be supplied simultaneously; otherwise, the phrase is ignored. The optional **ALIGN** keyword allows the user to control the vertical alignment of the window. Currently, **TOP**, **MIDDLE**, and **BOTTOM** alignments are supported. The optional keyword **PADX** allows the user to specify a space on both sides of the embedded window. The space is specified in screen pixels. Likewise, the optional keyword **PADY** specifies the spacing above and below the window in screen pixels. The optional keywords **ALTWIDTH** and **ALTHEIGHT** are available to control the respective width and height of the Netscape embedded window. They are ignored in the **itools** native browser. The optional keyword **ALT** specifies an alternate text for browsers that cannot display embedded windows. The **ALT** phrase is added for completeness but does not play any role in the EZ CAD System. All hypertext keywords are case sensitive and **MUST be either all lower or all upper case**.

Now we will turn our attention to an example of a Tk function. The Tk procedures below implement a pull down menu for changing the user mode.

```
proc ez_user {w} {
    # Get the global variable which holds the current user state.
    upvar #0 ICvarsG ic

    # Set the parameters which control the display of Tix combo box.
```

Ittools Documentation

```
set name [tixOptionName $w]
option add *$name*TixComboBox*label.width 16
option add *$name*TixComboBox*label.anchor e
option add *$name*TixComboBox*entry.width 20

# Create a combo box which we will configure to become a pull down menu.
tixComboBox $w.user -label "Level of Expertise" \
    -editable false -dropdown true -command "user:select_user $w.user" \
    -history 0

# Insert the options into the pulldown menu
$w.user insert end Novice
$w.user insert end Intermediate
$w.user insert end Expert
if {$ic(S_developer)} {
    $w.user insert end Developer
}

# Now lets set it to the current user mode.
tixSetSilent $w.user $ic(S_user)

# Now pack the widget so we can see the embedded window.
pack $w.user -side top -padx 20 -anchor center -pady 3
}

proc user:select_user {w s} {
    # Update the global user state.
    ICset_state -user $s

    # Change the value of the selector.
    tixSetSilent $w $s

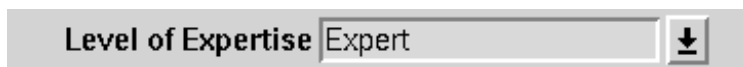
    # Tell the user the new mode.
    ic_message msg "Selected user mode:$s"

    # Re render the hypertext in the new user mode.
    ic:user_update render
}
```

To use this procedure, we just need to reference it in the hypertext. We would write:



and we do so below:



The Conditional Function

The conditional Javascript function controls the display and execution of the hypertext within its definition. The definition of the conditional function:

JS_COND(EXPR="..." , HTML="..." [, LOGIC=NEGATIVE])

or the alternate complete form

ltools Documentation

```
JS_COND (EXPR="..." , HTML="..." [, LOGIC=NEGATIVE] [,PROCEDURE="..."]

        [,TKWINDOW] [,ARGS="..." [,WIDTH="..." [,HEIGHT="..." [,ALIGN="..."]

        [,PADX="..." [,PADY="..." [,ALT="..." [,ALTWIDTH="..."]

        [,ALTHEIGHT] [,TKPLACEMENT="before"] )
```

The keyword **JS_COND** starts the conditional hypertext function. It must be followed with the **EXPR**, and **HTML** phrases. The **LOGIC** phase is optional. All hypertext keywords are case sensitive and **MUST be either all lower or all upper case**. However, the double quoted strings after the keywords are case sensitive.

The **EXPR** keyword states an expression for when the hypertext is displayed. The expression may be separated by the logical operators **||** (or), **&&** (and), and **!** (not). In addition, parentheses may be used for complex logic. It is necessary that the expression must be a valid Tcl expression. It is important to note that white space is preserved within the double quotes. For example,

```
EXPR="$ICMODE==standard cell tutorial||$ICMODE==macro cell tutorial,"
```

In the example, the condition code will be displayed if *"standard cell tutorial"* or *"macro cell tutorial"* mode is enabled. Notice that *"macro cell tutorial,"* ends in an OR operator. This ensures that *"macro cell tutorial"* is taken as one string instead of three individual ones named *"macro"*, *"cell"*, and *'tutorial'*.

A builtin Tcl procedure evaluates the expression. Positive logic is used in the decision to display the hypertext. That is, if the expression is enabled by the Tcl procedure, the hypertext will be shown. The optional phrase **LOGIC="negative"** allows the use of negative logic. This means that if the expression evaluates to false, the hypertext will not be shown; otherwise, it will be displayed. Nesting of conditional functions is **NOT permitted**. It is very important to remember that the HTML to be displayed must remain as one string. One can split the string over multiple lines in a file by using a backslash **"\"** character at the end of the line. In addition, double quotes within the HTML string must be escaped using the back-slash character **(\)**.

The alternate form of the command allows a Tk widget to be conditionally displayed. The Tk widget is rendered ***AFTER*** the accompanying HTML is displayed unless the **TKPLACEMENT="before"** option is supplied. This form inherits all of the requirements and options of the **JS_TK** function above. For example, if a Tk widget is to be displayed conditionally, the **PROCEDURE** and the **TKWINDOW** keywords *must both be present as the JS_TK function requires*.

Now, let us show an example:

This code will be displayed when the user mode is in intermediate or expert mode.

Notice in the latter script that the expression contains **\$ICvarsG(S_user)** and not **\$ICUSER**. As you can see **\$ICUSER** is a convenience variable and is a synonym for the Tcl variable **\$ICvarsG(S_user)**. Other convenience variables include **ICUSER**, **ICMODE**, **ICDESIGN**, and **ICFLOW**. A complete list is found [here](#).

The Tcl procedure *"iceval_expr"* which evaluates the expression is defined as follows:

```
proc iceval_expr {expression} {
    global ICvarsG
    if {[expr $expression]} {
```


```

    return true ;
}
return false ;
}

```

Above are two fragments of conditional hypertext. Both fragments are dependent on the expression `"$ICUSER==Intermediate||$ICUSER==Expert"` but they have different and opposite logic. The Tcl procedure `"iceval_expr"` is also presented. All convenience variables are substituted in the expression `"$ICUSER=Intermediate||$ICvarsG($S_user)==Expert"` and the expression is double quoted to form a valid Tcl expression. It is then passed into the procedure `iceval_expr` to be evaluated. The global array variable `ICvarsG($S_user)` contains the current state of the user mode. It is defined and maintained by the EZ CAD system. The procedure evaluates the expression and returns true if the user is either an Intermediate or an Expert. Otherwise, it returns **false**.

Now let us see the conditional code in action. We will change the color of the text to red so that the conditional hypertext will be apparent. Change modes and notice how the hypertext is displayed as the user mode is changed. Change the mode using the pulldown menu created below.

Level of Expertise


This code will be displayed when the user mode is in intermediate or expert mode. Change the pulldown menu and you will change the display of this hypertext.

Now one final example. Toggle the mode to *Intermediate* mode to see a button appear. You may view the hypertext using the *View Source command under a Netscape browser or by the clicking on the View or Edit icon found in the native browser*.

Summary

We have presented a revolutionary new active hypertext documentation system and graphical user interface. This system understands different user knowledge levels, actively integrates the program into the documentation, and incorporates the latest research in usability engineering.

Tk Windows

Copyright (c) 1999. InternetCAD, Inc. All right reserved.

The basic building block for a graphical user interface in Tk is a **widget**. A widget is a window with a particular appearance and behavior (the terms "widget" and "window" are used synonymously in Tk.) Widgets are divided into classes such as buttons, menus, and scrollbars. All the widgets in the same class have the same general appearance and behaviour. For example, all button widgets display a text string or bitmap and execute a Tcl command when they are invoked with the mouse.

Widgets are organized hierarchically in Tk, with names that reflect their positions in the hierarchy. The **main widget**, which appeared on the screen when you started **wish (EZ)**, has the the name ".". The name .b refers to a child b of the main widget. Widget names in Tk are like file names in UNIX except that they use . as a separator character instead of /. Thus .a.b.c refers to a widget that is a child widget of .a.b which in turn is a child of .a, which is a child of the main widget.

Ousterhout, John K., "Tcl and the Tk Toolkit", p. 10.

Getting Started

A valid license is ***NOT*** required to complete this page.

First, we will need to know your level of knowledge. Please set your [level of expertise](#) in the **itools** system.

Level of Expertise 

Experts have all capabilities but documentation is less verbose; just an enumeration of the options.

Next, we need to determine the execution mode for EZ CAD. The execution mode dynamically modifies the hypertext allowing the navigation of the **itools** tutorials. The default mode is the normal design flow.

iTools Mode 


Library/Foundry 

All of the information has been set for the gridded standard cell tutorial.

Ittools has detected work has been performed previously on this design.


There is no need to set any of the widgets on this page. The widgets below are shown in their correct state.

 /

-  Users
 -  bill
 -  .itools
 -  tutorials
 -  **gridded**
 -  itoolsdata
 -  floorplan
 -  groute
 -  place
 -  route

To set directory, hit
'Apply'
after choosing directory on left.

iTools Design Directory:

Design Name 

[Next Page:Inputs/Translation](#) =>

Available Tutorials

The table below lists the current tutorials in the EZ system. Each tutorial teaches a complete lesson in using **itools** on a design. In each tutorial, the correct options and files will be supplied for the user so that one can learn by emulation. The tutorials help to show the common and expected inputs of the system.

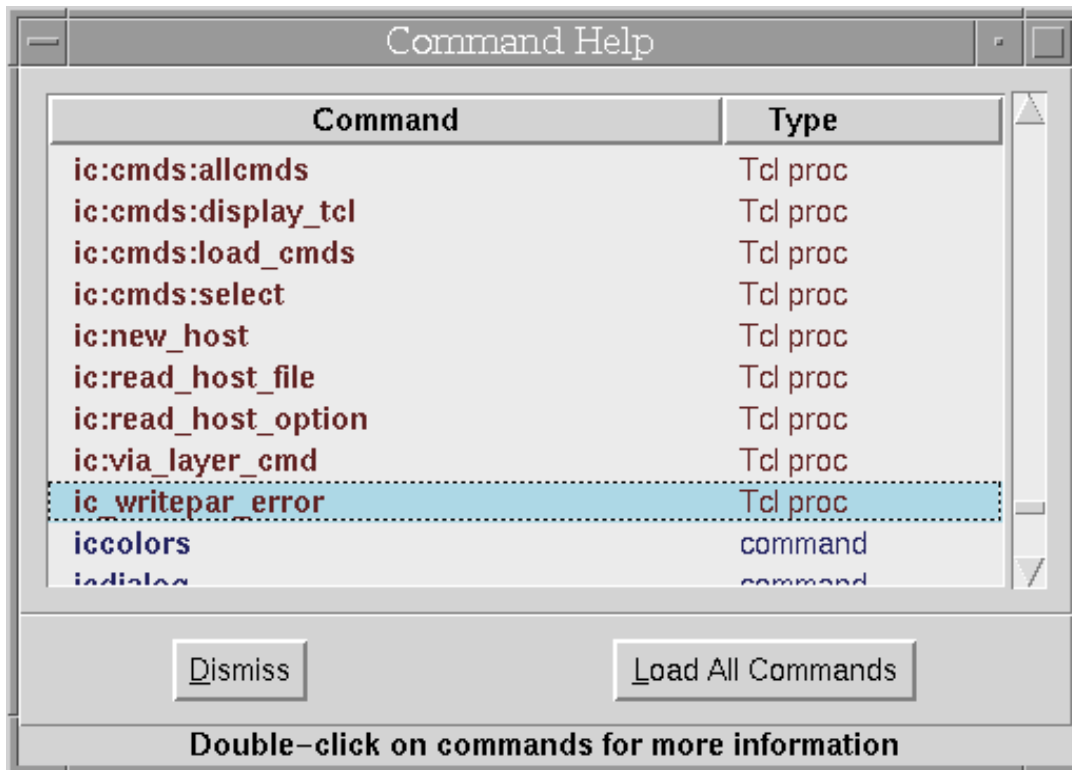
TUTORIAL	FUNCTION
<i>standard cell</i>	How to place and route a design with only row-based cells.
<i>macro cell</i>	How to place and route a design with only rectilinearly-shaped block cells.
<i>mixed cell</i>	How to place and route a design with macro blocks and row-based cells.

Auto Help Command

The iTools system consists of a set of programs which are Tcl programmable. The system may be extended by the users to suit their needs. The system contains two types of Tcl commands : *builtin* commands and Tcl procedures or *procs*. In order to document the available commands for each tool, an *autohelp* facility has been built into each of the tools. You may access the *autohelp* facility while in the graphics mode. You will find it under the *Help* menu that you find at the upper right corner of the graphics window. After choosing the *Commands* option, you will be presented with a window displaying the available commands in the current tool.

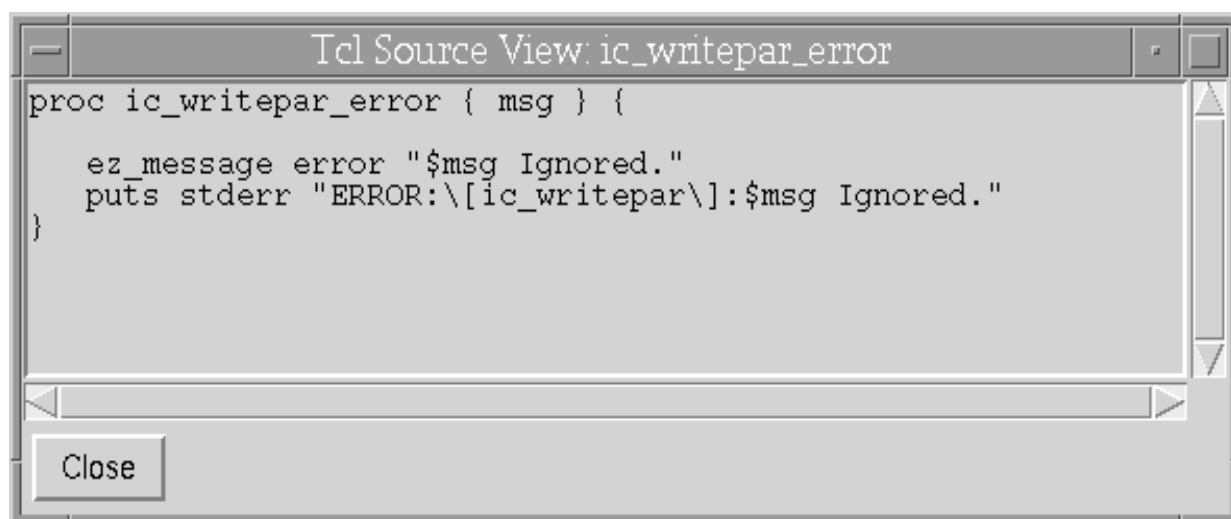


Each line contains the name of the command followed by its type. The user may receive more information for each command by double clicking on the desired line. If the command is a *builtin* command, the command will be executed with the *-help* argument and the result presented in the transcript window. If the command is a Tcl procedure, the source for the command will be presented in a scrollable text window.



At the bottom of the *Command Help* window are two buttons : *Dismiss* and *Load All Commands*. The *Dismiss* button will exit the *Command Help* window. The *Load All Commands* will load all Tcl procedures into the list of commands. Normally, the Tcl procedures are *auto-loaded*, that is, they are only loaded when they are reference. This makes the program load faster since only the minimal set of Tcl commands are loaded. However, the user may wish to see all of

the available commands. The *Load All Commands* function will load all of the Tcl files into the program and then display them.



Design Directory

Copyright (c) 1999–2005. InternetCAD, Inc. All right reserved.

The design directory serves as a depository for all of the files required and created by itools. Itools will create subdirectories (notably the *itoolsdata* directory which contains all of the files created by itools) for its own use but all design data for import into itools will reside in this directory.

Knowledge Criteria

Copyright (c) 1998–2005. InternetCAD, Inc. All right reserved.

EZ CAD allows users of all levels of expertise to use **itools** effectively. Currently, there are four levels of expertise: Novice, Intermediate, Expert, and Developer. The interface is simplified for Novices and explanations are as detailed as possible. No advanced options are available to the Novice. The Intermediate interface allows more options to the **itools** programs. The Expert mode allows the user access to all options to **itools**. The Developer mode allows the user to modify the hypertext in order to customize it for their own application. Reread this document in expert mode for more details on how to enter the Developer mode.

In order to enter "*Developer*" mode, you need to set the **ICDEVELOPER** environment variable in the shell before executing EZ CAD. For example, using **csh** you would enter:

```
setenv ICDEVELOPER 1
```

This extra step insures that ordinary users don't change modes and accidentally modify the hypertext.

Design or Root Name

Copyright (c) 1999–2005. InternetCAD, Inc. All right reserved.

The name of the design is substituted as the root of the **itools** file names. An **itools** file consists of the design name concatenated with a suffix which denotes the meaning of the file. There are several important **itools** file names: *.ckt*, *.lib*, *.con*, and *.par*. If the **design** name or **root** is "test", the file names will exist as *test.ckt*, *test.lib*, *test.con*, and *test.par* respectively. This should not be confused with the iTools root directory which represented by the environment variable **ICDIR**.

ITOOLS Inputs

A valid license is ***NOT*** required to complete this page. Translation does not require a license except for GDS2 processing.

Itools must have the following input files in the current design directory:

[*designName.lib*](#) describes the physical implementation of a design.


[*designName.ckt*](#) describes the logical netlist of a design.

[*designName.par*](#) controls the execution of **itools** programs.

The following input files are optional:

- The [*designName.con*](#) describes the physical design constraints, such as specifying placement of cells and pads, and timing constraints.
- The [*designName.host*](#) file which controls parallel execution of the placement program.

The input files are present. There is no need for translation. However, you may retranslate into itools using the [**itranslate**](#) program. Use the force translation button below to enable

translation **Force Translation**  on  off

[Edit Parameter File](#)

or choose the blue button to edit the parameter file (*designName.par*):

[=>](#)

[Next Page:Syntax/Flow](#)

[=>](#)

The Format of the designName.lib File

The file *designName.lib* contains the descriptions of the standard cells (row-based cells), macro cells and pads. Comments may be added using C-style comments (*/* */*). There is no special order for the description of cell models. The complete BNF for **itools** is given in Appendix B. Each description describes a model in the model library. Each model may have a set of versions. Each version *MUST* have the same number of netlist pins (the number of physical ports may vary among versions). All other physical attributes may differ between versions including the timing information.

The design library may be placed in separate files and included in the *designName.lib* by using the following construct:

```
%INCLUDE filename
```

For example, if you want to place the library file in "/home/user/design/ourlib", you would put the following in between any models in the *designName.lib* file.

```
%INCLUDE /home/user/design/ourlib
```

Include double quotes around the filename if spaces are present in the full pathname. You must supply either a fully-qualified pathname or pathname relative to the design directory. Any number of library files may be included but models may not be split over different files.

Valid Itool Strings

Itools accepts most unquoted strings (white space is not allowed within a string) as valid input. One must be careful about parentheses as parentheses are used to bind signals in the *designName.ckt* file. A string may not begin with either a left or right parenthesis. A string must contain matching parentheses in order to be valid. For example, the following are **invalid** strings : (*signalA*,)*signalA*, *signal*(, *signal*)*A*, *signalA*), and *signalA*). However, the following are **valid** strings: *signalA*, *signalA*(*A*), *signalA*() , and *signal*()(*A*). Itools will accept valid Verilog names.

The keyword **MODEL** begins the description of a model. The string following the keyword is the name of the model. This model name is referenced in the *designName.ckt* file. The type of model may be one of the following:

- **STANDARD** – row-based cell (standard cell or gate array cell).
- **PAD** – I/O cell ([see extra pad constraints at end of section](#)).
- **HARDCELL** – macro cell.
- **SOFTCELL** – flexible macro cell.
- **FEED** – row-based feed through cell.
- **SPACER** – row-based feed cell used to connect power/ground rails and wells through empty spaces.
- **FPGA** – field programmable gate-array.
- **ANTENNA** – special row-based cell for deep submicron antenna rules.
- **PORT** – special I/O cell ([see extra pad constraints at end of section](#)).

A model version begins with the **VERSION** keyword followed by the name of the version. A model may have any number of versions or templates. Each version is a valid implementation of the model. The number of signals connecting to a model must be the same over all versions but the physical implementation may be quite different. The placement programs will determine the best model version for each instance of the netlist based on area and timing constraints.

The optional **CLASS** restricts the model to the set of row classes. The set of classes are integers which immediately follow the **CLASS** keyword. Instances of this model inheritant the classes of the model. Each row in the *designName.blk* file may be assigned a class. If no class is explicitly assigned, the row defaults to class 1. Each instance may only be assigned to rows in which the classes match. Note, that a single instance may have more than one class assigned to it, whereas, each row may only be one class.

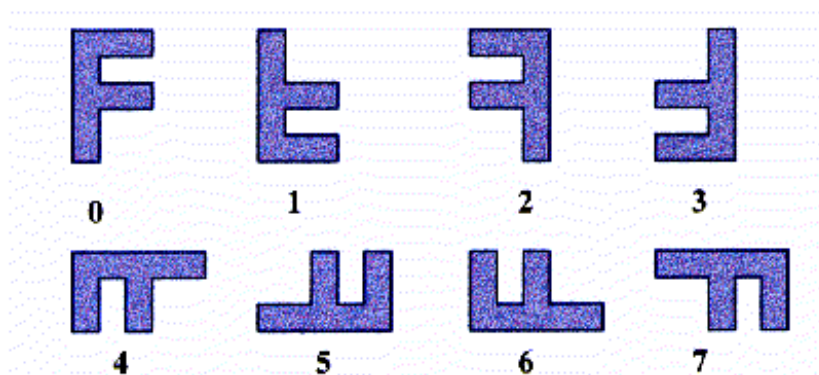
For each version, a model boundary is mandatory. A boundary is described by the **BOUNDARY** keyword followed by a vertex list, starting from the leftmost of the lowest vertices and proceeding in a clockwise manner around the cell. **Itools** handles cells of any rectilinear shape. The number of vertices is equal to the number of edges (or sides) of the cell (since the first vertex is not repeated). Each vertex is denoted by an x,y pair enclosed by parenthesis.

The keyword **ASPECT** is followed by a floating point number which represents the lower bound on the permissible aspect ratio for the cell (in the orientation as implied by the boundary definition, which supplied the cell geometry). An optional second floating point number represents the upper bound on the permissible aspect ratio for the cell. If only one floating point number is specified, then **itools** assumes that the aspect ratio must remain as given. Aspect ratio is defined as width divided by the height of the cell.

The optional aspect ratio range is only valid for **SOFTCELL** models. The keyword **ASPECT** is followed by a floating point number which represents the lower bound on the permissible aspect ratio for the cell (in the orientation as implied by the boundary definition, which supplied the cell geometry). An optional second floating point number represents the upper bound on the permissible aspect ratio for the cell. If only one floating point number is specified, then **itools** assumes that the aspect ratio must remain as given.

A keepout region is a region where a detailed-router is excluded. **Itools** allows the definition of a set of exclusion boundaries for any layer. A keepout region starts with the keyword **KEEPOUT**. Next, an arbitrary rectilinear boundary may be specified by a boundary construct. The boundary starts with the boundary keyword followed by a list of vertices. The vertices are listed starting from the leftmost of the lowest vertices and proceeding in a clockwise direction. The routing layer associated with the keepout is specified either by the integer following the keyword **LAYER** or by the layer name. The integer number cross-references the layer definitions found in the **RULES** section of the parameter file in the order that the layers were defined. The first layer defined in the **RULES** section will become layer 1, the second layer will be layer 2, and so forth. If the layer is unknown or does not matter, use layer 0. The use of layer numbers allow remapping of layer definitions whereas the layer name make it easier to read at the expense of the ability to remap the layers.

The detail router may abut routes against the keepout. It is the responsibility of the keepout generator to maintain design rule correctness. The second form of the keepout is supported for backwards compatibility but deprecated.

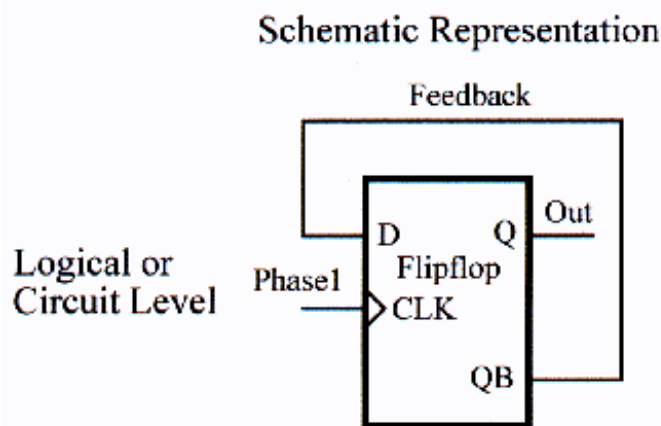


Itools orientations

The next optional structure is the orientation list. Following the keyword **ORIENT** is a list of valid orientations for the cell. There are 8 possible orientations for a cell as shown in the above figure. The first integer in the list specifies the initial orientation. The cell orientation numbering scheme employed by **itools** is: (0) Orientation 0 is the cell geometry as described above in the vertex list. (1) Orientation 1 is the cell geometry after mirroring the y coordinates with respect to orientation 0. (2) Orientation 2 is the cell geometry after mirroring the x coordinates with respect to orientation 0. (3) Orientation 3 is the cell geometry after a rotation of 180 degrees with respect to orientation 0 (which is the same as a mirror of the y coordinates with respect to orientation 0 followed by a mirror of the x coordinates). (4) Orientation 4 is the cell geometry after a combination of a mirror of the cell's x coordinates followed by a 90 degree rotation of the cell with respect to orientation 0. (5) Orientation 5 is the cell geometry after a combination of a mirror of the cell's x coordinates followed by a -90 degree rotation of the cell with respect to orientation 0. (6) Orientation 6 is the cell geometry after a 90 degree rotation of the cell with respect to orientation 0. (7) Orientation 7 is the cell geometry after a -90 degree rotation of the cell with respect to orientation 0.

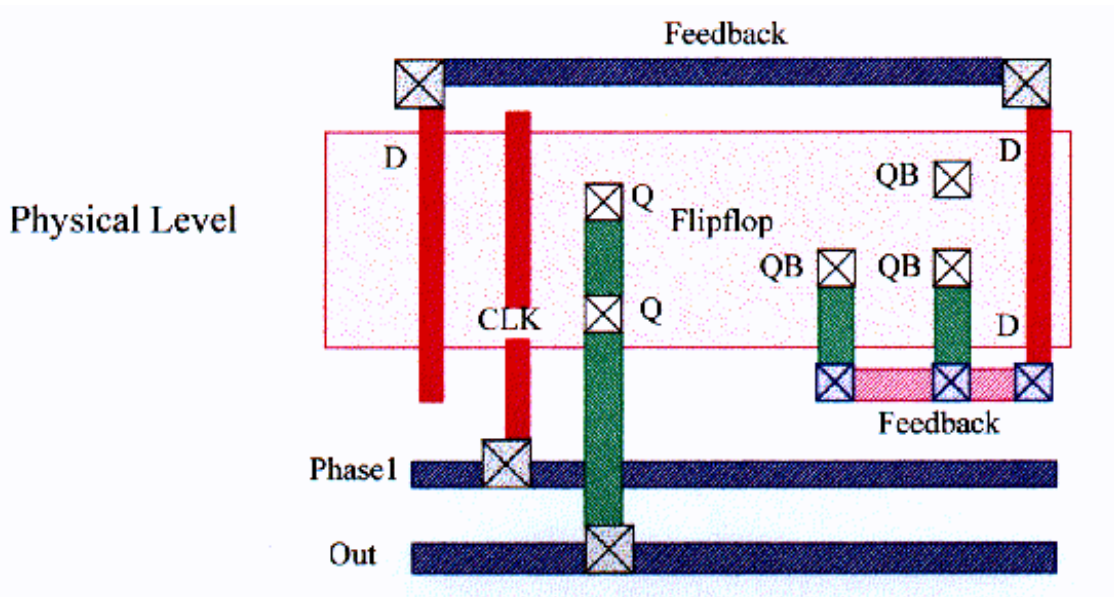
The strategy behind the numbering scheme is based on the fact that the first 4 orientations (numbered 0 through 3) have the same aspect ratio. The second 4 orientations (numbered 4 through 7) also have the same aspect ratio, which is the inverse of the aspect ratio of orientations 0 through 3.

The next structure describes the position of the physical ports and the model pin names associated with the signal or net names. A model pin provides the mapping between the netlist schematic pins and the physical ports. Each model pin contains a set of electrically equivalent ports. A schematic pin in the netlist may have more than one physical pin. This is accomplished by listing multiple model pins with the same model pin name. However, the set of model pins must be contiguous; no other model pins may appear in between. For example, let us present a small D flip-flop example. In the physical domain, both D input model pins must be connected, as well as both Q and QB model pins. The CLK pin requires only a single connection to one of the two electrically equivalent pins. The model definition for the D flip-flop is shown in the figure below.



The *circuitName.ckt* file would contain the following lines to represent this flip-flop:

```
INSTANCE inst1 Flipflop
(Phase1 CLK) (Feedback D) (Out Q) (Feedback QB)
```



Example of a flip-flop model showing model pins with multiple connections.

The *circuitName.lib* file for the example flip-flop circuit would contain the following:

```
MODEL Flipflop TYPE STANDARD
VERSION 1
BOUNDARY (-100 -30) (100 30)
ORIENT 0 1 2 3
PIN CLK I
  PORT CLK1 (-70 -40) LAYER POLY
  PORT CLK2 (-70 40) LAYER POLY
PIN D I
  PORT D1
    RECT (-85 -45) (-75 -35) LAYER POLY
  PORT D2
    RECT (-85 35) (-75 45) LAYER POLY
PIN D I
  PORT D3
    BOUNDARY (85 -45) (85 -35) (95 -35) (95 -45) LAYER POLY
  PORT D4
    BOUNDARY (85 35) (85 45) (95 45) (95 35) LAYER POLY
PIN Q O
  PORT Q1 (60 20) LAYER METAL2
PIN Q O
  PORT Q2 (60 -20) LAYER METAL2
PIN QB O
  PORT QB1 (40 -10) LAYER METAL2
PIN QB O
  PORT QB2 (60 10) LAYER METAL2
  PORT QB3 (60 -10) LAYER METAL2
ENDVERSION
ENDMODEL
```

There are two types of pins: hard pins and soft pins. Hard pins have all geometric details determined, whereas soft pins have unknown quantities to be determined. Each will be described in turn.

Hard pins begin with the keyword **PIN** followed by the model pin name. The pin type is denoted by the keyword **TYPE** followed by any of the following pin type descriptors:

- I – input
 - O – output
 - B – bidirectional pin.
 - F – feed through pin (implicit or build in feed through path).
 - PWR – power pin.
 - SHIELD – parasitic shield pin.
 - SPECIAL – distributed bus build into row-based cells.
 - GND – ground pin.
 - UKN – unknown pin type.
- HIGHWAY – pin which spans many grids and makes global connections.

Optionally, each pin may also be assigned a capacitance value. The capacitance of the pin is specified using the **CAPACITANCE** keyword followed by the capacitance in farads. The value may be modified by the [SCALE CAPACITANCE](#) keyword in the *designName.par* file. This information is used in the timing-driven placement algorithm. All pins should have their capacitance specified if timing-driven placement is desired. By assigning a capacitance to the pin, all ports of the pin assume this value. The capacitance of an individual port may be overridden by assigning a capacitance value to the port (see below).

For each model pin, a set of equivalent ports may be defined. A port definition begins with the keyword **PORT** followed by the port name string. Next, the x-y coordinates of the pin are specified in one of two ways. In the first method, the port center is given as a x-y tuple enclosed by parentheses. Using the second method, the port boundary is described using a boundary construct – the keyword **BOUNDARY** followed by a clockwise list of the boundary vertices starting from the leftmost of the lowest vertices and proceeding in a clockwise manner.

The routing layer associated with the port is specified by either an integer or string following the keyword **LAYER**. The integer form cross-references the layer definitions found in the RULES section of the parameter file in the order that the layers were defined. The first layer defined in the RULES section will become layer 1, the second layer will be layer 2, and so forth. If the layer is unknown or does not matter, use layer 0. In the alternate string definition, the layer name follows the keyword **LAYER**. The integer form allows the remapping of layers whereas the string form makes the physical description easier to read.

An optional current may be associated with the port by adding the **CURRENT** keyword followed by a floating point number representing the current in amps. Positive current flows into the port whereas negative current flows out of the port.

If only one of an electrically-equivalent pair of pins can be used (for example, because the resistance of the poly line connecting them is too high), the optional keyword **UNEQUIV** may be specified. In this case, only two **UNEQUIV** ports may be specified for a pin.

The driving strength resistance of the port may be specified using the **RESISTANCE** keyword followed by a floating point number in ohms. The value may be modified by the [SCALE RESISTANCE](#) keyword in the *designName.par* file. This information is used in the timing driven placement algorithm. It is important that all the output and bidirectional ports have their resistance specified. The resistance may also specify the transition by including either the keyword **HIGHLOW** for a high-to-low output transition or the keyword **LOWHIGH** for a low-to-high output transition. In addition, minimum, nominal, and maximum values may be specified using a triple set of numbers separated by colons. The value is chosen using the **time** parameter in the *designName.par* file. For example, the resistance may be specified as

```
RESISTANCE HIGHLOW (80:90:100)
```

or may be simplified (if the high-to-low and low-to-high values are the same and the nominal value is to be used)

RESISTANCE 90.0

It should be noted that this driving strength may also be specified for an individual delay path using the [DELAY](#) statement. It is recommended for accuracy that the driving port resistance be specified on a path basis rather than using **RESISTANCE** specified on a **PORT**.

The capacitance of the port is specified using the **CAPACITANCE** keyword followed by the capacitance in farads. The value may be modified by the [SCALE CAPACITANCE](#) keyword in the *designName.par* file. This information is used in the timing-driven placement algorithm. All ports should have their capacitance specified if timing driven placement is desired. The capacitance may also specify the transition by including either the keyword **HIGHLOW** for a high-to-low output transition or the keyword **LOWHIGH** for a low-to-high output transition. In addition, minimum, nominal, and maximum values may be specified. The value is chosen using the **time** parameter in the *designName.par* file.

In the case of soft macro cells, hard ports will retain the same relative position during aspect ratio changes, that is, hard ports will be scaled appropriately during aspect ratio changes. Hard pins may have any number of equivalent ports.

A soft pin may only occur on a soft macro cell. Soft pins begin with the keyword **SOFTPIN** followed by the model pin name. The pin type is denoted by the keyword **TYPE** followed by any of the following pin type descriptors:

- I – input
- O – output
- B – bidirectional pin.
- F – feed through pin (implicit or build in feed through path).
- PWR – power pin.
- GND – ground pin.
- UKN – unknown pin type.

Optionally, each softpin may also be assigned a capacitance value. The capacitance of the softpin is specified using the **CAPACITANCE** keyword followed by the capacitance in farads. The value may be modified by the [SCALE CAPACITANCE](#) keyword in the *designName.par* file. This information is used in the timing-driven placement algorithm. All pins should have their capacitance specified if timing-driven placement is desired. By assigning a capacitance to the pin, all ports of the pin assume this value. The capacitance of an individual port may be overridden by assigning a capacitance value to the port.

Soft pins may possess only soft ports. A soft port has an unspecified location and the routing layer may need to be determined. In addition to the optional keywords of a hard port, a soft port may also have additional restrictions. The optional keyword **ADDEQUIV** tells **itools** to add any number of equivalent soft ports to minimize the wire length.

In addition, restrictions may be added to limit the edges of the cell on which a port may appear. The keywords **RESTRICT SIDE** is followed by a list of integers. Each integer represents a valid cell edge for the port. The edges must be numbered in a clockwise fashion starting from the edge represented by the first two pairs of vertices. If the **RESTRICT SIDE** keywords are omitted, all cell edges are valid for the soft port.

The optional keyword **SIDESPAC** followed by a floating point number allows the user to place ports at a particular relative position along a side. The floating point number represents the fraction of the side length to which the center of the port is to be placed. For example, if a side measures 100 microns, and if the sidespace floating point number is 0.10, then the port will be placed at 10 microns from starting vertex of the side. The starting vertex is determined by

traversing the cell boundary in a clockwise manner starting from the lower left vertex. If a second floating point number is specified, then the two floating point numbers describe a valid range for the placement of the port. The first number is the lower bound and the second number is the upper bound on the fraction of the side length to which the center of the port may be placed. If the **SIDESPAC** constraint is omitted, the valid range is assumed to be [0.0,1.0].

The optional **PORTGROUP** construct allows the user to specify the common properties and ordering rules of a group of soft ports. The rules are specified with the properties **FIXED** and **PERMUTE**. **PERMUTE** is a property of the port group, and **FIXED** is a property of the port group members. If a port group has the *permute* property, then members of this group can exist in two configurations: the given ordering or its reverse. This option is useful for placing signal buses which require that the signals be placed in ascending or descending rank. If **NOPERMUTE** is specified, then the ordering rules are decided by the members' *fixed* property. If a soft port is fixed, then its rank in the group cannot be changed; otherwise, it can be moved freely within the group.

Port grouping is specified using the keyword **PORTGROUP** to begin the definition of the member ports. The keyword **PORTGROUP** is followed by a user specified group name. Ports belonging to this group *must* have unique names, and the keyword **FIXED** or **NONFIXED** must follow each soft port name. The port name used in the port group *must* be declared in the *designName.lib* file before the *port group* declaration. Each soft port can only belong to one port group directly. The soft ports belonging to the pin group must be listed in a clockwise order that follows the cell vertices, that is, they will appear bottom to top on a left cell edge, left to right on a top cell edge, top to bottom on a right cell edge, or right to left on a bottom cell edge.

Port groups can have restrictions just like normal soft ports. Furthermore, they can be used in subsequent port groups just like ordinary soft ports using the port group name instead of a pin name in the list of soft ports. As with ordinary soft ports, port groups nested within another group must have already been declared.

Pin to pin timing information is specified using the **DELAY** construct. This construct allows the specification of the intrinsic propagation delay between an input pin and an output pin of the cell. The **DELAY** construct consists of the keyword **DELAY** followed by the names of the pin pair. Following the pair of model pins is unate property definition. A pin pair may either be positive unate (**POS**), negative unate (**NEG**), or unknown (**UKN**). The unknown definition is used to model exclusive OR gates or gates in which the logic is unknown. The worst case analysis will be done for such delay paths. For example, the paths through an "AND" gate would require **POS**, those through a "NOR" gate would require **NEG**, and all paths through an "XOR" gate would require **UKN**.

The propagation delay from the input pin to the output pin when the output pin is a rising waveform is specified by a floating point number following the keyword **RISETIME**. The value may be modified by the [SCALE DELAY](#) keyword in the *designName.par* file. The propagation delay occurring when the output pin falls is specified using the **FALLTIME** construct. In addition, minimum, nominal, and maximum values may be specified using a triple set of numbers separated by colons. The value is chosen using the **time** parameter in the *designName.par* file. The driving port resistance may also be specified for this pin to pin delay. The resistance is given by the key phrase "**RESISTANCE =**" followed by the driving port resistance in ohms. Again, the value may be modified by the [SCALE RESISTANCE](#) keyword in the *designName.par* file. The equations used to calculate the delay times is shown in the section on the [designName.con](#) file.

The timing-driven placement algorithm only analyzes and optimizes combinational logic timing paths; sequential timing checks such as setup and hold times are not supported. It is assumed that the user specify other paths which insure that these conditions be met. However, preset/reset to output pin delay paths may be specified for latches and flip-flops.

Itools supports hierarchical netlists and the **INSTANCE** keyword allows the user to specify fixed constraints for subcomponents or instances of the model. The **INSTANCE** keyword is followed by two strings. The first string is the

instance name which must be unique and defined in the *designName.ckt* file. The second string denotes the referenced model. The fixed constraint follows next. Fixed constraints are all relative to the placement boundary of current model. The type of [fixed constraint](#) are described in detail in the *designName.con*. The user may alternatively add the constraint in the *designName.con* file and this description in the *designName.lib* file is added as a convenience. For example:

```
MODEL mydecoder TYPE HARDCELL
  VERSION 1
  .
  .
  .
  INSTANCE i1 inv1x FIXED RIGIDLY (100 100) CELL_ORIGIN L B ORIENT 0
  INSTANCE i2 inv2x FIXED NEIGHBORHOOD (0 0) CELL_ORIGIN L B
                                (100 100) CELL_ORIGIN R T VALID_ORIENTS 0 1 2 3
  .
  .
  .
  ENDVERSION
ENDMODEL
```

In this case, we have two instances with fixed constraints. The first instance *i1* is required to remain fixed at (100, 100) relative to the lower left boundary of the model *mydecoder*. The second instance *i2* is required to reside in a neighborhood defined by (0 0) (100 100). Notice that the cell origin of the upper right coordinate is specified as right top or **R T**. This insures that the entire cell remains inside the bounding box of the neighborhood. By default, the cell origin is the center of the placement boundary. Notice that the first instance has a fixed orientation and that the second instance may not change its aspect ratio (thru the use of orientations 0 1 2 3).

This concludes the definition of a version. A model may have any number of versions with the provision that the total number of schematic pins remains constant. The number of ports may vary among the versions.

This subsection illustrates the pad description format in the input file *designName.lib*. The description of the pad boundary is given in terms of a vertex list, starting from the leftmost of the lowest vertices and proceeding in a clockwise manner around the pad. However, all pads must be given as if they were to be placed on the bottom side of the chip, that is, with the bond pad at the bottom and the ports to the core at the top. **Itools** will perform the necessary rotations in order to achieve the correct orientation for each side. Since **itools** places the pads relative to the core area, the absolute coordinates of the vertices are, in fact, meaningless. Hence, it is recommended that the user let the lower left corner of every pad be coincident with the origin.

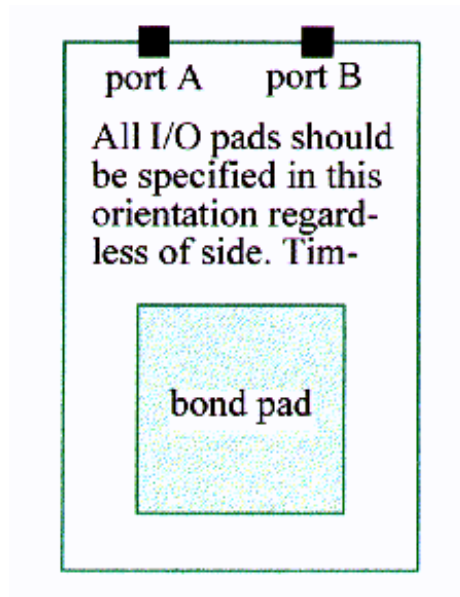
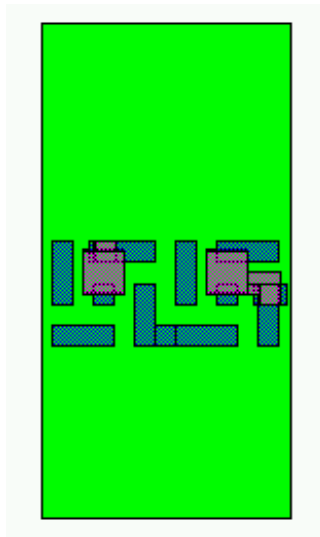
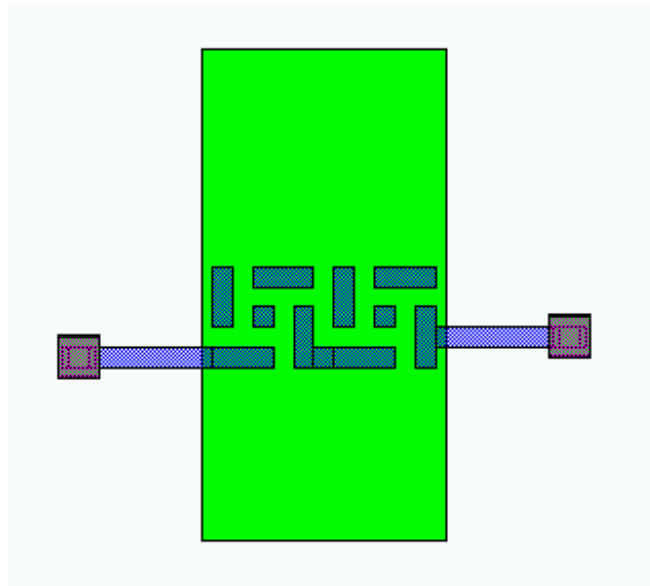


Figure Pad input orientation.

A special case of I/O cell is the **PORT** model. It acts like a pad cell but it is not constrained to remain outside the core. Instead, it may be placed anywhere and is not tested for overlap with other instances. Instead, it acts like a cover cell model. You may use a fixed constraint to place it where you desire; otherwise, the placer will seek to place it at a position which minimizes wire length. The pictures below show the same small design routed with ports and pads. In both cases, the I/O locations are the large squares.



Using Ports for I/O in Design.



Using Pads for I/O in Design.

Examples

Now we would like to present examples of standard cell definitions. In this example, we show a standard cell model named "_tif_1". It has but one version – "v0". It has one obstruction or keepout on layer 1. It has 4 pins: one input pin, two output pins, and one feedthru pin. Each pin has a capacitance associated with it of values 1.0, 1.5, and 0.8 Farads. Typically, the capacitance values will be scaled using the [SCALE CAPACITANCE](#) keyword in the *designName.par* file. There are two timing paths described: pin "I1" to pin "O1", and pin "I1" to pin "O2". The first delay is inverting, has a delay of 1.3 units when the pin "O1" rises, and a delay of 1.2 units when pin "O1" falls. This same path has a driving port resistance of 0.8 ohms when "O1" is rising and 1.1 ohms when "O1" is falling. Again, the resistance and delay units of this delay may scaled using [SCALE RESISTANCE](#) and the [SCALE DELAY](#) keywords in the *designName.par* file. The second delay path is similar to the first except that it is a non-inverting delay.

In this example, the port geometry may be specified either as a point such as PORT "I1_0", or as a rectilinear polygon similar to PORT "O2_0". If a point is given, it is expanded internally to a square centered at the specified point. The width and height will be determined by the layer's minimum width design rule.

```
MODEL _tif_1 TYPE STANDARD
VERSION v0
BOUNDARY (-30 -75) (-30 75) (30 75) (30 -75)
KEEPOUT LAYER 1 BOUNDARY (21.1 -75) (21.1 75) (29.1 75) (29.1 -75)
PIN I1 I CAPACITANCE 1.0
PORT I1_0 (-25 0) LAYER 1
PIN O1 O CAPACITANCE 1.5
PORT O1_0 (5 75) LAYER 1
PIN O2 O CAPACITANCE 0.8
PORT O2_0 (15 75) BOUNDARY (11 -5) (11 5) (19 5) (19 -5) LAYER 1
PIN tif_pin_3 F
PORT F1_0 (-15 75) LAYER 1
PORT F1_1 (-15 -75) LAYER 1
DELAY I1 O1 NEG RISETIME = 1.3 RESISTANCE = 0.8
FALLTIME = 1.2 RESISTANCE = 1.1
DELAY I1 O2 POS RISETIME = 1.3 RESISTANCE = 0.8
FALLTIME = 1.2 RESISTANCE = 1.1
ENDVERSION
```

ENDMODEL

Multiple Version Models

In this example, we show the above standard cell model named "_tif_1" in a multiple version or template configuration. It has two versions – v0 and v1. Both version have 3 signal pins: one input pin, and two output pins. However, the second version has a greater drive capability and hence a larger cell. Notice that the pins have a different placement and that even the boundary of the cell is changed. In addition, the second version doesn't have a blockage or any explicit feedthru pins (the global router will find the places to cross the cell).

```
MODEL _tif_1 TYPE STANDARD
VERSION v0
  BOUNDARY (-30 -75) (-30 75) (30 75) (30 -75)
  KEEPOUT LAYER 1 BOUNDARY (21.1 -75) (21.1 75) (29.1 75) (29.1 -75)
  PIN I1 I CAPACITANCE 1.0
    PORT I1_0 (-25 0) LAYER 1
  PIN O1 O CAPACITANCE 1.5
    PORT O1_0 (5 75) LAYER 1
  PIN O2 O CAPACITANCE 0.8
    PORT O2_0 (15 75) BOUNDARY (11 -5) (11 5) (19 5) (191 -5) LAYER 1
  PIN tif_pin_3 F
    PORT F1_0 (-15 75) LAYER 1
    PORT F1_1 (-15 -75) LAYER 1
  DELAY I1 O1 NEG RISETIME = 1.3 RESISTANCE = 0.8
    FALLTIME = 1.2 RESISTANCE = 1.1
  DELAY I1 O2 POS RISETIME = 1.3 RESISTANCE = 0.8
    FALLTIME = 1.2 RESISTANCE = 1.1
ENDVERSION

VERSION v2
  BOUNDARY (-50 -85) (-50 85) (50 85) (50 -85)
  PIN I1 I CAPACITANCE 1.0
    PORT I1_0 (-40 0) LAYER 1
  PIN O1 O CAPACITANCE 1.2
    PORT O1_0 (-30 0) LAYER 1
  PIN O2 O CAPACITANCE 1.2
    PORT O2_0 (0 0) LAYER 1
  DELAY I1 O1 NEG RISETIME = 1.0 RESISTANCE = 0.4
    FALLTIME = 0.9 RESISTANCE = 0.7
  DELAY I1 O2 POS RISETIME = 1.0 RESISTANCE = 0.4
    FALLTIME = 1.0 RESISTANCE = 0.7
ENDVERSION
ENDMODEL
```

The Format of the designName.ckt File

The *designName.ckt* file describes the netlist for the design. The complete Backus–Naur Form (BNF) for **itools** is given in [Appendix A](#). The *designName.ckt* file has the following syntax:

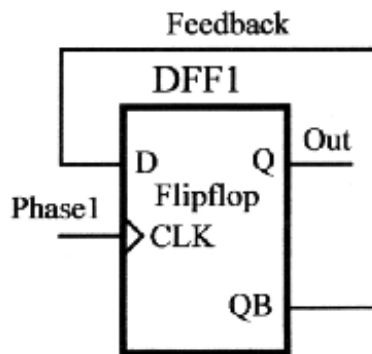
```
CIRCUIT string
INSTANCE string string
  string... |(string string)...]
  .
  .
  .
ENDCIRCUIT
```

The keyword **CIRCUIT** begins the description of the netlist. The string following netlist is the name of the design. The netlist consists of a set of instances. For example, in a digital design, the instances are the set of all the gates in the logic schematic. An instance definition begins with the **INSTANCE** keyword. Following this keyword are two strings. The first string is the name of this particular instance. It *MUST* be unique. The second string is the name of the model definition for the instance. This definition will be described in the *designName.lib* file. .

Valid Itool Strings

Itools accepts most unquoted strings (white space is not allowed within a string) as valid input. One must be careful about parentheses as parentheses are used to bind signals in the *designName.ckt* file. A string may not begin with either a left or right parenthesis. A string must contain matching parentheses in order to be valid. For example, the following are **invalid** strings : (*signalA*,)*signalA*, *signal*(, *signal*)*A*, *signalA*), and *signalA*). However, the following are **valid** strings: *signalA*, *signal*(*A*), *signalA*(, and *signal*()(*A*). Itools will accept valid Verilog names.

Schematic Representation



Textual Representations

INSTANCE DFF1 Flipflop

Phase1 Feedback Out Feedback

INSTANCE DFF1 Flipflop

(Feedback D) (Phase 1 CLK) (Out Q) (Feedback QB)

It is assumed that the model definition for the Flipflop model has pin ordered:

CLK, D, Q, QB

Two methods for specifying the signal binding.

The optional signal list may be specified by either of these two methods. In the first method, the signals are specified by listing them in order. All logical pins must be specified. Each signal is separated by white space. Unconnected pins must have a unique dummy signal name. The first signal in the list is bound to the first pin in the model description (described in the *designName.lib* file). The second pin is bound to the second pin in the model, and so forth. In the second method, each signal is bound to a pin in the model description. Each binding is specified as a pair of strings encapsulated by parentheses. The first string defines the signal net and the second string describes the pin name to connect. Either method may be used in any instance. However, for a single instance, only one signal specification method may be used. An example of the two formats is shown above.

The Hierarchical Format of the designName.ckt File

The *designName.ckt* file has been extended to describe hierarchical netlists. Such netlists contain instances which may contain subinstances which we define as *modules*. A module may be defined as follows:

```
_MODULE_ string
_PIN_ string {I | O | B | F | PWR | GND | SHIELD | SPECIAL | UKN}...
_INSTANCE_ string string
    string.[. |(string string)...]
    .
    .
    .
_ENDMODULE_
```

The keyword **_MODULE_** begins the description of a module. The string following the **_MODULE_** keyword is the name of the module. The name of the module may be any valid itool string. A module is similar in function to a model except that models are strictly physical entities; a module may or may not have a physical representation but reflects a logical partition of the circuit as deemed by the designer.

The list of input/output (I/O) pins follows next. A pin is defined by the **_PIN_** keyword followed by a string denoting the name of the pin and a pin attribute. The valid pin attributes are input (I), output (O), bidirectional (B), feedthru (F), power (PWR), ground (GND), shield (SHIELD), special global signals such as clock (SPECIAL), and unknown type (UKN). The I/O pins allow the connection of this module to its parent instance which references it. Any number of pins may be specified.

Each module may optionally have a set of submodules known as instances. These instances describe the implementation of the module. An instance begins with the **_INSTANCE_** keyword followed by two strings. The first string is the instance name and it should be a unique identifier within the module. The second string is the reference module. The reference module must either be a **MODEL** defined in the *designName.lib* file or a previously defined module in the *designName.ckt* file.

The instances are interconnected through the signal lists. Similar to the flat netlist, the optional signal list may be specified by either of two methods. In the first method, the signals are specified by listing them in order. All logical pins must be specified. Each signal is separated by white space. Unconnected pins must have a unique dummy signal name. The first signal in the list is bound to the first pin in the module/model description. (described in the *designName.ckt* or *designName.lib* file respectively). The second pin is bound to the second pin in the module/model, and so forth. In the second method, each signal is bound to a pin in the module/model description. Each binding is

specified as a pair of strings encapsulated by parentheses. The first string defines the signal net and the second string describes the pin name to connect. Either method may be used in any instance. However, for a single instance, only one signal specification method may be used.

Example of a Hierarchical Design

Below is an example of a hierarchical design. It is shown in a semi-flattened state, that is, some parts of the circuit exist in the normal flattened itools format. However, the module *mydecoder* is present and describe a subblock of the flat top level netlist. Notice that the instance name *i1* is used several places. It is valid because the string *i1* is unique within the module and unique within the flat top level netlist. Also notice the two styles of signal binding. The instance *pad1* shows the enumerated style.

```
CIRCUIT htest

_MODULE_ mydecoder
_PIN_ in I
_PIN_ out O
_INSTANCE_ i1 inv1x
    (in i) (in_bar o)
_INSTANCE_ i2 inv2x
    (in_bar i) (out o)
_ENDMODULE_

INSTANCE i1 mydecoder
    (master in) (master_dist out)
INSTANCE another_inst mydecoder
    (master in) (master_dist2 out)
INSTANCE pad1 port
    master
ENDCIRCUIT
```

The Format of the *designName.par* File

Ittools is a collection of interacting programs. Instead of having parameter files for each of the individual programs, the *designName.par* file contains the parameter specifications for all **ittools** programs.

The parameter file consists of two parts: the [design rule parameters](#) and the [program control parameters](#). The design rules may be placed in a separate file and included in the *designName.par* by using the following construct:

```
%INCLUDE filename
```

For example, if you want to place the design rules in "/home/user/design/designrules", you would put the following at the top of the *designName.par* file.

```
%INCLUDE /home/user/design/designrules
```

Include double quotes around the filename if spaces are present in the full pathname. You must supply a fully-qualified pathname or a filename relative to the design directory. Comments are similar to those found in shell languages; they are entered by placing a # in the first column of a line. [Here is an example of a parameter file.](#)

A valid parameter file exists for this design. Your options are:

1: Proceed to the next step (placement)



2: Read and edit a parameter file

Pick the file to edit by

- a: Read current parameter file: Read Current File

or

- b: Chose another parameter file: Select the Parameter File: Browse ... or
- c: Create a new parameter file: Create New Parameter File

• Edit the chosen parameter file

- Edit Parameter File

The Format of the designName.con File

All constraints are optional. Below is the supported constraints in the itools System:

```

ARRIVAL_MIN float
ARRIVAL_TIME string float
CLASS string integer
CLUSTER string {PERMUTE | NOPERMUTE} string ... ENDCLUSTER
CORE float float float float
DIEAREA float float float float
DIVIDER { . | / }
ECO ADDED CELL string
ECO DELETED CELL string
EQUIV NETS string MAXCAP float string string...
EQUIV PINS string (string/string string/string ...)
    string/$string string/string ... ) ...
FALSE_EDGE (string string string string)
FIXED string [{INITIALLY | APPROXIMATELY | RIGIDLY | NEIGHBORHOOD | VALID_ORIENTS}]
    { (number number) | (number BLOCK number)
    | (number FROM string BLOCK number)
    | (number FROM string number FROM string)}
    [CELL_ORIGIN {L|C|R} {B|C|T}]
    FORNET
    [VALID_ORIENTS ... ]
GROUP string
string string
NET string DO NOT GLOBAL ROUTE
NET string CAP FACTOR
NET string IGNORE
NET [CREATE] string GLOBAL string [string]
NET string RADIUS FACTOR float
NET string BUFFER SKEW float
    [INSERTION float] [USE ( <modelname expr> <modelname expr... )]
    [MATCH ( <net1> <net2> ... )] [STATE integer] [FIXEDSTAGES integer]
    [NONCRITICAL] [CAP_MATCH | MINIMIZE_WIRELENGTH | CLOCKGRID | FULL_CLOCKGRID ]
NET string [ROUTING | FIXED ROUTING]
    strRECT(float float) (float float) |
    flBECT(float) (float float) |
    strVIA (float float) |
    KEEPOUT(float float) (float float) |
    KEEPOUT(float) (float float) ...
PAD string
RESTRICT SIDE {L | R | B | T}]
    [EXACT float [float]
    [RELATIVE float [float] ] [CELL_ORIGIN {L|C|R} {B|C|T}]
    SIDESPAC float [float]]
    [ORIENT {0 1 2 3 4 5 6 7}]
ENDPAD
PADGROUP string {PERMUTE | NOPERMUTE}
string {FIXED | NONFIXED}...
    [RESTRICT SIDE {L | R | B | T}]
    float[$SIDESPAC
PADRING float float float float
PATH string... : float float integer
PATHCONSTRAINT {string... | (string string string string)...}
    float:float:float) (float:float:float) [MONITOR]
PATHCONSTRAINT [PATHPINPAIR] {string... | (string string string string)...}
    float:float:float) (float:float:float) [MONITOR]
REQUIRED_MAX float

```



```

REQUIRED TIME string float
SCANPATH string string string string (string string string string)
        string $string string string) ...
SOFTGROUP string [GROUPX | GROUPY] [STRENGTH float] {string string...} ...
VERSION string string

```

Optional row constraints. These constraints are normally created automatically by the floorplanner program: IFP

```

ROWS INTEGER
ROW float float float float
    MIRROR[| ALIGN CENTER | ALIGN BOTTOM | ALIGN TOP | ALIGN SUPPLY |
    CAPACITY float | CAPACITY ABOVE CENTER float | CAPACITY BELOW CENTER float |
    CAPACITY CHANNEL ABOVE ROW float | CAPACITY CHANNEL BELOW ROW float |
    EXCEPT float float | CLASS integer ]
ROW float float float float VERTICAL
ROW float float float float...
ENDROWS

```

Time Constraints

Itools supports the following types of timing constraints:

- [critical path using wire length constraints.](#)
 - [critical path using timing constraints.](#)
 - [critical path analysis using pin pair constraints.](#)
 - [zero-slack analysis using arrival and required times.](#)
-

Critical Path Using Wire Length Constraints

PATH string... : *number number* [MONITOR]

The simplest form of a timing constraint is one on the total wire length of a user specified critical path. For each critical timing path, the user supplies an upper and lower bound on the length of the path. The penalty assigned for a path p is the amount the length deviates from satisfying the bounds:

$$P_p = \begin{cases} \text{length}(p) - \text{upperBound}(p) & \text{if } \text{length}(p) > \text{upperBound}(p) \\ \text{lowerBound}(p) - \text{length}(p) & \text{if } \text{length}(p) < \text{lowerBound}(p) \\ 0 & \text{otherwise} \end{cases}$$

where the length of a path p is the sum of the half perimeter wire length of all the nets n in the path:

$$\text{length}(p) = \sum_{\forall n \in p} S_x(n) + S_y(n)$$

Since the simulated annealing algorithm ensures that the sum of the lengths of the individual nets in the critical path meet the given bounds, there is no need for the user to partition the path length between the individual signals of the path. Previously reported systems have used net weights on individual nets in an attempt to achieve timing driven placement. This method is superior to net weighting techniques because it overcomes the partition problem and

Optional row constraints. These constraints are normally created automatically by the floorplanner program: IFP

reflects more accurately the true timing constraints to be satisfied.

To add a wire length constraint, add the following line to the *designName.con* file:

PATH net1 [net2]... : *lowerBound upperBound* [**MONITOR**]

where *lowerBound* and *upperBound* specify the length in design units. The **MONITOR** keyword is optional. If **MONITOR** is present, the path is monitored but does not enter the cost function to be optimized. Otherwise, the path cost is optimized during placement and routing.

An example constraint consisting of three nets whose total length must not exceed 1000 units is given below:

PATH signal_in inver signal_out : 0 1000

Critical Path Using Timing Constraints

Although length constraints have successfully been applied to the timing design of several microprocessors, this method fails to account for the differences in drive strength. **Ittools** expresses drive strength using driver source resistance. The propagation time for a signal path is defined as follows:

The arrival time for a path *p* is the summation of all the net delays for the path.

$$T_a = \sum_{\forall n \in p} D_n$$

The delay for a single net *n* is the sum of the intrinsic gate delay associated with the driver of the net *n*, and the product of the equivalent driver resistance, and the total load capacitance seen by the driver.

$$D_n = T_n + R_n C_n$$

where the user has specified an upper (*Tru*) and lower bound (*Trl*) on the required arrival times.

To add a critical path constraint, add the appropriate values of resistance and capacitance for all the pins in the *designName.lib* file and add one or more of the following lines to the *designName.con* file:

```
PATHCONSTRAINT pin_instances times [monitor]
  where
    pin_instances      := pin_instance
                      |= pin_instances pin_instance
    pin_instance      := string
                      |= (string string string string)
    times              := time
                      |= upper_bound
                      |= lower_bound upper_bound
    lower_bound        := risetime falltime
    upper_bound        := risetime falltime
    risetime           := time
    falltime           := time
                      |= (number:number:number)
    time               := (number)
                      |= (number:number:number)
```

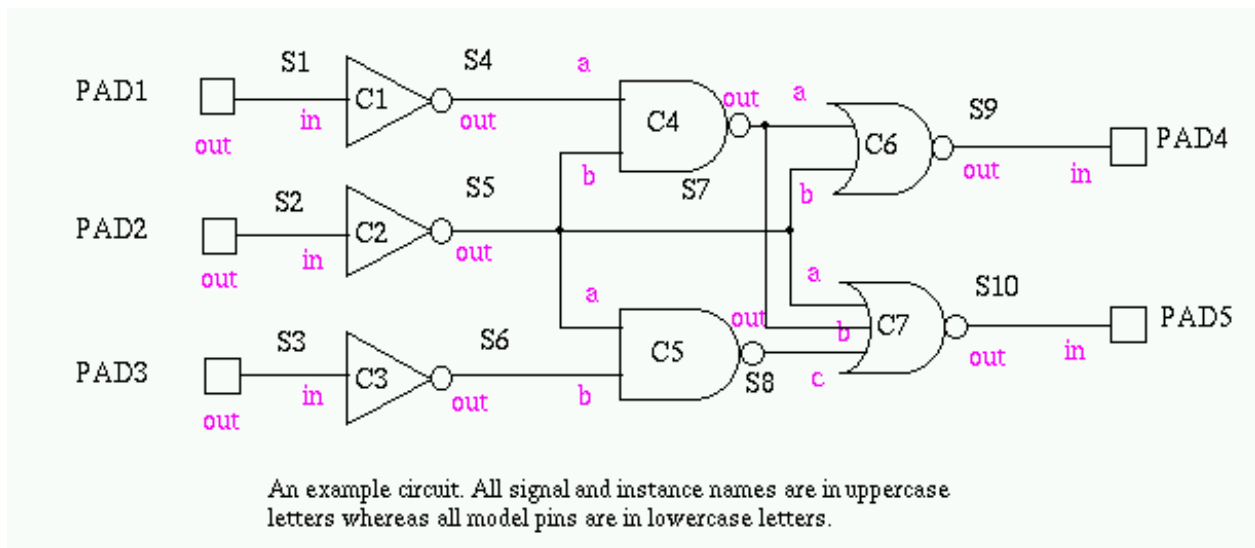
```
monitor_flag      := {MONITOR}
```

A **PATHCONSTRAINT** is a timing constraint on a set of pins. A *pin instance* may be defined using either the "Standard Delay Format" (SDF) or instance pin-pairs styles. In the SDF style, the instance name is separated by the pin name or the hierarchy divider character. The default hierarchy divider character is the forward slash "/" but it may be changed to a period using the **DIVIDER** keyword in the *designName.con* file. The instance name must correspond to the instance name given in the *designName.ckt* file and the pin name must correspond to a pin of that instance in the *designName.lib* file. The first pin in the sequence should be an output pin and the last should be an input pin. This also implies that there should be an even number of pin instances. The second form consists of four strings (*inst1 pin1 inst2 pin2*) surrounded by parentheses. The set of strings describes an output model pin (pin1) on instance 1 connected to an input model pin (pin2) on instance 2. An error message is generated if the pin instances in the *designName.con* file do not correspond to the netlist described in the *designName.ckt* and *designName.lib* files.

The timing constraint value follows the set of pin instances. In the general case, four sets of timing values may be specified – the rise and fall time of the lower bound on the arrival time followed by the rise and fall time of the upper bound on the arrival time. Each time may be specified either as a single real number or a triple set of numbers separated by colons. The triple describes the minimum, nominal, and maximum sets of time. The default is to use the nominal times in optimization; but can be modified using the **TIME** keyword in the *designName.par* file. Several simplifications of the constraint are available. Whenever the lower bound on rise and fall time is zero, the first set of rise and fall times may be omitted. In addition, if the rise and fall time are considered equal, the fall time may be omitted. If only the fall time is to be considered, set the rise time to be zero. The time constraints are specified in seconds. The units may be modified using the **SCALE DELAY** keyword in the *designName.par* file.

If the keyword **MONITOR** is present, the path is monitored but does not enter the cost function to be optimized. Otherwise, the path cost is optimized during placement and routing.

An example of a critical time constraint is given for the following circuit:



A path constraint whose rise and fall delay must not exceed 10 nsecs. is given below:

```
PATHCONSTRAINT(PAD2 out C2 in) (C2 out C4 b) (C4 out C7 b) (C7 out PAD5 in) (10.0)
The constraints below are equivalent representations of the previous constraint.
PATHCONSTRAINT PAD2/out C2/ in C2/out C4/b C7/out PAD5/in (10.0)
PATHCONSTRAINT (PAD2 out C2 in) (C2 out C7a) (C7 out PAD5 in) (10.0:10.0:10.0)
```

It is assumed that the *designName.par* file has the following line:

SCALE DELAY 1E-9

Note that each pin pair constitutes a net. The path could also be specified as consisting of nets S2, S5, S7, and S10. The pin pair specification insures that reconvergent and false paths will be handled properly.

Now suppose we want the upper bound on the rise time to be 10nsecs and the lower bound on the fall time to be 2nsec for the second constraint above. We would enter:

```
PATHCONSTRAINT PAD2/outC2/inC2/outC5/aC5/outC7/cC7/out PAD5/in (0) (2) (10.0) (0)
```

Critical Path Analysis Using Pin Pair Constraints

Often users don't not know the many critical paths in the circuit. Instead, they are concerned with the timing delays between the primary inputs and the primary outputs of the integrated circuit. Between a primary input pin and a primary output pin there may be many unique paths. The shortest non-false path will set a lower bound on the time delay whereas the longest non-false path will determine the upper bound. The timing delay has a parasitic component due to the wiring between components as shown in the equations above. Different placements yield different wiring and thus timing delays. A particular path between two pins, for example, may be the longest path in one placement, but the shortest path in another placement. Therefore, it is important to optimize all non-false paths between the two pins to ensure that timing specifications are met.

To add a critical pin pair constraint, use the same format as a PATHCONSTRAINT except only specify the source and sink ports. **Ittools** will search all the paths between the source and the sink ports automatically. All constraint times include the intrinsic gate delays found in a path. A delay from the input to the output of a gate must be specified in order for a path to be found during the path enumeration. The optional keyword **PATHPINPAIR** may be specified to distinguish paths that must be treated as pin-pairs.

For example, a path constraint whose rise and fall delay must not exceed 10 nsecs. is given below:

```
PATHCONSTRAINT PATHPINPAIR PAD2/out PAD5/in (10.0)
```

Again, it is assumed that the *designName.par* file has the following line:

SCALE DELAY 1E-9

The above constraint is equivalent to the following three constraints:

```
PATHCONSTRAINT (PAD2 out C2 in) (C2 out C4 b) (C4 out C7 b) (C7 out PAD5 in) (10.0)
PATHCONSTRAINT (PAD2 out C2 in) (C2 out C5 a) (C5 out C7 c) (C7 out PAD5 in) (10.0)
PATHCONSTRAINT (PAD2 out C2 in) (C2 out C5 a) (C5 out C7 c) (C7 out PAD5 in) (10.0)
PATHCONSTRAINT (PAD2 out C2 in) (C2 out C7 a) (C7 out PAD5 in) (10.0)
```

Zero-Slack Analysis

Another alternative timing-driven placement methodology is zero-slack analysis. The zero-slack analysis determines the minimum and maximum delays for each circuit which permits the performance specifications to be met. These delays are then transformed into length bounds for each net. This process is more efficient than other timing methodologies since only a single constraint is associated with a net unlike path and pinpair constraints where a net may occur in multiple paths. The user must furnish the **input arrival times** and **output required times**.

The **input arrival time** of a pin instance in the circuit is the minimum time in which a signal propagates through the circuit arriving at that pin. The arrival time of a pin may be constrained by adding either of the following forms:

```
ARRIVAL_TIME string float
ARRIVAL_TIME ( string string ) float
```

The two forms correspond to the two different [pin instance](#) forms. In the first form, the pin instance expressed in SDF style follows the keyword **ARRIVAL_TIME**. The arrival time is expressed as a floating-point number which is scaled by the delay factor found in the parameter file. In the second form, the pin instance is expressed in pin-pair style. For example, the two constraints below specify that the arrival time of pin "in" of instance C1 is 2 nanoseconds:

```
ARRIVAL_TIME C1/in 2
ARRIVAL_TIME (C1 in) 2
```

Again, it is assumed that the *designName.par* file has the following line:

```
SCALE DELAY 1E-9
```

The **output required time** of a pin in the circuit is the maximum time in which a signal may arrive at a pin and still function. The required time of a pin may be constrained by adding either of the following forms:

```
REQUIRED_TIME string float
REQUIRED_TIME ( string string ) float
```

The two forms correspond to the two different [pin instance](#) forms. In the first form, the pin instance expressed in SDF style follows the keyword **REQUIRED_TIME**. The required time is expressed as a floating-point number which is scaled by the delay factor found in the parameter file. In the second form, the pin instance is expressed in pin-pair style. For example, the two constraints below specify that the required time of pin "out" of instance C7 is 12 nanoseconds:

```
REQUIRED_TIME C7/out 12
REQUIRED_TIME (C7 out) 12
```

Again, it is assumed that the *designName.par* file has the following line:

```
SCALE DELAY 1E-9
```

Normally, unconnected input pin instances are assigned the default arrival time of negative infinity (defined by machine constant DBL_MIN). The user may set the default arrival time by following the keyword **ARRIVAL_MIN** with a floating-point number:

```
ARRIVAL_MIN float
```

It is common for users to set the default to 0. For example:

```
ARRIVAL_MIN 0
```

Normally, unconnected output pin instances are assigned the default required time of positive infinity (defined by machine constant DBL_MAX). The user may set the default required time by following the keyword **REQUIRED_MAX** with a floating-point number:

```
REQUIRED_MAX float
```

This parameter is usually not needed if all output pin instances are defined. It should be set to at least the maximum required time in the circuit. For example, the maximum is set to 10 thousand nanoseconds:

REQUIRED_MAX 10000

If **ARRIVAL_MIN** and **REQUIRED_MAX** are both specified, the ZSA algorithm will be invoked even if no pin instances are given. All unconnected input pins will be assigned **ARRIVAL_MIN** and all unconnected output pins will be assigned **REQUIRED_MAX**.

Net Constraints

NET *string* [**IGNORE** | **DO_NOT_GLOBAL_ROUTE** | **CAP_FACTOR** *float* | **REROUTE** | **PRIORITY** *integer* | **RADIUS_FACTOR** *float*]

A net may be ignored during placement if the keyword **IGNORE** is used in conjunction with the **NET** keyword. If a net is ignored, the net wire length is not optimized during placement. For example, to ignore the net VCC during placement use

NET VCC IGNORE

in the *designName.con* file.

NET [CREATE] *string* **GLOBAL** <*modelPinName*> [<*instanceName*>]

A net may be designated as a global net if the keyword **GLOBAL** is used in conjunction with the **NET** keyword. A global net contains one or more *global* pins. A *global* pin is a pin of type **POWER**, **GROUND**, or **SPECIAL**. Global pins are extracted from the library models and do **not** need to (but may) appear explicitly in the netlist. Non global pins are pins which do not have pin type **POWER**, **GROUND**, or **SPECIAL** and thus a member of the set {**INPUT** **OUTPUT** **BIDIRECTIONAL** **FEED** **UNKNOWN**}. In addition, non global pins **must** appear explicitly in the netlist.

A global net has two states in detailed routing: enabled and disabled. When enabled the global net acts as a connected net, non global pins may validly connect to the closest global pin. The global pins are not required to be connected while the global property is enabled. When disabled this special property is not enforced, that is, all global pins must be connected. Disabling this global property allows the global net to be connected among themselves. Enabling and disabling of this command is performed in the detail router thru the use of the **icglobal** command.

The **NET** [CREATE] *string* **GLOBAL** command controls the implicit extraction of global pins from the instantiated models found in the netlist. In the first form of the command, the string occurring after the **GLOBAL** keyword determines the name of the modelPin used in the extraction. All pins which match this model pin name will be added as pins to the specified net. The optional keyword **CREATE** forces the creation of the net if it doesn't exist in the netlist. It is safe to use the **CREATE** keyword even if the signal does exist in the netlist. The second form of the constraint allows the specification of the instanceName so that one can implement multiple power supplies or multiple clock signals. Each instance of a given net must be specified or the row number must be specified as **ICROW#**. For example, if we want rows 1 thru 5 of a row-based design to be connected to the low voltage supply *vlo* we would enter

```
NET CREATE vlo GLOBAL VDD ICROW1
NET CREATE vlo GLOBAL VDD ICROW2
NET CREATE vlo GLOBAL VDD ICROW3
NET CREATE vlo GLOBAL VDD ICROW4
NET CREATE vlo GLOBAL VDD ICROW5
```

The **CAP_FACTOR** keyword allows a net to be weighted during placement. The floating point number that follows

must be greater than zero. The larger the weight, the more priority the net will be given in the placement cost function. By default, the cap factor is 1.0.

The **DO_NOT_GLOBAL_ROUTE** keyword allows a net to be omitted during global routing. Again, to omit the VCC and GND during global routing use

```
NET GND DO_NOT_GLOBAL_ROUTE
```

```
NET VCC DO_NOT_GLOBAL_ROUTE
```

in the *designName.con* file.

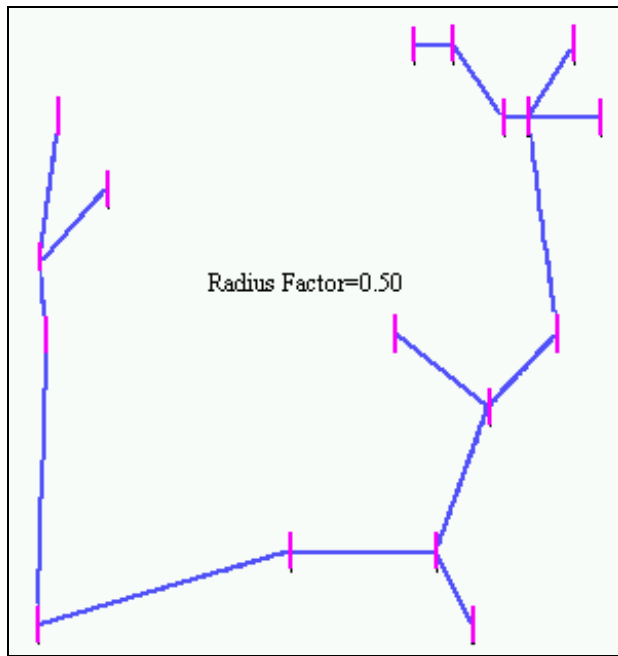
A net may be selectively rerouted during incremental global routing by using the **REROUTE** keyword. Any net which has been specified with the **REROUTE** keyword will undergo optimization during incremental global routing. The reroute constraint has no meaning during a normal (non-incremental) global route.

The keyword **PRIORITY** allows the user to control the assignment of nets to *feed through pins* during global routing. Feed through pins are positions where the global router is able to cross the row of cells. Each net may be assigned a priority by specifying a nonnegative integer after the keyword **PRIORITY**. Nets with the highest priority (largest integer) are assigned first. All nets with the same priority are assigned simultaneously. The default priority for all nets is 0; that is, all nets will be assigned simultaneously. Only a small group of time critical nets should have a positive priority; sequentially assigning feed through pins will degrade the performance of the feed through assignment algorithm.

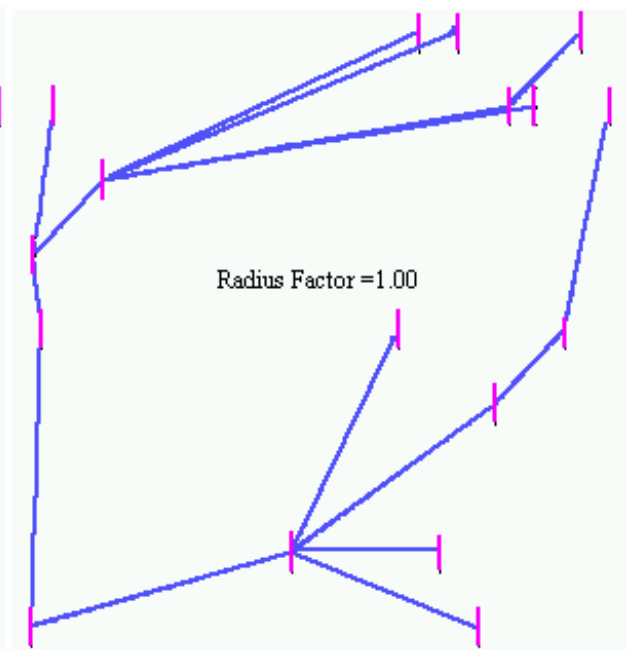
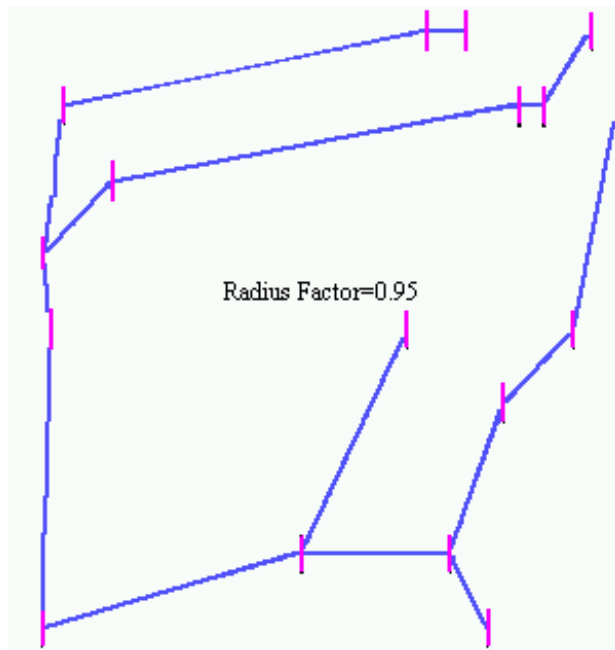
For example, nets A and B will be assigned simultaneously before net C. These three nets will be assigned before the remaining nets.

```
NET A PRIORITY 2
NET B PRIORITY 2
NET C PRIORITY 1
```

The keyword **RADIUS_FACTOR** followed by a floating point number in the interval [0.0 1.0] controls the Steiner tree radius during global routing for the specified net. For proper operation, the net must consist of a single driver and any number of loads. The pin types must be properly defined in the *designName.lib* file. For small radius factors, wire length is more important than the timing from the source to the farthest load. For radius factors approaching 1, the Steiner tree becomes a shortest path tree. The **RADIUS_FACTOR** may also be set globally from the global routing



parameter controls in the



Net Prerouting

```

NET [CREATE] netName[ROUTING | FIXED_ROUTING]
    layBEame (x1 y1) (x2 y2) |
    xRECT) (x2 y2) |
    viaNAme (x y) |
    KEEPOName (x1 y1) (x2 y2) |
    KEEPOUT (x2 y2) ...
    
```

Itools supports prerouting nets through the use of the **NET** *netName* **ROUTING** and **NET** *netName* **FIXED_ROUTING** constraints. Nets which have the **FIXED_ROUTING** property may not be ripped up by the detail router whereas nets with the **ROUTING** keyword may be removed and rerouted by the detail router. The optional **CREATE** keyword allows one to create new nets which do not occur in the netlist. Currently, three types of prerouted

objects are supported: rectangular routing, vias, and keepouts. A routing rectangle begins with the **RECT** keyword followed by the layer name. If the layer name is not specified, it is assigned to the last defined layer name for this net. After the layer definition follows the two points of the rectangle, lower left followed by upper right. Each point is enclosed by parenthesis. A via is specified by the **VIA** keyword followed by the name of the via followed by the center point of the via. A keepout is specified by the **KEEPOUT** keyword followed by the layer name. Like the rectangle, keepouts may optionally omit the layer name if one has previously been defined. In addition, the boundary of the keepout is specified by the lower left and upper right points enclosed by parenthesis. It should be noted that keepouts in this system do not include spacing constraints; it is assumed that routing may abut any keepout. In addition, keepouts will not appear in the output; only rectangles and vias will be present. A keepout may be associated with any net with more than one pin. The net chosen does not affect the result. For example, the following constraint assigns keepouts to the ground signal:

```
NET gnd FIXED_ROUTING
    KEEPOUT metall ( 2.35 -2.85 ) ( 398.35 -2.70 )
    KEEPOUT metall ( 2.35 -0.20 ) ( 398.35 1.85 )
    KEEPOUT metall ( 2.35 9.75 ) ( 398.35 11.55 )
    KEEPOUT metall ( 2.35 11.15 ) ( 398.35 12.95 )
    KEEPOUT metall ( 2.35 33.50 ) ( 398.35 35.30 )
```

The detail router has the ability to ignore prerouting. If you intend to use prerouting, do not use the **-noprerouting** option in the routing *do* script.

Fixed Constraints

```
FIXED string [{INITIALLY | APPROXIMATELY | RIGIDLY | NEIGHBORHOOD | VALID_ORIENTS}]
    { (number number)
    | (number BLOCK number)
    | (number FROM string BLOCK number)
    | (number FROM string number FROM string) }
CELL_ORIGIN {L|C|R} {B|C|T}
ORIENT [integer]
VALID_ORIENTS integer ... ]
```

Preplacing a cell is optional. The user should be aware that in general fixed constraints reduce the quality of the final placement. In fact, initial placements do not improve the quality of the **itools** placement tools.

A fixed constraint is constructed with the keyword **FIXED** followed by one of five options for the subsequent keyword: **INITIALLY**, **RIGIDLY**, **APPROXIMATELY**, or **NEIGHBORHOOD** or **ORIENT**. If no option is specified, the cell is to remain fixed at the specified location. **Ittools** will place the cell as close as possible to this coordinate. If the **INITIALLY** keyword is specified, the cell is placed at the coordinates specified at the start of the placement algorithm. However, the cell is free to move during the placement. This constraint is useful in supplying an initial placement to **itools** for calculating wire length[1]. If the **APPROXIMATELY** fixed option is specified, **itools** will initially place the cell at the coordinates given. During the placement algorithm, the cell is not moved directly but other cells may displace it causing it to deviate from its original spot. In the approximate mode, the cell will stay in its initial row. If the **RIGIDLY** keyword is supplied, **itools** attempts to align the edge of the cell to the exact location specified by adding spacer cells if necessary. The **NEIGHBORHOOD** keyword constrains a cell to a set of rectangular areas for standard cells and a single rectangular area for macro cells. The **VALID_ORIENTS** option allows the user to override the inherited valid orientations for a specific instance.

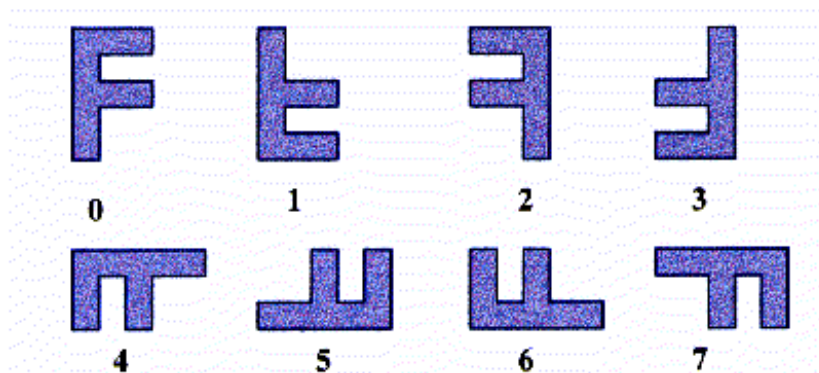
The cell's location may be specified in four ways. All coordinates are relative to the center of the cell by default. The placement origin may be changed to a cell edge using the **CELL_ORIGIN** option. The first method of locating the cell uses the x-y coordinates of the center or edge of the cell. It is applicable to all types of cells. The second and third methods apply only to row-based cells. In the second method, the first number after the parenthesis specifies the

x-coordinate of the cell's center. The integer after the **BLOCK** keyword assigns the row number to the cell. This integer represents a row number, in which the rows are numbered starting from the bottom of the layout. In the third method, the number represents how far from the left or right end of a row the cell should be placed. Following the keyword **FROM** the user selects either **LEFT** or **RIGHT** to indicate whether the number represents a distance from the left or right end of the row. If the user selects **LEFT**, the distance is the amount the center (edge) of the cell is to be placed from the left end of the block (row). On the other hand, if the user selects **RIGHT**, the distance is the amount the center (edge) of the cell is to be placed from the right end of the block. The integer after the **BLOCK** keyword assigns the row number to the cell.

The final location method is restricted to macro cells. It is used to specify a corner of the neighborhood rectangle. The first number represents how far from the left or right end of the core region the cell should be placed. Following the keyword **FROM** the user selects either **LEFT** or **RIGHT** to indicate whether the number represents a distance from the left or right end of the core region. If the user selects **LEFT**, the distance is the amount the cell center is to be placed from the left end of the core. On the other hand, if the user selects **RIGHT**, the distance is the amount the center of the cell is to be placed from the right edge of the core. The second number represents how far from the top or bottom edge of the core region the cell should be placed. Following the keyword **FROM** the user selects either **TOP** or **BOTTOM** to indicate whether the number represents a distance from the top or bottom edge of the core region. If the user selects **TOP**, the distance is the amount the cell center is to be placed from the top edge of the core. On the other hand, if the user selects **BOTTOM**, the distance is the amount the center of the cell is to be placed from the bottom edge of the core.

The **CELL_ORIGIN** option allows the placement location to be referenced by the cell's edges rather than the cell's center. The **CELL_ORIGIN** keyword is followed by the x-reference and y-reference strings. The x-reference string may be one of **L**, **C**, or **R**. This allows the x coordinate of the placement origin to be the left, center, or right edge of the cell respectively. Similarly, the y-reference sets the y coordinate placement to refer to the bottom, center, or top edge of the cell through the use of **B**, **C**, or **T**.

The orientation of the cell may be specified by the integer following the **ORIENT** keyword. The orientation numbering scheme is presented in the figure below, with orientation 0 being the base. Below are valid examples of fixed constraints.



The **VALID_ORIENTS** constraint is provided as a mechanism to override the default inherited model valid orientations. There is no need to specify a fixed location if constraining the orientation is the only need. However, if the user needs to constrain a specific instance to a set of orientations which are different than the model constraints and a fixed location is desired, the **VALID_ORIENTS** clause may be added to any of the other fixed constraints. There is no need to specify a **VALID_ORIENTS** clause if the **ORIENT** option has been specified. An override of the model orientation constraints (former case) is specified as

```
FIXED <instname> ORIENT orient1 orient2 ...
```

whereas an example of overriding valid orientation in addition to fixing a position (latter case) is specified as

```
FIXED <instname> (location clauses) VALID_ORIENTS orient1 orient2 ...
```

Fixed Constraint Examples

```
FIXED inst1 (234 954) ORIENT 0
```

Cell (of any type) whose center is fixed at (234,954). The orientation of the cell is unconstrained.

```
FIXED standard1 INITIALLY (234 BLOCK 4) CELL_ORIGIN L C ORIENT 2
```

Row-based cell in row 4 whose left edge is placed initially at x = 234. The orientation of this cell is fixed at orient 2.

```
FIXED standard2 RIGIDLY (234 FROM RIGHT BLOCK 4) CELL_ORIGIN R C
```

Row-based cell in row 4 whose right edge is rigidly placed 234 units from the right edge of the row. If the coordinates of row 4 are (0, 0) (1000, 100), the right edge of the cell would be at 766.

```
FIXED standard3 NEIGHBORHOOD (200 BLOCK 1) CELL_ORIGIN L C (1000 BLOCK 3) CELL_ORIGIN L C ORIENT 2
```

Row-based cell which is constrained to remain in a neighborhood. The left edge of the cell must remain between 200 and 1000 and the cell must remain in row 1 thru 3. In addition, the cells orient is fixed at 2.

```
FIXED standard4 NEIGHBORHOOD (200 200) (1000 1000) (500 1200) (1000 2000)
```

Row-based cell which is constrained to be in one of two possible neighborhoods: either the neighborhood area bounded by the rectangle (200,200) (1000,1000) or the neighborhood bounded by the rectangle (500,1200) (1000,2000). The cell's orientation is unconstrained.

```
FIXED macro1 (0 FROM LEFT 0 FROM BOTTOM) (100 FROM LEFT 100 FROM BOTTOM)
```

Macro cell constrained in the neighborhood (0,0) (100,100). Placement origin is the center of the macro and the orientation is unconstrained.

Fixing Orientations

As an example, suppose we have a row-based model named *flipflop*. In addition, suppose there are four instances of this cell in the netlist description named *ff1*, *ff2*, *ff3* and *ff4*. Now let's look a few constraints. Suppose we want to restrict all instances of *flipflop* to orientations 2 and 3. We could add four fixed constraints but it would be easier and more appropriate to change the default valid orientations for a row-based cell. We do this by adding the **ORIENT** constraint to the model definition in the library description.

```
MODEL flipflop TYPE STANDARD
  VERSION 1
  BOUNDARY (0 0) (30 10)
  ORIENT 2 3
```

```
  .
  .
  .
```

```
ENDMODEL
```

All flipflop instances are required to be either orientation 2 or orientation 3.

Next, we wish to override the valid orientations of *ff1* so that only orientations 0 and 1 are valid. In addition, there is no constraint on where the instance may be placed. Assuming the previously defined model definition, this would mean

ff1 is restricted to orientations 0 and 1 whereas *ff2*, *ff3*, and *ff4* are constrained to only orientation 2 and 3. We would add the following constraint to the *circuitName.con* file:

```
FIXED ff1 VALID_ORIENTS 0 1
```

Finally, we need to constrain *ff4* to 512 from the right of block 2 and with the addition restriction that the valid orientations of 0 and 2. We would add the following constraint to the *circuitName.con* file:

```
FIXED ff4 RIGIDLY (512 FROM RIGHT BLOCK 2) CELL_ORIGIN R C VALID_ORIENTS 0 2
Row-based cell in row 2 whose right edge is rigidly placed 512 units from the
right edge of the row and whose orientations are restricted to 0 and 2. If the
coordinates of row 2 are (0, 0) (1000, 100), the right edge of the cell would be at 488.
```

Notice that there is a problem with this definition if the row 2 is mirrored because both the 0 and 2 are invalid when the row is mirrored. There is no need to add fixed valid orient constraints to deal with mirrored rows; the placement and routing tools understand the required orientation constraints automatically.

Engineering Change Orders

ECO_ADDED_CELL *string*

ECO_DELETED_CELL *string*

Another optional feature is the Engineering Change Order (ECO) handling capability. This feature is useful when a small perturbation is needed in the design. Additional standard cells may be added or deleted in the netlist. The placement of the "old" cells should remain essentially in the same positions as in their previous run. An initial placement must be given through the restart mechanism. The initial positions of the cells using the result of a previous placement are specified using the *designName.pll_in* file. The program will look for the existence of the *designName.pll_in* file iff a) **ECO_ADDED_CELL** or **ECO_DELETED_CELL** constraints are found in the *designName.con* file or b) the *ICSC*eco_placement_improve : on* appears in the *designName.par* file.

Engineering change orders may be accomplished by entering the keyword **ECO_ADDED_CELL** followed by the new cell instance which was added to the netlist. Any **ECO_ADDED_CELL** will have its initial placement information completely disregarded. Instead, a quick procedure which is a variant of force-directed placement is used to place the new cell to minimize total wire length. For example, to add a new cell called *inst_fix*, add the following line to the *designName.con* file:

```
ECO_ADDED_CELL inst_fix
```

A placement improvement algorithm is normally performed after the force-directed placement. This may be avoided by specifying only fixed or rigidly fixed constraints in the *designName.con* file or by placing *ICSC*eco_placement_improve : off* in the *designName.par* file. Avoiding placement improvement will minimize the movement of the original cells at the expense of the solution quality.

It is possible to control the movement of the "old" or constrained cells during ECO placement improvement using the *ICSC*eco_constraint_factor* in the *designName.con* file. By default, the eco constraint factor is set to -1 which adds a **FIXED** constraint to each of the "old" cells. This keeps the cell at approximately the requested place but may move minimally in order to make sure cells do not overlap. The user may fix the cell **RIGIDLY** (which forces the placement of the cell at the exact specified location) by setting the eco constraint factor to -2. These negative settings of the eco constraint factor limit the range which "old" cells can move and disables placement improvement. When the eco constraint factor is non-negative, the program will add **NEIGHBORHOOD** constraints for the "old" cells. The eco

constraint factor controls the size of the neighborhood. The "old" cells may move anywhere within the neighborhood bounds centered at the original placement of the cell. If the eco constraint factor is zero, cells must remain in their row but may move within a short distance of their original positions. If the eco constraint factor is increased to 1, the constrained cells may stay in the same row or move to the row above or below the current row. In addition, the x dimension of the neighborhood is increased as well. As the eco constraint factor is increased, the size of the neighborhood is increased. As the number of ECO cells are increased, the eco constraint factor should be increased. From our experience, the best results occur when the eco constraint factor is in the range of [0..2]. Note: the user may constrain individual cells by specifying individual fixed constraints in the *designName.con* file.

As a convenience function, cells may be deleted from the netlist without modifying the *designName.ckt* or *designName.pll_in* files. Just add **ECO_DELETED_CELL** followed by the deleted instance name in the *designName.con* file. An error will be issued if the cell does not exist in the *designName.ckt* file.

Class constraints

[**CLASS** *string integer...*]

The **CLASS** feature allows the user to constrain each row-based instance to a set of block classes. The keyword **CLASS** is followed by the instance name, followed by a list of integers -- each one of which is a block class as defined in the Genrows program. The block class definitions are generated by the Genrows program and output into the *itools.con* file. Genrows will allow the user to associate a class number with any of the rows as shown in the examples below. The blocks listed in the *itools.con* file are listed from the top but numbered from the bottom on the screen. Classes are only valid for row-based instances.

EXAMPLE:

CLASS latch 8 5 9

left ...

.
.
.

This implies that the instance named *latch* must be confined to the rows whose block class is one of 8, 5 or 9. Keep in mind that any number of rows may have been specified as having a given block class.

If a cell is to be allowed in any row, then simply don't include a constraint.

EXAMPLE 2:

designName.con file:

```
ROWS 6
ROW 0 11      123      37      CLASS 1 (first row)
ROW 0 63      123      89      CLASS 1
ROW 0 115     123      141     CLASS 2
ROW 0 187     123      193     CLASS 2
ROW 0 219     123      245     CLASS 3
ROW 0 271     123      297     CLASS 3 (last row)
```

Suppose that the following constraint was added in the `designName.con` file:

```
CLASS instance1 1 2
```

This specifies that `instance1` is confined to rows 1 through 4. Further, suppose that the description of a row-based instance includes the following line:

```
CLASS instance2 1 3
```

This specifies that the instance is confined to rows 1 or 2, or, 5 or 6. Note that a instance without any constraint can go into any of rows 1 through 6.

Note: currently class constraints are not obeyed during hierarchical placement. In addition, class and fixed constraints are mutually exclusive.

Version initialization

[**VERSION** *string string*]

The version initialization constraint allows the user to initialize a model version for an instance. The model version is specified after the instance is named. The default model version is the first version specified for the model in the *designName.lib* file. For example, to initialize the model version name *nandv2* for instance *inst3*, enter the following in the *designName.con* file:

```
VERSION inst3 nandv2
```

Core and Pad Region Definitions

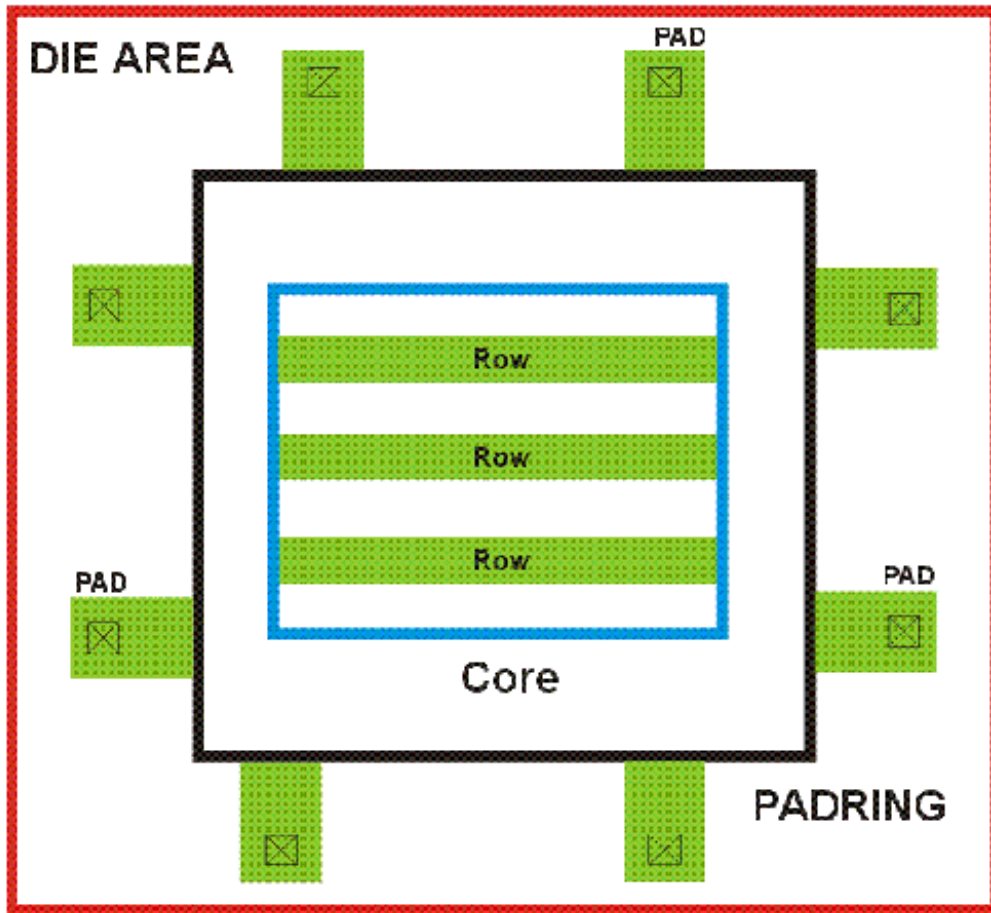
```
[CORE x1 y1 x2 y2]
[PADRING x1 y1 x2 y2]
[DIEAREA x1 y1 x2 y2]
```

The optional keyword **CORE** allows the description of the core or non-pad area of the die. The core rectangle contains all row-based areas and macro cell areas. The core area is specified by four numbers: lower left x, lower left y, upper right x, and upper right y, respectively. The blue rectangle in the figure below shows the demarcation line of the core.

The optional keyword **PADRING** allows the description of the pad area of the die. The pad placement algorithm will abut all of the pads against the pad ring. The core area is specified by four numbers: lower left x, lower left y, upper right x, and upper right y, respectively. The black rectangle in the figure below shows the pads abutting against the pad ring.

The area between the core and the pad ring is the I/O routing region and the specification of the pad ring and core can define the size of this region.

The pad ring must always be equal or larger than the core; otherwise, an error is generated. If a core constraint is present, but the pad ring constraint is omitted, the pad ring will be set to the core constraint.



The optional keyword **DIEAREA** allows the description of the complete die area. The DIEAREA rectangle includes all fixed and placeable objects. All routing must lie inside the DIEAREA rectangle. The rectangle is specified by four numbers: lower left x, lower left y, upper right x, and upper right y, respectively. A keepin constraint will be automatically generated from this data in the detail router to insure that routing does not occur outside this area. The die area is shown as a red rectangle in the figure above.

Pad constraints

```
[PAD string
  [RESTRICT SIDE {L | R | B | T}]
  [SIDESPACE float [float]]
  [RELATIVE float] ] [CELL_ORIGIN {L|C|R} {B|C|T}]
  [EXACT float float [float float] ORIENT {0 1 2 3 4 5 6 7}]
ENDPAD]
[PADGROUP string {PERMUTE | NOPERMUTE}
  string {FIXED | NONFIXED}...
  [RESTRICT SIDE {L | R | B | T}]
  [SIDESPACE float [float]]]
```

Pad instances may be constrained using a description format that begins with the keyword **PAD** followed by the instance name. The pad may be constrained to a side of the chip using the keywords **RESTRICT SIDE** followed by a string containing the characters:

L , T , R , or B

If the **RESTRICT SIDE** option is given for the pad, **itools** is constrained to place the pad on the given side or sides of the chip. **L**, **T**, **R**, and **B** stand for the left, top, right, and bottom sides of the chip, respectively. More than one side may be specified. For example, **RESTRICT SIDE LR** allows the program to place the pad on either the left or right side of the chip where its final position is determined by the side which minimizes wire length and satisfies pad area constraints. If no side restriction is present, **itools** is free to place the pad on the side which minimizes wire length for that pad and satisfies all other constraints.

The optional keyword **SIDESPAC** followed by a floating point number allows the user to place the pads at a particular relative position. For left or right side pads (**RESTRICT SIDE L** or **R**), the floating point number represents a decimal fraction which specifies the fraction of the bottom-to-top span of the core of the chip to which the center of the pad is to be placed. For example, if the vertical span of the chip is 10000 microns, and if the sidespace floating point number is 0.10 and a pad is to be placed on the left side, then the pad will have its y-center placed at 1000 microns from the bottom edge of the core. For bottom or top side pads (**RESTRICT SIDE B** or **T**), the floating point number represents a decimal fraction which specifies the fraction of the left-to-right span of the core to which the center of the pad is to be placed. For example, if the horizontal span of the core is 10000 microns, and if the sidespace floating point number is 0.70 and a pad is to be placed on the top side, then the pad will have its x-center placed at 7000 microns from the left edge of the core. It should be noted that the sidespace coordinate system is relative to the core definition. It may be specified with the **CORE** definition.

Note that the **SIDESPAC** keyword can be used to force an order on a given side. If the sidespace keyword is not present for a pad, by default, **itools** will place the pad so as to minimize wire length. In summary, to constrain a pad to a side or sides of a chip, use the **RESTRICT SIDE** option and to constrain a pad to a particular relative position, using the **SIDESPAC** option. The user may also use padgroups to specify an ordering. See below for details.

The optional keyword **SIDESPAC** followed by two floating point numbers allows the user to set a valid window for pad placement. The first number represents the lower bound of the window and the second floating point number represents the upper bound. Again, each floating point number represents a decimal fraction which specifies the fraction of the left-to-right (bottom-to-top) span of the core to which the center of the pad is to be placed. It should be clear that the first form of the sidespace parameter is a compact form for the case when the lower bound equals the upper bound.

The **RELATIVE** option works in a similar manner to the **SIDESPAC** option except that the floating point number following the **RELATIVE** keyword describes the distance along the specified chip side rather than the fraction of the side. This option **must** be used with a side restriction. The origin for pads on the left and bottom sides is the lower left corner of the pad region. The origin for pads on the top side is the upper left corner of the pad region while the origin for the right pads is the lower right corner of the pad region. As a convenience, the user may specify a second coordinate as well. This allows one to convert **EXACT** pad constraints into **RELATIVE** pad constraints just by changing the keyword from **EXACT** to **RELATIVE**. The program will determine which coordinate is fixed relatively and which coordinate is flexible. Optionally, the user may add the **CELL_ORIGIN** keyword followed by any of {C, L, R, B, T} which allows for the user to specify the placement origin of the pad cell itself. By default, the cell origin is assumed to be the center of the cell boundary.

The **EXACT** option allows the user to set the exact location of the pad. The **EXACT** keyword is followed by two or four floating point numbers which describe the position of the pad. If two floating point numbers follow the keyword, they are interpreted to be the x and y location of the center of the pad. The four floating point numbers represent the lower x, lower y, upper x, and upper y coordinates of the pad, respectively. The **EXACT** option should be used in conjunction with a side restriction and/or an **ORIENT** option so that pad rotation can properly be performed.

The **ORIENT** option allows the user to specify the orientation of the pad using the [rotate the pad automatically](#).

As an example, consider the following four pads: *pad1*, *pad2*, *pad3*, and *pad4*. The first pad is unconstrained and may be placed on any side. There is no constraint in the *designName.con* file. The program will attempt to place the pad in such a way that the wire length to the core is minimized. This is useful for subchips where the optimal I/O point is unknown. The second pad is constrained to be placed either on the top or bottom side of the core whereas the third and fourth pads must be placed on the left side of the core. In all cases, the pads are unordered on that side.

```
PAD pad2 RESTRICT SIDE TB
PAD pad3 RESTRICT SIDE L
PAD pad4 RESTRICT SIDE L
```

The example below shows the same four pads as in the example above, however, in this case the user has requested that the pads appear at specific relative positions. Now *pad1* may occur on any side at 50% of the span of that side. Note that pads *pad3* and *pad4* are ordered bottom to top on the left side of the core. Also note that *pad2* may be placed on either the **T**, **B**, or **R** sides in a window starting at 30% and ending at 50% of the side's span. In the case of *pad4*, it will be placed at 20 units from the bottom left corner of the pad region.

```
PAD pad1 SIDESPAC 0.5
PAD pad2 RESTRICT SIDE TBR SIDESPAC 0.3 0.5
PAD pad3 RESTRICT SIDE L SIDESPAC 0.8
PAD pad4 RESTRICT SIDE L RELATIVE 20
```

Another way to order a group of pads is with the **PADGROUP** construct which allows the user to specify the common properties and ordering rules of a group of I/O pads. The rules are specified with the properties **FIXED** and **PERMUTE**. **PERMUTE** is a property of pad groups, and **FIXED** is a property of the pad group members. If a pad group has the **PERMUTE** property, then members of this group can exist in two configurations: the given ordering or its reverse. This option is useful for placing I/O buses which require that the signals be placed in ascending or descending rank. If **NOPERMUTE** is specified, then the ordering rules are decided by the members' **FIXED** properties. If a pad is fixed, then its rank in the group cannot be changed; otherwise, it can be moved freely within the group. It should be clear that if a pad group has the **PERMUTE** property, the pad member's **FIXED** property is ignored.

Pad grouping is specified using the keyword **PADGROUP** to begin the definition of the member pads. The keyword **PADGROUP** is followed by a user specified group name. Pads belonging to this group must have unique names, and the keyword **FIXED** or **NONFIXED** must follow each pad name. The pad name used in the pad group must be declared in the *designName.ckt* file. Each pad can only belong to one pad group directly. The pads belonging to the pad group must be listed in order: from left to right for **T**, **B**, and **TB** side restrictions, and bottom to top for **L**, **R**, and **LR** side restrictions.

By default, members of the pad groups are placed contiguously. In other words, no nonmember pads can be placed between the member pads. To change the setting to noncontiguous, the keyword [contiguous pad groups](#) must be turned off in the *designName.par* file. In this case, nonmember pads will be placed between the pads.

Pad groups can have restrictions just like single pads. Furthermore, they can be used in subsequent pads groups just like ordinary pads using the padgroupname instead of a padname in the list of pads. As with ordinary pads, pad groups nested within another group must have already been declared.

Below is an example of a pad group consisting of two ordinary pads and a previously defined padgroup which is to be placed in either increasing or decreasing order:

```
PADGROUPPERMUTE
pad1 NONFIXED
group1 NONFIXED
pad2 NONFIXED
```

Below is an example of a pad group consisting of three ordinary pads which can be placed in any order but is restricted to the top side. Notice that the padgroup property is **NOPERMUTE** and that all pad group members are **NONFIXED**.

```

PADGROUP NOPERMUTE
  pad1 NONFIXED
  pad3 NONFIXED
  pad2 NONFIXED
  RESTRICT SIDE T

```

When [contiguous_pad_groups](#) is off in the parameter file, this constraint is exactly equivalent to

```

PAD pad1 RESTRICT SIDE T ENDPAD
PAD pad2 RESTRICT SIDE T ENDPAD
PAD pad3 RESTRICT SIDE T ENDPAD

```

If the keyword **contiguous_pad_groups** is enabled, the two sets of constraints will only be equivalent if there is only one pad group on a given side and all pads of that side are contained in the pad group. In the example, all top pads must reside in pad group *group3*.

Cluster Constraints

CLUSTER *string* {PERMUTE | NOPERMUTE} *string* ... ENDCLUSTER

A cluster constraint allows the user to group cell instances together so to be placed and routed as one entity. A cluster constraint starts with the **CLUSTER** keyword followed by a string denoting the name of the cluster. The cluster name must be unique. Following the cluster name is the permutation type which may be either **PERMUTE** or **NOPERMUTE**. If **NOPERMUTE** is given the order within the cluster is fixed by the order in which the instances are specified; if **PERMUTE** is given the place and route programs are free to rearrange the order of the cells in the cluster. The members of the cluster are specified by listing their instance names. The cluster is terminated by the use of the **ENDCLUSTER** keyword. Clusters may be defined hierarchically but they must be defined before they are used in a definition.

```

CLUSTER c1 PERMUTE
  |insta
  |X4
ENDCLUSTER

CLUSTER c2 PERMUTE
  |instb
  |X4
  instc
ENDCLUSTER

CLUSTER c3 NOPERMUTE
  c1
  c2
  instd
ENDCLUSTER

```

The example above prescribes three clusters. Clusters *c1* and *c2* are both defined as permutable clusters where cluster *c3* may not be rearranged and consists of clusters *c1*, *c2*, and primitive instance *instd*. Currently, itools only supports row-based clusters but this will change in the future.

Scanpath Constraints

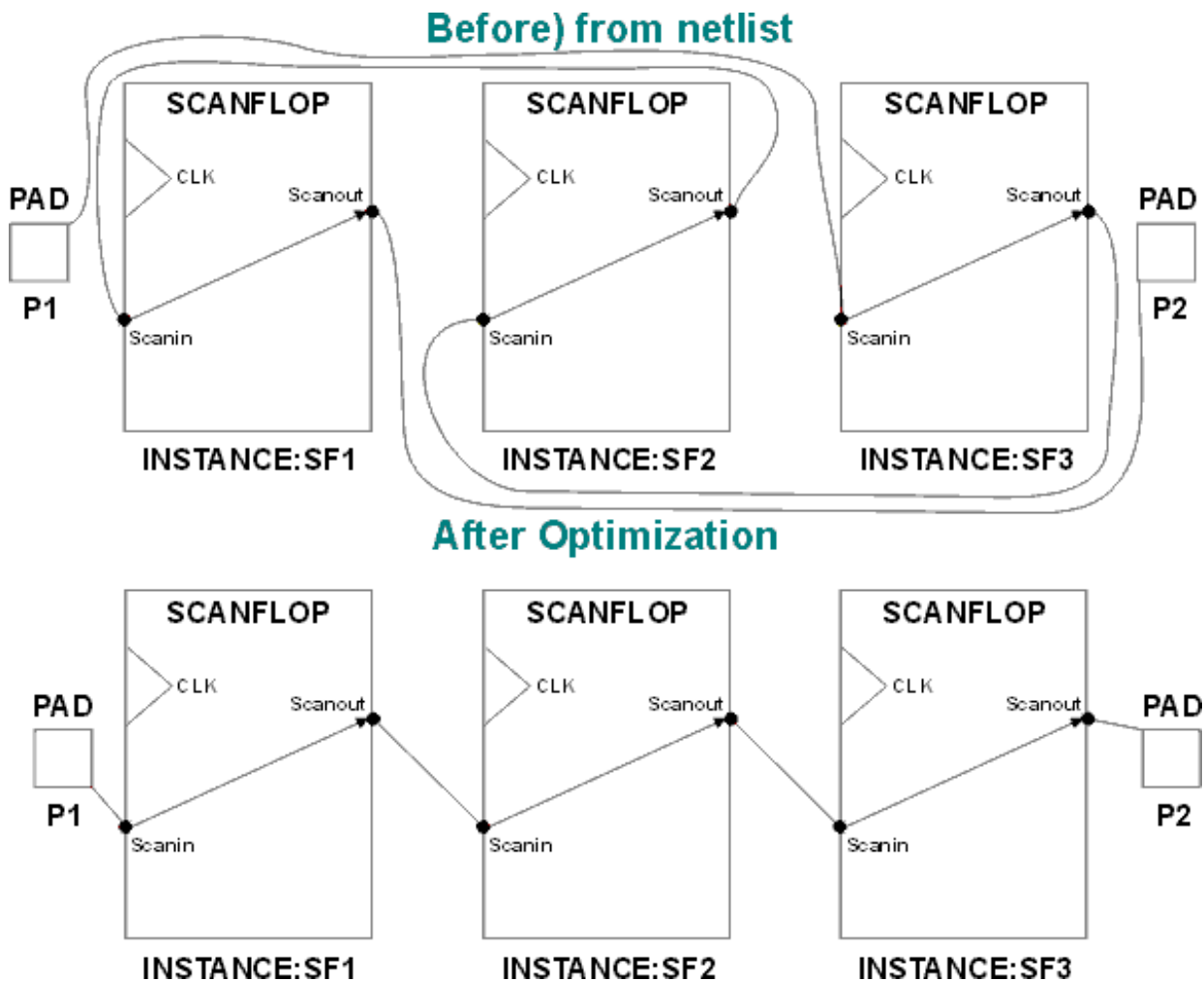
SCANPATH *startInstance startModelPin endInstance endModelPin (scanFlopInstance startFlopPath scanFlopInstance endFlopPath) ...*

where

SCANPATH *string string string string (string string string string) (string string string string) ...*

The **SCANPATH** keywords begins a scanpath constraint. There may be any number of scanpath constraints. Each scanpath has a definitive starting and ending pin. The first pair of strings following the **SCANPATH** keyword describes the starting pin of the scanpath. The starting pin is defined by its instance name followed by its model pin name. The second pair of strings define the ending pin of the scan chain. Again, it is defined by its instance name followed by its model pin name. Next, the set of scan flop components are defined. Each component is a path from the input of a scan flop to its output. This is specified by two *instance – model pin* pairs enclosed by parentheses. (Note: it is possible to specify different instances in cases where the user has prerouted a set of flip-flops). Below is an example of a scanpath constraint. The constraint would be specified as:

```
SCANPATH P1 PAD P2 PAD
(SF1 Scanin SF1 Scanout)
(SF2 Scanin SF2 Scanout)
(SF3 Scanin SF3 Scanout)
```



The top frame of the figure shows the configuration after placement as the scanpath nets are ignored during placement and have no influence on the positions of cells. The location of instances *SF1*, *SF2*, and *SF3* are determined by the wiring and timing costs of other nets which connect to these instances but are not shown for clarity. The bottom frame of the figure shows the configuration after the scanpath algorithm has been run during global routing.

Softgroup Constraints

SOFTGROUP *groupName* [GROUPX | GROUPY] [STRENGTH *float*] {*instanceName1 instanceName2...*} ...

The **SOFTGROUP** keywords begins a softgroup constraint. A softgroup defines a soft clustering of cells during the placement program. There may be any number of softgroup constraints. Each softgroup should have a unique *groupName*. The group name will be used to display the group in the placement program. Normally, the softgroup's wire length is minimized in both the x and y directions. The optional **GROUPX** and **GROUPY** keywords allow groups to be constrained in a single dimension. The optional **STRENGTH** keyword followed by a positive real number modifies the relative importance of the softgroup. The default softgroup strength is 1.0. The list of instances comprising the softgroup completes the constraint. For example, the constraint below defines a soft group named *a* which consists of three instances *insta*, *instb*, and *instc* whose bounding box should be minimized and whose relative strength is twice the default :

```
SOFTGROUP a STRENGTH 2.0
          insta instb instc
```

Equivalent Group Processing

Equivalent Group processing allows the placement program to exchange elements of equivalent sets in order to achieve a higher quality placement. Placement quality should be increased because the placement program has more degrees of freedom. Two types of equivalent sets are possible: pin sets and net sets.

Equivalent Pin Constraints

Pin sets define equivalent pins which may be exchanged freely between the sets. The size of the equivalents sets must all be equal. The special keyword **EQUIV_NC** may be used as a placeholder in a set in order to insure the set size requirement. The equivalent pin set may be defined using either the instance pinname form or the SDF form. The format for the constraint is

```
EQUIV PINS group_name (instname1/pin1 instname2/pin2 ... instname_k/pin_k)
                      (instname_k+1/pin_k+1 ... instname_2k/pin_2k) ...

                      or equivalently

EQUIV PINS group_name ((instname1 pin1) (instname2 pin2) ...
                      (instname_k pin_k))
                      ((instname_k+1 pin_k+1) ... (instname_2k pin_2k))...
```

where

groupname -- is an arbitrary string used to
define each group

(instname1 pin1) is the instance pin name form
of a global pin in the netlist. The pin can be
uniquely determined by specifying the instance
reference name and the model pin of the instance.

Itools Documentation

instname1/pin1 is the SDF form of a global pin in the netlist. Again, the pin can be uniquely determined by specifying the instance reference name and the model pin of the instance. The default sdf hierarchy divider character is '/'. It can be changed to other characters using the DIVIDER definition preceding any EQUIV constraint.

Equivalent Net Constraints

A net set is a special case of a pin set. A net set is a set of nets which are interchangeable. Furthermore, each net has exactly one driver and one or more loads. The placement program is allowed to exchange any/all pins of the set between the various nets of the defined set in order to improve placement quality. The placement program is constrained to place no more than MAXCAP capacitance on any one net. In addition there must be one driver per net. This constraint is very useful in generating clock trees as it states that all the leaves of the clock tree are equivalent.

The format of the net set equivalent constraint is:

```
EQUIV NETS group_name MAXCAP < value > netname1 netname2 ...
```

where

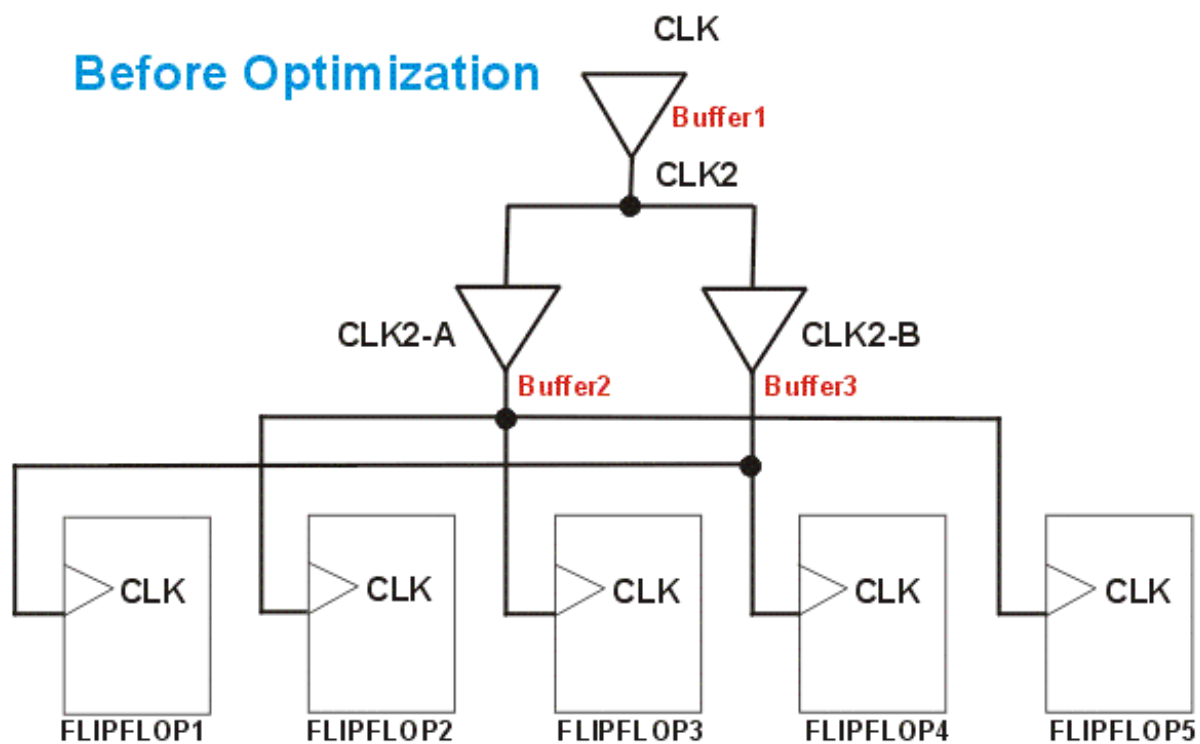
groupname -- is an arbitrary string used to define each group

netname1 .. netnameN -- are valid design netnames

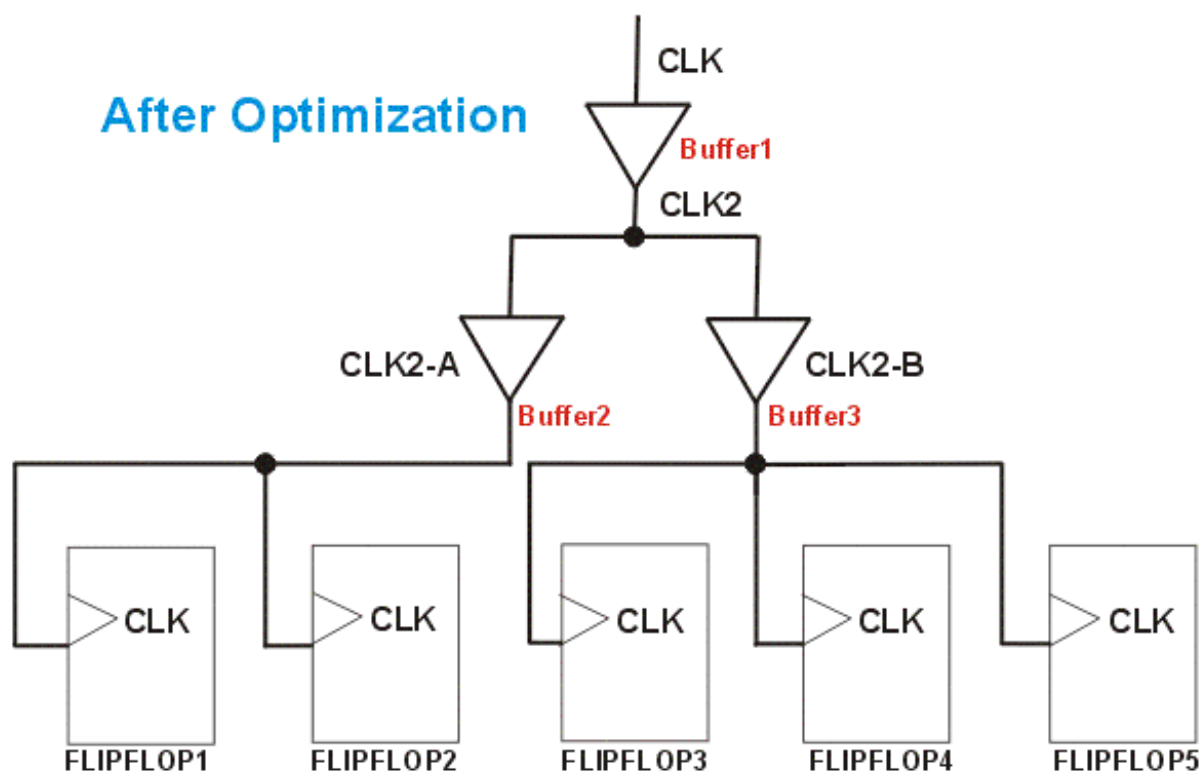
MAXCAP <value> -- defines the maximum capacitance allowed for any nets of the group. This lumped capacitance is the sum of the gate and wire capacitances.

Below are two figures showing the benefits of the equivalent net constraints:

Before Optimization



After Optimization



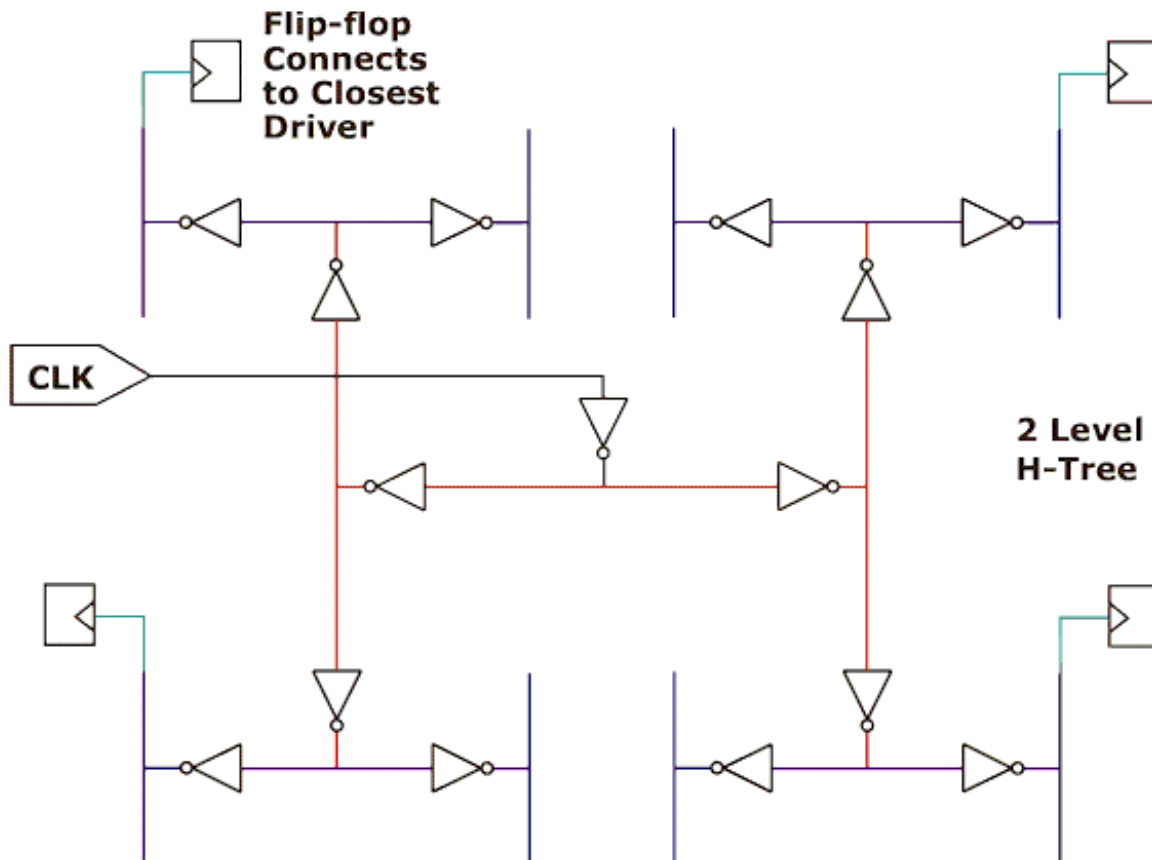
Buffered Net Constraints

Clock Tree Synthesis

The purpose of clock tree synthesis is to minimize the amount of **skew** on the clock tree network. Ideally, it takes the identical amount of time for the signal to travel from the source or *driver* to any of the sinks or *loads* of the clock signal. However, due to the physical positioning of the sinks, the clock signal may not arrive simultaneously. Skew is the maximum time difference of the arrival time of the clock signal between any two load pins. We would like to minimize skew so that all clock signals arrive at their loads simultaneously.

Normally, a clock is supplied to many load pins. This results in a large capacitive load which needs to be driven. Driving this large capacitance from a single gate results in a slow and poorly defined clock signal.

Itools solves the problem of skew and buffer requirements through the use of a buffered H-tree distribution network as shown in the picture. This network is synthesized during placement and the connections between the clock driver and the load are optimized so that the load will be attached to its closest driver. The results of this optimization are reported in the *circuitName.bout* file under the *placesubdirectory*.



The format of the **NET BUFFER** constraint is given by:

```
NET string BUFFER SKEW float
  [INSERTION float] [USE ( <modelname expr> <modelname expr... )]
  [MATCH ( <net1> <net2> ... )] [STATE integer] [FIXEDSTAGES integer]
  [NONCRITICAL] [CAP_MATCH | MINIMIZE_WIRELENGTH | FULL_CLOCKGRID | CLOCKGRID ]
```

The floating number which follows the **SKEW** describes the maximum permitted skew between load pins. The value is interpreted in seconds unless a **SCALE DELAY** statement occurs in the *design.par* file. The optional **INSERTION** controls the maximum allowed insertion delay from the topmost driver to the slowest load. By default, the driver configuration is the smallest area configuration found among the top 10% fastest delay configurations. This gives a relatively good insertion delay with reasonable area. However, in some cases the insertion delay is not critical and one may achieve a smaller area by specifying a larger insertion delay. The optional **USE** keyword allows the user to select the models to be used in the clock tree.

There are two ways of specifying the model : enumeration and regular expressions. For example, suppose we are using an Artisan® library and want to use the clock buffer cells CLKBUF2, CLKBUF3, CLKBUF4, and CLKBUF8 in our clock tree. We could simply enumerate them with the phrase:

```
USE (CLKBUF2 CLKBUF3 CLKBUF4 CLKBUF8)
```

We could also specify the use of all of the clock buffers by using the wildcard in a regular expression such as

```
USE (CLKBUF*)
```

The optional keyword **MATCH** followed by a net or set of nets tells the program to try to minimize the insertion times of the nets in the match set.

The user can specify the desired H-tree buffer configuration by specifying the keyword **STATE** followed by the configuration integer. The possible configurations for a clock tree are enumerated in the *circuitName.bout* file which can be found under the *place* subdirectory. This file is generated during placement and so the user must run the placer at least once in order to generate this file. If the user changes the available buffers, this configuration will change and the placement will need to be rerun.

The user can set the depth or the number of stages of buffering by using the **FIXEDSTAGES** keyword followed by the desired number of stages. Currently, the tool supports from one to four stages of buffering.

The optional **NONCRITICAL** keyword tells the clock tree algorithm to initially place the drivers in an H-tree but allows the placer to move the drivers to minimize wire length. Because the H-tree property is not enforced, the skew constraint is ignored during placement optimization. This option is useful in buffering asynchronous reset networks where skew is not a concern but where the user wishes to buffer the network.

The user may also select the algorithm used in synthesis. Currently, there are four options: capacitance matching, wire length minimization, clocktree mesh synthesis, and full clocktree mesh synthesis. Capacitance matching is chosen through the use of the **CAP_MATCH** keyword. In this mode, the load capacitance is matched as much as possible for all clock buffer drivers. If **MINIMIZE_WIRELENGTH** is chosen, the load is moved to the closest driver in the H-tree regardless of loading. If **FULL_CLOCKGRID** is present, the algorithm will find the closest driver as in the minimize wire length case. In addition, the global router will *wire-or* all of the outputs of the driving buffers so to minimize skew. This results in extremely low skew at the expense of capacitance and wiring resource. The **CLOCKGRID** option tries to minimize some capacitance by only wiring together the buffers that actually drive loads. In most cases, the **CLOCKGRID** and **FULL_CLOCKGRID** yield identical results.

The *designName.host* File

The itools place and route system supports parallel processing on a network of *loosely-coupled heterogenous* workstations. A loosely-coupled heterogenous network consists of workstations running possibly different operating systems connected by a LAN or local area network.

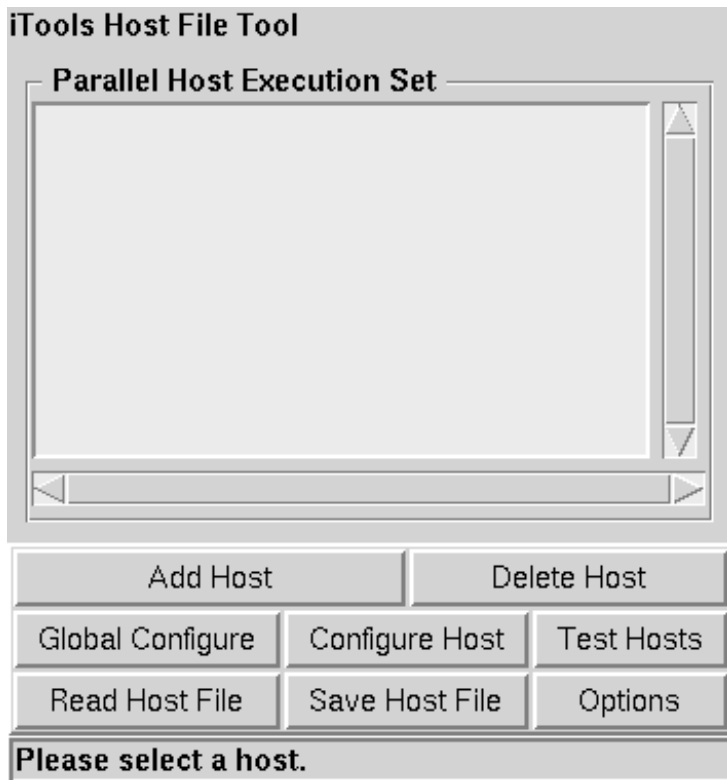
The parallel processing mode is enabled by entering the [parallel](#) option in the *circuitName.par* file. It may be enabled for [placement](#) or [routing](#). When this option is selected, a *designName.host* file must be present in the design directory to describe the available processors on the network. The *designName.host* describes the hosts to be used in the parallel process and their attributes. Here is an [example host file](#).

The [iTools Host File](#) tool below will help you create this file. First, pick **Add Host** to search the network for the available hosts. This procedure uses UDP to broadcast to all of the *portmap* programs on the network. In order for a host to appear in the list, it must be running *portmap*. To add an available host to the list, double click the desired host using the left mouse button. The hostname will appear in the main window. A host may appear multiple times in the main window and each occurrence corresponds to a concurrent process on that host. For example, suppose the user a quad multicore host named *fasthost* available for use. The user would enter *fasthost* four times in the host list as follows:

```
fasthost
fasthost
fasthost
fasthost
```

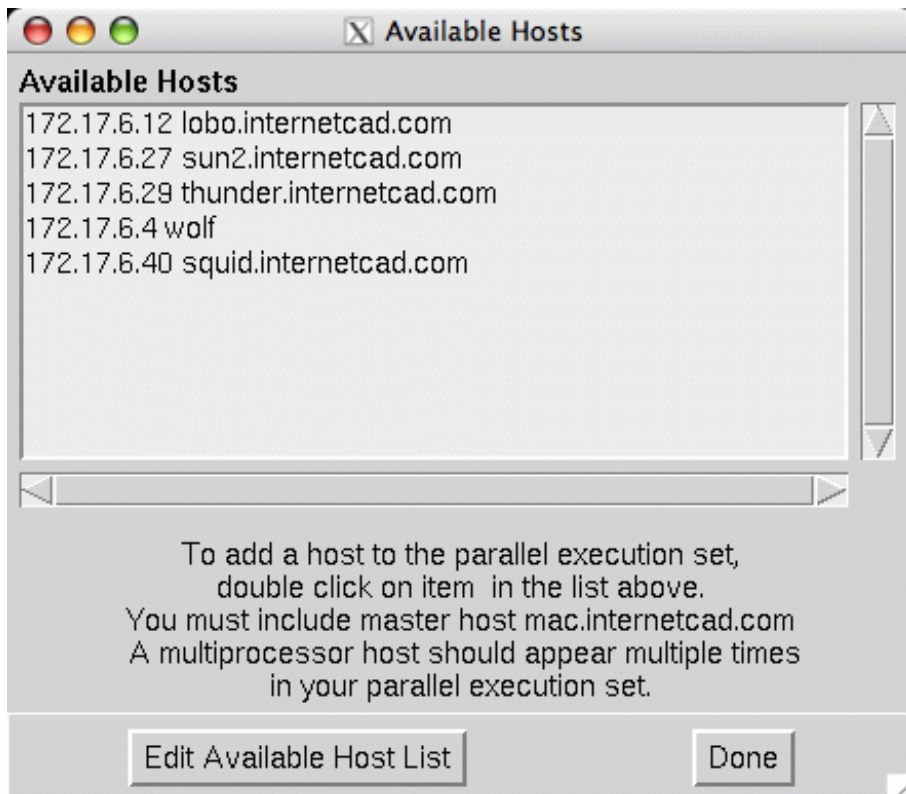
When all desired hosts have been added, use the **Done** button to close the *Available Hosts* dialog box.

Next, use the command **Global Configure** to set the configure all of the hosts defined in the host list and use the command **Configure Host** to individually tailor each host. You may test your set of host by using the **Test Hosts** button. When you are satisfied with your host definitions you may save the definitions to a host file using the **Save Host File** button. A previously defined host file may be read into the current state using the **Read Host File** button.

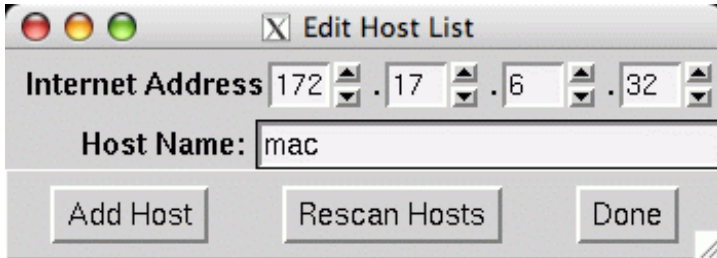


Using Add Host

The image below shows the **Add Host** dialog box. By double clicking on the host in the **Available Hosts** list, you will add it to the **Parallel Host Execution Set**. The dialog box will remind you to include the master host.



Since not all machines in the network may not respond to the portmap query, there is a mechanism to add hosts to the **Available Hosts** manually. The **Edit Available Host List** button will display the **Edit Host List** dialog box. You should supply an Internet Address and a Host Name for any desired host **not** found in the **Available Host List**. The **Rescan Hosts** command reinitializes the **Available Host** list with a new portmap query. When you are finished adding hosts, use the **Done** button to close the **Edit Host List** dialog box.



Using Global Configure

The image below shows the format of the Configure dialog box. An option here applies to all hosts and will reset all hosts configurations. Therefore, one should use the global configure option first to set the baseline, and then configure individual hosts. There are seven options: **Relative Computational Power**, **Login Name**, **Ittools Directory**, **Display**, **Geometry**, **Mounting Prefix**, and **Working Directory**. Each option may be set to its default value by selecting the **default** radio button. In order to customize an option, the customize button must be chosen. This will allow you to enter data in the entry form below the option.

The **Relative Computational Power** is a non-negative floating-point number which describes the relative speed of the host to others in the file. The default value is 1.0. For example, a value of 2.0 means the host is twice as fast as the default entry.

Host Configuration Dialog

Parallel Configure Options for all hosts

Relative Computational Power	<input type="radio"/> default <input checked="" type="radio"/> customize
2.0	
Login Name	<input type="radio"/> default <input checked="" type="radio"/> customize
bills	
iTools Directory	<input type="radio"/> default <input checked="" type="radio"/> customize
/home/itools.1.4.0	
Display	<input type="radio"/> default <input checked="" type="radio"/> customize
wolf:0	
Geometry	<input type="radio"/> default <input checked="" type="radio"/> customize
400x500+399+1	
Mounting Prefix	<input type="radio"/> default <input checked="" type="radio"/> customize
/mnt	
Working Directory	<input type="radio"/> default <input checked="" type="radio"/> customize
/home/bills/my_design	
Remote Login Program	<input type="radio"/> rsh <input checked="" type="radio"/> ssh
Remote Program Argument Order	<input checked="" type="radio"/> args first <input type="radio"/> host first
Remote Login Program Options	<input checked="" type="radio"/> default <input type="radio"/> customize
<input type="button" value="Accept"/> <input type="button" value="Cancel"/>	

The **Login Name** allows the user to adjust the login name on a host by host basis. The user may have rsh privileges on the hosts to be used. The default is the current login name.

The **iTools Directory** allows the user to adjust the path of the **itools** root directory on a host by host basis. The **itools** directory may or may not be cross mounted on all hosts via NFS. This option allows you to adjust the path on each host. The default is the current value of the environment variable **ICDIR**. In the example, the **itools** root or toplevel directory is */twolf/itools*.

The **Display** and **Geometry** options allows the user to adjust the graphics output of the **itools** programs. The display is a valid X11 host display. The geometry is a valid X11 geometry specified by the form **WIDTHxHEIGHT+XOFF+YOFF** (where **WIDTH**, **HEIGHT**, **XOFF**, and **YOFF** are numbers) for specifying a preferred size and location (measured in pixels) for this application's main window. The **XOFF** and **YOFF** parts are used to specify the distance of the window from the left or right and top and bottom edges of the screen, respectively. Both types of offsets are measured from the indicated edge of the screen to the corresponding edge of the window. In the example, the display name is *wolf:0* and the geometry has a width of 400 and height of 500 pixels. This geometry

has its window origin at 399,1.

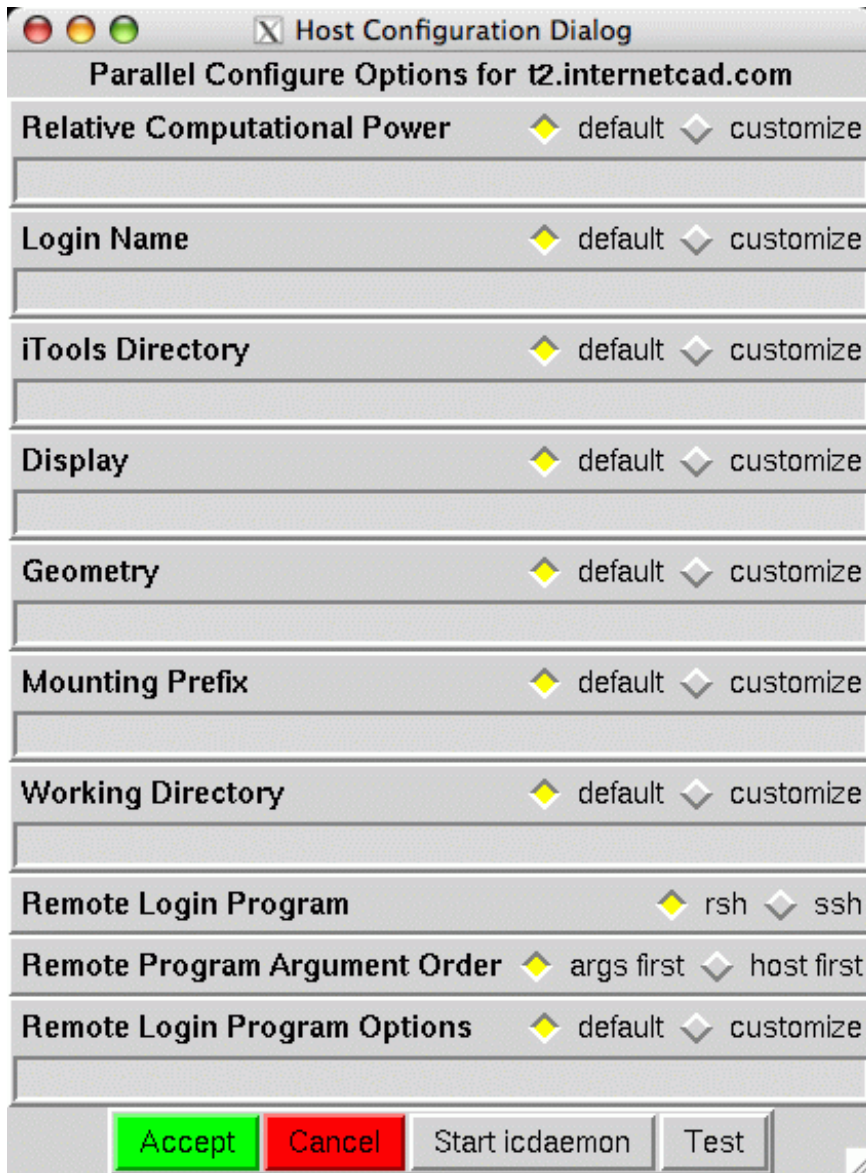
The **Mounting Prefix** is used to [adjust for automount mounting point](#) prefixes. If present, the prefix will be prepended to both the *working directory* and the **itools root directory**. In the example, */mnt* will be prepended to the **itools** directory to form: */mnt/twolf/itools*

The **Working Directory** is the path of the current design directory. This option makes it possible to describe a different directory for each host. In the example, the design directory is */mnt/home/bills/my_design*. It should be noted that the design directory should represent the same NFS mounted directory. The designated user must have read and write permissions in the design directory. Normally with NFS mounted directories, the working directory option is not needed as the mount option is able to accomodate automounter path differences. It should also be noted that if the login user is the same for all hosts, the uid (user id) of the login name should be the same on all hosts; otherwise, there will be a user mismatch on the NFS file system.

If [icdaemons](#) are not utilized, the user will need to furnish the method to start programs on the remote host. The **Remote Login Program** is used to start a program on the remote host. The user may choose between remote shell (rsh) or secure shell (ssh). The user may customize the order and pass custom options to the login programs to achieve a login. The user must be able to remotely login with one of the two remote login programs without having to supply a password. Please consult your system administrator for help. It is recommended that the user use icdaemons so remote login is not an issue.

Using Configure Host

The image below shows the **Configure Host** dialog box. It is similar to the [Global Configure](#) dialog box but works on an individual host. Two additional buttons are present in this dialog. The first button allow the user to start the icdaemon whereas the second **Test** button is furnished so that you can test the behaviour of this host. In both cases, the current state is used not the saved state.



Using icdaemon as the Remote Startup Mechanism

In order to overcome the difficulties in starting programs on remote hosts, the *icdaemon* facility was developed. The *icdaemon* is a Tcl-based client-server application where servers run on each of the hosts listed in the *designName.host* file. The Tcl for this server can be found in the [icdaemon](#) file located in the itools bin directory. The **Start icdaemon** button on the **Host Configuration Dialog** displays the following **Start icdaemon** dialog:

icdaemon Start Options

login: bill

passwd: *****

command: /Users/bill/version1/itools.current/itools/bin/icos -override icdir /Users/bill/version1/itools.curre

prompt: \$

salt: 3694084

script: /Users/bill/version1/itools.current/itools/bin/iclogin.ssh t2.internetcad.com bill

remote pgm: ssh

view passwd: ☐

debug: ☐

status: Idle

Start Daemon Done

The dialog box enables the user to control and test the icdaemon. The first field (which cannot be edited) is the login name. If you need to change the login name, please go back and change it in the **Configure Host** dialog box. The second hidden field is the passwd field which contains the users password for that host. The user may show the contents of the password field by checking the **view passwd** field below. The third field is the command that will be executed on the remote host. On the creation of the dialog box, this field will be filled with the proper command to start the icdaemon on the remote host. However, the user can modify the command to their needs. For example, during debug of the remote login process, the user may change this command to a simple common Unix command such as **date** in order establish the correctness of the basic login process. To return the command to its default contents just hit the **Done** button to close the dialog box and then reopen it with the **Start icdaemon** command. In order to login onto the remote machine, icdaemon uses the **Expect** library bundled with itools. **Expect** is a Tcl extension which automates the execution of programs. In this case, **Expect** is used to automatically login to the remote host using either **rsh** or **ssh**. The script that is used for the login process is found in the script field. The user can furnish their own Expect script by modifying the contents of the script field. You will find the iclogin.ssh and **iclogin.rsh** scripts in the itools bin directory. The **Expect** script is written such that it *expects* a prompt in the remote shell before sending the remote command found in third entry box. For this reason, the user should enter the characters found in their shell command prompt. The default is the Bash shell command prompt.

In order to ensure reasonable security, the user's password is not normally displayed and the results are encrypted. The salt field is used as a key in encoding and decoding the passwd. Note, the passwd is encrypted during the execution remote command. The encrypted passwd is deleted as soon as possible, that is, immediately after the remote command is executed in order to minimize security risk. The user can change the salt to any valid Tcl string and a generalized string if they enclose it with curly braces (so Tcl recognizes it as a string).

The debug checkbox is available to turn of verbose debugging feedback to help the user debug the login process if something goes wrong. Automating the login process is fraught with problems as there are many possible unforeseen situations. The automatic process is useful when many different hosts need to be started. However, one can always start the icdaemon manually.

Manually controlling the icdaemon

If the automatic process fails or the login process is unique and difficult, it is possible to start the icdaemon manually. First, login to the remote host. Next, change directory to the [itools root directory](#) and [source the appropriate initialization script](#). Now you are ready to start the icdaemon. Just enter the command:

```
icdaemon
```

The icdaemon will report if a icdaemon is already running. On the occasion that the user wants to kill the current icdaemon, enter the command:

```
icdaemon -suicide
```

It is also possible to perform both tasks by entering the command:

```
icdaemon -restart
```

When installing a new version of itools, it is best to issue a restart especially if the installation path changes. If the icdaemon is unresponsive, use the restart mechanism to rectify the problem.

Mounting Prefix Option

The **Mounting Prefix Option** is available to adjust automount/mount path differences. Itools needs to find the working directory of the design on the remote machine and often the path varies on the remote machine. For example, suppose on the local machine named *mylhost*, the working direction and directory listing looked like:

```
mylhost>pwd
/attic/data/tests/designs/large_design/itoolsdata
mylhost>ls
ldesign.ckt      ldesign.lib
ldesign.con      ldesign.par
```

Now suppose on the remote machine named *fasthost*, the working directory and directory listing looked like

```
fasthost>pwd
/mnt/attic_copy/data/tests/designs/large_design/itoolsdata
fasthost>ls
ldesign.ckt      ldesign.lib
ldesign.con      ldesign.par
```

Your host file would need to contain the following mounting host options:

```
mylhost 1.0 -mount /attic
fasthost 1.0 -mount /mnt/attic_copy
```

As you can see, one only needs to provide the mounting path prefix, in this case */attic* on the local host and */mnt/attic_copy* for the remote host. You do not need to supply the common parts of the path as itools can calculate it. The algorithm for this adjustment is given in the Tcl procedure `::ichost::test_adjust_mount_point` found in the file `htool.tcl`.

ITools Host File Tool

The ichost tool may be executed in three different ways: from EZ, command line graphics user interface, and command line text mode. In EZ, the graphical interface is presented and the user only needs to interact with the buttons which are presented. The host tool may also be invoked in a standalone mode either in graphical or text mode. To invoke graphical mode, use

```
ichost
```

on the command line after [sourcing the appropriate itools initialization](#) script. The graphical mode works exactly like it does in EZ and the documentation above applies.

The text mode of ichost is invoked on the command line by entering:

```
ichost -nogui
```

and is used exclusively for testing. The top menu for ichost in text mode looks like:

```
Host Tool Top Menu
1) Read Host File
2) Test Single Host
3) Test All Hosts
4) Check Icd daemon Registration
5) Enter Tcl Shell
6) Exit Host Tool
```

Enter operation:

You should enter **1** in order to read a host file. When prompted, you should enter the path of the host file. You may then test all hosts or a single host given in the host file. The testing procedures will help you troubleshoot parallel execution. You should also check the icd daemon registration if you are using the icd daemons as this test checks whether the license server is recent enough. Itools version 2.0.0pre37i (or later) is required to use icd daemons properly.

Universal Host File

Itools supports the use of a universal host file in substitution of the *designName.hostfile*. Itools will look for the existence of an environment variable called **ICHOSTFILE** before searching for the *designName.host* file. The **ICHOSTFILE** should be defined as the pathname to a valid host file. This allows all designs to use the same host file. For example, here is an example in the Bash shell:

```
export ICHOSTFILE=/home/user/itools/myglobalhostfile
```

Automatic Testing of ITools Host File

Now it possible to test the host file for correctness during the the execution of the itools flow. Turn the following option on



in the *designName.par* file. During the execution of the syntax program, the host file will be read into the ichost tool in text mode. The itools ichost tool will wait for the user to begin testing.

Itranslate Program

Copyright (c) 1996–2005. InternetCAD, Inc. All right reserved.

The **itranslate** program is a general purpose translator which permits the translation between various netlist and data formats. ITOOLS supports translation from the following CAD systems:

- Cadence using LEF/DEF
- Cooper and Chyan Technologies
- Silicon Valley Research

In addition, ITOOLS is able to translate the following common library formats:

- CALMA GDSII

the following common netlist formats

- structural verilog
- EDIF 2.0.0

and timing constraints using

- Standard Delay Format (SDF)

and physical constraints using

- Physical Design Exchange Format (PDEF)
-

ITRANSLATE Description

The **itranslate** program is a Tcl–based interpreter which enables the user to write scripts controlling the translation process. Typing "itranslate" on the command line shows its proper usage:

iTools translator

itranslate version:v1.4.0b2

iTools compilation:Tue Sep 21 15:40:02 CDT 1999 by bills using Linux 2.0.27 (i686)

Copyright (c) 1993–99 InternetCad.com, Inc. All rights reserved.

ERROR[Yunix_parse]:option -- unknown

Usage: itranslate [-pl-debug] [-Dl-memdebug] [-bl-bcompat] [-el-echo]

```
[-C|-Compact] [-d|-do <do>] [-v|-verbose] <circuit>
[<args>...]
```

Description:

This program is a TCL-based translation program which translates between external CAD languages and the iTools language. Among the languages supported are LEF/DEF, EDIF 2.0.0, SDF, verilog, and GDS2. For more information, type 'ichelp' at the interpreter prompt.

Options/Arguments:

-p -debug	debug mode on	(off)
-D -memdebug	debug memory	(off)
-b -bcompat	backwards compatibility	(off)
-e -echo	echo commands	(off)
-C -Compact	compact files & save space	(off)
-d -do <do>	do file name	(none)
-v -verbose	verbose mode on	(off)
<circuit>	design name	
<args>	do file script arguments	(none)

itranslate terminated abnormally with 1 error[s] and 0 warning[s]

itranslate may also be executed interactively with EZ-CAD.

The **itranslate** program has a help command which lists the available commands. It is executed by typing:

```
ichelp
```

at the prompt. The **ichelp** command currently returns

```
% ichelp
In addition to the basic Tcl commands,
The following commands are available
Itools read commands:
  icread_blk
  icread_ckt
  icread_con
  icread_lib
  icread_routing
  icread_par
  icread_placement
  icread_ic
Itools write commands:
  icwrite_blk
  icwrite_ckt
  icwrite_con
  icwrite_lib
  icwrite_routing
  icwrite_par
  icwrite_placement
  icwrite_ic
Cadence read commands:
  icread_lef
  icread_def
Cadence write commands:
  icwrite_lef
  icwrite_def
CCT (Cadence) commands:
  icread_cct
  icwrite_cct
Silicon Valley Research commands:
  icread_cdl
  icread_pdl
  icread_pif
  icread_sdl
  icread_srf
  icread_tdl
```

```

icwrite_cdl
icwrite_pdl
icwrite_pif
icwrite_sdl
icwrite_tdl
Edif commands:
  icread_edif
  icwrite_edif
Verilog commands:
  icread_verilog
  icflatten_verilog
  icwrite_verilog
CIF commands:
  iccif_define
  icwrite_routing_cif
GDS2 commands:
  icread\_gds2
  icwrite\_gds2
  icgds\_define
SDF commands:
  icread_sdf
  icwrite_routing_sdf
PDEF commands:
  icread_pdef
Other commands:
  icadd_virtual_rows
  icblock_align
  iccase
  iccheck_ports
  iccompare_netlists
  iccontext_free
  icdesign_name
  icequiv_cons
  icexit_on_error
  icfree_instances
  icfree_models
  icfeedthru
  icmodel
  icnumber_rows
  icnum_instances
  icread_spice
  icrename
  icrouting_frame
  icrouting_grid
  icrouting_origin
  icscale
  icscript_args
  icsite_type
  icstatus
To get addition information on an individual command,
type the command name followed by "-help"

```

Programming ITRANSLATE

ITRANSLATE is a Tcl-based program. However, the user does not require extensive knowledge of Tcl in order to effectively use the tool. In fact, almost all scripts consist exclusively of the "**ic**" commands listed above. Nevertheless, the full power of the Tcl language is at your disposal. Unfortunately, it is beyond the scope of this documentation to present the Tcl language and it is recommended that the user read one of the many books on Tcl. Again, for most scripts this is unnecessary – in fact, EZ-CAD can generate most scripts for you.

In order to use **itranslate** correctly, one must understand the design data dependencies of CAD systems. At the bottommost level is the technology or design rules information. The second level or library level, requires knowledge of the first; the library needs to know about layers, in order to meaningfully describe the physical attributes of a cell or model. The third level or netlist level, references the library models in the description of the circuit. The fourth and uppermost level of the hierarchy is the constraint/state layer. The constraint level may describe a state of the design, or constrain a design parameter or aspect of the design. The constraint level may affect any or all of the lower levels of the design.

In the ITOOLS system, each level is partitioned into a separate file. For example, the technology is in the *designName.par* file, the library information in *designName.lib*, the circuit description in *designName.ckt*, and the constraints in the *designName.con* file. However, in ITOOLS the state information resides in many separate files: row topology information in the *designName.blk* file, placement state in the *designName.pl1* and *designName.pl3* files, and global routing information in the *designName.pin* file.

Other CAD systems partition the levels in various manners. For example, the Cadence tools partition the technology and library information into the LEF file and the circuit, constraints, and state into the DEF file. Silicon Valley Research is similar to ITOOLS in that each level is partitioned into a separate file whereas the Cooper and Chyan Technologies allows the data to be represented in just one file.

Regardless of the system, the design data hierarchy must be observed; the level of data input to **itranslate** program must progressively increase, starting at level one. It is not necessary for all levels to be from the same source as long as the order is followed. This is to insure proper construction of the design data structures. For example, the design rules could be from SVR, the library from GDS2, and the circuit from DEF as long as the files are read in this order. Once the input phase is complete, and the design data structures are built, the user may modify the data and then output the data in another form. In general, it is safest to output the data in the reverse order – from highest to the lowest.

Detailed Description of Tcl Commands

GDS2 Commands

There are three GDS2 commands: **icread_gds2**, **icwrite_gds2**, and **icgds_define**. The first two commands convert from GDS2 to ITOOLS and from ITOOLS to GDS2 respectively. The third command, **icgds_define** is needed to compensate for weaknesses in the GDS2 language.

The **icread_gds2** command reads a GDS2 file and converts it into the ITOOLS internal representation. The conversion takes place in a two step process. In the first step, the GDS2 is converted into an ascii representation of the binary GDS2 format. Then the ascii is parsed with a yacc/lex parser into the internal representation. The format of the command is as follows:

```
icread_gds2 [-ascii] [<filename>] [-help]
```

The **ascii** option allows one to read the intermediate representation into ITOOLS

The **icwrite_gds2**

The **icgds_define**

Translating to and from Cadence LEF/DEF

Below is a simple script to translate from Cadence to ITOOLS. Notice all commands are "**ic**" commands. Notice that none of the read and write commands take any arguments. This is a common feature of all the translator read and write commands. If there aren't any arguments to a command, the filename is assumed to be the design name appended with the commands default suffix. For example, the **icread_lef** command would read from the file *mydesign.lef* if **itranslate** was invoked using "*itranslate mydesign*".

```
iccheck_ports off
icread_lef
icread_def
icwrite_ic
```

We see in this example that we first read levels 1 and 2 using **icread_lef** and then read levels 3 and 4 using **icread_def**. To write the data in ITOOLS format, we use the convenience function **icwrite_ic**. In reality, this function is implemented using

```
icwrite_blk
icwrite_con
icwrite_ckt
icwrite_lib
icwrite_par
```

If you are using just the placement program of ITOOLS, converting back to Cadence is just as easy. Here we need to read the ITOOLS input files, obtain the state information from a placement file, and then write the Cadence files.

```
iccheck_ports off
icread_ic
icread_placement mydesign.pl3
icwrite_lef
icwrite_def
```

Notice in this script, we specifically designated *mydesign.pl3* which is the placement after global router as the file to read. If we had not supplied an argument to **icread_placement**, it would read the default *mydesign.pl1* file instead.

Problems with ITOOLS keywords

The ITOOLS Placement and Routing Package is case-sensitive. Trouble may arise if commands are specified with upper case. All *itools* commands are lower case.

Cluster Parameters

There are no user parameters for cluster.

Constraints Tutorial

All constraints to **itools** programs are specified in the *designName.con* file. [This link contains the full specification of the constraints file](#) The following sections describe some of the commonly used constraints and give examples of their use.

[1. Fixing pad positions.](#)

[2. Fixing cell positions and ECOs.](#)

[3. Path length constraints.](#)

[4. Path timing constraints.](#)

[5. Pin pair timing constraints.](#)

[6. Ignoring selected nets during placement](#)

Fixing Pad Positions

A line such as the following is added to the *designName.con* file:

```
PAD name RESTRICT SIDE side SIDESPAC pos1 pos2
```

The *side* may be one or more of R, L, T or B. The letters stand for right, left, top or bottom. The number *pos1* is the lower bound of the pad position along the side, as a decimal fraction. *pos2* is the upper bound of the position along the side. For example, to restrict the CIN1 pad to the bottom or left sides, between 10% and 15% of the distance along the side from the lower left, the following line is added to the constraints:

```
PAD CIN1 RESTRICT SIDE BL SIDESPAC 0.1 0.15
```

Fixing Cell Positions and ECOs

Any cell in the design can be given an initial position. From this initial position the cell can move in 3 possible ways during placement: It can be fixed in the initial position, it can move slightly from the initial position or it can be free to move anywhere in the design. To give a cell an initial position the following line is added to the constraints file:

```
FIXED cellname fix-type (x y) ORIENT orientation
```

The *fix-type* must be one of RIGIDLY, APPROXIMATELY or INITIALLY, corresponding to the three cases above. The (*x y*) co-ordinates give the initial fixing position and the *orientation* must be an integer between 0 and 7 inclusive, to represent the [orientation](#) of the cell. **Ittools** reads the initial placements for a new circuit from previous placements by adding the **restart pl1_absolute** or **restart pl1_relative** commands to the *designName.par* file. The *designName.pl1* file from the previous run must be renamed to *designName.pl1_in* for the new run. Fixing cells can be used when the design must be slightly modified by ECOs. Two commands can be used to add new ECO cells, or delete old cells from the previous version of the design:

```
ECO_ADDED_CELL cellname
```

```
ECO_DELETED_CELL cellname
```


Deleted cells may appear in the list of INITIALLY FIXED cells, but not in the *designName.ckt* file. Added cells should be in the *designName.ckt* file.

Path Length Constraints To limit the wire length of any given net the following line is added to the constraints file:

```
PATH net1 net2 . . . netN : min-length max-length
```

The minimum and maximum lengths of the sum of the net lengths for *net1* to *netN* are given in the same units as the library cell dimensions. To observe any paths that violate path length constraints, without having **itools** try to place the net within the constraint, the MONITOR keyword is added to the end of the PATH constraint line. For example the following line will show if net S84 is longer than 4000:

```
PATH S84 : 0 4000 MONITOR
```

Removing the MONITOR keyword makes **itools** attempt to keep net S84 under 4000.

Path Timing Constraints A more accurate method of controlling circuit delays takes driver strengths and routing capacitance into account. **Ittools** takes account of driver strengths as specified in the *designName.lib* file and capacitance for routing layers as given in the *designName.par* file. Constraints on delay through selected paths can be specified in the constraints file as follows:

```
PATHCONSTRAINT (instance1 output_pin1 instance2 input_pin2) (instance2 output_pin2 instance3 input_pin3) . . .  
(instance_n-1 output_pin_n-1 instance_n input_pin_n) (delay)
```

Ittools will attempt to keep the path delay under the specified delay. As in the path length constraint the MONITOR keyword can be added to the end of the constraint line to make **itools** print out violations without attempting to change placement. The delay defaults to nanoseconds but the scale can be changed using **scale delay** in the *designName.par* file.

Path timing constraints can be imported from synthesis systems such as Synopsys by using the [SDF](#) file format.

Pin Pair Timing Constraints In cases when the explicit paths are not known it is necessary to specify timing constraints using just the external pins of the circuit. In these cases **itools** will enumerate all possible paths from the specified primary input to primary output pin pairs and attempt to constrain path delay to the specified amount. To constrain delays between input output pin pairs the following line is used in the constraint file:

```
PATHCONSTRAINT (input_pad/output_pin output_pad/input_pin) (delay)
```

As before, pin pair delays can be monitored without affecting placement, by adding the MONITOR keyword to the end of the constraint line.

Ignoring Selected Nets Sometimes it is necessary to ignore the effects of selected nets during cell placement. Nets such as Reset lines may be very large and the timing may not be critical. Trying to minimize the net length of the Reset line may adversely affect lengths of more critical nets. To ignore a net during placement use the following line in the constraint file:

```
IGNORE netname
```

Escape Characters in Verilog and SDF

The specification of a component or net identifier is handled differently in Verilog and the Standard Delay Format (SDF) files.

In SDF, identifiers can be up to 1024 characters long. Characters that you can use in an identifier are:

- Alphanumeric characters, that is, alphabetic letters – uppercase and lowercase letters are considered different, all of the numbers and the underscore (_).
- Hierarchy divider character may appear in an identifier without escaping it.
- Escape character. Each special character used as part of an identifier must be escaped. The special characters in SDF are
 • ! " # \$ % & ' () * + , - . / : ; < = > ? @ [\] ^ { | } ~

The backslash character must precede each of these special characters.

- Bit specs. Place bit specs at the end of identifiers with no separating white space. Include bit specs between square brackets ([and]). If bit spec is a range, use the colon (:) to separate the range, for example, **[4]**, **[3:31]** and **[15:0]**.
- Never use white space in an identifier.

For example, here is an escape identifier in SDF:

- my_ram/address\ (4\)

In contrast, Verilog's escape mechanism uses one escape or backslash (\) before the identifier and a white space after the identifier when the identifier contains a special character. For example, here is the same identifier in Verilog:

- \my_ram/address(4)

Because of these incompatibilities, it is best to use *iTools* canonical form which strips all escape characters from the identifiers. This is now the default in the translator. So using the canonical form, these identifiers will become:

- my_ram/address(4)

in *iTools*.

It is also possible to prevent such translation using the `-literal` switch to the commands **icread_verilog** and **icread_sdf**. It is also possible to add custom translation functions to these routines using the `-transcmd <Tcl proc>` switch. This allows the user to add their own custom translation routines in Tcl. They will be documented at a later date.

Itools Documentation

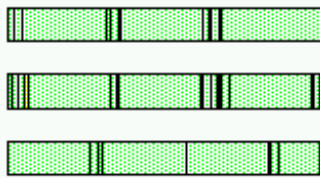
```
bulldog 1.3 -login bill -icdir /bulldog/itools -wd /common/work/bills/stdcell -mount /mnt -display wolf:0
hp 1.0 -display wolf:0
lobo 1.0 -display wolf:0
wolf 1.0 -display wolf:0
indy 1.0 -display wolf:0
sun 1.0 -display wolf:0
powerpc 1.0 -display wolf:0
```

General Parameters

Below are the general parameters available for edit. It is also possible to [edit the rules and other program parameters](#).

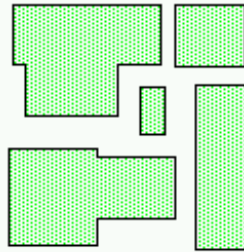
*autodetect_script:	<input type="radio"/> off <input type="radio"/> on
*auto_wire_weight:	<input type="radio"/> off <input type="radio"/> on
*background:	<input type="text"/>
*check_ports:	<input checked="" type="radio"/> off <input type="radio"/> on
*context_free:	<input type="radio"/> off <input type="radio"/> on
*contiguous_pad_groups:	<input type="radio"/> off <input type="radio"/> on
*design_style:	<input type="radio"/> gate_array_with_sites <input type="radio"/> gate_array <input type="radio"/> stdcell
*graphics:	<input type="radio"/> off <input type="radio"/> on
*graphics_wait:	<input type="radio"/> off <input type="radio"/> on
*library:	<input type="text"/>
*memory_messages:	<input type="radio"/> off <input type="radio"/> on
*minimum_pad_space:	<input type="text"/>
*padspaceing:	<input type="radio"/> abut <input type="radio"/> exact <input type="radio"/> uniform <input type="radio"/> variable
*rowSep:	<input type="text"/>
*random_seed:	<input type="text" value="1"/>
*scale:	<input type="text"/>
*total_row_length:	<input type="text"/>
*vertical_path_weight:	<input type="text"/>
*vertical_wire_weight:	<input type="text"/>

Design Styles



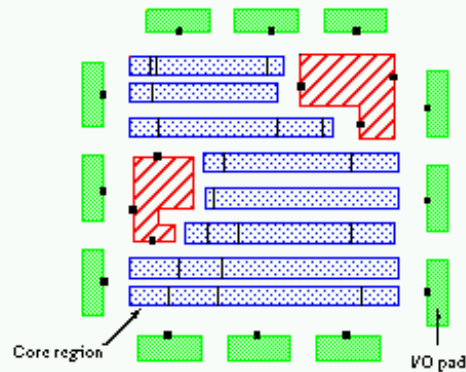
Standard cell design style

Suits random logic



Macro cell design style

Suits array architectures

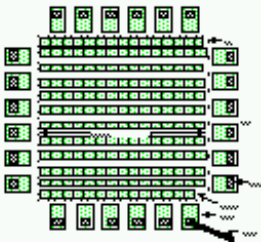


Mixed Macro/Standard Cell style

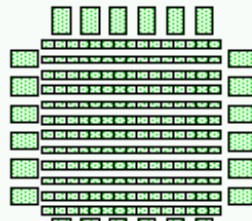
Ittools can handle designs with macro cell designs, row-based designs, or a mixed combination as shown above. **Ittools** automatically detects the presence of macros and row-based designs and executes the appropriate commands.

However, row-based designs may be further qualified into four broad categories: standard cell, gate_array, sea-of-gates, and island style gate arrays as shown in the picture below. **Ittools** can now handle all four methodologies but it requires the execution of different algorithms. The **design_style** keyword in the *designName.par* controls the selection of the algorithms necessary to handle these four methodologies. The standard cell, gate array, and sea-of-gates styles are placed using IPLACE whereas the island style gate array must be placed with IPLACEGA. The methodologies placed by IPLACE are siteless, that is, a cell may be placed anywhere in the row (in the absence of constraints). The island style gate array specifies sites in which the cells must align and this is implemented in IPLACEGA.

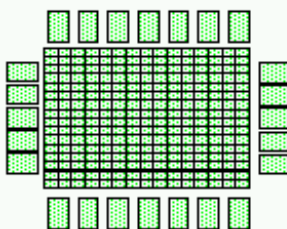
Row-based Design Methodologies



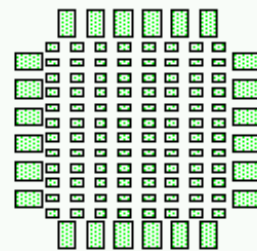
Standard Cell



Gate Array






Sea-of-Gates



Island style gate array

Cadence Cell3 style designs should be run with **gate_array** mode and not **gate_array_with_sites**.

***design_style:**  gate_array_with_sites  gate_array  stdcell

Itools can handle designs with port data outside the cell boundary. However, if requested, itools checks and records all data which resides outside the given cell boundary as an error. The option **check_ports** is read and utilized by many of the itools programs. If the data should be inside the cell boundary, turn on the check; otherwise turn the check off.

Recommended setting: off

***check_ports:**  off  on

The itools language may be transformed into a context-free parsing language for itools versions 1.3.0 and above. This feature forces all itools keywords in the *designName.lib*, *designName.ckt*, and *designName.con* files to begin with **IC**. This mode reduces the chance that the user would have a name clash which results in a syntax error. For example, the itools keyword **PORT** normally requires the user to avoid the use of *PORT* as an identifier. Since itools is case sensitive, the user could use *Port* or *port* but must not use *PORT*. Otherwise, itools will reject the input as an error. If the context free mode is enabled, itools will reject *PORT* as a keyword and instead require the port keyword to be **ICPORT**. Now the user is free to use the common *PORT* as an identifier but must refrain from using the obscure keyword **ICPORT**. If you plan to use an earlier version of itools, this mode must be disabled. If one can avoid the use of itools keywords, you can save file space by disabling this feature. However, if the design uses itools keywords this mode is available to remedy the situation. By default, this mode is disabled. Recommended setting: off

***context_free:**  off  on

The optional **library** keyword allows the user to specify more than one library (*.lib*) file for the design. The string following the library keyword specifies the pathname of the library. The user may specify multiple libraries by listing each library separately.

***library:**

The optional **memory_messages** keyword allows the user to control the output of memory usage messages to the console. Whenever program memory is increased, the itools memory manager issues a message stating the current memory. This command allows the user to suppress these messages. The default is to enable the display of the messages.

***memory_messages:**  off  on

The keyword **random_seed** is useful when the output data files have been deleted and the data needs to be regenerated. If the input files are identical, a second run using the same **random_seed** value will yield the exact same output. The random number generator seed is printed in the output files of each of the respective programs. The seed may be set identitically for all programs here or may be set uniquely for each program.

***random_seed:**

The required keyword **rowSep** is followed by a floating point number representing the desired amount of separation between the rows. The amount of separation between the rows is this number times the average height of the rows. That is, if you want the row separation to be equal to the average row height, then this number should be 1.0. On the other hand, if you want the row separation to be twice the height of the rows, then this number should be 2.0. Normally, a value of 1.0 is appropriate.

*rowSep:

The optional keyword **total_row_length** is followed by an integer specifying the total available row length, for a gate array circuit (only). That is, this keyword should only be used when the total row length is fixed to a value larger than the total cell width.

*total_row_length:

Wire Weighting Parameters

The keyword **auto_wire_weight** controls the interpretation of the **vertical_wire_weight** parameter. If **auto_wire_weight** is enabled (default), the internal vertical wire weight is calculated by IPLACE using Equation 3.2. Otherwise, IPLACE will use the **vertical_wire_weight** value supplied by the user for the internal vertical wire weight.

*auto_wire_weight: ◇ off ◇ on

The keyword **vertical_wire_weight** is optional. The floating point number represents the cost for one unit of vertical wire length, given that the cost for one unit of horizontal wire length is unity. This features allows the user to simulate different row separations without changing the *designName.blk* file. Internally, **itools** uses the following formula for weighting vertical wire:

$$\text{internal_vertical_weight} = \frac{|\text{vertical_wire_weight}|}{\text{average row separation}}$$

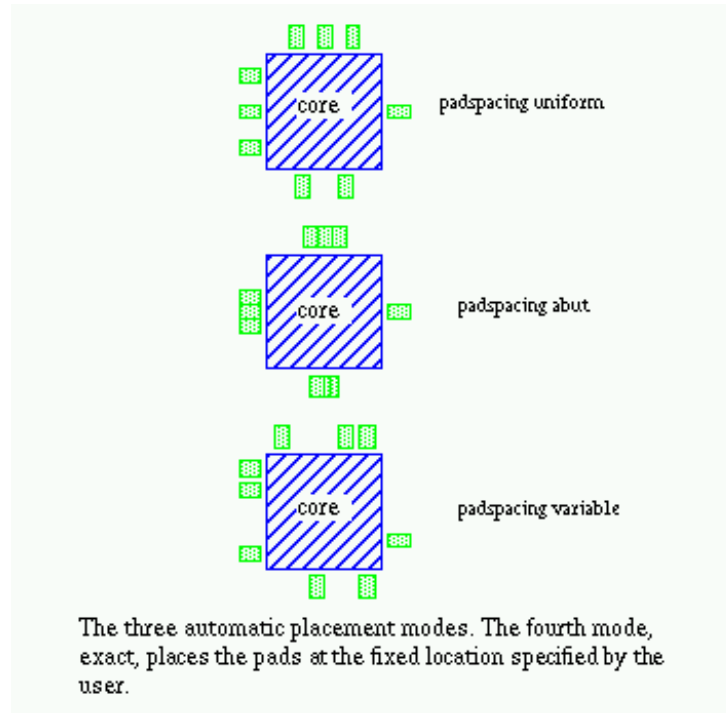
where **internal_vertical_weight** is used directly in the cost function, **vertical_wire_weight** is given by the user, and **average_row_separation** is the mean of the row spacing supplied in the *designName.blk* file. If **vertical_wire_weight** is not specified, IPLACE will assume a value of 1.0. An effective row separation of 1.0 will achieve the best results for designs with adequate feed through resources. When feed through resources are scarce, the user may wish to set an effective row separation greater than 1.0 to minimize the number of feed throughs added to the design. In this case, IPLACE will suggest a value for the **vertical_wire_weight**. If the **auto_wire_weight** is disabled, IPLACE will ignore the row separation in the *designName.blk* file and set the **average_row_separation** to 1.0. This mode simulates previous versions of **itools**.

*vertical_wire_weight:

The keyword **vertical_path_weight** is required. The floating point number represents the cost for one unit of vertical path length, given that the cost for one unit of horizontal path length is unity. This features allows the user to specify that the capacitance (or, in some sense, the delay) per unit length is different for the vertical routing layer as opposed to the horizontal routing layer. IPLACE will seek to ensure that for each path specified in the *designName.con* file, the horizontal path length plus the **vertical_path_weight**, times the vertical path length is between the upper and lower bound for that path.

*vertical_path_weight:

I/O Pad Parameters



Pad spacing in **itools** has three automatic placement modes. The fourth mode, **exact**, places the pads at the fixed location specified by the user.

The optional keyword **padspacing** controls the pad spacing mode. There are four modes of operation: **uniform** pad spacing, **abutting** pad spacing, **variable** pad spacing, and **exact** pad spacing as shown in the figure above. Uniform pad spacing spaces the pads evenly on each of the sides. In the abut mode, pads are forced to touch one another. The variable pad spacing mode places each pad such that the wire length is minimized. The last mode turns off the pad spacing algorithm and the pads remain in the place specified by the user in the *designName.con* file. In the first three cases, the [side](#) and [sidespace](#) constraints are observed. The default mode is uniform padspacing.

***padspacing:** ☐ abut ☐ exact ☒ uniform ☐ variable

By default, members of pad groups are placed contiguously. In other words, no nonmember pads can be placed between the member pads. To change the setting to noncontiguous, the parameter value **off**, must follow the keyword **contiguous_pad_groups**. In this case, nonmember pads could be placed between the pads.

***contiguous_pad_groups:** ☐ off ☒ on

The optional keyword **minimum_pad_space**, allows the user to specify a minimum space between the I/O pads. The default minimum space between pads is 0.

***minimum_pad_space:**

Graphics User Interface

By default, each itool program on startup looks for the appropriate Tcl script file based on a suffix assigned to each program and executes it. The optional **autodetect_script** allows the user to turn off this behaviour. The assigned suffixes are given below:

Program	Code	Suffix
<i>ifp</i>	<i>ICFP</i>	.fdo
<i>igp</i>	<i>ICFP</i>	.fdo
<i>iplacesc</i>	<i>ICSC</i>	.pdo
<i>grouter</i>	<i>ICGR</i>	.gdo
<i>igrouter</i>	<i>ICGR</i>	.gdo
<i>iroute</i>	<i>ICRT</i>	.rdo
<i>idetailer</i>	<i>ICDR</i>	.ddo

◇ off ◇ on

The optional **background** keyword allows the user to modify the background color. By default, the background color is black. The user may enter any valid X11 color. For example, ***background : red** and ***background : #ff0000** are both valid inputs and both set the background color of the graphics to red.

The optional **graphics** keyword allows the user to turn off the X11 graphics display for **itools**. It is recommended to leave the graphics on (default) until it can be determined that the input data is correct.

◇ {DEFAULT=on ◇ off ◇ on

The optional **graphics_wait** keyword allows the user to control whether **itools** will enter a wait state after the execution of each step of the placement and global routing algorithms. The default is not to wait.

◇ off ◇ on

Genrows Parameters

Below are the parameters specific to the genrows subprogram of the floorplanning program. It is also possible to edit the [general parameters](#).

GENR*block_alignment:	<input type="radio"/> ALIGN_CENTER <input type="radio"/> ALIGN_BOTTOM <input type="radio"/> ALIGN_TOP <input type="radio"/> ALIGN_SUPPLY_PINS
	Layer: <input type="text"/>
GENR*doubleback_rows:	<input type="radio"/> off <input type="radio"/> on
GENR*double_height_rows:	<input type="radio"/> off <input type="radio"/> odd <input type="radio"/> even
GENR*feed_percentage:	<input type="text"/>
GENR*feed_taper:	<input type="radio"/> off <input type="radio"/> on
GENR*generate_rows:	<input type="radio"/> off <input type="radio"/> on
GENR*graphics_wait:	<input type="radio"/> off <input type="radio"/> on
GENR*minimum_row_len:	<input type="text"/>
GENR*numrows:	<input type="text"/>
GENR*rowSep:	<input type="text"/>
GENR*row_to_tile_spacing:	<input type="text"/>
GENR*taper_percentage:	<input type="text"/>
GENR*xgrid:	<input type="text"/>
GENR*ygrid:	<input type="text"/>

Function

The function of the **Genrows** module of the floorplanner is to define and modify the rows or blocks of a standard or mixed cell design.

Parameters

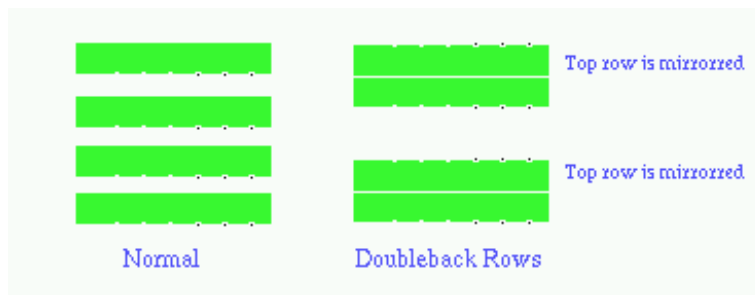
The optional keyword **block_alignment** controls the y-alignment of standard cells within the standard cell rows. This has meaning when variable height standard cells exist in the design. The figure below shows the four methods of aligning the cells within the row. The default is to align the cells by their center. The **ALIGN_SUPPLY_PINS** option may be followed by the layer of interest if the supply is implemented in more than one layer.



GENR*block_alignment: ☐ ALIGN_CENTER ☐ ALIGN_BOTTOM ☐ ALIGN_TOP ☐ ALIGN_SUPPLY_PINS

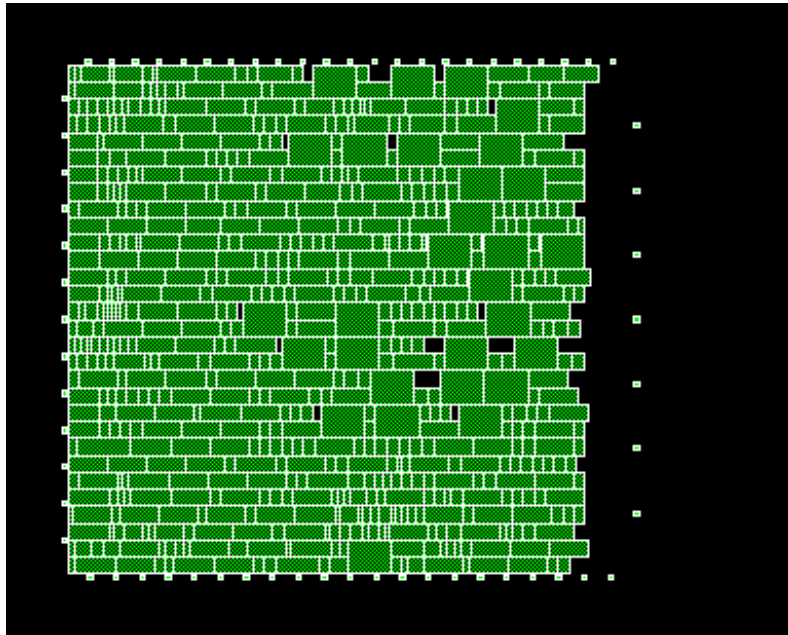
Layer:

The optional keyword **doubleback_rows** enables the doubleback row generation code. A double back row is defined as two rows grouped as a single entity with the second row mirrored around the x-axis as shown in the figure. The default is for double back generation to be turned off.

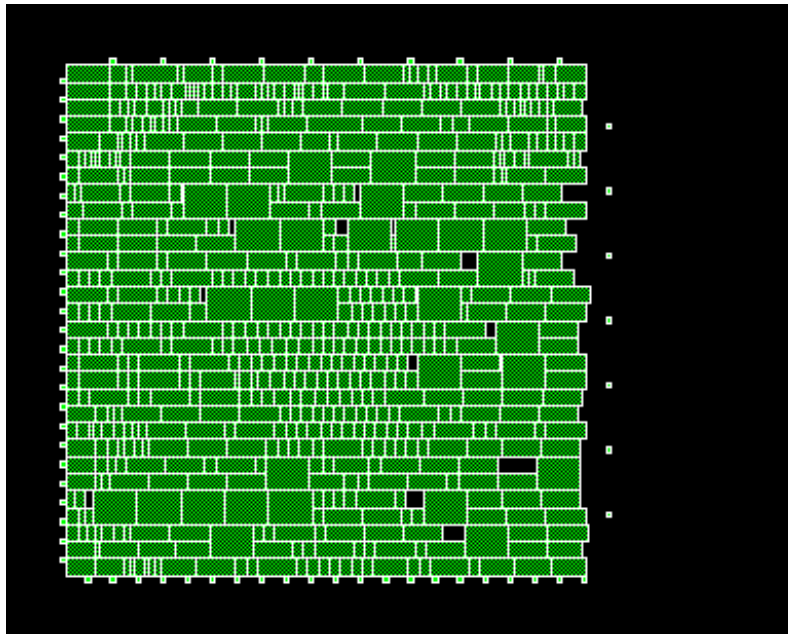


GENR*doubleback_rows: ☐ off ☐ on

The optional keyword **double_height_rows** enables the double height row placement code. A double height cell is a cell which spans two rows. A double height row is defined as two rows grouped as a single entity as shown in the figure. Due to power supply concerns, all double height cells must have a consistent origin within a row. Two global choices are possible: *odd* and *even*. If *odd* is chosen, the origin (lower left of cell) must occur in odd rows whereas if *even* is chosen, the origin must occur in even rows. The rows are numbered from 1 starting at the bottommost row. The default is for double height row definitions to be turned off.



Double height cells origin occur in odd number rows



Double height cells origin occur in even number rows

GENR*double_height_rows:

☒ off ☐ odd ☐ even

The **keyword feed_percentage** is followed by a floating point number, which specifies the amount of space to be reserved for feed through cells. The amount of cell width reserved will be the *feed_percentage* multiplied by the total width of the row-based cells. **Iplace** reports the feed percentage of the current execution at the bottom of the *designName.out* file, if global routing has been requested.

GENR*feed_percentage:

The **feed_taper** keyword enables or disables the tapering of rows to accommodate feed throughs. For designs which do not have sufficient implicit feed through cells, it is recommended that feed tapering be enabled.

GENR*feed_taper: ◇ off ◇ on

The optional keyword **generate_rows**, enables all output from the genrows module of the floorplanner. If **generate_rows** is turned off, the program will execute, but no output will be generated. This is useful if the user has previously generated a *itools.con* file.

GENR*generate_rows: ◇ off ◇ on

The **graphics_wait** keyword allows the user to control whether Genrows will enter a wait state after configuring the rows.

GENR*graphics_wait: ◇ off ◇ on

Genrows breaks the core area into tiles. The keyword **minimum_row_len**, sets a limit on the size of a valid tile, that is, any tile whose width is smaller than the **minimum_row_len** will not have rows.

GENR*minimum_row_len:

In the case of designs consisting only of row-based cells, the number of cell rows may be set to the value following the keyword **numrows**. This parameter has precedence over the rowSep parameter in calculating the spacing between standard cell rows.

GENR*numrows:

The required keyword **rowSep** is followed by a floating point number representing the desired amount of separation between the rows. The amount of separation between the rows is this number times the average height of the rows. That is, if you want the row separation to be equal to the average row height, then this number should be 1.0. On the other hand, if you want the row separation to be twice the height of the rows, then this number should be 2.0. Normally, a value of 1.0 is appropriate.

GENR*rowSep:

The class for a row may be set using the keyword **row_class**. The **numrows** keyword must be specified before any row_class parameter may be set. For example, suppose there are three rows in a design and the following is specified in the *design.par* file:

```
numrows:3
row_class:12
row_class:23
```

In this case, the first row is assigned class 2, and the second row is assigned class 3. The third row which was not specified will be assigned the default class 1.

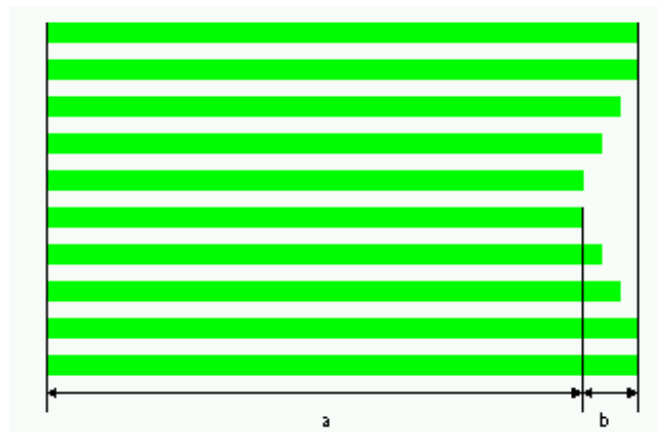
The optional keyword **row_to_tile_spacing** allows the user to modify the distance between the edge of tile and the beginning and end of a row. The default distance is 0.

GENR*row_to_tile_spacing:

The amount of feed taper can be controlled by using the **taper_percentage** keyword. The floating point number controls the amount of indentation. Figure 3.3 shows an example of feed tapering. The amount of tapering is determined by:

$$\text{taper_percentage} = 100 \frac{b}{a + b}$$

where a is length of the shortest row and b is the maximum amount of indentation. The default taper percentage is 10% and should be sufficient for most designs without feedthrus. **Taper_percentage** must be between 0 and 50%.



Example of feed tapering.

GENR*taper_percentage:

The optional keyword **xgrid** is followed by a floating point number, which causes the lower left coordinate of the rows of standard cells to fall on the specified grid. An optional second floating point number specifies an offset to the x grid. By default, the grid has zero offset. For example, if "GENR*xgrid : 5" is specified, the standard cell rows' x-position will be forced to be a multiple of 5. If "GENR*xgrid : 5 1" is given, the valid x-positions for a row are ..., -9, -4, 1, 6, 11,

GENR*xgrid:

The optional keyword **ygrid** is followed by a floating point number, which causes the lower left coordinate of the rows of standard cells to fall on the specified grid. An optional second floating point number specifies an offset to the y grid. By default, the grid has zero offset. For example, if "GENR*ygrid : 7.1" is specified, the standard cell rows' y-position will be forced to be a multiple of 7.1. If "GENR*ygrid : 7.1 1" is given, the valid y-positions for a row are ..., -6.1, 1, 8.1, 15.2,

GENR*ygrid:

[Go on to next program:Simplify](#)

Itools Documentation

```
#!/bin/sh
# This comment make this script an executable. Do *NOT* delete eval exec ${ICDIR}/bin/icos ICTk -n -d

#
# $Id: icdaemon,v 1.8 2007/07/16 18:36:04 bills Exp $
# $Log: icdaemon,v $
# Revision 1.8 2007/07/16 18:36:04 bills
# Now log stderr messages to a log file.
#
# Revision 1.7 2007/07/13 16:17:16 bills
# Now icdaemon uses countdown counter during waits for prettier output.
# In addition, we now remove stale suicide files so we don't kill ourselves
# unnecessarily.
#
# Revision 1.6 2007/06/25 00:13:39 bills
# Added -- to end the list of arguments to ICTk.
#
# Revision 1.5 2007/06/25 00:09:10 bills
# Added the -suicide and -restart options for ease of use.
#
# Revision 1.4 2007/06/17 03:02:09 bills
# Now allow arguments to icdaemon command. We also rename the exit
# commands so that the user doesn't accidentally kill the daemon. Now
# you must explicitly enter the icexit_server command to kill the icdaemon
# server.
#
# Revision 1.3 2006/05/07 13:22:20 bills
# Now allow a -debug switch from the command line.
#
# Revision 1.2 2005/12/09 06:10:37 bills
# Now icdaemon commits suicide when the file icdaemon_suicide is present
# rather than when icdaemon is not present so we don't have trouble
# when NFS acts up.
#
# Revision 1.1 2005/07/02 21:45:03 bills
# Initial revision
#
#
# The defaults settings.
set debug 0
set restart_flag 0
set commit_suicide 0

global env
global argv
global icdirG
if {[info exists env(ICDIR)]} {
    set icdirG $env(ICDIR)
} {
    puts stderr "ERROR:cannot get iTools environment variable:ICDIR"
    exit 1
}
lappend auto_path $icdirG/tcl/common

set args $argv
set num_args [llength $args]
for {set i 0} {$i <$num_args} {incr i} {
    set arg_el [lindex $args $i]
    if {$arg_el == "-debug"} {
        set debug 1
    }
}
```


Ittools Documentation

```
    puts stderr "debug mode enabled.\n"
} elseif {$arg_el == "-suicide" } {
    puts stderr "commit suicide mode enabled.\n"
    set commit_suicide 1
} elseif {$arg_el == "-stop" } {
    puts stderr "commit suicide mode enabled.\n"
    set commit_suicide 1
} elseif {$arg_el == "-restart" } {
    puts stderr "restart mode enabled.\n"
    set commit_suicide 1
    set restart_flag 1
}
}

set hname [exec hostname]
set lockfile [file native ~/.itools/icdaemon.${hname}.pid]
set locked [icfilelock test $lockfile]
if {$locked} {
    if {$commit_suicide} {
        global mystatusG myoutG myerrorG

        set rpid [exec cat $lockfile]
        puts stderr "Detected a running Remote Startup Daemon. Process id:$rpid"
        puts stderr "Suicide begins..."
        puts stderr "First try graceful exit..."
        # Try file shutdown.
        set suicide_file [file join $icdirG icdaemon_suicide]
        set f [open $suicide_file w]
        puts $f "die"
        close $f
        # wait 30 seconds for suicide
        puts -nonewline stderr "Waiting 1:30 for suicide to take effect..."
        ::icschedule::countdown 90
        vwait ::icschedule::waitvarS
        puts stderr "done."

        # Remove suicide file
        file delete -force -- $suicide_file
        set locked [icfilelock test $lockfile]
        if {$locked} {
            puts stderr "Suicide failed. Applying more drastic means..."
        } else {
            if {$restart_flag} {
                puts stderr "Suicide sucessful. Now attempting restart..."
                set cancel_id [after 10000 {set mystatusG timeout}]
                if {[catch {eval bgexec mystatusG -output myoutG -error myerrorG  ${icdirG}/bin/icos icdaemon &
                    puts stderr "ERROR:$result\n"
                    if {[info exists myoutG]} {
                        puts stderr "$myoutG\n"
                    }
                } else {
                    after cancel $cancel_id
                    if {[info exists myerrorG]} && ($myerrorG != "") {
                        puts stderr "$myerrorG\n"
                    } else {
                        if {[info exists result]} && ($result != "") {
                            puts stderr "$result"
                        }
                    }
                }
            } else {
                puts stderr "Suicide sucessful. You may now restart the daemon..."
            }
        }
    }
}
```

```

    }
    exit 0
}
exec kill -9 $rpid
puts -nonewline stderr "Waiting to kill program pid:$rpid..."
::icschedule::countdown 10
vwait ::icschedule::waitvarS
puts stderr "done."
set locked [icfilelock test $lockfile]
if {$locked} {
    puts stderr "Suicide failed again. Please kill $rpid manually. Perhaps we don't have permission."
    exit 1
}
puts stderr "Suicide sucessful. Now we must cleanup..."
set cancel_id [after 10000 {set mystatusG timeout}]
set results ""
if {[catch {eval bgexec mystatusG -output myoutG -error myerrorG echo license taginfo cleanup| $}]} {
    puts stderr "ERROR:$result\n"
    if {[info exists myoutG]} {
        puts stderr "$myoutG\n"
    }
} else {
    after cancel $cancel_id
    if {[info exists myerrorG] && ($myerrorG != "")} {
        puts stderr "$myerrorG\n"
    } else {
        if {[info exists result] && ($result != "")} {
            puts stderr "$result"
        }
    }
}
puts -nonewline stderr "Now waiting for tags to re-register..."
::icschedule::countdown 60
vwait ::icschedule::waitvarS
puts stderr "done."
if {$restart_flag} {
    puts stderr "Now restarting icdaemon..."
    set cancel_id [after 10000 {set mystatusG timeout}]
    if {[catch {eval bgexec mystatusG -output myoutG -error myerrorG ${icdirG}/bin/icos icdaemon &}]} {
        puts stderr "ERROR:$result\n"
        if {[info exists myoutG]} {
            puts stderr "$myoutG\n"
        }
    } else {
        after cancel $cancel_id
        if {[info exists myerrorG] && ($myerrorG != "")} {
            puts stderr "$myerrorG\n"
        } else {
            if {[info exists result] && ($result != "")} {
                puts stderr "$result"
            }
        }
    }
} else {
    puts stderr "You may now restart the daemon..."
}
exit 0

} else {
    puts stderr "ERROR: Remote Startup Daemon already running. Process id:[exec cat $lockfile]"
    exit 0
}
}

```

Itools Documentation

```
} else {
    # Suicide file should not be present delete it if it exists.
    set suicide_file [file join $icdirG icdaemon_suicide]
    if {[file exists $suicide_file]} {
        # Remove suicide file
        icmessage warnmsg icdaemon "attempting to delete stale suicide file..."
        file delete -force -- $suicide_file
        icmessage warnmsg null "done.\n"
    }

    if {$commit_suicide} {
        if {$restart_flag} {
            icmessage warnmsg icdaemon "no Remote Startup Daemon running.\n"
        } else {
            icmessage errmsg icdaemon "no Remote Startup Daemon running.\n"
            exit 1
        }
    }
}

# Because users may type exit in the client, we wouldn't want that to get
# interpreted to be a server exit.  For that reason we rename exit to
# icexit_server

rename exit ""
rename icexit icexit_server

if {$debug} {
    icremote_server -comment {Remote Startup Daemon} -debug -debug
    icwait
}

package require Expect

proc keep_alive { } {
    global icdirG

    # Suicide by this method wasn't any good because file system problem (NFS)
    # can cause a suicide unexpectedly.  Better to make it explicit.
    # So to make the daemon die just create icdaemon_suicide at the top level.
    #
    # First make sure I still exist
    # set me [file join $icdirG bin icdaemon]
    # if {!([file exists $me])} {
    #     # Suicide is painless
    #     exit 0
    # }

    # New explicit version
    set me [file join $icdirG icdaemon_suicide]
    if {[file exists $me]} {
        # Suicide is painless
        exit 0
    }
    icremote_advertize
    after 60000 {keep_alive}
}

set pid [fork]
if {$pid==0} {
    disconnect
}
```

Itools Documentation

```
iclog stderr -noappend [file native ~/.itools/icdaemon.${hname}.log]
icremote_server -comment {Remote Startup Daemon}
set locked [icfilelock create $lockfile]
keep_alive
icwait
}
puts stdout "Icdaemon started.  Process id = $pid.\n"
flush stdout
exit
```

Itranslate Tcl Command : icvias

A via defines the connection between two routing layers. To add a via to the design without a technology file, we must use the **icvias** Tcl command. The form of the command is as follows

```
icvias add viaName layer1 layer2 geom1 geomc geom2
where

layer1, layer2
    are the two routing layers of the via

geom1, geomc, and geom2
    are the geometries of the via on each layer
    specified using Tcl lists in the form: {layer llx lly urx ury}
    There should be two routing rectangles and one cut rectangle in a
    via definition. Each geometry specifies the geometry relative to
    the center of the via.
```

For example:

```
icvias add via12 METAL1 METAL2 \
    {METAL1 -1.0 -1.0 1.0 1.0} \
    {VIA -0.5 -0.5 0.5 0.5} \
    {METAL2 -1.0 -1.0 1.0 1.0}
```

In this example, we are creating a via called *via12* which connects between routing layers *METAL1* and *METAL2*. The via consists of three geometries: two metal rectangles and a cut geometry. As you can see, the geometries are relative to the center of the via. Please note that in Tcl, this command must reside on a single line. If you wish to split it over different lines, use the backslash character as shown.

ICDR – IDETAILER parameters (Detailed Router) parameters

Below are the parameters specific to the detail router program. It is also possible to edit the [general parameters](#).

ICRT*check_ports:	<input type="radio"/> off <input type="radio"/> on
ICRT*contiguous_pad_groups:	<input type="radio"/> off <input type="radio"/> on
ICRT*graphics:	<input type="radio"/> off <input type="radio"/> on
ICRT*graphics_wait:	<input type="radio"/> off <input type="radio"/> on
ICRT*minimum_pad_space:	<input type="text"/>
ICRT*padspacing:	<input type="radio"/> abut <input type="radio"/> exact <input type="radio"/> uniform <input type="radio"/> variable
ICRT*script:	<input type="text"/>

The optional keyword **minimum_pad_space** allows the user to specify a minimum space between the I/O pads.

By default, members of the pad groups are placed contiguously. In other words, no nonmember pads can be placed between the member pads. To change the setting to noncontiguous, the parameter value **off** must follow the keyword **contiguous_pad_groups**. In this case, nonmember pads could be placed between the pads.

The graphics system has three control keywords: **graphics** which allows control of the X11 graphics display, **graphics_wait** which tells IPLAN to wait for the user to enter commands after each step in the process, and **graphics_update** which can turn off the graphics update until the end of the simulated annealing run. To continue execution from a graphics wait loop, the user clicks on the FILE menu and selects CONTINUE PGM. See the section on graphics for more details concerning the graphics capabilities and commands.

The optional keyword **padspacing** controls the pad spacing mode. There are four modes of operation: **uniform** pad spacing, **abutting** pad spacing, **variable** pad spacing, and **exact** pad spacing as shown in Figure 3.4. Uniform pad spacing, spaces the pads evenly on each of the sides. In the abut mode, pads are forced to touch one another. The variable pad spacing mode places each pad such that the wire length is minimized. The last mode turns off the pad spacing algorithm and the pads remain in the place specified by the user in the *designName.con* file. In the first three cases, the side and sidespace constraints are observed. The default mode is uniform padspacing.

[Return to Editing the Parameter File](#)

ICFP – IPF parameters (Floorplanner) parameters

Below are the parameters specific to the floorplanning program. It is also possible to edit the [general parameters](#).

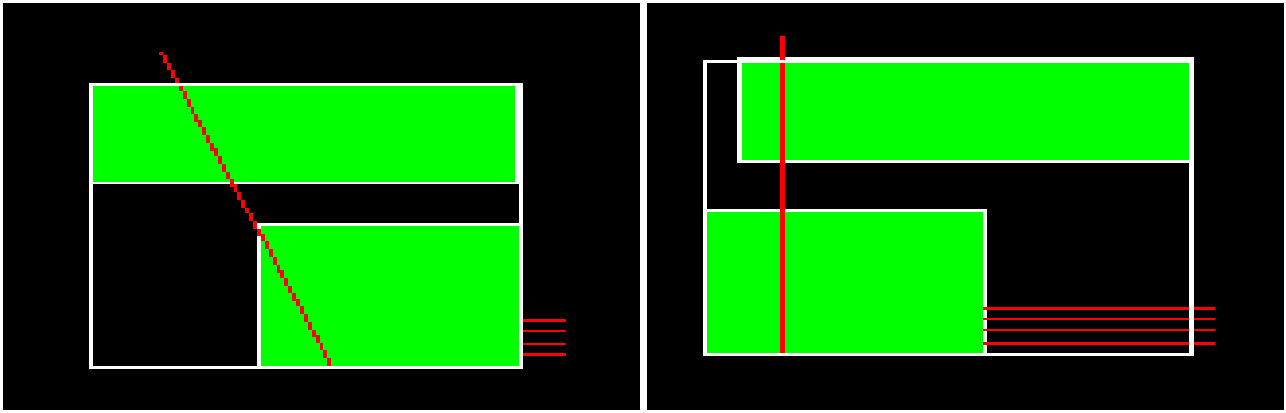
ICFP*autodetect_script:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICFP*background:	<input type="text"/>
ICFP*bendcost_threshold:	<input type="text"/>
ICFP*check_ports:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICFP*chip_aspect_ratio:	<input type="text"/>
ICFP*cost_only:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICFP*contiguous_pad_groups:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICFP*core:	<input type="text"/>
ICFP*core_to_padspace:	<input type="text"/>
ICFP*default_tracks_per_channel:	<input type="text"/>
ICFP*default_block_xspace:	<input type="text"/>
ICFP*default_block_yspace:	<input type="text"/>
ICFP*default_block_yspace:	<input type="text"/>
ICFP*detail_script_file:	<input type="text"/>
ICFP*do_compaction:	<input type="text"/>
ICFP*fast:	<input type="text"/>
ICFP*graphics:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICFP*graphics_update:	<input type="text"/>
ICFP*graphics_wait:	<input checked="" type="checkbox"/> off <input type="checkbox"/> on
ICFP*gridOffsetX:	<input type="text"/>

ICFP*gridOffsetY:	<input type="text"/>
ICFP*gridX:	<input type="text"/>
ICFP*gridY:	<input type="text"/>
ICFP*minimum_pad_space:	<input type="text"/>
ICFP*origin:	<input type="text"/>
ICFP*padspacing:	<input type="radio"/> abut <input type="radio"/> exact <input type="radio"/> <u>u</u> niform <input type="radio"/> variable
ICFP*random_seed:	<input type="text"/>
ICFP*restart:	<input type="radio"/> off <input type="radio"/> on <input type="radio"/> pl1_relative <input type="radio"/> pl1_absolute
ICFP*rowSep:	<input type="text"/>
ICFP*slow:	<input type="text"/>
ICFP*vertical_path_weight:	<input type="text"/>
ICFP*vertical_wire_weight:	<input type="text"/>

By default, each itool program on startup looks for the appropriate Tcl script file based on a suffix assigned to each program and executes it. The optional `autodetect_script` allows the user to turn off this behaviour. In the case of floorplanning, the startup script would be named *designName.fdo*. The default behaviour is for the floorplanner to source this Tcl file if present.

ICFP*autodetect_script:	<input type="radio"/> off <input type="radio"/> <u>o</u> n
-------------------------	--

The keyword **bendcost_threshold** is followed by a floating point number which specifies the amount of wire length which may be traded off in order to make *bend-free* connections during placement. As a convenience, the value may be suffixed with the letter *t* which multiplies the given value by the average track or pitch size. For example, **ICSC*bendcost_threshold : 10t** would make the threshold equal to 10 routing pitches. This option pertains only to two-pin nets. The default value for `bendcost_threshold` is 0.0, that is, no tradeoff of wire length is made. The left picture below shows the result of placement using a `bendcost_threshold` of 0.0. Notice that the many connections between the bottom cell and the I/O pads draw that cell to the right. However, this increases the wire length in the x-direction of the connection between the top I/O cell and the bottom cell. In the picture on the right, the `bendcost_threshold` was set to *20t* or 20 tracks. Now the x component of the wire length is reduced to 0 between these two pins (no bends) at the cost of a maximum of 20 tracks worth of wire length. This constraint works best in the second-generation floorplanner *igp*.



ICFP*bendcost_threshold:

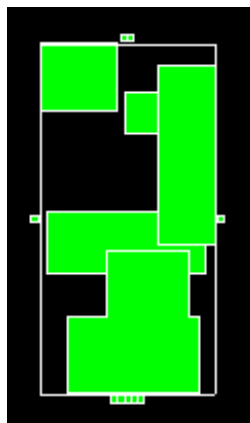
The keyword **default_block_xspace** is followed by a floating point number which specifies the minimum amount of space between cells in x direction. The default amount of space is the average pitch of all vertical routing layers. The number may be zero if the cells touch in the x direction. This space may be overridden by the use of the *cells_may_abut* keyword which will set the default block space to zero.

ICFP*default_block_xspace:

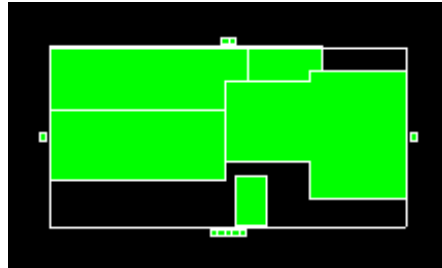
The keyword **default_block_yspace** is followed by a floating point number which specifies the minimum amount of space between cells in y direction. The default amount of space is the average pitch of all horizontal routing layers. The number may be zero if the cells touch in the y direction. This space may be overridden by the use of the *cells_may_abut* keyword which will set the default block space to zero.

ICFP*default_block_yspace:

The keyword **chip_aspect_ratio** is followed by a floating point number which specifies the desired aspect ratio for the chip. ICFP uses this parameter to compute the dimensions of the core area. The cell placement is influenced in such a manner as to yield an aspect ratio close to the specified value. The aspect is defined as the height of the chip to the width of chip. For example, our first picture below has an aspect ratio of 2 whereas the second picture has an aspect ratio of 0.5.



Aspect ratio set to 2.0



Aspect ratio set to 0.5

ICFP*chip_aspect_ratio:

The optional keyword **core** allows the user to specify the exact positions of the chip core area. The four integers following the **core** keyword specifying the dimensions are left side, bottom side, right side, and top side of the chip, respectively. If fixed cells are present in the *designName.con* file, the chip core dimensions should be specified so that a frame of reference is available for determining the fixed cells positions. If the keyword **initially** is specified, IFP will determine the core area after considering the fixed cells; otherwise, IFP is constrained to place the cells in the given core area. Unless the user is certain of the routing area of all of the cells, the keyword **initially** *should* be specified.

ICFP*core:

The optional keyword **core_to_padspace** allows the user to specify the separation between the core area and the I/O pads. The **core_to_padspace** keyword is followed by one, two or four floating point numbers which describe the spacing between the core area and the inside boundary of the I/O pads. When a single number follows **core_to_padspace**, all four sides (left right bottom top) are given the spacing equally. If two numbers are given, the first number will set the core to pad spacing for the left and right pads and the second number will describe the spacing for the top and bottom pads. Four numbers will describe the spacing on the left, bottom, right, and top side respectively.

ICFP*core_to_padspace:

The optional keyword **default_tracks_per_channel** tells the compaction program and global router to allocate an additional number of tracks (above density) in each channel. The space allocated is the given number of tracks multiplied by the track pitch calculated for that direction value.

ICFP*default_tracks_per_channel:

The optional keyword **detail_script_file** is the name of the Tcl script file which contains commands to direct the detail router. The global router will call the detail router if the floorplanning script executes the [icdetail](#) Tcl command. The **icdetail** Tcl command may override this setting or provide a default script. This option is a convenience function.

ICFP*detail_script_file:

Four optional parameters are available for fixing the lower left hand corner of a cell to a grid or lattice, often useful for PCB applications. The parameters **gridX** and **gridY** define the grid to grid spacing in the horizontal and vertical directions, respectively, and the parameters **gridOffsetX** and **gridOffsetY** allow the grid to be shifted from the origin. All four must be specified simultaneously.

The optional keyword **minimum_pad_space** allows the user to specify a minimum space between the I/O pads.

The optional keyword **origin** allows the user to specify the origin of the core region. The two integers following the **origin** keyword specify the lower left corner of the core area.

The keyword **vertical_path_weight** is required. The floating point number represents the cost for one unit of vertical path length, given that the cost for one unit of horizontal path length is unity. This features allows the user to specify that the capacitance (or, in some sense, the delay) per unit length is different for the vertical routing layer as opposed to the horizontal routing layer. IFP will seek to ensure that for each path specified in the *designName.con* file, the horizontal path length plus the vertical_path_weight times the vertical path length is between the upper and lower bound for that path.

The keyword **cost_only** is an optional entry in the *designName.par* file allowing for the bypass of the simulated annealing placement algorithm. Its presence will result in IFP reading the input file, generating an initial placement from the coordinates given in the input data, computing the initial cost, generating output files, and then terminating.

By default, members of the pad groups are placed contiguously. In other words, no nonmember pads can be placed between the member pads. To change the setting to noncontiguous, the parameter value **off** must follow the keyword **contiguous_pad_groups**. In this case, nonmember pads could be placed between the pads.

ICFP*contiguous_pad_groups:

◇ off ◇ on

If **do_compaction** is present in the *designName.par* file, the program will iterate the following flow, given the number (specified by the integer) of times: compaction, channel generation, and global routing. This is the placement modification phase. After each iteration, the space required for routing is accounted for by the compactor, and in the next cycle the program attempts to minimize the chip area using the current knowledge of the routing. Usually three iterations are sufficient for the placement to converge.

ICFP*do_compaction:

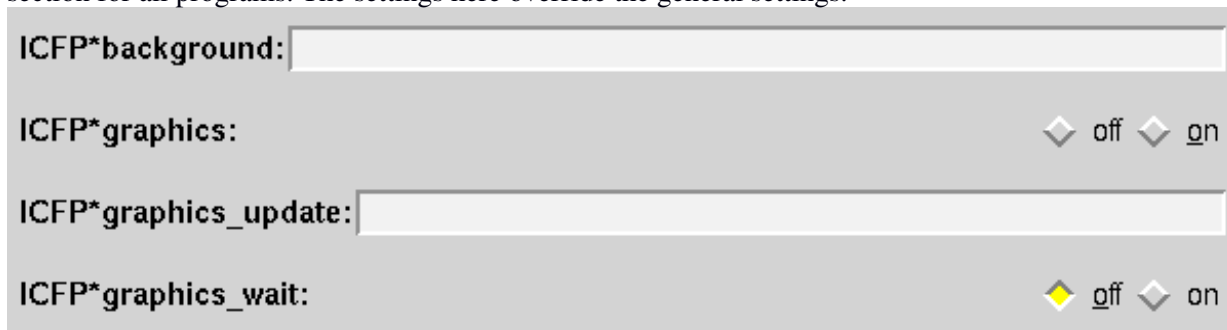
By default, IFP performs the simulated annealing placement algorithm. The user may control the run time of the

simulated annealing algorithm. The keyword **fast** followed by an integer number shortens the running time of the simulated annealing algorithm by the specified integer factor (possibly at the expense of placement quality). To increase the placement quality (at the expense of running time) use the keyword **slow** followed by an integer multiplying factor. Usually the default amount is sufficient but it is recommended that the **fast** option be used on initial runs. The placement is normally the output of the simulated annealing algorithm; however, the user may specify a placement by fixing all the cells in the *designName.con* file and using the keyword **cost_only** to avoid IFP's placement algorithm.

The graphics system has several control keywords: **background** keyword allows the user to modify the background color, **graphics** which allows control of the X11 graphics display, **graphics_wait** which tells IFP to wait for the user to enter commands after each step in the process, and **graphics_update** which can turn off the graphics update until the end of the simulated annealing run. To continue execution from a graphics wait loop, the user clicks on the green

Continue

Continue button found in the lower left corner of the graphics display. See the section on graphics for more details concerning the graphics capabilities and commands. In addition, these options may be set in the general section for all programs. The settings here override the general settings.



ICFP*background:

ICFP*graphics: ☐ off ☒ on

ICFP*graphics_update:

ICFP*graphics_wait: ☒ off ☐ on

The optional keyword **padspacing** controls the pad spacing mode. There are four modes of operation: **uniform** pad spacing, **abutting** pad spacing, **variable** pad spacing, and **exact** pad spacing as shown in Figure 3.4. Uniform pad spacing, spaces the pads evenly on each of the sides. In the abut mode, pads are forced to touch one another. The variable pad spacing mode places each pad such that the wire length is minimized. The last mode turns off the pad spacing algorithm and the pads remain in the place specified by the user in the *designName.con* file. In the first three cases, the side and sidespace constraints are observed. The default mode is uniform padspacing.

The optional keyword **print_pins** causes IFP to output the names of all the one pin nets in the design into the *designName.mout* file.

The keyword **random_seed** is useful when the output data files have been deleted and the data needs to be regenerated. The random number generator seed is printed in the *designName.mout* file. If the input files are identical, a second run using the same **random_seed** value will yield the exact same output. The *designName.mest* file should not exist if the run was the first execution of IFP.

The optional keyword **restart** must be present in order to resume an execution of IFP. This is useful for resuming a run after a hardware crash or other termination of a run.

[Go on to next program: IPLACE](#)

ICGR – IGRROUTER (Row-based global router) parameters

Below are the parameters specific to the global routing program. It is also possible to edit the [general parameters](#).

ICGR*adjust_macros:	<input type="radio"/> off <input type="radio"/> on
ICGR*compaction:	<input type="radio"/> off <input type="radio"/> on
ICGR*contiguous_pad_groups:	<input type="radio"/> off <input type="radio"/> on
ICGR*check_pin_access:	<input type="radio"/> off <input type="radio"/> on
ICGR*do_global_route:	<input type="radio"/> off <input type="radio"/> on
ICGR*end_around:	<input type="radio"/> off <input type="radio"/> on
ICGR*fast:	<input type="radio"/> off <input type="radio"/> on
ICGR*feed_search:	<input type="text"/>
ICGR*fixed_width:	<input type="radio"/> off <input type="radio"/> on
ICGR*graphics:	<input type="radio"/> off <input type="radio"/> on
ICGR*graphics_update:	<input type="radio"/> off <input type="radio"/> on
ICGR*graphics_wait:	<input type="radio"/> off <input type="radio"/> on
ICGR*gr_placement_improve:	<input type="radio"/> off <input type="radio"/> on
ICGR*incremental_routing:	<input type="radio"/> off <input type="radio"/> on
ICGR*input_format:	<input type="radio"/> version6 <input type="radio"/> version1
ICGR*ignore_feeds:	<input type="radio"/> off <input type="radio"/> on
ICGR*iterations:	<input type="text"/>
ICGR*library:	<input type="text"/>
ICGR*minimum_pad_spacing:	<input type="text"/>

ICGR*modify_placement:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*modify_do_file:	<input type="text"/>
ICGR*one_pad_connect:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*output_at_density:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*output_pinfile:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*output_regions:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*otc_river_route:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*otc_feed:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*otc_vert:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*over_the_cell:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*over_the_cell_feed_layer:	<input type="text" value="METAL2 METAL4"/>
ICGR*padspacing:	<input type="checkbox"/> abut <input type="checkbox"/> exact <input checked="" type="checkbox"/> uniform <input type="checkbox"/> variable
ICGR*print_pins:	<input type="text"/>
ICGR*random_seed:	<input type="text"/>
ICGR*radius_factor:	<input type="text"/>
ICGR*steiner_points:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*strip_feeds:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*vertical_path_weight:	<input type="text"/>

The optional keyword **library** allows the user to specify more than one library (.lib) file for the design. The string following the library keyword specifies the pathname of the library. The user may specify multiple libraries by listing each library separately.

The optional keyword **minimum_pad_space** allows the user to specify a minimum space between the I/O pads.

The keyword **vertical_path_weight** is required. The floating point number represents the cost for one unit of vertical path length, given that the cost for one unit of horizontal path length is unity. This features allows the user to specify

that the capacitance (or, in some sense, the delay) per unit length is different for the vertical routing layer as opposed to the horizontal routing layer. ITOOLS will seek to ensure that for each path specified in the *designName.con* file, the horizontal path length plus the vertical_path_weight times the vertical path length is between the upper and lower bounds for that path.

By default, members of the pad groups are placed contiguously. In other words, no nonmember pads can be placed between the member pads. To change the setting to noncontiguous, the parameter value off must follow the keyword **contiguous_pad_groups**. In this case, nonmember pads could be placed between the pads.

The keyword **do_global_route** allows the user to turn off the global routing from the *designName.par* file. The default is for global routing to be performed. It is recommended to execute the global router to insert feed throughs even if other global routing information is not utilized.

The **end_around** keyword controls the global routing style for nets near the end of the row. If **end_around** is enabled, the global router may avoid a feed through for nets near the edge of the rows by routing in the I/O channels. The default is to force the routes to stay within the core region.

The **fast** keyword is an optional parameter for controlling the trade-off between CPU time and solution quality. The default is for the best solution quality.

The optional **feed_search** parameter controls the distance that the global router will search for an available implicit feed through from the desired crossing point during feed through assignment. If the global router does not find a feed through in that area, it will add an explicit feed through. The floating point number following the keyword denotes the percentage of the total row length which will be searched. The default is the entire row (100%). The default is recommended. **Feed_search** values less than 10% will cause the addition of excessive feed throughs. It is recommended to use NET PRIORITY to route the time critical nets rather than using the **feed_search** parameter.

The optional **fixed_width** keyword specifies whether the global router is able to add additional explicit feed throughs in order to complete the routing. The default is to allow the global router to add the explicit feeds. If not enough implicit feeds are present to complete the routing and the **fixed_width** option is enabled, the global router will exit with an error.

The optional **graphics** keyword allows the user to turn off the X11 graphics display for IGROUTER. It is recommended to leave the graphics on (default) until it can be determined that the input data is correct.

The optional **graphics_update** keyword allows the user to control whether IGROUTER draws the results after routing. The default is to draw the results.

The optional **graphics_wait** keyword allows the user to control whether IGROUTER will enter a wait state after each routing iteration. The default is not to wait.

By default, the global router performs a placement improvement algorithm to reduce wire length and density. The user may turn this stage off in order to maintain a placement using the optional **gr_placement_improve** keyword. However, the results may suffer.

Incremental global routing is enabled using the **incremental_routing** keyword. In the incremental mode, the global router will read the *designName.pin_incr* and *designName.pl3_incr* files to initialize the global router. Global routing will be performed on all the nets connected to the ECO_ADDED_CELLS and any nets with a REROUTE constraint given in the *designName.con* file. The *designName.pin_incr* and *designName.pl3_incr* files are created by renaming the *designName.pin* and *designName.pl3* files respectively.

The IGRouter global router has a state-of-the-art feed through assignment algorithm. When reading the placement input, the global router will, by default, ignore any explicit feed through cells in the design and use the assignment algorithm to add any necessary feed throughs. The optional parameter **ignore_feeds** allows this function to be turned off. In the incremental global routing mode, the **ignore_feeds** option is always disabled.

The number of times that the global routing algorithm is performed may be controlled using the **iterations** keyword. The default number of iterations is one.

IGROUTER can handle designs in which the cell ports are internal to the cell when the **over_the_cell** option is enabled. Otherwise, IGRouter warns about cell ports which are not on the boundary of the cell. Sea-of-gates circuits should enable the **over_the_cell** parameter.

ICGR*over_the_cell:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*otc_river_route:	<input type="checkbox"/> {DEFAULT=off <input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*otc_feed:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*otc_vert:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICGR*over_the_cell_feed_layer:	<input type="text" value="METAL2 METAL4"/>

By default, IGRouter outputs the global routing at the row separation specified in the input placement file. If the **output_at_density** parameter is enabled, the global router will space the rows according to the calculated density. This option is not well-defined for designs incorporating macro cells.

The optional keyword **padspacing** controls the pad spacing mode. There are four modes of operation: **uniform** pad spacing, **abutting** pad spacing, **variable** pad spacing, and **exact** pad spacing as shown in Figure 3.4. Uniform pad spacing spaces the pads evenly on each of the sides. In the abut mode, pads are forced to touch one another. The variable pad spacing mode places each pad such that the wire length is minimized. The last mode turns off the pad spacing algorithm and the pads remain in the place specified by the user in the *designName.con* file. In the first three cases, the side and sidespace constraints are observed. The default mode is uniform padspacing.

The keyword **random_seed** is useful when the output data files have been deleted and the data needs to be regenerated. The random number generator seed is printed in the *designName.gout* file. If the input files are identical, a second run using the same **random_seed** value will yield the exact same output. The random seed may be set as a general parameter. Use the HTML button below to travel to the general parameters.

ICGR*random_seed:	<input type="text"/>
-------------------	----------------------

The radius factor is a floating point number in the interval [0.0 1.0] which controls the Steiner tree radius during global routing for all nets. For proper operation, each net must consist of a single driver and any number of loads. The pin types must be properly defined in the *designName.lib* file. For small radius factors, wire length is more important than the timing from the source to the farthest load. For radius factors approaching 1, the Steiner tree becomes a

shortest path tree. The **RADIUS_FACTOR** may also set on a [net basis from the *designName.con*](#) file.

The radius factor may also be specified as a function of the number of pins on a net. The format of this command is as follows:

```
ICGR*radius_factor : float numpins integer integer
```

where the floating-point number specifies radius factor, and the two integers specify the bounds for the cardinality of pins on a net for which it is valid. For example, the sequence of statements creates a two piece radius function: a value of 0.95 for nets less than or equal to 6 pins and a value of 0.0 for pins greater than 6.

```
ICGR*radius_factor : 0.95 numpins 1 6
```

```
ICGR*radius_factor : 0.0 numpins 7 600
```

Note: that the valid range of pins is [1,600] and that segments may not overlap.

ICGR*radius_factor:

[Go on to next program:iroute](#)

ICSC – IPLACE (Row-based placement) parameters

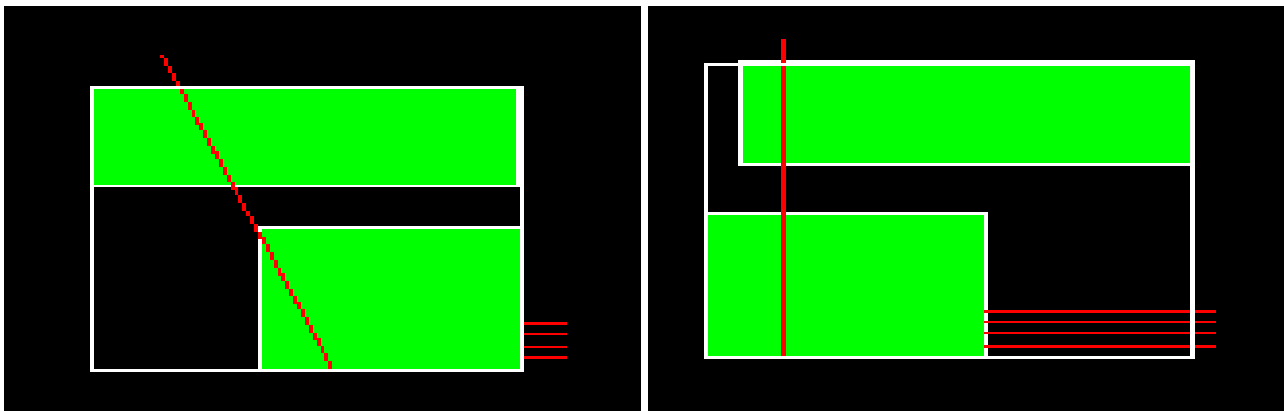
Below are the parameters specific to the placement program. It is also possible to edit the [general parameters](#).

ICSC*add_feeds:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*add_rigid_spacers:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*allow_long_rows:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*allow_pad_overlap:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*auto_wire_weight:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*bendcost_threshold:	<input type="text"/>
ICSC*check_ports:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*cost_only:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*contiguous_pad_groups:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*design_style:	<input checked="" type="checkbox"/> gate_array_with_sites <input type="checkbox"/> gate_array <input type="checkbox"/> stdcell
ICSC*dont_output_pads:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*eco_placement_improve:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*even_placement:	<input type="checkbox"/> off <input checked="" type="checkbox"/> default
ICSC*except_threshold:	<input type="text"/>
ICSC*fast:	<input type="text"/>
ICSC*fix_orientation_problems:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*graphics:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on
ICSC*graphics_update:	<input type="text"/>
ICSC*graphics_wait:	<input type="checkbox"/> off <input checked="" type="checkbox"/> on

ICSC*library:	<input type="text"/>
ICSC*minimum_pad_space:	<input type="text"/>
ICSC*mode:	<input type="radio"/> hierarchical <input type="radio"/> flat
ICSC*max_numrows:	<input type="text"/>
ICSC*old_placement_format:	<input type="radio"/> off <input type="radio"/> on
ICSC*orientation_optimization:	<input type="radio"/> off <input type="radio"/> on
ICSC*orient_opt_unlap:	<input type="radio"/> off <input type="radio"/> on
ICSC*overlap_mode:	<input type="radio"/> allow_space <input type="radio"/> compact <input type="radio"/> left_justify <input type="radio"/> off
ICSC*padding:	<input type="radio"/> abut <input type="radio"/> exact <input type="radio"/> uniform <input type="radio"/> variable
ICSC*parallel:	<input type="radio"/> off <input type="radio"/> probes <input type="radio"/> multi <input type="radio"/> maximum
ICSC*print_pins:	<input type="text"/>
ICSC*probes:	<input type="text"/>
ICSC*processor_limit:	<input type="text"/>
ICSC*prune_amount:	<input type="text"/>
ICSC*random_seed:	<input type="text"/>
ICSC*restart:	<input type="radio"/> off <input type="radio"/> on <input type="radio"/> pl1_relative <input type="radio"/> pl1_absolute
ICSC*row_var_percent:	<input type="text"/>
ICSC*save_file:	<input type="radio"/> off <input type="radio"/> on
ICSC*slow:	<input type="text"/>
ICSC*taper_percentage:	<input type="text"/>
ICSC*time_update:	<input type="text"/>

ICSC*total_row_length:	<input type="text"/>
ICSC*update_overlap:	<input type="text"/>
ICSC*vertical_path_weight:	<input type="text"/>
ICSC*vertical_wire_weight:	<input type="text"/>
ICSC*width_optimize:	<input type="text"/>
ICSC*width_tolerance:	<input type="text"/>

The keyword **bendcost_threshold** is followed by a floating point number which specifies the amount of wire length which may be traded off in order to make *bend-free* connections during placement. As a convenience, the value may be suffixed with the letter *t* which multiplies the given value by the average track or pitch size. For example, **ICSC*bendcost_threshold : 10t** would make the threshold equal to 10 routing pitches. This option pertains only to two-pin nets. The default value for `bendcost_threshold` is 0.0, that is, no tradeoff of wire length is made. The left picture below shows the result of placement using a `bendcost_threshold` of 0.0. Notice that the many connections between the bottom cell and the I/O pads draw that cell to the right. However, this increases the wire length in the x-direction of the connection between the top I/O cell and the bottom cell. In the picture on the right, the `bendcost_threshold` was set to *20t* or 20 tracks. Now the x component of the wire length is reduced to 0 between these two pins (no bends) at the cost of a maximum of 20 tracks worth of wire length.



ICSC*bendcost_threshold:	<input type="text"/>
--------------------------	----------------------

The keyword **cost_only** is an optional entry in the file *designName.par*. Its presence will result in IPLACE reading the input files, generating an initial placement, computing the initial cost, generating the output files, and then a graceful death. It is highly recommended to include this keyword on the first run of the input files. Any errors will be directed to the output file called *designName.out*.

ICSC*cost_only:	<input type="checkbox"/> off <input type="checkbox"/> on
-----------------	--

The optional keyword **eco_placement_improve** controls whether the placement improve algorithm is executed after engineering change order (ECO) cells have been added to the design.

ICSC*eco_placement_improve:

◇ off ◇ on

The optional keyword **even_placement** controls the automatic feed evening step with IPLACE. Normally enabled, IPLACE will automatically indent the rows to compensate for *row bowing* due to the necessary addition of explicit feed throughs. The even_placement algorithm reads the placement file and performs compensation in a subsequent placement run. The even placement algorithm will be executed only if the addition of feedthru cause the rows to become uneven.

ICSC*even_placement:

◇ off ◇ default

The optional keyword **orientation_optimization** instructs IPLACE to perform only the following steps: read the initial placement information from the *designName.ckt* and *circuitfile*, and optimize the orientation of the cells.

ICSC*orientation_optimization:

◇ off ◇ on

The keyword **random_seed** is useful when the output data files have been deleted and the data needs to be regenerated. The random number generator seed is printed in the *designName.out* file. If the input files are identical, a second run using the same **random_seed** value will yield the exact same output.

ICSC*random_seed:

The optional keyword **restart** must be present in order to resume an execution of IPLACE. This is useful for resuming a run after a hardware crash or other termination of a run. Crash recovery is possible if IPLACE has created a *designName.sav* file. In order to recover from a crash, you must rename the *designName.sav* file to *designName.res* and set restart to *on*. In addition, the restart option is useful for entering an initial placement into ITOOLS. Using the **pl1_relative** or **pl1_absolute** keywords causes ITOOLS to read the *designName.pl1_in* file for the initial placement. The **pl1_absolute** mode uses the exact positions given in the *designName.pl1_in* file whereas the **pl1_relative** mode guarantees that the order of the cells in a row will be maintained. To create the *designName.pl1_in* file just copy or rename the *designName.pl1* file.

ICSC*restart:

◇ off ◇ on ◇ pl1_relative ◇ pl1_absolute

The amount of variance between the lengths of rows is controlled by the optional keyword **row_var_percent**. The floating point number following the keyword denotes deviation in row length permitted. The deviation is specified as a percentage of the total row length. It is recommended that the internal value be used.

ICSC*row_var_percent:

Feed Through Options

The optional keyword **add_feeds** controls whether IPLACE will add feed throughs to the design to create a routable design. By default, IPLACE will not add feed throughs; instead, the IGROUTER global router will. This global router has a very sophisticated feed through assignment and area minimization algorithm. However, if the user chooses not to run IGROUTER (and use other-party tools to complete the routing), **add_feeds** should be enabled to create a routable design. **It is recommended that you run IGROUTER and not enable this option. You cannot use this option if you intend to use the itools detail router.**

ICSC*add_feeds:

☐ off ☒ on

Whenever rigidly fixed cells do not align properly with the specified coordinates, *spacer* (feed through) cells are placed before the rigidly fixed cells in order to occupy any empty space. Spacers are placed between the right edge of the last placed cell to the left of the rigidly placed cell and the left edge of the rigidly placed cell. However it is possible to leave the empty spaces and insert spacer cells.

ICSC*add_rigid_spacers:

☐ off ☒ on

Overlap and Row Control Options

The **allow_long_rows** option controls whether IPLACE will strictly obey the row constraints in the *designName.blk* file. If enabled, every cell will be placed within the boundaries of the block or row definition in the *designName.blk* file. Only in the case of an infeasible solution, will this be violated. The option should be turned on to place a gate array. The gate array should be similar to the gate array model used by Cadence's CELL3.

ICSC*allow_long_rows:

☐ off ☒ on

There are now four modes in which ITOOLS will remove any residual cell overlap. The first mode **off** is only valid when **orientation_optimization** is requested. In this mode, ITOOLS will respect the initial placement and only rotate the cell about its center. The **left_justify** mode places cells contiguously starting from the left edge of the block. Any extra space in the row occurs to the right of the cells. This is ITOOLS' default mode. The **compact** mode is similar to the left_justify mode in that there is no space between any of the cells but unlike left_justify the empty space may occur on either end of the row or both. In the last mode, **allow_space**, empty space may occur between any two cells but no two cells overlap. **The left_justify and compact modes are recommended. The allow_space mode will generally result in higher wire length and is not recommended.**

ICSC*overlap_mode:

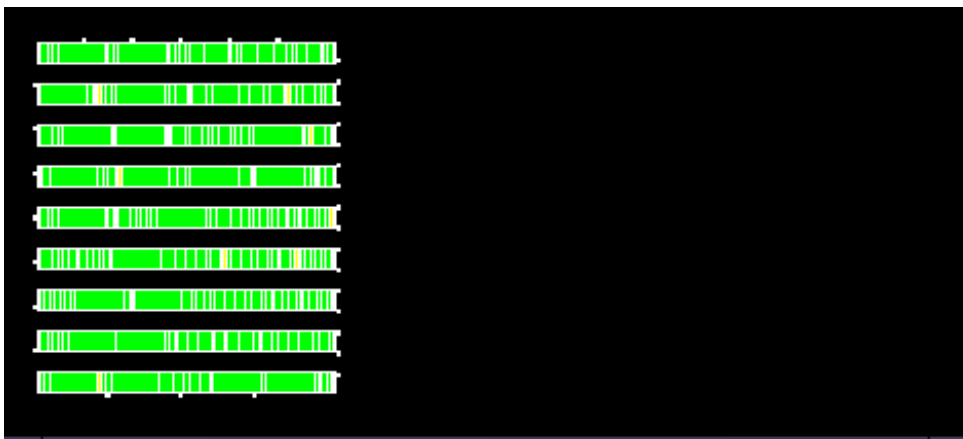
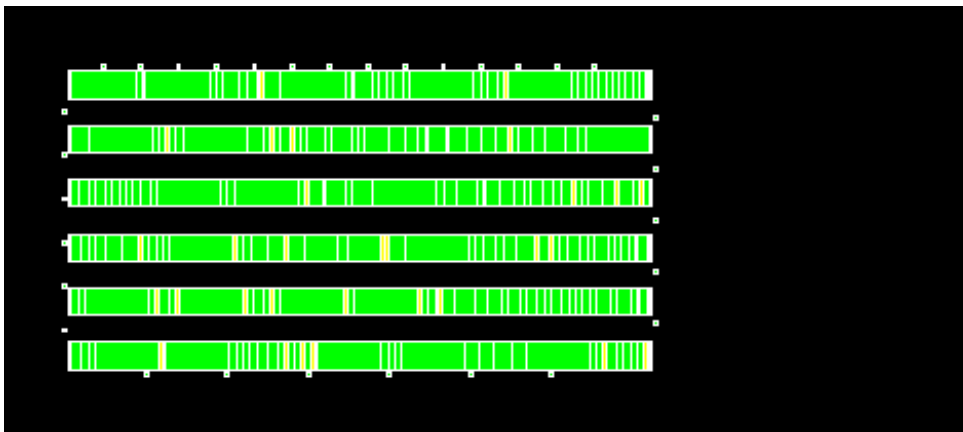
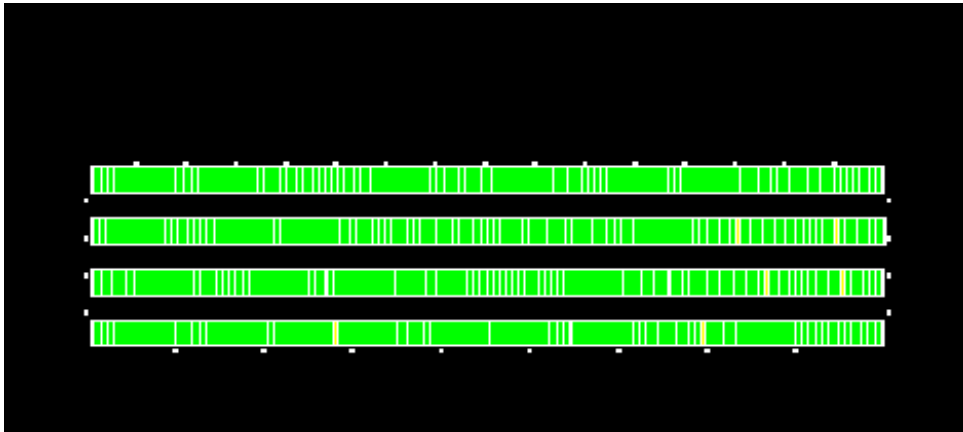
☐ allow_space ☐ compact ☒ left_justify ☐ off

The **allow_pad_overlap** option controls whether it is legal for I/O pads to overlap. By default, it is illegal for pads to overlap and ITOOLS will remove the overlap. If pad overlap is allowed, ITOOLS will honor the pad placement given by the user.

ICSC*allow_pad_overlap:

☐ off ☒ on

The **width_optimize** option enables width optimization when present in the parameter file. The floating point number following the keyword specified the target width of the row-based designed. The **width_optimize** option only has meaning when a width optimization flow has been selected; otherwise it is ignored. Width optimization consists of a series of placement and global routing steps where the number of rows are varied in order to adjust the width of the design. The global router is always able to calculate the final width of a design and so no detail routing is required until the width is finalized. Below is a series of snapshots of a design after global routing. In the first picture, the design is much too wide and too few rows are present. The second picture shows an intermediate step and the final picture show the tradeoff of width versus number of rows and total chip height.



ICSC*width_optimize:

The width optimization target number is implemented as an interval defined by the target width specified by the user plus and minus a tolerance amount. By default, the tolerance is 0.1 time the target or 10 percent. The user may modify the tolerance by entering a non-negative number. Supplying too small a tolerance may cause the program not to

converge.

ICSC*width_tolerance:

The width optimization algorithm varies the number of rows in order to set the width of the design. By default, the maximum number of rows in the design is limited to 1000 during row optimization. The user may use the **max_numrows** option in order to limit the maximum height of the design.

ICSC*max_numrows:

The orientation of cells is extremely important in placement and routing. The itools placer obeys the row definitions for mirroring and orientation. All cells must obey the requirements of the row definitions even fixed cells in which the user has specified an orientation. By default, all cells (including fixed cells) will be swapped to orientations which obey the constraints. However, thru the use of the **fix_orientation_problems** parameter, the user may modify this behavior so that it only applies to free cells; user specified fixed orientation cells will **NOT** be corrected when **fix_orientation_problems** is set to off.

ICSC*fix_orientation_problems:

☐ off ☒ on

Program Speed

There are no required parameters for controlling the quality of the solution and the CPU time used by IPLACE. That is, by default IPLACE is set up to yield what we feel approximates the best attainable solutions. However, experienced IPLACE users may wish to experiment with the optional parameters for controlling the trade-off between CPU time and solution quality. The keywords fast and slow represent the set of optional parameters.

ICSC*fast:

The optional keyword **fast** is followed by an integer n which will result in an execution time that is n times faster than the default. The quality of the placement will tend to decrease as n is made larger than one (the smaller the value of n , the better the result). For chip-planning applications, a value of n in the range of five to ten is recommended.

The keyword **slow** is an optional entry causing IPLACE to be executed about n times slower than the default. The value of n is specified by the integer following the keyword slow. In some cases, you may get a slightly better result. However, only use the slow option if CPU time is of no interest to you.

ICSC*slow:

The optional **graphics_update** keyword allows the user to control whether IPLACE draws the placement after each execution of the outer loop of the simulated annealing algorithm. The default is to draw the placement after each iteration.

ICSC*graphics_update:

The optional **mode** keyword allows the user to force the placement algorithm. If unspecified, ITOOLS will automatically determine the placement algorithm which will give the best results. The flat mode gives the best results for small circuits, that is, for circuits with less than 1000 placeable objects. Otherwise, ITOOLS will chose the hierarchical mode which gives faster and better solutions than flat mode for large circuits. Normally, this should be left unspecified. One notable exception is ECO execution where the mode must be set to flat.

ICSC*mode:

☐ hierarchical ☐ flat

When the hierarchical mode is used, placement is performed in multiple stages. The result after the first two stages is a very good predictor of the final result. We call the execution of the first two stages a *probe*. ITOOLS, by default, will perform a single *probe*. If multiple probes are specified, the first two stages are executed multiple times. After all probes are completed, the best candidate will be run to completion. The probes are independent and may be run in parallel. If a *designName.host* file exists, the probes will automatically be executed in parallel on a network of workstations; otherwise, the probes will be executed sequentially. Any positive number of probes may be performed.

ICSC*probes:

The optional keyword **time_update** controls the frequency in which longest path algorithm is performed during the course of the annealing algorithm. The integer following `time_update` specifies the number of iterations to wait before executing the longest path algorithm. The default is every iteration. For designs with many pin pairs, the longest path search may dominate the run time. This option trades off accuracy for CPU time. Note: this setting has no effect for fully specified timing path constraints; only pin pairs constraints are affected.

ICSC*time_update:

The keyword **prune_amount** is an optional parameter which allows the user to control the number of active paths considered during timing driven placement. The integer following the `prune amount` specifies the number of path segments considered during placement. The default value is recommended.

ICSC*prune_amount:

Miscellaneous Options

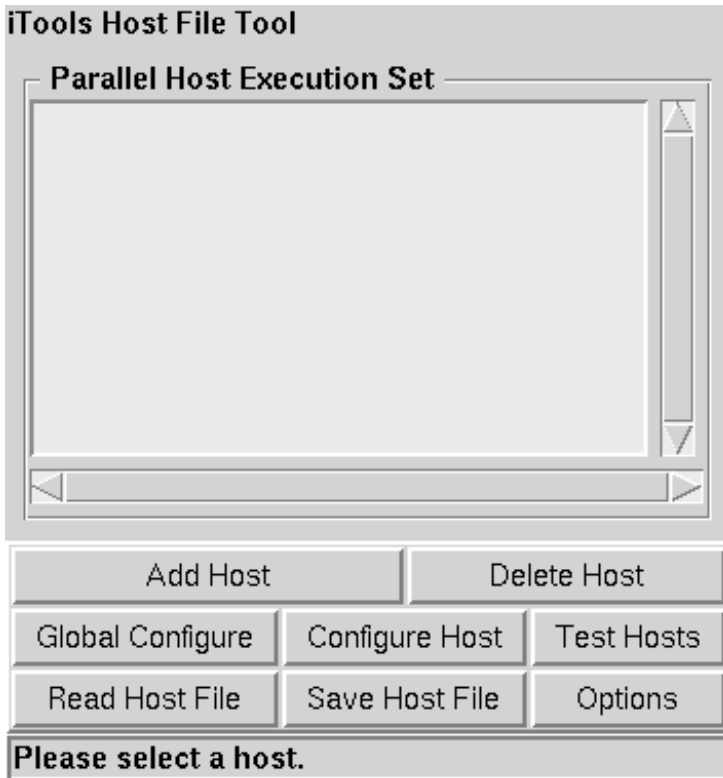
The optional **graphics_wait** keyword allows the user to control whether IPLACE stops at the end of the placement program to enter a graphics wait loop. This loop allows user interaction.

ICSC*graphics_wait:

☐ off ☐ on

Parallel Placement

The **parallel** parameter enables the parallel processing mode. The parallel processing mode recruits network processors to speed the execution of the simulated annealing algorithm. When this option is selected, a *designName.host* file must be present in the design directory to describe the available processors on the network. The tool below will help you create this file. First, depress the **Add Host** button to search the network for available hosts.



The parallel option has four modes: **off**, **probes**, **multi**, and **maximum**. The default is to run ITOOLS in a single process. The **probes** option uses multiple processors to execute a set of probes in parallel. However, in this mode, each ITOOLS stage is executed sequentially. In the **multi** mode, only a single probe is executed but each ITOOLS stage is executed in parallel. The **maximum** mode executes multiple probes and runs each of the stages in parallel. For small designs (< 1000 cells), the **probes** option will give the best results. For larger designs, the **multi** and **maximum** modes are recommended. The **multi** mode is the fastest execution mode whereas the **maximum** will in general yield better results.

ICSC*parallel: ☐ off ☐ probes ☐ multi ☐ maximum

The processor limit controls the maximum number of processors available to the ITOOLS parallel algorithm. Normally, all processors listed in the *designName.host* file are used during the parallel placement algorithm. This option allows the user to limit the number of processors to the amount specified. You may limit processors for each stage of the hierarchical algorithm. By default, all hosts specified in the *designName.host* file are used.

ICSC*processor_limit:

[Go on to next program:IGROUTER](#)

ICSC – IPLACEGA (Row-based gate-array placement) parameters

Below are the parameters specific to the gate-array placement program. It is also possible to edit the [general parameters](#).

ICSC*add_feeds:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*add_rigid_spacers:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*allow_long_rows:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*allow_pad_overlap:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*auto_wire_weight:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*check_ports:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*cost_only:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*contiguous_pad_groups:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*design_style:	<input type="checkbox"/> gate_array_with_sites <input type="checkbox"/> gate_array <input type="checkbox"/> stdcell
ICSC*dont_output_pads:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*eco_placement_improve:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*even_placement:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*except_threshold:	<input type="text"/>
ICSC*fast:	<input type="text"/>
ICSC*graphics:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*graphics_update:	<input type="text"/>
ICSC*graphics_wait:	<input type="checkbox"/> off <input type="checkbox"/> on
ICSC*library:	<input type="text"/>
ICSC*minimum_pad_space:	<input type="text"/>

ICSC*mode:	<input type="radio"/> hierarchical <input type="radio"/> flat
ICSC*old_placement_format:	<input type="radio"/> off <input type="radio"/> on
ICSC*orientation_optimization:	<input type="radio"/> off <input type="radio"/> on
ICSC*orient_opt_unlap:	<input type="radio"/> off <input type="radio"/> on
ICSC*overlap_mode:	<input type="radio"/> allow_space <input type="radio"/> compact <input type="radio"/> left_justify <input type="radio"/> off
ICSC*padding:	<input type="radio"/> abut <input type="radio"/> exact <input type="radio"/> uniform <input type="radio"/> variable
ICSC*parallel:	<input type="radio"/> off <input type="radio"/> probes <input type="radio"/> multi <input type="radio"/> maximum
ICSC*print_pins:	<input type="text"/>
ICSC*probes:	<input type="text"/>
ICSC*processor_limit:	<input type="text"/>
ICSC*prune_amount:	<input type="text"/>
ICSC*random_seed:	<input type="text"/>
ICSC*restart:	<input type="radio"/> off <input type="radio"/> on <input type="radio"/> pl1_relative <input type="radio"/> pl1_absolute
ICSC*row_var_percent:	<input type="text"/>
ICSC*rowSep:	<input type="text"/>
ICSC*save_file:	<input type="radio"/> off <input type="radio"/> on
ICSC*slow:	<input type="text"/>
ICSC*taper_percentage:	<input type="text"/>
ICSC*time_update:	<input type="text"/>
ICSC*total_row_length:	<input type="text"/>
ICSC*update_overlap:	<input type="text"/>

ICSC*vertical_path_weight:	<input type="text"/>
ICSC*vertical_wire_weight:	<input type="text"/>

The keyword **cost_only** is an optional entry in the file *designName.par*. Its presence will result in IPLACE reading the input files, generating an initial placement, computing the initial cost, generating the output files, and then a graceful death. It is highly recommended to include this keyword on the first run of the input files. Any errors will be directed to the output file called *designName.out*.

ICSC*cost_only: ◇ off ◇ on

The optional keyword **eco_placement_improve** controls whether the placement improve algorithm is executed after engineering change order (ECO) cells have been added to the design.

ICSC*eco_placement_improve: ◇ off ◇ on

The optional keyword **even_placement** controls the automatic feed evening step with IPLACE. If enabled, IPLACE will automatically indent the rows to compensate for *row bowing* due to the necessary addition of explicit feed throughs. By default, the even placement algorithm will be executed if necessary.

ICSC*even_placement: ◇ off ◇ on

The optional keyword **orientation_optimization** instructs IPLACE to perform only the following steps: read the initial placement information from the *designName.ckt* and *circuitfile*, and optimize the orientation of the cells.

ICSC*orientation_optimization: ◇ off ◇ on

The keyword **random_seed** is useful when the output data files have been deleted and the data needs to be regenerated. The random number generator seed is printed in the *designName.out* file. If the input files are identical, a second run using the same **random_seed** value will yield the exact same output.

ICSC*random_seed:

The optional keyword **restart** must be present in order to resume an execution of IPLACE. This is useful for resuming a run after a hardware crash or other termination of a run. Crash recovery is possible if IPLACE has created a *designName.sav* file. In order to recover from a crash, you must rename the *designName.sav* file to *designName.res* and set restart to *on*. In addition, the restart option is useful for entering an initial placement into ITOOLS. Using the **pl1_relative** or **pl1_absolute** keywords causes ITOOLS to read the *designName.pl1_in* file for the initial placement. The **pl1_absolute** mode uses the exact positions given in the *designName.pl1_in* file whereas the **pl1_relative** mode guarantees that the order of the cells in a row will be maintained. To create the *designName.pl1_in* file just copy or rename the *designName.pl1* file.

ICSC*restart: ◇ off ◇ on ◇ pl1_relative ◇ pl1_absolute

The amount of variance between the lengths of rows is controlled by the optional keyword **row_var_percent**. The floating point number following the keyword denotes deviation in row length permitted. The deviation is specified as a percentage of the total row length. It is recommended that the internal value be used.

ICSC*row_var_percent:

Feed Through Options

The optional keyword **add_feeds** controls whether IPLACE will add feed throughs to the design to create a routable design. By default, IPLACE will not add feed throughs; instead, the IGROUTER global router will. This global router has a very sophisticated feed through assignment and area minimization algorithm. However, if the user chooses not to run IGROUTER, **add_feeds** should be enabled to create a routable design. **It is recommended that you run IGROUTER and not enable this option.**

ICSC*add_feeds:

◇ off ◇ on

Whenever rigidly fixed cells do not align properly with the specified coordinates, *spacer* (feed through) cells are placed before the rigidly fixed cells in order to occupy any empty space. Spacers are placed between the right edge of the last placed cell to the left of the rigidly placed cell and the left edge of the rigidly placed cell. However it is possible to leave the empty spaces and insert spacer cells.

ICSC*add_rigid_spacers:

◇ off ◇ on

Overlap and Row Control Options

The **allow_long_rows** option controls whether IPLACE will strictly obey the row constraints in the *designName.blk* file. If enabled, every cell will be placed within the boundaries of the block or row definition in the *designName.blk* file. Only in the case of an infeasible solution, will this be violated. The option should be turned on to place a gate array. The gate array should be similar to the gate array model used by Cadence's CELL3.

ICSC*allow_long_rows:

◇ off ◇ on

There are now four modes in which ITOOLS will remove any residual cell overlap. The first mode **off** is only valid when **orientation_optimization** is requested. In this mode, ITOOLS will respect the initial placement and only rotate the cell about its center. The **left_justify** mode places cells contiguously starting from the left edge of the block. Any extra space in the row occurs to the right of the cells. This is ITOOLS' default mode. The **compact** mode is similar to the **left_justify** mode in that there is no space between any of the cells but unlike **left_justify** the empty space may occur on either end of the row or both. In the last mode, **allow_space**, empty space may occur between any two cells but no two cells overlap. **The left_justify and compact modes are recommended. The allow_space mode will generally result in higher wire length and is not recommended.**

ICSC*overlap_mode:

◇ allow_space ◇ compact ◇ left_justify ◇ off

The **allow_pad_overlap** option controls whether it is legal for I/O pads to overlap. By default, it is illegal for pads to overlap and ITOOLS will remove the overlap. If pad overlap is allowed, ITOOLS will honor the pad placement given by the user.

ICSC*allow_pad_overlap:

◇ off ◇ on

except_threshold {on | off}

Program Speed

There are no required parameters for controlling the quality of the solution and the CPU time used by IPLACE. That is, by default IPLACE is set up to yield what we feel approximates the best attainable solutions. However, experienced IPLACE users may wish to experiment with the optional parameters for controlling the trade-off between CPU time and solution quality. The keywords **fast** and **slow** represent the set of optional parameters.

ICSC***fast**:

The keyword **fast** is an optional entry in the file *designName.par*. The inclusion of this keyword will cause IPLACE to be executed about n times faster, where n is the value of the integer following the keyword **fast**. The quality of the placement will tend to decrease as n is made larger than one (the smaller the value of n , the better the result). However, it is our experience that IPLACE will outperform other placement algorithms even with n set in such a manner that the run time of IPLACE matches the run time of the other (faster) algorithm. For chip-planning applications, a value of n in the range of five to ten is recommended.

The keyword **slow** is an optional entry causing IPLACE to be executed about n times longer than the default. The value of n is specified by the integer following the keyword **slow**. In some cases, you may get a slightly better result. However, only use the slow option if CPU time is of no interest to you.

ICSC***slow**:

The optional **graphics_update** keyword allows the user to control whether IPLACE draws the placement after each execution of the outer loop of the simulated annealing algorithm. The default is to draw the placement after each iteration.

ICSC***graphics_update**:

The optional **mode** keyword allows the user to force the placement algorithm. If unspecified, ITOOLS will automatically determine the placement algorithm which will give the best results. The flat mode gives the best results for small circuits, that is, for circuits with less than 1000 placeable objects. Otherwise, ITOOLS will chose the hierarchical mode which gives faster and better solutions than flat mode for large circuits. Normally, this should be left unspecified. One notable exception is ECO execution where the mode must be set to flat.

ICSC***mode**:

◇ hierarchical ◇ flat

When the hierarchical mode is used, placement is performed in multiple stages. The result after the first two stages is a very good predictor of the final result. We call the execution of the first two stages a *probe*. ITOOLS, by default, will perform a single *probe*. If multiple probes are specified, the first two stages are executed multiple times. After all probes are completed, the best candidate will be run to completion. The probes are independent and may be run in parallel. If a *designName.host* file exists, the probes will automatically be executed in parallel on a network of workstations; otherwise, the probes will be executed sequentially. Any positive number of probes may be performed.

ICSC***probes**:

The optional keyword **time_update** controls the frequency in which longest path algorithm is performed during the course of the annealing algorithm. The integer following **time_update** specifies the number of iterations to wait before executing the longest path algorithm. The default is every iteration. For designs with many pin pairs, the longest path search may dominate the run time. This option trades off accuracy for CPU time. Note: this setting has no effect for

fully specified timing path constraints; only pin pairs constraints are affected.

ICSC*time_update:

The keyword **prune_amount** is an optional parameter which allows the user to control the number of active paths considered during timing driven placement. The integer following the prune amount specifies the number of path segments considered during placement. The default value is recommended.

ICSC*prune_amount:

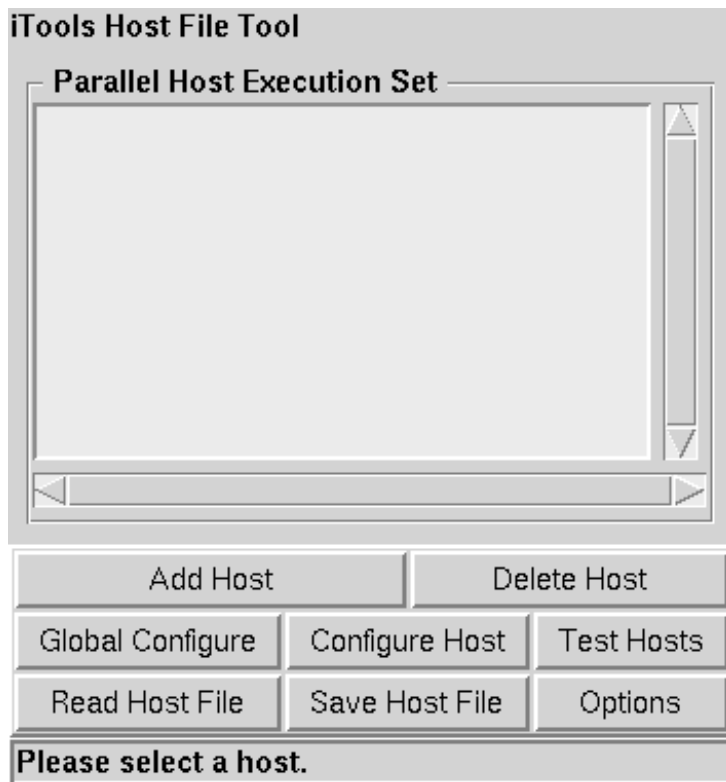
Miscellaneous Options

The optional **graphics_wait** keyword allows the user to control whether IPLACE stops at the end of the placement program to enter a graphics wait loop. This loop allows user interaction.

ICSC*graphics_wait: ☐ off ☐ on

Parallel Placement

The **parallel** parameter enables the parallel processing mode. The parallel processing mode recruits network processors to speed the execution of the simulated annealing algorithm. When this option is selected, a *designName.host* file must be present in the design directory to describe the available processors on the network. The tool below will help you create this file. First, depress the **Add Host** button to search the network for available hosts.



The parallel option has four modes: **off**, **probes**, **multi**, and **maximum**. The default is to run ITOOLS in a single

process. The **probes** option uses multiple processors to execute a set of probes in parallel. However, in this mode, each ITOOLS stage is executed sequentially. In the **multi** mode, only a single probe is executed but each ITOOLS stage is executed in parallel. The **maximum** mode executes multiple probes and runs each of the stages in parallel. For small designs (< 1000 cells), the **probes** option will give the best results. For larger designs, the **multi** and **maximum** modes are recommended. The **multi** mode is the fastest execution mode whereas the **maximum** will in general yield better results.

ICSC*parallel:

☐ off
 ☐ probes
 ☐ multi
 ☐ maximum

The processor limit controls the maximum number of processors available to the ITOOLS parallel algorithm. Normally, all processors listed in the *designName.host* file are used during the parallel placement algorithm. This option allows the user to limit the number of processors to the amount specified. You may limit processors for each stage of the hierarchical algorithm. The default is to use all of the hosts specified in the *designName.host* file.

ICSC*processor_limit:

[Go on to next program:IGROUTER](#)

ICRT – Iroute parameters (Master Routing Control)

Below are the parameters specific to the iroute program. It is also possible to edit the [general parameters](#).

ICRT*check_ports:	<input type="radio"/> off <input type="radio"/> on
ICSC*design_style:	<input type="radio"/> gate_array_with_sites <input type="radio"/> gate_array <input type="radio"/> stdcell
ICRT*graphics:	<input type="radio"/> off <input type="radio"/> on
ICRT*graphics_wait:	<input type="radio"/> off <input type="radio"/> on
ICRT*overlap_mode:	<input type="radio"/> allow_space <input type="radio"/> compact <input type="radio"/> left_justify <input type="radio"/> off
ICRT*read_pn2:	<input type="radio"/> off <input type="radio"/> on
ICRT*router_node:	<input type="text"/>
ICRT*padspacing:	<input type="radio"/> abut <input type="radio"/> exact <input type="radio"/> uniform <input type="radio"/> variable

The optional keyword **minimum_pad_space** allows the user to specify a minimum space between the I/O pads.

By default, members of pad the groups are placed contiguously. In other words, no nonmember pads can be placed between the member pads. To change the setting to noncontiguous, the parameter value **off** must follow the keyword **contiguous_pad_groups**. In this case, nonmember pads could be placed between the pads.

The graphics system has three control keywords: **graphics** which allows control of the X11 graphics display, **graphics_wait** which tells IPLANMC to wait for the user to enter commands after each step in the process, and **graphics_update** which can turn off the graphics update until the end of the simulated annealing run. To continue execution from a graphics wait loop, the user clicks on the FILE menu and selects CONTINUE PGM. See the section on graphics for more details concerning the graphics capabilities and commands.

The optional keyword **padspacing** controls the pad spacing mode. There are four modes of operation: **uniform** pad spacing, **abutting** pad spacing, **variable** pad spacing, and **exact** pad spacing as shown in Figure 3.4. Uniform pad spacing, spaces the pads evenly on each of the sides. In the abut mode, pads are forced to touch one another. The variable pad spacing mode places each pad such that the wire length is minimized. The last mode turns off the pad spacing algorithm and the pads remain in the place specified by the user in the *designName.con* file. In the first three cases, the side and sidespace constraints are observed. The default mode is uniform padspacing.

[Go on to next program: idetailer](#)

Creating an Itranslate Command Script

First select the path of the script to be generated. If you leave the form blank, the default name will be *itoolscmds.tcl*

Select the Translate Commands Script:

Now generate the script:

Generate Translation Commands Script

You can look at the generated *itoolscmds.tcl* file or your custom script using the editor of your choice using the control

Translation Cmds File

☒ tk
 ☐ vi
 ☐ emacs

Mode: ☒ Read-Only

above.

Editing the Parameter File

Design Rules and Technology Information

The menus below allow you to view and modify the **RULES** section of the parameter file.

The first menu is for the layers you want **itools** to use for routing. New layers must be given a layer name. Resistance is specified in ohms per square, capacitance in Farads per square micron.

Layers

Name:

Color:

Resistance (kohm/sq):

Capacitance (pF/sq. micron):

Direction: ☐ HORIZONTAL ☐ VERTICAL ☐ ORTHOGONAL

Type: ☐ ROUTING ☐ BASE ☐ CUT
☐ METAL ☐ POLY
☐ NDIFF ☐ PDIFF ☐ NWELL ☐ PWELL

Width (micron):

Spacing (micron):

Apply Edit
 New Layer
 Delete Layer

This menu allows you to modify via information.

Vias

↓

Name:

Color:

Layer 1:

↓

Layer 2:

↓

Width (micron):

Spacing (micron):

Geometry:

Apply Edit

New Via

Delete Via

↓

LLX

LLY

URX

URY

Apply Edit

New Layer

Delete Layer

It is also possible to [output the rules of the parameter file as commands](#) to be supplied to the translator in order to regenerate the parameter file. This is useful to learn the commands and capabilities of the translator. Each command in the parameter file has an equivalent command in the translator.

Program Parameters

You can now modify the program parameters that control individual components of the **itools** system. The parameter file controls execution of the programs given in the table below. You may edit the program parameters out of sequence using the links in the table below or in sequence using the button.

In parameter selection menus the default choice is shown in *italics*. The current selection is shown in **bold**. Information about each parameter is displayed when that parameter name is clicked. Parameters are changed by clicking on the selection or entering data in the form.

[Start Editing of Program Parameters](#)

Program	Function
*	General parameters for all programs.
GENR	Genrows : row-based configuration.
CLUS	Cluster : Hierarchical clustering tool.
SIMP	Simplify : Net list simplifier.
ICFP	Ifp / Igp : Floorplanning tools.
ICSC	Iplace : Row-based placement program.
ICGA	Iplacega : Strict gate array placement tool.
ICGR	grouter / igrouter : Global routers
ICRT	Iroute : Detailed-routing control program.
ICDR	Idetailer : Detailed-router.

Updating the Parameter File

Now that you have made changes to the parameter file, you may update the parameter file. There are two options: **Write New Parameter File** and **Overwrite Parameter File With Backup**. The first option creates a *designName.par.new* file. The user must rename the file to *designName.par* for the placement and routing system to utilize it. The second option, **Overwrite Parameter File With Backup** first copies the contents of *designName.par* to *designName.par:#* where # is the version number. The version number is determined by the presence of existing files with the same *base* name. This means all files of the form *designName.par:#* are considered in order. The new name of the file will be the last version number plus one. Version numbers begin with the first version or *designName.par:1*. The parameter file is recognized immediately by the system.

While we have tried to make it easy to edit the parameter file in EZ, you are free to edit the *designName.par* file with your favorite text editor.

[Write New Parameter File](#)

[Overwrite Parameter File With Backup](#)

Now we are ready to return to the top of the **Input/Translation** documentation and continue on with the design process.

[Continue](#)

Example of a Parameter File.

Below is an example par file. Note that wildcarding is permitted by preceding the parameter with an asterisk.

Sample xxx.par template file

```
# This is a default parameter file for the itools system.
# Please change the variables below to their appropriate values.
#

RULES
  MANUFACTURING_GRID 0.5
  LAYER metal GDS2 10 COLOR blue 0.05 1E-14 HORIZONTAL METAL
  LAYER poly GDS2 12 COLOR cyan 20.0 1E-14 VERTICAL POLY
  LAYER contact GDS2 14 COLOR gray 0.0 0.0 ORTHOGONAL CUT
  VIA contact metal poly
  GEOMETRY RECT metal -2.0 -2.0 2.0 2.0
    RECT contact -1.0 -1.0 1.0 1.0
    RECT poly -2.00 -2.0 2.0 2.0
  WIDTH metal 5.0
  SPACING metal metal 5.0
  WIDTH poly 5.0
  offset poly 0.
  SPACING poly poly 5.0
  SPACING contact contact 5.0
  WIDTH contact 5.0
  SCALE RESISTANCE 1000
  SCALE CAPACITANCE 1E-12
  SCALE DELAY 1E-9

ENDRULES

# General parameters controlling the itools system.

*vertical_wire_weight :10
*vertical_path_weight :10
*rowSep :10
*padspacing :abut

# Parameters controlling ifp.

#ICFP*slow :2

# Parameters controlling iplace.

ICSC*fast :1
ICGR*maze_route :off

#Parameters controlling genrows configuration program found in ifp.

GENR*feed_percentage :30.0
GENR*row_to_tile_spacing :1
```

Program Parameters

The second part of the parameter file is devoted to the program control parameters. The format for this file is similar to the .Xdefaults format:

*programName*parameter: parameterValue*

where *programName* may be one of the following:

- **GENR**– genrows: Row–based configuration program now in floorplanner.
- **CLUS**– cluster: Hierarchical clustering tool.
- **SIMP**– simplify: Netlist simplification program.
- **ICDR**– iroute: Detailed–router (future tool).
- **ICGA**– iplacega: Strict gate–array placement tool.
- **ICGR**– grouter/igrouter: Row–based global routing program.
- **ICFP**– ifp/igp: Floorplanner/macro cell placement program.
- **ICRT**– iroute: Detailed–routing control program.
- **ICSC**– iplace: Row–based placement program.

If no parameter file exists for a design, the **itools** system automatically copies a default template for the parameter file from the **./itools/defaults** directory into the current design directory. The file copied will be **xxx.par** when row–based cells are present, **xxx.macro.par** when only macros are present in the design, or **xxx.mixed.par** for designs with both row–based cells and macro cells. Next, the user will be asked to edit the file using vi, or the editor specified in the **EDITOR** cshell environment variable. The user should edit the default values to their appropriate values. Any parameters which are common to all designs should be entered as defaults in the parameter template files. Note that wildcarding is permitted by preceding the parameter with an asterisk. [An example of the xxx.par template for the parameter file.](#) *It is recommended that you use EZ CAD to generate the parameter file rather than using the default mechanism*



Use the button to edit each of the program parameters in turn.

Row constraints

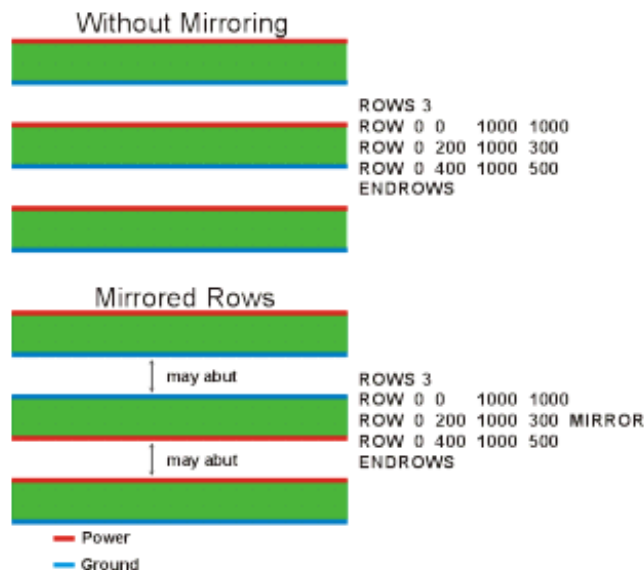
Row constraints are normally created automatically by the floorplanner programs: **ifp** or **igp**. However, the user may supply the row information directly. These constraints **MUST** appear at the top of the *circuitName.con* file before any other constraints. The format of the constraints is as follows:

```
ROWS INTEGER
ROW float float float float
  MIRROR[| ALIGN CENTER | ALIGN BOTTOM | ALIGN TOP | ALIGN SUPPLY |
  CAPACITY float | CAPACITY ABOVE CENTER float | CAPACITY BELOW CENTER float |
  CAPACITY CHANNEL ABOVE ROW float | CAPACITY CHANNEL BELOW ROW float |
  EXCEPT float float | CLASS integer ]
ROW float float float float VERTICAL
ROW float float float float...
ENDROWS
```

Row constraints begin with the mandatory keyword **ROWS** followed by the number of rows to follow. Rows may occur in any order.

A row is defined by the **ROW** keyword followed by the lower left x, lower left y, upper right x, and upper right y coordinates of the row respectively. By default, the row is assumed to be oriented horizontally, that is, standard cells are to be placed and abutted horizontally in the defined row. If the optional keyword **VERTICAL** appears, the row is declared a vertical and standard cells are abutted from the bottom to the top of the row. It is recommended that the rows be defined horizontally if at all possible since all programs understand horizontal rows but only a subset understand vertical rows at this time. There a number of options which may further describe a row:

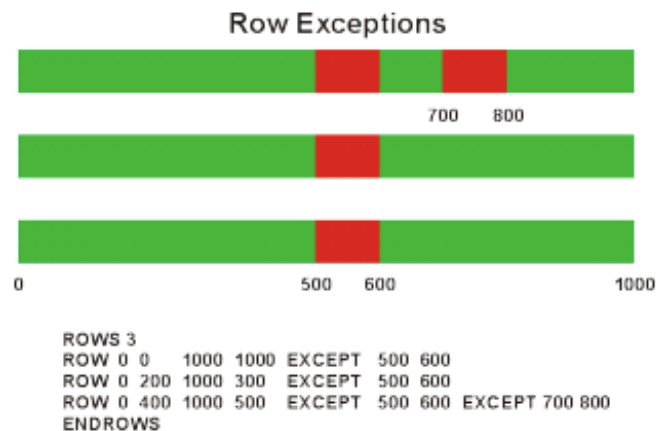
A row may be mirrored in order to share power and ground routing between adjacent rows. In order to achieve row mirroring, every other row should contain the keyword **MIRROR** after the row definition. The figures below show the mirroring of horizontal rows.



The cells in a row may be aligned in various ways. By default all of the cells in a row are aligned by their center position. Other alignments are bottom alignment, top alignment, and alignment by power and ground pins as shown in the figure below:



A row exception is a portion of a row which is not available for placement. One could block the portion of the row using rigidly placed spacer cells ; this option conveniently achieves the same goal. There may be any number of exceptions in a row. However, increasing the number of exceptions increases the difficulty of the placement problem. The figure below shows an example of exceptions in a row topology.



The row definitions are concluded with the keyword **ENDROWS**. The row definitions *must* be the first set of constraints in the constraint files as many other constraints are based on the row topology information.

Spacing constraints

Row spacing constraints are handled as a design rule, specifically as spacing constraints. These are added into the rule section as described below

[SPACING ROW ROW <value>](#)

[SPACING ROW ROW -1 RANGE <lo range> <hi range>](#)

The minimum row separation is defined using a row-to-row spacing constraint. The keyword **SPACING** is followed by the keywords **ROW ROW** and a number which defines the minimum spacing between rows. The spacing is measured from the bottom boundary of the top row to the top boundary of the bottom row. The spacing may be 0.0 to denote that the rows may abut.

Illegal spacings may be specified by using a **RANGE** constraint. A spacing range constraint is defined using the keyword sequence **SPACING ROW ROW** followed by the spacing value of -1 illustrating that the spacing is illegal. The illegal range is defined by the **RANGE** keyword followed by two number defining the illegal interval.

The range constraint is normally used in conjunction with the minimum row spacing constraint to define the row spacing. In many standard cell libraries, well to well spacing design rules require the wells to either overlap (in a mirrored arrangement of rows) or are required to be spaced one track apart. For example, let's use UW Cadlab's 0.18 um standard cell library. It is required that the rows either abut to share wells or they must be placed 0.72 um apart. To

do this we would add:

```
SPACING ROW ROW 0.0
```

```
SPACING ROW ROW -1 RANGE 0.0 0.72
```

to the rules. As a convenience, this can be added during translation using the *itranslate* commands

```
icrule add SPACING ROW ROW <minimum_value>
```

```
icrule add SPACING ROW ROW -1 RANGE <lo_value> <hi_value>
```

Our example would look like:

```
icrule add SPACING ROW ROW 0.0
```

```
icrule add SPACING ROW ROW -1 RANGE 0.0 0.72
```

Rules Section

The format for the design rule parameters are as follows:

RULES

SCALE	<i>float</i>
MANUFACTURING_GRID	<i>float</i>
LAYER	<i>layername</i> [GDS <i>gdslayer</i>] [COLOR <i>colorname</i>] [STIPPLE <i>bitmap</i>] <i>resistance</i> <i>capacitance</i> { VERTICAL HORIZONTAL ORTHOGONAL MINIMAL NONE } { <i>layertype</i> }
VIA	<i>vianame layername layername</i> [DEFAULT] [TOPOFSTACKONLY] [<i>geometry</i>] [RESISTANCE { <i>float</i> }]
WIDTH	{ <i>layername</i> <i>vianame</i> } <i>float</i>
SPACING	{ <i>layername</i> <i>vianame</i> } { <i>layername</i> <i>vianame</i> } <i>float</i>
PITCH	{ <i>layername</i> <i>vianame</i> } <i>float</i>
OFFSET	{ <i>layername</i> <i>vianame</i> } <i>float</i>
OVERHANG	{ <i>layername</i> <i>vianame</i> }{ <i>layername</i> <i>vianame</i> } <i>float</i>

•
•
•

SCALE RESISTANCE	<i>float</i>
SCALE CAPACITANCE	<i>float</i>
SCALE DELAY	<i>float</i>
SCALE PRECISION	<i>float</i>
TIME	MIN NOM MAX

ENDRULES

The mandatory keywords **RULES** and **ENDRULES** delimit the set of design rules.

Scale

The optional keyword **SCALE** allows the user to override the autoscaling feature of **itools**. For the autoscaling to be properly overridden it is important that the scale rule appear as the first line in the RULES section of the parameter file. Scale controls the internal representation of numbers in the ITOOLS system, namely, the number of significant digits to the right of the decimal point. It is the number which multiplied by a fixed point number converts that number into an integer. For example, **SCALE 100** converts 8.0843 to ITOOLS' internal integer representation of 808. When this number is output, it will appear as 8.08. It is strongly recommended that scale rule be omitted from the RULE section and allow ITOOLS to determine the internal scale.

Manufacturing Grid

Related to the *scale* rule is the **MANUFACTURING_GRID** rule for it is the inverse of the scale rule and is generally more convenient. Often the user knows the manufacturing grid of a process; it is the precision of the geometries which will be output to the mask. For example, if the user specified

```
MANUFACTURING_GRID 0.005
```

the manufacturing grid was 0.005 (corresponding to a scale of $1/0.005 = 200$), and all geometries will be a multiple of 0.005. Unlike the scale rule, if the user knows the manufacturing grid of the process, it is best to specify it.

Layer

The keyword **LAYER** defines a layer whose name is given by *layername*. The order of the layers in this file corresponds to the layer number which is found in the *designName.lib* file. The first layer appearing in this file becomes layer 1; the second becomes layer 2, and so forth. In addition, the user may optionally furnish a gds layer name which consists of a layer id and a data type separated by a colon. If the colon and the data type are omitted, it is assumed that the data type is zero. For example, the lines

```
LAYER metall GDS 10:0 1.00e+00 1.00e+00 VERTICAL METAL
LAYER metal GDS 10 1.00e+00 1.00e+00 VERTICAL METAL
```

both describe a layer named metall whose gds2 layer is 41 and data type is 0. This information can be used by the *itranslate* program to output the final routing as GDS2.

Layer Color

Optionally, the user may specify a drawing color for the layer. The color may be any of the single word colors found in the X11 [rgb.txt](#) file (normally found in the /usr/lib/X11 directory) or specified by its RGB values directly using the following notations:

COLOR #rrggbb


























where the rr, gg, bb, are the hexadecimal representation of the red, green, and blue intensities respectively. For example, each of the following layer definitions specify the color *yellow*:

```
LAYER boun COLOR yellow 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
LAYER boun COLOR #FFFF00 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
LAYER boun COLOR #ffff00 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
```

Layer Stipple

Additionally, the user may specify a stipple pattern for the layer. The stipple pattern is a bitmap normally 16 bits wide by 16bits high. There are a set of bitmaps which are supplied by itools and may be found in the *tcl/stipple* subdirectory under the itools root directory. The user may supply their own stipple pattern by placing the bitmap in the *~/itools/stipple* directory or supplying the full pathname of the bitmap. If the bitmap exists in either the itools stipple directory or the user's home stipple directory, the shortened form of the bitmap may be used as an argument to the **STIPPLE** keyword. It is mandatory that this shortened file name end in the *.bmp* suffix. Otherwise, the full pathname of the bitmap must be preceded by the @ sign so the bitmap is a valid Tk bitmap string. The user may also use any of the builtin Tk bitmaps which are currently *gray12*, *gray25*, *gray50*, and *gray75*. A solid or no stipple pattern may be offered using the keyword **none**. Below are the available stipple patterns in iTools. The first row contains the Tk default definitions and the remainder of the table enumerates the predefined iTools stipple patterns found in the

tc/stipple directory under the itools root directory. The default stipple pattern is *gray50*.

 none	 gray75	 gray75	 gray50
 gray25	 gray12	 icbitmap01.bmp	 icbitmap02.bmp
 icbitmap03.bmp	 icbitmap04.bmp	 icbitmap05.bmp	 icbitmap06.bmp
 icbitmap07.bmp	 icbitmap08.bmp	 icbitmap09.bmp	 icbitmap10.bmp
 icbitmap11.bmp	 icbitmap12.bmp	 icbitmap13.bmp	 icbitmap14.bmp
 icbitmap15.bmp	 icbitmap16.bmp	 icbitmap17.bmp	 icbitmap18.bmp
 icbitmap19.bmp	 icbitmap20.bmp	 icbitmap21.bmp	 icbitmap22.bmp
 icbitmap23.bmp	 icbitmap24.bmp	 icbitmap25.bmp	 icbitmap26.bmp
 icbitmap27.bmp	 icbitmap28.bmp		

Below are several examples of valid stipple definitions. Notice that the suffix *.bmp* may be dropped as in third and fourth examples below. The fourth example must reside in the file *~/itools/stipple/mybitmap.bmp*. In the fifth example, we show how to describe a fully-qualified bitmap which may reside anywhere.

```
LAYER boun STIPPLE gray75 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
LAYER boun STIPPLE icbitmap01.bmp 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
LAYER boun STIPPLE icbitmap12 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
LAYER boun STIPPLE mybitmap 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
LAYER boun STIPPLE @/mnt/mydrive/stipple.bmp 1.00e+00 1.00e+00 ORTHOGONAL BOUNDARY
```

Layer Resistance, Capacitance, and Direction

Each layer has associated with it, a parasitic resistance and capacitance given in ohms per square and farads per square micron respectively. In addition, the preferred routing direction to be associated with the layer must also be specified. The choices for layer direction are: **VERTICAL**, **HORIZONTAL**, **ORTHOGONAL**, **MINIMAL**, and **NONE**. If the **VERTICAL** direction is specified, routing geometries on this layer will be favored in y-direction and penalized in the x-direction. If **HORIZONTAL** attribute is given to this layer, routing geometries will be favored in the x-direction and penalized in the y-direction. An **ORTHOGONAL** layer allows routing in both directions; neither direction is penalized. A layer which has been specified as a **MINIMAL** layer is penalized in order to minimize wire length on this layer. Polysilicon layers are frequently specified as **MINIMAL** layers. If a layer is specified as **NONE**, no routing may be added by the detail router on this layer. The preferred direction is only used for routing estimation and costing. The actual routing for a given layer may occur in either direction and may be redefined in the detail router. At least one layer must be specified in both the horizontal and vertical directions.

Layer Type

Lastly, the layer type should be specified. The layer type is defined to be one of the following types:

```
{ METAL | POLY | NDIFF | PDIFF | NWELL | PWELL | CUT | ROUTING | BASE | BOUNDARY }
```

A layer may belong to one of the four general layer classes : **ROUTING**, **CUT**, **BASE**, or **BOUNDARY**. A **ROUTING** layer is used for interconnection of signals. A routing layer may be further differentiated as either **METAL** or **POLY**. This further classification may be of use to detailed-routers. A **CUT** layer is the pseudo layer used to connect the two layers of a via. The third category **BASE** is used to distinguish the layers which are not to be

manipulated by a detailed-router. These layers generally are the layers which are used to construct the transistors of the cell. The **BASE** class may be further differentiated as **NDIFF**, **PDIFF**, **NWELL**, or **PWELL**. While it is not necessary to use the detailed classes at this time, it may become useful for future products. The last category **BOUNDARY** is used to denote layers which define the boundary of the cell. If no layer type is given, the layer is assumed to be a **ROUTING** layer

Any number of layers may defined. Layer definitions must precede all other rules (except scaling/manufacturing grid rules).

The screenshot shows a 'Layers' dialog box with the following elements:

- Title Bar:** 'Layers' with a close button.
- Input Fields:**
 - Name:** A text input field.
 - Color:** A color selection field.
 - Resistance (kohm/sq):** A text input field.
 - Capacitance (pF/sq. micron):** A text input field.
 - Width (micron):** A text input field.
 - Spacing (micron):** A text input field.
- Direction:** A dropdown menu with options: ☐ HORIZONTAL, ☐ VERTICAL, ☐ ORTHOGONAL.
- Type:** A dropdown menu with options: ☐ ROUTING, ☐ BASE, ☐ CUT, ☐ METAL, ☐ POLY, ☐ NDIFF, ☐ PDIFF, ☐ NWELL, ☐ PWELL.
- Buttons:**
 - Apply Edit**
 - New Layer**
 - Delete Layer**

The connections between layers are defined by using the keyword **VIA**, followed by the name given to the via cell and the two layers which are to be connected. The via name can then be used in subsequent width and spacing rules.

The optional keyword **DEFAULT** tells the routers that this via is the preferred via for connections between the two defined layers. If a **DEFAULT** via is not defined the detail router will warn the user and attempt to use all of the vias defined for the pair of via layers. A non-default via may be supplied for special occasions such as a double width via used to lower via resistance and improve reliability or as a **TOPOFSTACKONLY** via.

The optional **TOPOFSTACKONLY** keyword marks the via as appropriate for solving minimum area rules during routing. A **TOPOFSTACKONLY** via meets all of the minimum area requirements and may become a stacked via. In general, **TOPOFSTACKONLY** vias take up more area than default vias. **TOPOFSTACKONLY** vias are only useful when minimum area rules are supplied and do not need to be supplied when minimum area rules are absent.

Next, the geometry of the via may be given. However, the via geometry is mandatory when invoking either a global router or detail router in the flow. The geometry of the via is defined as a set of rectangles preceded by the **GEOMETRY** keyword. A rectangle is defined by the keyword **RECT** followed by the routing layer of the geometry followed by floating point numbers representing the lower left and upper right coordinates of the geometry, namely $x1\ y1\ x2\ y2$. Normally, three rectangles are supplied – the bottom layer of the via, the via cut layer, and the upper layer of the via. Each geometry on a routing layer should satisfy the minimum width for that layer. The geometry of the cut layer should be symmetric around zero; otherwise the router may trouble satisfying same-net rules. The cut geometry does *NOT* need to be square but may be rectangular. For example, we define a default via named *via1* between the first two layers of metal by

```
VIA via1 METAL1 METAL2 DEFAULT
```

```
GEOMETRY
```

```
RECT metal1 -0.190 -0.140 0.190 0.140
```

```
RECT via12 -0.130 -0.130 0.130 0.130
```

```
RECT metal2 -0.225 -0.225 0.225 0.225
```

```
RESISTANCE 1.4
```

The via resistance is the final option of a via definition. It is specified by the keyword **RESISTANCE** followed by the resistance of the via in ohms.

Currently, itools does *not* support multiple layer vias, or vias, that jump more than one layer. For example, we could define a multiple layer via named *via13* which connects layers *metal1* and *metal3*. However, the syntax does support them and the router will support them in the future.

The image shows two overlapping dialog boxes from the ITOOLS software. The top dialog is titled 'Vias' and contains fields for 'Name', 'Color', 'Layer 1', 'Layer 2', 'Width (micron)', and 'Spacing (micron)'. It has buttons for 'Apply Edit', 'New Via', and 'Delete Via'. The bottom dialog is titled 'Geometry' and contains a 'Layer' dropdown, four coordinate fields (LLX, LLY, URX, URY), and a large empty rectangular area. It has buttons for 'Apply Edit', 'New Layer', and 'Delete Layer'.

The keyword **width**, allows the definition of a given layer's minimum routing width or via width in microns. Similarly, the **spacing** keyword, begins the definition of the minimum distance between any defined layers or vias.

The keyword **pitch** defines the routing grid for a layer. It is the minimum separation between two adjacent tracks using a gridded detailed-router. If the **pitch** is omitted in the *design.par* file, its value becomes the sum of the **width** and the **spacing** for that routing layer.

The keyword **offset** specifies a distance between the cell and the first track in the channel for a given layer. Its default is the **pitch** value of the routing layer.

The keyword **overhang** specifies that a layer or via must overlap another layer or via by the amount specified by the floating point number.

The **scale** parameters are used to control the timing-driven placement features of ITOOLS. The resistance values in the *designName.lib* file may be multiplied by the **scale resistance** value. Likewise, the capacitance values may be scaled using the **scale capacitance** keyword. In addition, the times specified for delays may be scaled using the **scale delay** keyword. The resistance, capacitance, and delay scale parameters default to unity. The **scale precision** keyword

allows the user to adjust the internal time measurement system in ITOOLS. The default time scale is in tenths of nanoseconds. This should be adequate for most purposes. Warning: increasing the precision increases the chances of integer overflow during program execution.

For example, given that the *designName.par* file specified:

```
scale resistance 1000
scale capacitance 1E-12
scale delay 1E-9
scale precision 1E-9
```

and the *designName.lib* file specified:

```
PIN a I
PORT a_0 (15 58) LAYER 1 CAPACITANCE 0.1
PIN q O
PORT q_0 (-1 58) LAYER 1 RESISTANCE 3.0
DELAY a q NOT FALLTIME =3.8 RISETIME 2.9
```

The circuit would be converted to

```
PIN a I
PORT a_0 (15 58) LAYER 1 CAPACITANCE 1E-13
PIN q O
PORT q_0 (-1 58) LAYER 1 RESISTANCE 3000
DELAY a q NOT FALLTIME = 3.8E-9 RISETIME 2.9E-9
```

Internally, ITOOLS will measure all time delays in nanoseconds.

The **time** keyword selects the set of time parameters to be used during optimization. It may have one of the following values: **min**, **nom**, or **max**. All timing information may be entered as a triple set of numbers separated by colons or a single number. The numbers in the triple set represent the minimum, nominal, and maximum values respectively. If only one number is supplied, this number will be used for all time settings. For example, if the resistance of a port was specified as:

```
RESISTANCE (80:100:130)
```

and **time** was set to **min**, the value of the resistance would be 80 ohms. If the time keyword is not present in the parameter file, the nominal values will be used during optimization.

Simplify Parameters

Below are the parameters specific to the simplify program. It is also possible to edit the [general parameters](#).

SIMP*check_ports:	<input type="checkbox"/> off <input type="checkbox"/> on
SIMP*library:	<input type="text"/>
SIMP*net_ignore_limit:	<input type="text"/>
SIMP*rowSep:	<input type="text"/>
SIMP*scale:	<input type="text"/>

Function

The function of the **Simplify** program is to prepare the netlist for the placement module. One pin nets are removed as well as nets which have more than **net_ignore_limit**.

Parameters

ITools can handle designs with port data outside the cell boundary. However, if requested, ITools checks and records all data which resides outside the given cell boundary as an error. The option **check_ports** is read and utilized by the simplify program. However, **check_ports** is a general parameter and should be set at the general level. Use the hypertext button on the widget below to return to the general level.

SIMP*check_ports:	<input type="checkbox"/> off <input type="checkbox"/> on
-------------------	--

The optional **library** keyword allows the user to specify more than one library (*.lib*) file for the design. The string following the library keyword specifies the pathname of the library. The user may specify multiple libraries by listing each library separately. The option **library** is read and utilized by the simplify program. However, **library** is a general parameter and should be set at the general level. Use the hypertext button on the widget below to return to the general level.

SIMP*library:	<input type="text"/>
---------------	----------------------

The required keyword **rowSep** is followed by a floating point number representing the desired amount of separation between the rows. The amount of separation between the rows is this number times the average height of the rows. The **rowSep** parameter is read and utilized by the simplify program. However, **rowSep** is a general parameter and should be set at the general level. Use the hypertext button on the widget below to return to the general level.

SIMP*rowSep:	<input type="text"/>
--------------	----------------------

The optional keyword **net_ignore_limit** controls the pruning of nets with high number of pins from the placement netlist. By default, the simplify program will remove all nets which have more than 100 individual pins from the netlist. Our studies have shown that neglecting nets with large number of pins does not affect the quality of the result but leads to a great improvement in time of execution. Increasing this number will increase the run time. Reducing the

number below 100 may reduce the quality of the result.

SIMP*net_ignore_limit:

[Go on to next program:IFP](#)

Notes on Creating Verilog for Itools

Itools expects structural Verilog as an input for translation. While it knows the syntax of the Verilog language, it will only process module definitions, module references, and the wires that connect them. The itools translator does understand assignment statements which assign one signal to another. Assignment of a signal to a function is not valid as that does not constitute structural verilog.

Power Supplies

Power supplies can be defined in Verilog using the `supply1` and `supply0` directives. For example,

```
supply1 vdd ;
supply0 gnd ;
```

would define the signal `vdd` as the power supply and `gnd` as the ground for the module. These definitions are detected as global signals by the translator and allow the definition of implicit net connections.

Tie off of Signal Pins to Power Supplies The proper way to tie off unused gate signals to the power supply or ground is to use an assignment statement. In this example, we are tying the `D` input of the flip-flop to ground.

```
assign gnd = 1'b0 ;
.
.
.
DFFRX1 state0 ( .D(1'b0), .CK(clk), .RN(reset), .QN(out) );
```

This will be translated in iTools to

```
DFFRX1 state0 (gnd D) (clk CK) (reset RN) (out QN)
```

Multiple Power Supplies

If your design has multiple power supplies or multiple grounds, the easiest and best method is to explicitly specify the power network in the Verilog. For example:

```
supply1 vdd ;
supply1 vlo ;
supply0 gnd ;
.
.
.
DFFRX1 state0 ( .D(1'b0), .CK(clk), .RN(reset), .QN(out) .VDD(vlo), .VSS(gnd) );
```

In this example, the flip flop is connected to the `vlo` supply rather than the main supply `vdd`. The library pin `VSS` is tied to the common ground network. The program supports multiple power and ground networks. While this can be handled [implicitly](#), if specified explicitly, the tools have the ability to partition the design based on power supply networks.

Power Supply IO Pins

If you wish for the power and ground pins to have I/O pads to be generated, include them in the modules IO list. For example,

Itools Documentation

```
module my_design ( clk, add1, add2, out_1, vdd, gnd ) ;
    input clk;
    input [16:0] add1 ;
    input [16:0] add2 ;
    output [17:0] out_1 ;
    input vdd ;
    input gnd ;
    .
    .
    .
endmodule
```

In this case, **iTools** will automatically generate I/O pads at the boundary of the design. The *circuitName.ckt* file will contain the following instances:

```
INSTANCE vdd ICPORTINPUT
(vdd port)
INSTANCE gnd ICPORTINPUT
(gnd port)
```

Additional Constraints and Syntax Checking

A valid license is ***NOT*** required to complete this page. Syntax checking does not require a license.

Global or Implicit Signal Definitions

Now that we have translated the design, we need to define the *global* signals or the signals which implicitly connect to the cells in the design. We need to define them since they are not present in the netlist especially in standard cell circuits. Usually, these are the power and ground networks. In order to define these nets we need to make a mapping from the signal name to the implicit pin of the cell. This mapping is added to the *circuitName.con* file. For example, suppose we want the signal **vdd** to connect to all of the VDD pins found in library. This would entered as

```
NET CREATE vdd GLOBAL VDD
```

in the *circuitName.con* file. The **CREATE** keyword creates the net even if it doesn't exist in the netlist. Use the tool below to add the implicit pin definitions:

Global Net Definition Tool

```
NET CREATE vdd GLOBAL vdd!
NET CREATE gnd GLOBAL gnd!
```

New Global Net

Edit Global Net

Delete Global Net

When you are finished creating the constraints, you may add them to the constraint or *circuitName.con* file.

Update Constraint File

The constraints have been put into the constraint file for you.

Notes about multiple power supplies

Multiple power supplies can be handled in two ways: the netlist can have the power supply explicitly in the netlist, or you can specify the implicit global information on a [row or instance](#) basis.

Reserved Routing Layers

Frequently, a process technology has more layers defined than required for the current task. For example, when routing a *subchip* or partition of the entire design, it is useful to reserve routing layer[s] for power and ground and/or global interconnects. Use the tool below to add layers to the set of served layers. There is no need to reserve via

layers; routing layers are sufficient.

Reserved Layer Definition Tool

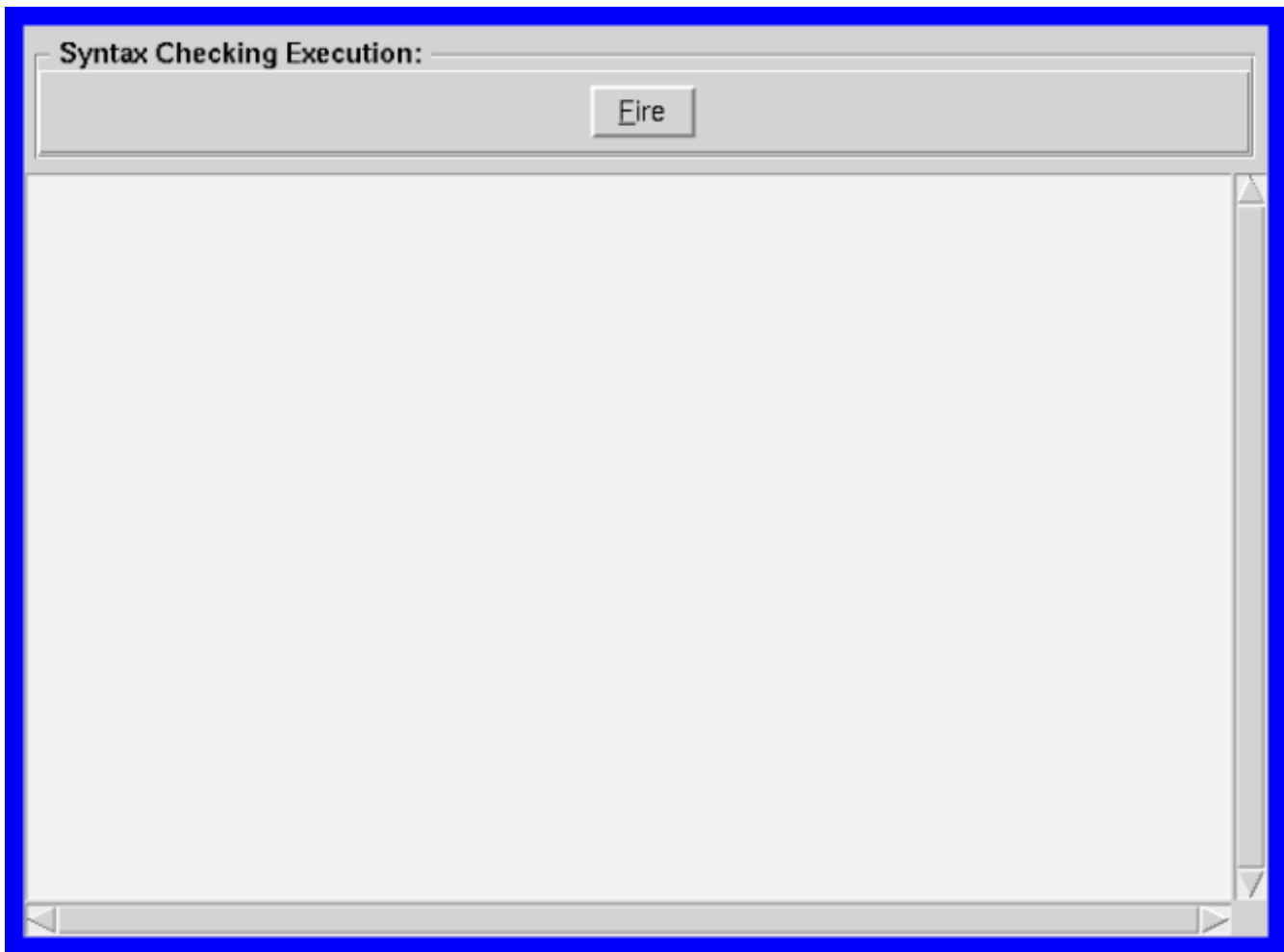
RESERVED_LAYERS METAL6 ENDRESERVED

When you are finished creating the constraints, you may add them to the constraint or *circuitName.con* file.

The constraints have been put into the constraint file for you.

Syntax Checking

The mandatory input files are present. We can perform syntax checking of the design. Syntax checking will report problems in the input data as well as recording vital statistics of the design. In addition, the syntax checker will determine the [design type](#): standard (row-based), macro (block), or mixed macro-standard cell design styles. Syntax checking of the design may be accomplished by pressing the button below.



Flow Selection



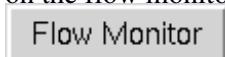
Now that we have chosen a flow, **we should update the parameter file based on the chosen flow**. For example, if we chose the *flow.onlyroute1* flow, the parameter file needs to be updated so the global router knows we are in the route_only mode. Use the button below to update the parameter. If you are curious, use the second button to

understand what settings need to change.



Flow Monitoring

Now that we have selected a flow, EZ can display and monitor our progress in that flow. Use the button below to turn on the flow monitor. You can also find the flow monitor in the **Tools** top level menu.





If you do not have a valid license, you will need to [obtain a license file](#) from InternetCAD, Inc. in order to continue.

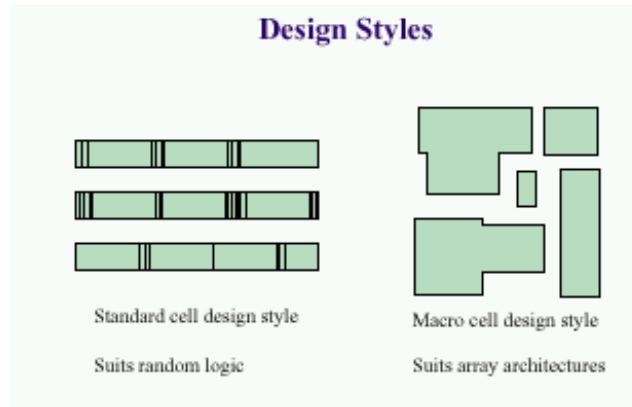
[Next Page:Placement](#)



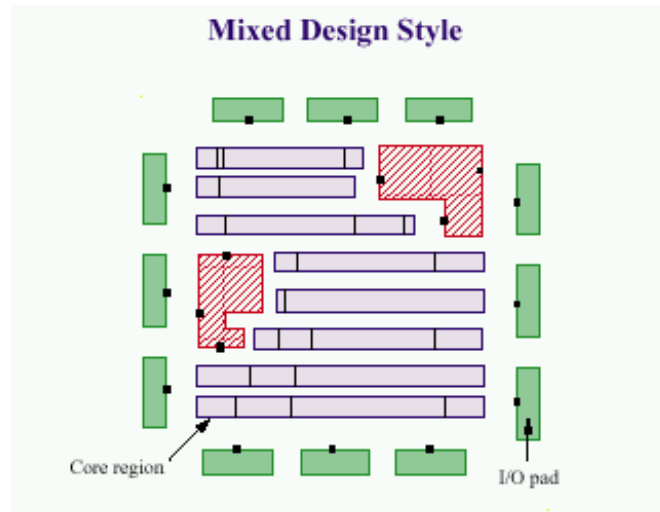
Design Style or Type

Copyright (c) 1999–2004. InternetCAD, Inc. All right reserved.

All designs fall into one of three design styles : standard (row-based), macro (block), or mixed macro–standard cell. The two major classes are row-based which are also known as standard cells and non-row based designs known as macro or block designs as shown below. We will use these terms interchangeably throughout EZ.



But it also possible to mix both design styles together to form the most general and most difficult case:



A flow consists of the necessary steps to process each of these design styles. The abstraction into three separate styles permits the efficient solution of each of the design types. Different tools are required for each style.

Available Flows

The table below lists the current flows in the EZ system. Each [flow](#) contains a complete set of programs to be executed for each of the possible design types: standard (row-based), macro (block), or mixed macro-standard cell design styles.

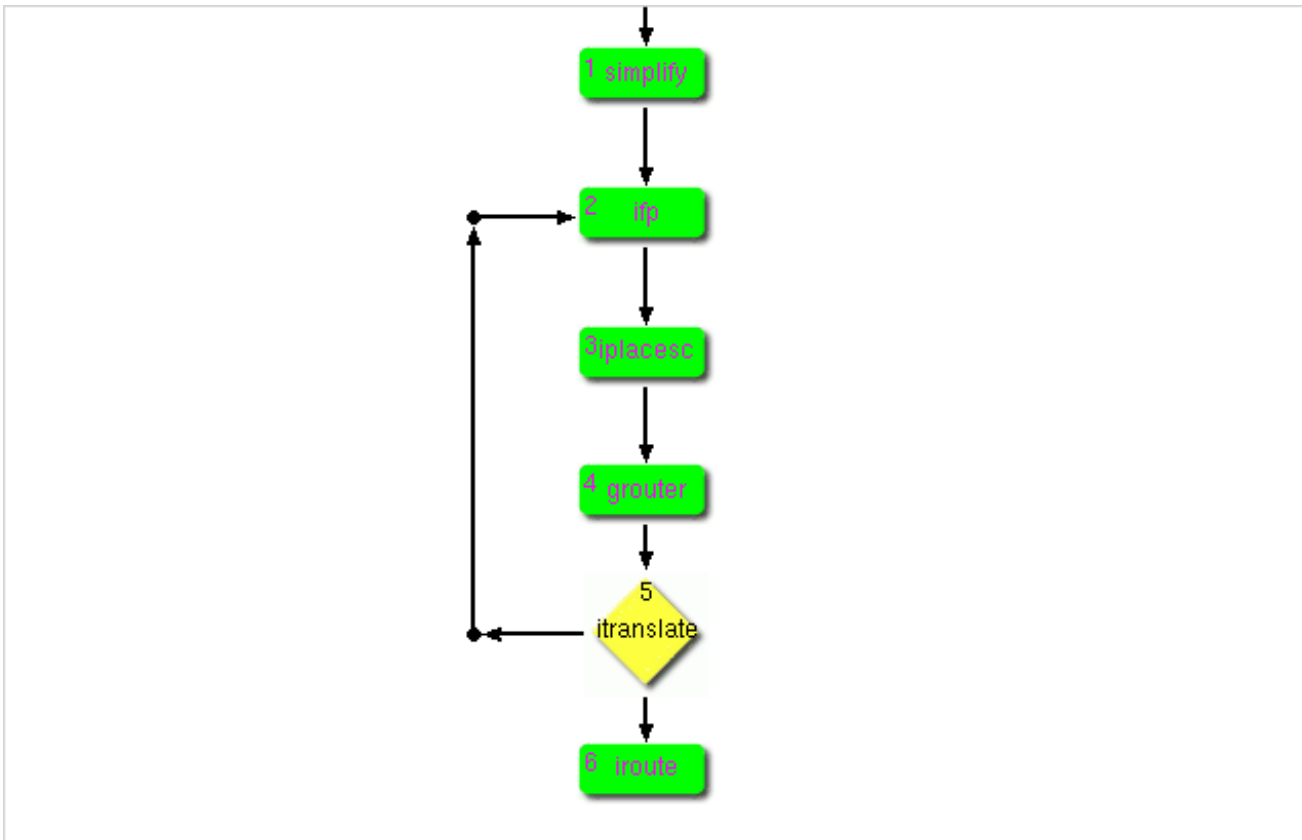
FLOW	FUNCTION
<i>flow.iplace</i>	Placement only flow.
<i>flow.iplace2</i>	Second generation placement only flow.
<i>flow.parallel</i>	Place, global and detail route in parallel mode.
<i>flow.route1</i>	Place, global and detail route using first generation global router. First generation only handle horizontal rows and is not fully automatic.
<i>flow.route2</i>	Second generation place, global, and detail route.
<i>flow.onlyroute1</i>	First generation global, and detail route only. Must supply a valid placement.
<i>flow.widthopt1</i>	First generation width optimization. Uses a decision object to iterate until target width is achieved. Currently, only strictly row based designs are supported.

Running width optimization

In order to run width optimization, you must supply a target width. To do this you use [ICSC*width_optimize:<targetWidth>](#) in the parameter file to active optimization. This target is subject to a default tolerance of 10%. The user may change the tolerance by using the [ICSC*max_numrows:<integer>](#) in the parameter file.

Program Flows

The sequence of the place and route programs is controlled by the master flow program called *iflow*. The program *iflow* creates a flow chart in the graphics window (as shown below) and automatically shows the current status of the layout process by highlighting the current node in the flow chart. A graph is used to describe the design flow. The programs or shell scripts in the flow are nodes in the graph (flowchart) and the edges between the nodes are valid program execution sequences. The program uses file dependencies in order to determine if a program or shell script needs to be executed, similar in function to the Unix *make*.



Input

In order to make the system flexible from the user's perspective, *icflow*'s sequence of programs is not predefined; instead, it is defined by an ASCII input file. A portion of the *iflow* input format is shown below. In order to make things easier, we have also supplied a [flow editor](#) called *flowER* which is able to create and edit flow files.

```

numobjects 6
.
.
.
pobject igrouter 4: 3
path :
drawn : 450 50 750 150
edge 3:
ifiles: $.lib $.ckt $.par $.con* itools.con* itools.lib* itools.ckt*
ofiles: route/$.gout
args : -w @WINDOWID $
args debug : -dv -w @WINDOWID $

```

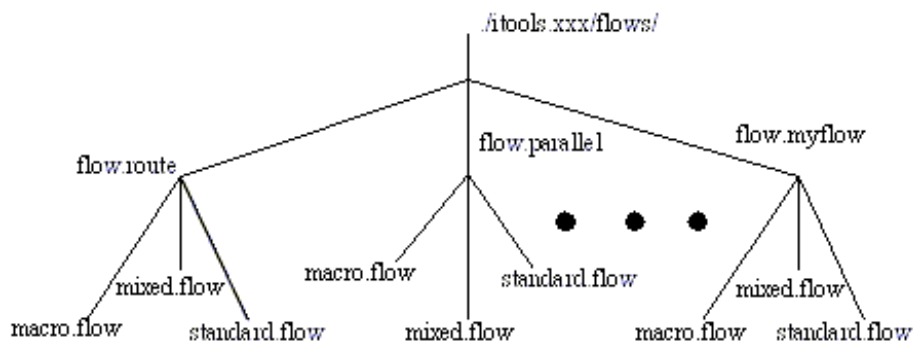
```
args nographics : -n $
drawn : 600 150 600 250
        600 150 620 170
        600 150 580 170
.
.
.
```

A portion of the icflow file

Each program object or *pobject* is numbered as shown above. The program **igrouter**, for example, is program number 4 and it may be executed in a valid sequence after program 3. The path of the program may be changed from its default place and the user may choose how he wishes to draw the object. Next, each edge is specified, that is, how it is to be executed and what files it depends on. The \$ is a special string substitution character which stands for the design name. For each edge we list the input and output files that are required and generated by the program. By looking at the time stamp of each of these files we can determine if this program needs to be executed. In addition, each edge allows for different argument lists to be passed to the executed program. The key word @WINDOWID tells icflow that the child process is capable of inheriting the graphics interface. Optional files are followed by an asterisk. For example, \$.con* is an optional input file to **igrouter** on edge 3. Two argument lists are possible: the default *args*, and the optional *nographics* list. The nographics list is called when **itools** is executed with the -n command line switch.

It is very easy to modify the sequence of programs in order to customize it to particular applications; for example, another flow file could be generated which allows the addition of front-end and back-end translation programs or perhaps another would omit the @WINDOWID key words for the batch mode processing.

Iflow recognizes two distinct flow file hierarchies: a directory exists for each high level design strategy, and a file exists for each design style (standard, mixed, and macro) within that strategy. Two examples of high level strategies are the basic placement and routing strategy, flow.route, and the parallel placement and routing strategy, flow.parallel. The figure above graphically depicts the flow file structure. The user may add their own high level strategies by creating a directory in the ./itools/flows directory and building the three design style files within that strategy (macro.flow, standard.flow, mixed.flow). The flow directory name must contain the prefix "flow.". The user may set the default flow with change_flows or specify the flow explicitly on the **itools** command line.



File structure

Output

The output of iflow is the *designName.stat* file which contains the design statistics used by the other programs for efficient allocations of resources.


Input


Flow Editor

A Tk-based flow editor has been supplied in order to make flow creation easy and error free. It is called flowER and can be executed in several ways. First, it may be called from the iflow program itself from the **Edit Flow File** command under the **Flow** menu. You may put the iflow program in a graphics event loop by entering :

- `iflow -p <designName>`
on the command line. Secondly, you may enter the following command to invoke the flow program directly:
- `ICTk -i $ICDIR/tcl/iflow/flowER.tcl`
on the command line where ICDIR is the itools root directory environment variable. Lastly, you may call flowER from the tutorial below.

Flow Tutorial

Here is a tutorial on using the flow editor: flowER. First, click the  button to open the Tk editor.

Let us start by first loading a new flow file to look at it and its parameters. We will  the maxwidth.flow file from the \$ICDIR/flows/subflows directory.

A valid license is ***REQUIRED*** from this point onward in the flow. The [license server](#) will start automatically but may be started manually.

Floorplanning

Do you want an embedded (inline window) or remote (external window) display?

This is the transcript window for the floorplanning process. To start the floorplanning process, hit the "Fire" button.

You may resize the window by using the first mouse button on any corner of the display window.

Do you have any I/O constraints? You should add or edit them now.

[=>](#)

Now we may optionally generate a Tcl script known as the *design.fdo* file. The *design.fdo* file controls the floorplanning algorithm. Normally, this doesn't need to be specified as the floorplanner has a default script.

☐ yes
 ☒ no

You may want to edit the parameters which control the execution of the

floorplanner.


[=>](#)

Floorplanner Execution:

Hint: you can resize the window below by using the first mouse button on any corner.

Placement



Clock tree synthesis

Does your design have clock signals which need to be synthesized as a distribution network? If so, you should define these nets now.  =>

Power and ground network

At this point we need to determine when we are going to instantiate the power and ground network. Normally, we defer this step to detailed routing but we can instantiate a cell-based construction of the power and ground networks at either the placement or global routing steps. It is often necessary to place cells which [strap the interior of the core](#). The user can create the layout of these cells using Tcl scripts. Use the tool below to choose the power and ground configuration from the list of possible choices.

Now we may optionally generate a Tcl script known as the *design.pdo* file. The *design.pdo* file controls the placement algorithm. Normally, this doesn't need to be specified as the placer has a default script. You will have to supply a script if you wish to instantiate a power and ground network at the placement stage.

Custom Placement Script  yes  no

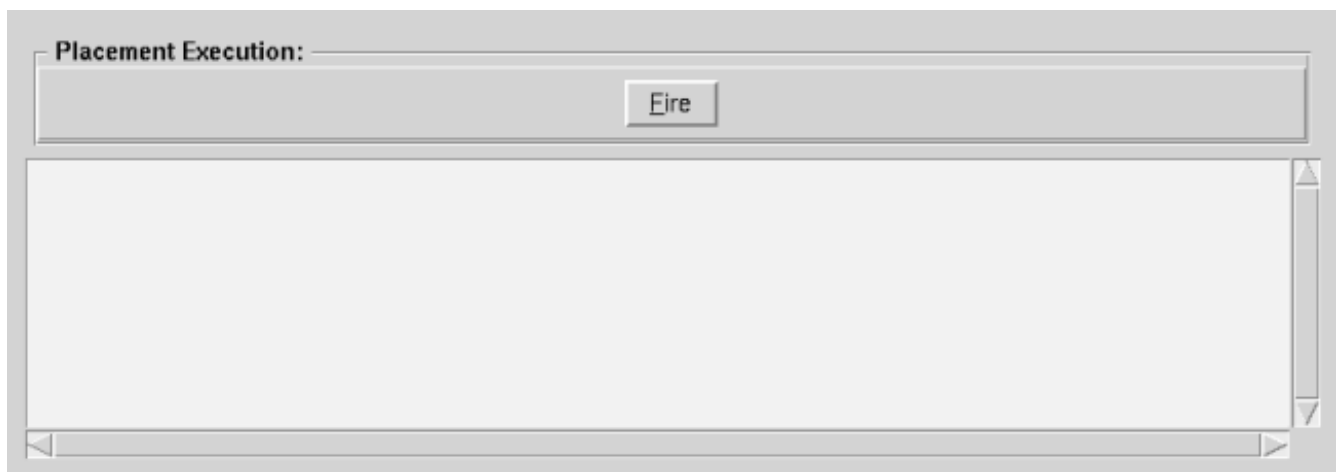
 =>

You may want to edit the parameters which control the execution of the placer.



This is the transcript window for the placement process. To start the placement process, hit the *Fire* button.

You may resize the window by using the first mouse button on any corner of the display window.



Hint: you can resize the window below by using the first mouse button on any corner.

[View Placement](#)

Use the button below to view the placement whenever the button is green.

[Placement Output Files](#)

[Next Page:Routing](#)

[=>](#)

License Server Features


The **itools** license server supports both floating, and fixed program licenses. In addition, several options are available to the system administrator to help limit the rights to program execution for a set of [hosts, users, groups, or operating systems](#). In addition, versions 1.3.4 and above support [redundant license servers](#).

The license server consists of two programs: the license daemon *iclicensed* and the license administration program *ladmin*. The license daemon will be started automatically during the execution of the **itools** program. The license daemon reads the license file, which resides in the license directory under the **itools** root directory or in `${ICDIR}/license/license` where ICDIR is an environment variable which contains the path of the **itools** root directory.

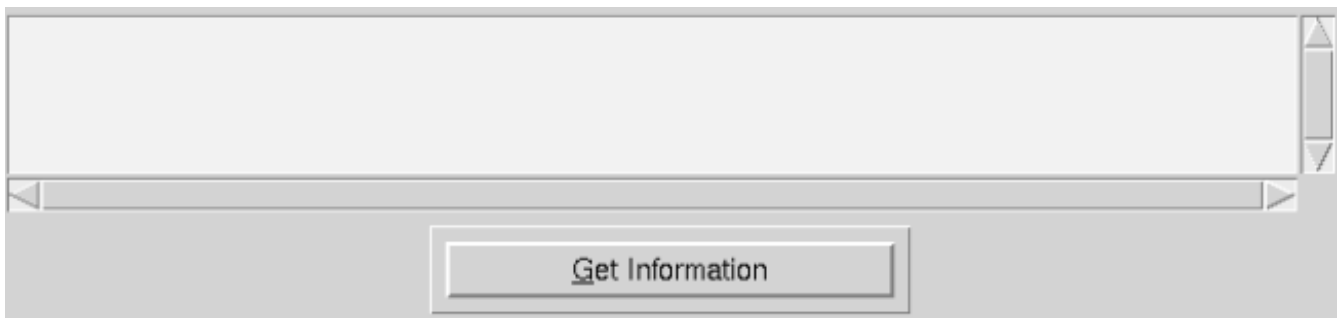
The system administrator who installs **itools** may need to [adjust the location of the itools root directory](#) in the license file.

1. Obtaining a License File

Now we need to gather the information needed to generate a license file. First, we must determine the license server host. The default is the current host. If other machines in the network have access to **itools**, you may use "Find Available Hosts" to find the names of hosts in the network. In order for a host to respond, it must have the portmap program running on that machine.



Next, we ask *ladmin* on the chosen machine to send us the license file information. The widget below executes *ladmin -qi*. If the license server host is not the local host, it uses a remote shell (rsh,remsh,etc.) to communicate with *ladmin* on the remote machine. It should be noted that the machine will not respond properly if that machine does not have *ladmin* installed on it.



For your convenience, this information obtained from *ladmin -qi* can be sent to us to generate your license file.

Send information to Internet CAD, Inc.	
Name:	<input type="text"/>
Affiliation:	<input type="text"/>
Email Address:	<input type="text"/>
<input type="button" value="Send"/>	
Status:	<input type="text"/>

2. Installing the License File

First, we need to supply the license file. All we need is to locate the mail message containing the license file or the license file itself. In order to send the license file over the Internet, the file was converted to an email attachment using the Unix commands

```
tar cvf - license | compress -c | uuencode tar.pw.Z > mail.pw
```

However, **itools** can recognize either the mail attachment or the license file itself. There is no need to decode the license file, EZ will do it for you below.

Select the License File
<input type="text"/> <input type="button" value="Browse ..."/>

Now install the license file. Normally, the license file resides in the `${ICDIR}/license/license` file where ICDIR is the itools root directory environment variable. However, the path of the license file can be altered with the ICPASSWD environment variable. The ICPASSWD environment variable should be set to the fully qualified path to the license file. Only advanced users should use this method of specifying the license file. The button below will install the license file in its normal residence.

Install the license file.
<input type="button" value="Install"/>

The license server needs to verify the time on your system. The easiest, fastest, and most reliable way is to connect to the [itools time standard via the Internet](#). However, this may not be possible in the default configure if you are behind a firewall. The license server knows how to [use a HTTP proxy](#) to traverse the firewall. If you are directly connected to the Internet, there is no need for the proxy. Note that this is the same proxy used by browsers. Consult your system administrator for the proxy machine and port. The optional proxy timeout given in seconds is the amount of time to wait before terminating a Internet connection.

Add HTTP Proxy.	
Proxy Host	<input type="text"/>
Proxy Port	<input type="text"/>
Proxy Timeout	<input type="text"/>
<input type="button" value="Add Proxy"/>	

If you plan to execute iTools on a host other than the license server host and you wish to start the license server remotely, you can allow iTools to automatically start the license server for you using a remote login program. The default remote login program is the remote shell or rsh (remsh on HP-UX). However with today's security concerns, iTools now supports ssh and custom login commands to remotely login to the license server host from any other host in the network. For example, if you wanted to use SSH-1 with RHOSTS authentication (not recommended), you would enter in the custom form:

```
ssh -1 -o UsePrivilegedPort=yes
```

Please contact your system administrator for the correct option. Note: this option is irrelevant if you always run your iTools programs on the same host as the license server.

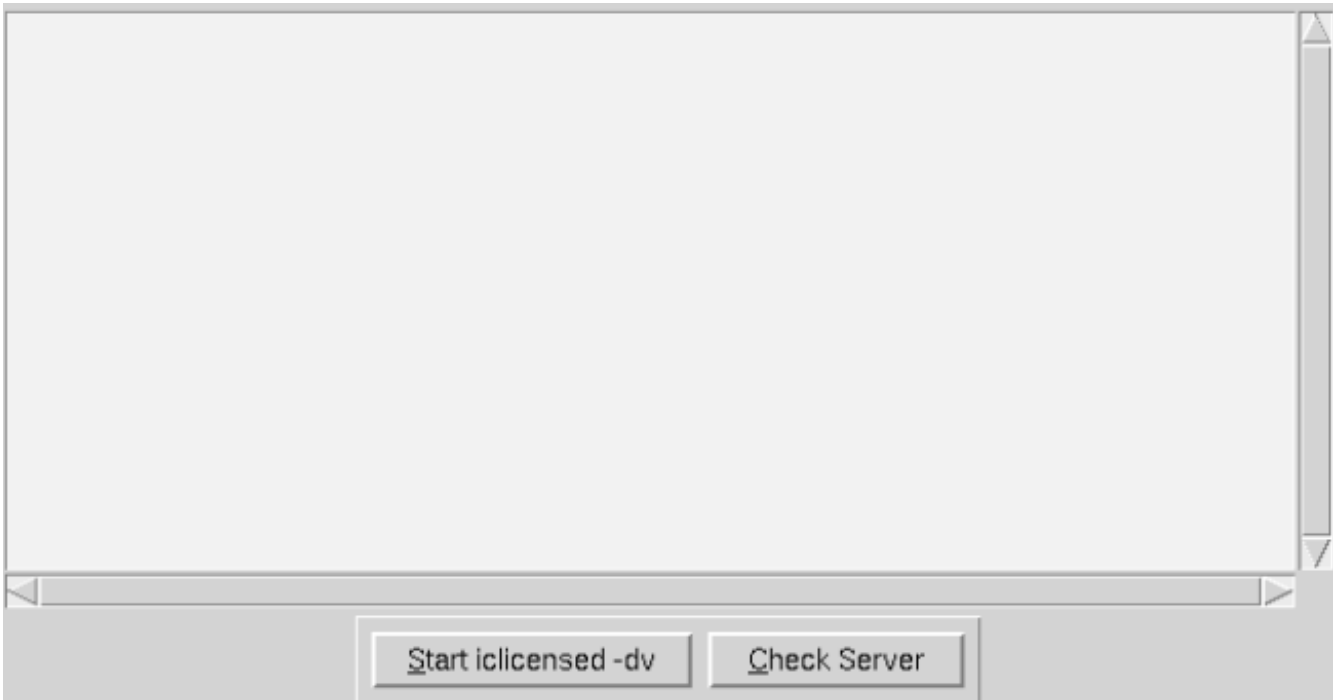
Remote Start Login Program	
Rsh (default)	<input checked="" type="radio"/>
Ssh	<input type="radio"/>
Custom	<input type="radio"/>
Command	<input type="text"/>
<input type="button" value="Update License File"/>	

3. Starting the License Server the First Time

The license server daemon will automatically be started during the first execution of the **itools** program. However, problems may arise due to incorrect installation. If the license server does not successfully start automatically, the server may be started manually in the debug mode. This will allow you to acquire more information on the nature of the problem. To start the server manually, login onto the host where the license server is to be executed. On a shell command line, you would enter:

iclicensed -dv

If you are starting the license server for the first time, it is a good idea to start it as a foreground process with the debug and verbose switches enabled. For your convenience we have build a text widget which executes "*ic -dv*" in a shell and displays its output in the scrollable text area. Press the button to start the license server.



4. Starting the License Server Remotely

The license server is automatically started when **itools** is executed or the license administration program **ladmin** is invoked. Here we start **itools** (if it is not already running) by executing the **ladmin** program.

The license administration program **ladmin** is executed by typing:

ladmin

on the command line provided that **ladmin** is in your search path. It can be found in the `${ICDIR}/bin/${ICOS}` directory where **ICDIR** is an environment variable containing the **itools** root directory and **ICOS** is an environment variable containing the name of the operating system as described in [Table 1](#). **Ladmin** executed in this manner will check out a **LOOK1.0** license. By default, all users are given 1000 **LOOK1.0** licenses. This allows all users to look at the current status of the license server.

Click the button to start the license server. If **ladmin** does not load into the window below, please check the console window for errors. The administration license may be checked out and you may need to start without administration rights.



Ladmin allows a user to look at the current status of the license server. If a license server is not currently running, ladmin will start it automatically. Ladmin is an interpreter, based on tcl, which talks to the license server. Table 2 gives a list of useful ladmin commands. The most useful command is the *license info* command which provides information on the state of the license server.

Command	Description
help	shows the available client commands
license help	shows the license server commands
free_licenses	shows the number of free licenses
license info	shows the status of the license server
license statistics	shows the license server usage statistics
license statistics reset	resets the license server usage statistics*
license statistics start	starts the license server usage statistics*
license statistics stop	stops the license server usage statistics*
exit	exits the ladmin program
license kill	kills all bogus license connections*
license timeout <time>	sets the idle license time-out *

<code>license exit</code>	stops the license server gracefully *
<code>license background</code>	detaches server to background *

Table 2 Ladmin commands. Commands with the asterisk are privileged commands and must be executed in administration mode.

The last three commands in Table 2 are privileged commands. In order to execute these commands, ladmin must be run in the administration mode. Do this by typing:

ladmin -a

on the command line. Only one user may be the administrator at any time. The *license kill* command, frees any licenses which remain locked after severe network problems. Most networks will never require the use of this command. The *license timeout* command sets the idle license time-out. If a license has been idle for more than the time-out amount, the license will be freed automatically. The default idle time-out is one day. The *license exit* command gracefully exits the license server. This command is the preferred method for stopping the license server. If the UNIX kill command is used, the license server's lock files may not be deleted properly.

5. Starting the iclicensed daemon manually

The license server may be executed manually by typing on the command line:

iclicensed -dv

This command will start the license server with debug messages allowing you to determine and correct any errors.

If iclicensed is not in your search path, it may be found in the `${ICDIR}/bin/${ICOS}` directory where ICDIR is an environment variable containing the **itools** root directory and ICOS is an environment variable containing the name of the operating system as described in Table 2 above.

If the iclicensed daemon starts manually, but fails to start automatically, run the remote shell tests described in [section 7](#).

6. Starting the iclicensed daemon in background

The license server may be executed manually by typing on the command line:

iclicensed -bv

This command will start the license server in the background from a startup script. You must set the **itools** environment variable **ICDIR** before executing this command. If the license server does not start properly, use the form "*ic -dv*" to determine and correct the problem.

If iclicensed is not in your search path, it may be found in the `${ICDIR}/bin/${ICOS}` directory where ICDIR is an environment variable containing the **itools** root directory and ICOS is an environment variable containing the name of the operating system as described in Table 2 above.

If the `iclicensed` daemon starts manually, but fails to start automatically, run the remote shell tests described in [section 7](#).

7. Common Problems

Remote Starting Problems

Most failures of the license server are due to insufficient rights to the `/tmp` directory or permission problems related to the remote shell command. The license program needs to create two files in this directory: `iclicensed.lock` and `iclicensed.pid`. These files are created and owned by the user who initially starts the license server. Before the license server can be executed, any locked files from previous runs must be removed.

The following are installation problems that prevent the license server from starting automatically.

1. The user does not have permission to login to the license server host without a password. You can check this by entering the command:

```
rsh[1] <hostname> ls
```

If you get a message similar to "`hostname: connection refused`", then either the license server host machine is down or your network is not operational. Contact your system administrator for help.

If you get a message similar to "`hostname: permission denied`", then you need to edit your `/etc/hosts.equiv` file on the machine `hostname`.

If you get a message similar to "`hostname: unknown host`", then the license was not generated for the correct host or the license file has an error in it.

2. The user does not have the `ICDIR` environment variable set on the license server host machine. You can check this by entering the command:

```
rsh hostname env | grep ICDIR
```

If you do not see a line similar to

```
ICDIR=/home/cad/bin/itools
```

then you must set the environment variable in your `.cshrc`, or `.profile`.

[1]. This command is `remsh` on HP machines. On SCO machines, this command is `remd`. If you are using the secure shell, this command will be `ssh`.

Time Verification Problems

The license server must verify the system time before starting to insure the integrity of the current time. The verification occurs once before the license server enables licenses. There are two mechanisms for verifying time: wide-area and local-area time verification. Wide-area time verification uses the Internet to transport a known time standard. If the Internet time standard is not available, the license server performs local-area time verification. A random set of machines in the local area network is chosen and their time is obtained. If the license server's local time is not within one day of the local area time, the license server is terminated.

The user should facilitate wide-area time verification if possible. The license server knows how to use a proxy server to work behind a firewall. If you are behind a firewall, use the **FIREWALL** statement in the license file. It must be the last non-comment statement in the license file. The **FIREWALL** declaration takes two arguments: the proxy host and the proxy port.

```
FIREWALL <proxy host> <proxy port>
```

The proxy is the HTTP proxy used by WEB browsers. Some examples:

```
FIREWALL www.internetcad.com 81
```

```
FIREWALL 199.1.128.171 81
```

As you can see above, either symbolic or numeric forms are accepted. You can test your firewall statement independent of the license server by executing the following sequence on the command line:

```
telnet <proxy host> <proxy port>
```

```
GET http://www.internetcad.com/cgi-bin/test_time
```

You should receive the following reply:

```
We are able to talk to the time server at www.internetcad.com.
```

If you do not get a reply at either *http://www.internetcad.com/cgi-bin/test_time* or *http://www.internetcad.com/cgi-bin/test_time*, you have either network problems or you have the wrong proxy. Please consult with your system administrator.

Example telnet session

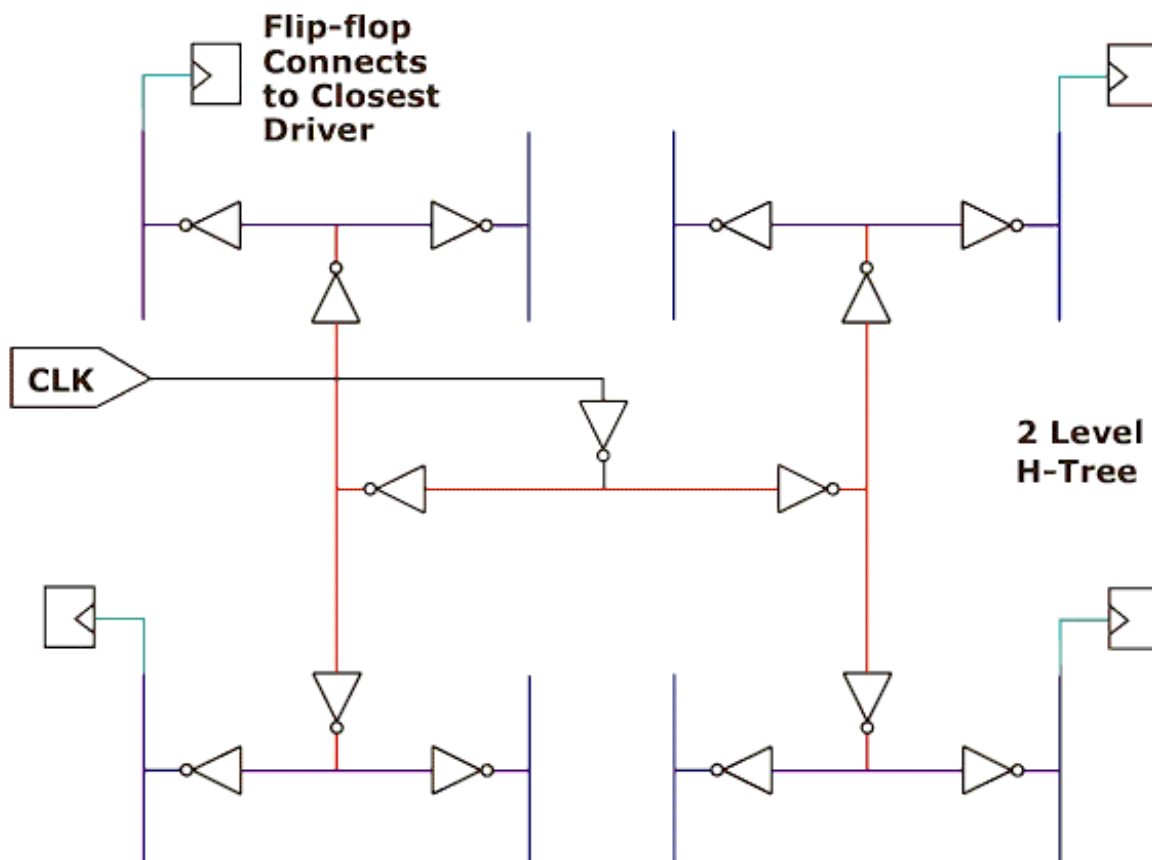
```
(19)>>>telnet ftp 81
Trying 199.1.128.5...
Connected to ftp.internetcad.com.
Escape character is '^]'.
GET http://www.internetcad.com/cgi-bin/test_time
We are able to talk to the time server at www.internetcad.com.
Connection closed by foreign host.
```

Clock Tree Synthesis

The purpose of clock tree synthesis is to minimize the amount of **skew** on the clock tree network. Ideally, it takes the identical amount of time for the signal to travel from the source or *driver* to any of the sinks or *loads* of the clock signal. However, due to the physical positioning of the sinks, the clock signal may not arrive simultaneously. Skew is the maximum time difference of the arrival time of the clock signal between any two load pins. We would like to minimize skew so that all clock signals arrive at their desired loads as simultaneously or *synchronously* as possible.

Normally, a clock is supplied to many load pins. This results in a large capacitive load which needs to be driven. Driving this large capacitance from a single gate results in a slow and poorly defined clock signal.

Itools solves the problem of skew and buffer requirements through the use of a buffered H-tree distribution network as shown in the picture. This network is synthesized during placement and the connections between the clock driver and the load are optimized so that the load will be attached to its closest driver.



Parameters

ITools can handle designs with port data outside the cell boundary. However, if requested, ITOOLS checks and records all data which resides outside the given cell boundary as an error. The option **check_ports** is read and utilized by the simplify program. However, **check_ports** is a general parameter and should be set at the general level. Use the [hypertext button](#) on the widget below to return to the general level.

SIMP*check_ports:

off on

Floorplanning Command Line Options

There are two generations of floorplanning tools : **ifp** and **igp**. The first generation floorplanner will be used exclusively in first generation flows whereas the second generation floorplanner will be called from second generation flows. We will discuss each program in turn.

First Generation Floorplanner : ifp

To execute the first generation floorplanner, type

```
ifp <circuitName>
```

where *circuitName* is the name of your design.

The command line options for the program can be displayed by typing

```
ifp
```

You should see something similar to the following:

```
>>>ifp
```

```
Macro cell Placement and Routing/Floorplanning Program
```

```
ifp version:v2.0.0
```

```
iTools compilation:Fri Dec 17 15:15:40 CST 2004 by bills using SunOS 5.7 (sun4u)
```

```
Copyright (c) 1993-2004 InternetCad.com, Inc. All rights reserved.
```

```
ERROR[check_args]:circuit required
```

```
On the command line the user typed:
```

```
ifp
```

```
Usage: ifp [-C|-Compress] [-n|-nog] [-p|-pads] [-q|-quickroute] [-r|-rewrite]
          [-v|-verbose] [-w|-windowid <windowid>] <circuit>
```

```
Description:
```

```
    This program is used to perform floorplanning, or the planning of the
    overall physical structure of the integrated circuit. This is
    accomplished by placing and routing blocks or macro cells.
```

```
Options/Arguments:
```

-C -Compress	compact files	(off)
-n -nog	no graphics mode	(off)
-p -pads	pads only mode on	(off)
-q -quickroute	quick route mode on	(off)
-r -rewrite	rewrite data in route directory	(off)
-v -verbose	verbose mode on	(off)
-w -windowid <windowid>	parasite graphics mode	
<circuit>	design name	

```
ifp terminated abnormally with 1 error[s] and 0 warning[s]
```

Second Generation Floorplanner : igp

To execute the second generation floorplanner, type

```
igp <circuitName>
```

where *circuitName* is the name of your design.

The command line options for the program can be displayed by typing

```
igp
```

You should see something similar to the following:

```
>>>igp
```

```
Macro cell Placement and Routing/Floorplanning Program
```

```
igp version:v2.0.0
```

```
iTools compilation:Fri Dec 17 16:07:08 CST 2004 by bills using SunOS 5.7 (sun4u)
```

```
Copyright (c) 1993-2004 InternetCad.com, Inc. All rights reserved.
```

```
ERROR[check_args]:circuit required
```

```
On the command line the user typed:
```

```
igp
```

```
Usage: igp [-C|-Compress] [-n|-nog] [-d|-display <display>] [-d|-do <do>]
          [-G|-Geometry <geometry>] [-i|-interpret] [-p|-pads] [-q|-quickroute]
          [-r|-rewrite] [-s|-sync] [-v|-verbose] [-w|-windowid <>windowid>]
          <circuit>
```

Description:

This program is used to perform floorplanning, and placement. It can be called as the second generation floorplanner in order to plan the overall physical structure of the integrated circuit. It also may be used to place both macro and row-based cells.

Options/Arguments:

-C -Compress	compact files	(off)
-n -nog	no graphics mode	(off)
-d -display <display>	display name	
-d -do <do>	do file name	(none)
-G -Geometry <geometry>	display geometry (500x500+1+1)	
-i -interpret	tcl stdin interpreter on	(off)
-p -pads	pads only mode on	(off)
-q -quickroute	quick route mode on	(off)
-r -rewrite	rewrite data in route directory	(off)
-s -sync	use synchronous mode for display server	(off)
-v -verbose	verbose mode on	(off)
-w -windowid <>windowid>	parasite graphics mode	
<circuit>	design name	

igp terminated abnormally with 1 error[s] and 0 warning[s]

Floorplanning Features

- Advanced simulated annealing algorithm.
 - Two floorplanners available: *ifp* (first generation) and *igp* (second generation).
 - Fast state-of-the-art timing driven placement algorithm.
 - No constraints on the number of instances.
 - Floating point data supported.
 - Macro cells of any rectilinear shape.
 - Hard and soft macro cells.
 - X11 (X11R2 – X11R6 inclusive) Tk-based graphics interface.
 - Signal path-based timing driven.
 - Upper and lower bounds on path lengths and time constraints.
 - Wire length calculations are based on actual pin locations.
 - Flexible pad placement algorithm.
 - Ability to generate placements close to that obtained from a previous run.
 - CPU time control via fast/slow option.
 - Scripting thru the use of Tcl scripts.
 - Fixed and neighborhood constraints supported.
-

Program Parameters

Floorplanning Parameters at a Glance :

[autodetect_script](#) : control automatic execution of *designName.fdo* script.
[background](#) : sets background color of graphics display.
[bendcost_threshold](#) : trade off bends versus wire length.
[block_alignment](#) : controls y – alignment of horizontally placed rows of standard cells.
[check_ports](#) : check data outside of cell placement boundary.
[chip_aspect_ratio](#) : controls aspect ratio of chip.
[contiguous_pad_groups](#) : allow nonmember pads to exist in a padgroup.
[core](#) : define the core area of the chip.
[core_to_padspace](#) : define the spacing between core and I/O pads.
[default_block_xspace](#) : set minimum amount of space between cells in x direction.
[default_block_yspace](#) : set minimum amount of space between cells in y direction.
[default_tracks_per_channel](#) : set number of tracks in channels between macros.
[detail_script_file](#) : supply a Tcl script for detail routing.
[do_compaction](#) : specify the number of placement refinement cycles.
[do_partition](#)
[double_height_rows](#) : controls the definition of double height rows.
[fast](#)
[floorplan](#)
[generate_rows](#) : control output to *itools.con* file.
[graphics](#)
[graphics_wait](#)
[ignore_keepouts](#)
[minimum_pad_space](#)
[mirror](#) : %s\n",
[no_edit_during_quickroute](#) : %s\n",
[off_corner_distance](#) : %s\n",
[origin](#) : %s %s\n",
[over_the_cell](#) : %s\n",
[padspacing](#) : uniform\n") ;
[random_seed](#) : %u\n", randVarG) ;
[require_vias](#) : %s\n", (RMODE(R_REQUIRE_VIAS)?"on":"off")) ;
[restart](#) : %s\n", (restartG?"on":"off")) ;
[row_spacing](#) : %s\n",
[rowSep](#) : %.3f\n", row_sepG) ;
[save_routes](#) : %s\n",
[scale](#) : %–4.2f\n", Yscale_get()/Yscale_modifier(–1)) ;
[script](#) : %s",
[slow](#) : %d\n", tw_slowG) ;
[softpin_pitch_factor](#) : %.3f\n", softpin_pitch_factorG) ;
[vertical_wire_weight](#) : %4.3f\n",
[wire_estimation](#) : %s\n",
[wire_estimation_factor](#) : %4.3f\n", wireest_factorS) ;
[wire_estimation_factor](#) : N/A\n") ;
[wiring_reduction](#) : calculated\n") ;

[xgrid](#) : %s\n",
[ygrid](#) : %s\n",

Floorplanning Tcl Commands at a Glance :

I/O Constraints

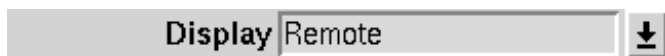
The itools program controls the placement of pads thru the use of **PAD** constraints in the [circuitName.con](#) filename. I/O constraints can be added and edited in several ways:

1) I/O constraint editor

Using the graphical I/O editor

The I/O pads can be manipulated graphically using the *itools* translator. **For the proper operation, you must click on all of the buttons. Each button will execute a command and put the program in its proper state.**

Do you want an embedded (inline window) or remote (external window) display? For small screens, a remote window is recommended.



Here we will describe the steps of operation in using the I/O Editor. First we will start the I/O pads I/O Editor:

Start I/O Editor By clicking on the *Start I/O Editor* button, we are executing the command:

```
itranslate -v -g designName
```

in a Bourne command shell. Next, we will need to read in the design data which is found in the design rule, cell library, netlist and design constraint files. We can do this by using the **icread_ic** menu option. Once we have the design loaded, we can enter the I/O editor by setting the graphics context to *ioeditor* by either the **command line** or thru the **menu interface.**

The I/O editor allows us to preview the pad placement for the design. In this editor, we have several functions which are helpful to manipulate the pad placement. Most importantly is the

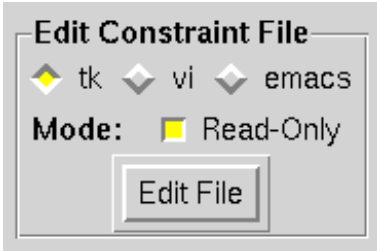
Place Pads function which is found under the I/O Editor menu. A dialog box will appear showing the four methods of pad placement : abut, uniform, variable, and exact. The abut, uniform, and variable methods are depicted in small pictures next to their selection button. The exact method means all pad locations are fixed based on

user-defined positions. To start, let us just **place the pads** using the uniform settings. The blue box shows the estimated core area which we can manipulate using the **Core Constraints** dialog box. The default core area is a square or an aspect ratio of 1.0. We can change the aspect ratio to **2.0** for example. You should see the blue box change its shape. Afterwards, you need to **place the pads** once again. It is also possible to draw the nets connecting the pads using the **icdraw nets on** command. There are a number of editing function at your disposal. You may edit a pad's position or add a constraint to pad. You may **list the constraints** you have defined.

Once you are satisfied with your pad placement, you may save the results into the **constraints file** so the placement and routing software may utilize it.

2) Directly editing the constraint file

You can directly edit the *circuitName.con* file by using the editor of your choice. Here we supply a few of the common editors. This script reads the **EDITOR** environment variable so if your favorite editor isn't listed, try setting the **EDITOR** environment variable before starting EZ.



License Server Features

The system administrator may also want to limit access to a program or programs. This is accomplished by adding the keyword **SYS_ADMIN** to the bottom of the license file. After this keyword, any number of restrictions may be added for each licensed program. The names of the license programs are found in the program information section. Each licensed program is specified:

PROGRAM <program Name> **LICENSE** # # XXXXXXXXXX

The program Name field should be used in all restriction commands. For example, the license administration program name is **LADM1.0**.

An individual program may be restricted to a set of hosts, users, groups, and operating systems. Any number of restrictions are possible. Restrictions are added to the license file with the following syntax:

RESTRICT <program Name> [**HOST** | **USER** | **GROUP** | **OS**] <hostname>

For example, to limit the program LADM1.0 to the host bulldog, you would enter

RESTRICT LADM1.0 **HOST** bulldog

If you wish to limit the program LADM1.0 to the user cadman, you would enter

RESTRICT LADM1.0 **USER** cadman

Restricting the ladmin program to just the members of the group staff, would be achieved by:

RESTRICT LADM1.0 **GROUP** staff

Restricting the ladmin program to the Sun operating system, would be accomplished by:

RESTRICT LADM1.0 **GROUP** SunOS4

Platform	Operating System
<i>DEC Alpha</i>	<i>OSF1</i> or <i>Linux-alpha</i>
<i>HP 9000 Series 700</i>	<i>HP-UX</i>
<i>IBM RS/6000</i>	<i>AIX</i>
<i>SGI Iris</i>	<i>IRIX</i>
<i>Sun SPARCstation</i>	<i>SunOS4</i> (4.x) or <i>SunOS5</i> (Solaris)
<i>Intel x86</i>	<i>Linux</i> or <i>SunOS5-i86pc</i> (Solaris)

Table 2.2 Operating system keywords for supported operating systems.

Itools Documentation

The following, is an example of a license file where the system administrator has added the above restrictions. There are no restrictions on the LOOK1.0 program.

```
# Itools License File
ITools VERSION 1.0
PASSWORD VERSION 1 774806146
# Program information:
PROGRAM LADM1.0 LICENSE 1 0 B/}HY'eYPROGRAM LOOK1.0 LICENSE 1000 0 'b'u*s]f30
# License Server information:
LICENSE SERVER HOST twolf6 /home/cad/itools 0 :C&X1*TBm.
# Host information:
# HOST hostname hostid expiration-date
HOST twolf6 08:00:2b:16:44:87 0
# Any license encoded restrictions
RESTRICT NONE
# All sys admin restrictions must be added below SYS_ADMIN
SYS_ADMIN
RESTRICT LADM1.0 HOST bulldog
RESTRICT LADM1.0 USER cadman
RESTRICT LADM1.0 GROUP staff
RESTRICT LADM1.0 GROUP SunOS4
```

Changing the ICDIR Field

The system administrator who installs iTools may need to adjust the location of the **itools** root directory in the license file. It is assumed that all machines on the network refer to the **itools** directory using the same pathname, that is, the mounting point pathname is identical. The system administrator may hard code the pathname of the **itools** root directory by modifying the sixth field of the **LICENSE SERVER HOST** statement. DO NOT change any other field of this line, as it will invalidate the license file. All keywords below are in boldface type. All statements are case sensitive. The format of this command is:

LICENSE SERVER HOST <hostname> <hostid> <itools root directory> <expiration date> <licensekey>

For example, if **itools** was installed in /home/cad/itools, change the line to

LICENSE SERVER HOST twolf6 08:00:2b:16:44:87 /usr/local/bin/itools 915170400 XXXXXXXX

or if the mounting points are all the same within the network use

LICENSE SERVER HOST twolf6 08:00:2b:16:44:87 \$ICDIR 915170400 XXXXXXXX

The last format is the recommended and default format.

Redundant License Servers

Ittools now supports redundant or secondary license servers. The redundant license servers are started automatically from a remote shell when the primary license server fails to start. This may be due to a network or machine failure. The redundant license servers will serve a reduced number of licenses according to the formula:

$$reduced_licenses = MAX (1, num_licenses / num_servers)$$

where

- ***num_licenses*** is the original number of licenses available.
- ***num_servers*** is the total number of primary and secondary (redundant) servers.

When the primary license server returns online, the redundant servers are killed automatically. In normal operation, only the primary license server runs on the network; secondary servers are started on a demand basis.

Starting a Redundant License Server Manually

Normally, the secondary license server will be started automatically from the program needing the license. However, this may fail if the user does not have remote shell privileges on the machine hosting the redundant server or the [root directory](#) is not specified correctly. To start a redundant server manually in debug mode type:

iclicensed -drv

To start the redundant server in background use:

iclicensed -brv

The redundant license server has all the properties and features of the primary license server except that the number of licenses are reduced.

Ittools Placement Output Files

Message Files

All error messages are normally directed to the screen and appended to the *designName.log* file. In addition, other messages may also be written to one of the **ittools** output files depending on the program that is being executed. The following programs have output files:

Program	Function	Output Message File
-----	-----	-----
igrouter		designName.out Global Router
ifp/igp		designName.out MacroCell Placer
iplace		designName.out Row-Based Placer

Each file contains data concerning the executed program. Data is also printed in these files following each iteration of the simulated annealing programs: **ifp**, **igp**, and **iplace**. Much, although not all, of the information available in these files is self explanatory.

Placement Output Description Files

File Format

There is one basic file format describing the placement information in the **ittools** system. The format for each line is as shown below:

string string string number number number number integer integer

with the following meaning:

Instance Model Version x1 y1 x2 y2 Orientation Row_number

NOTE: this output format differs from previous versions of **ittools**. Use

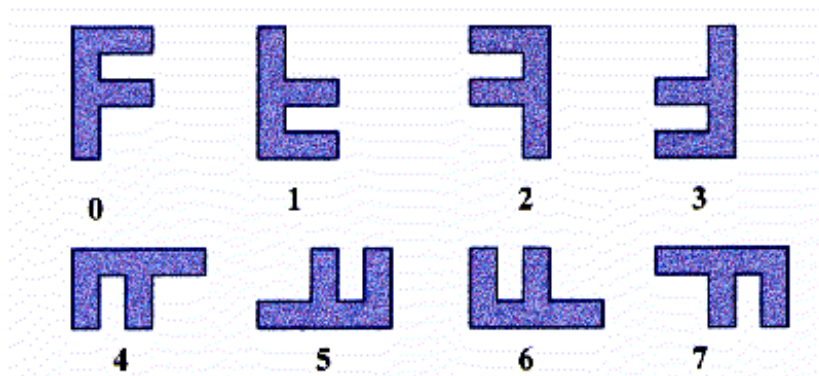
*old_placement_format : on

in the *designName.par* file to revert to the old format. The default is the new format.

All *designName.pl{x}* where x is 1..3 use this format. The first string is the name of a cell, macro block, row block, or pad. The second string is the name of the library model or the block name. The third string indicates the version which has been selected. The pair of numbers following the third string represent the x,y-coordinates of the lower left corner of the cell, macro block, pad, or row. The third and fourth numbers represent the x,y-coordinates of the upper right corner of the cell, macro block, pad, or row. The number of decimal points is controlled by the [scale](#) parameter. **Ittools** autoscales its numbers and uses integers whenever possible.

The next integer represents the orientation of the cell, macro block, or pad. This field is set to 0 for rows. All cells are presumed to have been entered in orientation 0 in *designName.lib*. The orientation number, if different from 0, represents a change in a cells orientation. The orientation numbering scheme was presented above during the description of the contents of the file *designName.lib*. However, due to its immediate relevance here, it will be repeated.

The **itools** orientation numbering scheme is as follows: Orientation 0 is that description of the geometry appearing for cells, pads and macro blocks. The other 7 possible orientations are as follows: (1) Orientation 1 represents a mirror of the cell's y, that is, a mirror about the x-axis with respect to orientation 0. (2) Orientation 2 represents a mirror of the cell's x-coordinates, that is, a mirror about the y-axis with respect to orientation 0. (3) Orientation 3 represents a 180 degree rotation of the cell's coordinates with respect to orientation 0. (4) Orientation 4 represents a combination of a mirror of the cell's x-coordinates followed by a 90 degree rotation of the cell with respect to orientation 0. (5) Orientation 5 represents a combination of a mirror of the cell's x-coordinates followed by a -90 degree rotation of the cell with respect to orientation 0. (6) Orientation 6 represents a 90 degree rotation of the cell with respect to orientation 0. (7) Orientation 7 represents a -90 degree rotation of the cell with respect to orientation 0.



Ittools orientations.

The sixth integer represents the row number to which a cell belongs. This field is set to zero for all macro blocks. For pads on the left side of the chip, the field is set to -1. For pads on the right, bottom, and top sides of the chip, it is set to -2, -3, and -4 respectively.

Generated Files

As stated earlier, **itools** generates multiple placement files throughout the placement and routing process. The table below displays the purpose of each file.

File	Program	Function
-----	-----	-----
<i>designName.pl1</i>	iplace/igp	
<i>designName.pl2</i>	igrouter	Global router placement
<i>designName.pl3</i>	igrouter	Row evening placement

The first entries of detailed placement files are the cell's placement information. These entries are sorted by row number. That is, the cells for row number 1 appear first, then the cells for row number 2 appear second, and so on. Furthermore, the cells for each row are sorted in left to right order. Following the cell placement information, the pad and macro block bounding box placement information appears.

If the floorplanning tool, **imacro** is executed, it will produce a *designName.pl1* output file. Only the bounding box of the macro cell will be given in the *designName.pl1* file; the *designName.mlib.out* file should be analyzed for the final vertex description. All macro cells are converted to hard macro cells with their vertices and ports chosen. The *designName.mlib.out* file has the same format as the [designName.lib](#) file. Note: earlier versions of **imacro** required the appearance of **ICMC*sc_output_files : on** in the *designName.par* file for the creation of the *designName.pl1* file.

Itools Documentation

The coarse placement description files contain the placement information for each of the rows, pads and macro blocks. Note that footprint files do not contain the row cell placement information. The intention is for this file to aid the user in determining the proper configuration of the circuit, that is, the number of rows, the macro block placement, and the pad placement.

The first entries in the coarse description files are the rows placement information. These entries are sorted by row number. Following the row placement information, the pad and macro block placement information appears.

The *designName.pl1* file describes the cell's configuration after placement. In contrast, the *designName.pl2* file describes the placement configuration after the global routing step. Explicit feed through cells appear in the *designName.pl2* file if the design requires them. The *designName.pl3* only appears when row evening is performed. Row evening is automatically performed when the design requires large number of explicit feedthrough cells.

Silicon Ensemble I/O Format

There currently is some rudimentary support of the Silicon Ensemble format. We only support reading pad name and side information.

Use the button to read in SE I/O file information.

Read SE I/O File

Global and Detailed Routing

Global Routing

Do you want an embedded (inline window) or remote (external window) display?

Display

Remote

↓

Power and ground network

At this point we need to determine when we are going to instantiate the power and ground network. Normally, we defer this step to detailed routing but we can instantiate a cell-based construction of the power and ground networks at either the placement or global routing steps. It is often necessary to place cells which [strap the interior of the core](#). The user can create the layout of these cells using Tcl scripts. Use the tool below to choose the power and ground configuration from the list of possible choices.

You may either allow EZ to generate the script for you or supply your own script:

Generate Global Routing Script

Select the Global Routing Script:

Browse ...

or

Supplying your own global routing script is only recommended for advanced users. This script will override all settings in this subsection. It is ok to leave this blank. If the setting is blank, the generated script will be used instead. Generate the [custom detail routing Tcl](#) and then locate the Tcl script. If you do supply a script, you will need to update the parameter file.

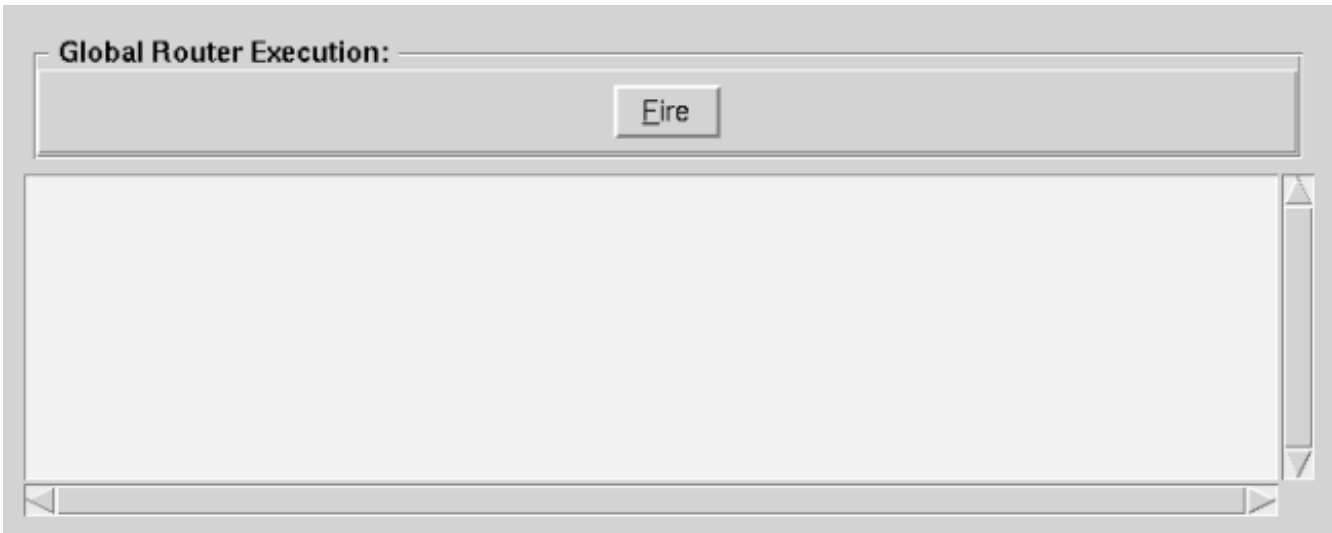
You may want to edit the parameters which control the execution of the global

Edit Global Router Parameters

=>

Update Parameter File

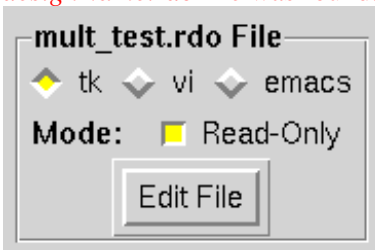
This is the transcript window for the global routing process. To start the global router, hit the "Fire" button. You may resize the window by using the first mouse button on any corner of the display window.



[Global Routing Output Files](#)

Detailed Routing

Iroute is a Tcl-driven program and needs a set of [routing commands](#) in order to execute the routing process. **A *designName.rdo* file was found.** You may view the file using the button below or completely recreate it.



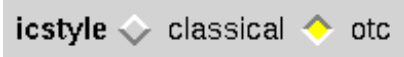
If you are satisfied with the contents of the routing script, you may [execute the detail router](#).

Creating Routing Scripts

In this section, we will create the mandatory parts of the routing scripts. There are two levels of scripts: the ***designName.rdo*** script for the high-level routing control program *iroute* and lower level ***regions.ddo*** and ***entire.ddo*** scripts to control the detail router program *idetailer*. **The tutorial has set all of the buttons for you. You only need to execute the programs.**

1. Routing style

First, we need to determine the routing style. There are two possible routing styles: ***classical*** and ***otc***. The ***classical*** style only allows routing in the space between cell instances whereas the ***otc*** or over-the-cell style does not put any restrictions on where routing may occur. Both styles must obey the keepout regions (or user defined regions where routing is not permitted) and design rules. The default design style is over-the-cell.



2. Core Routing

Next, we perform the core routing using the region command. This will route the regions between the row-based cells. Connections between the row-based cells and the pads will be omitted. If this is a macro-cell only design, you should chose the **omit** option. Under the proper conditions and proper library design, it is possible to route all row based channels in [parallel](#) simultaneously.

icroute ☒ regions ☐ parallel ☐ omit step

Since we are going to route core area in separate steps, we may route using either the variable or fixed die methodology. In the variable die approach, the router is allowed to expand the region between the rows in order to complete the routing. This methodology guarantees routing completion regardless of the congestion at the expense of area. The fixed die approach does not allow the router to add area and hence the area is fixed. The fixed methodology is the approach offered by the rest of the CAD industry. The variable die methodology expands congested regions rather than forcing the maze router to find long and circuitous paths in the congested regions. It also allows the user to complete designs using less layers of metal. One can even trade off routed area versus number of layers of metal. The variable die approach minimizes the routing region automatically and will route the design without extra whitespace if possible.

methodology ☒ variable ☐ fixed

Now we need to set the iTools router framework. Currently, there are two frameworks: contour (shape-based) and gridded. The contour framework is completely griddless and suitable for analog design and small designs as the framework requires significant runtime. The gridded router is suitable for digital designs and runs quickly.

region framework ☐ contour ☒ gridded

The gridded framework needs a virtual uniform grid definition. Use the tool below in order to enter the grid values for each routing layer.

Layer Grid Definition Tool

icgrid layer METAL1 {0.72 0.36 0.56 0.28}
 icgrid layer METAL2 {0.72 0.36 0.56 0.28}
 icgrid layer METAL3 {0.72 0.36 0.56 0.28}
 icgrid layer METAL4 {0.72 0.36 0.56 0.28}
 icgrid layer METAL5 {0.72 0.36 0.56 0.28}
 icgrid layer METAL6 {0.72 0.36 0.56 0.28}

Auto Extract : ☐ None ☒ Offset Only ☐ Pitch & Offset

New Layer Grid

Edit Layer Grid

Delete Layer Grid

Now that we have selected the methodology, we need to generate the region Tcl script known as the *regions.ddo* file. The *regions.ddo* file controls the detail routing algorithm. EZ will generate a different script based on our choice of

methodology above. If you change methodology, you must regenerate the script. You may either allow EZ to generate the script for you or supply your own script:

or

Select the Detail Routing Script:

Alternately, it is possible to customize the detail router algorithm using Tcl. **This is only recommended for advanced users. This script will override the regions.ddo file generated above. It is ok to leave this blank.** Generate the [custom detail routing Tcl](#) and then locate the Tcl script.

Regions.ddo File

☒ tk
 ☐ vi
 ☐ emacs

Mode: ☒ Read-Only

You can look at the generated *regions.ddo* file or your custom script using the editor of your choice.

3. Full Chip Routing including I/O, Power/Ground, and Antenna Rule Routing

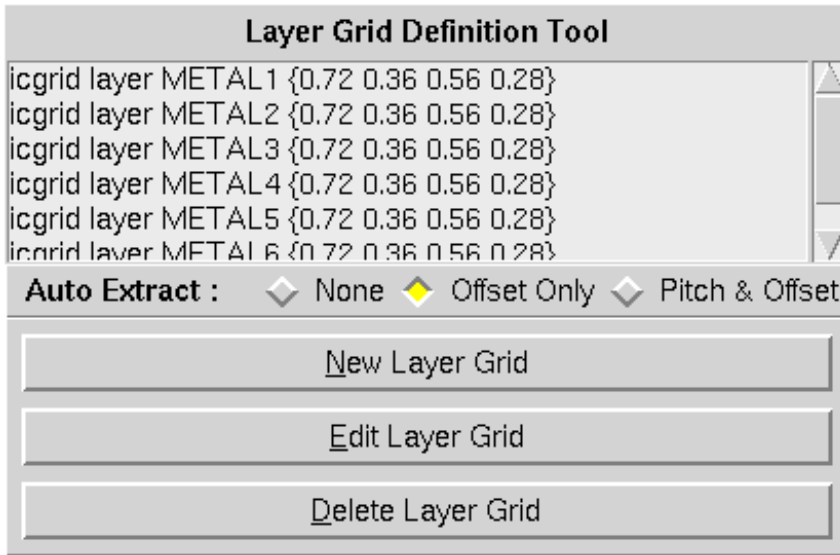
We now route the pads to the core using the **icroute entire** command. In addition, in this step we will connect the power and ground routing as well as performing antenna rule corrections on the final route. This step will read in the routing performed on a region basis as prerouting to this step. **This step is mandatory under most circumstances**

icroute
☒ entire
 ☐ omit step

First we must set the iTools router framework. Currently, there are two frameworks: contour (shape-based) and gridded. The contour framework is completely griddless and suitable for analog design and small designs as the framework requires significant runtime. The gridded router is suitable for digital designs and runs quickly.

framework
☐ contour
 ☒ gridded

The gridded framework needs a virtual uniform grid definition. Use the tool below in order to enter the grid values for each routing layer.



If there are pads present, we may need to perform final placement of the pads. The *Core To PadRing* entry form controls the spacing between the core and the pad ring. There is a more complete graphical interface within the detail routing tool but the following interface allows one to set up the basics.

Usually, power and ground signals are handled as a special set of signals. In the detail router, you can use the graphical interface to design your power and ground network which will give you immediate feedback. However in EZ, you can only generate a template for the detail router. There are two types of constructive power networks available: *ring* and *stripe* networks. In *ring* construction, the power signals form a ring around the core of placeable objects with straps which connect the ring to the core. In *stripe* construction, vertical stripes are placed at the left and right of the core and straps are projected into the core.

power routing ☒ ring ☐ stripe ☐ none

This is the ring graphical interface found in the detail router. At the top of the form is the field for the nets to be routed. Enter the signal names separated by a space. The ring consists of two vertical segments and two horizontal segments. Use the pulldown menu to choose the layer for each direction. Enter the desired width of the core ring in the *Core Ring Width* fields. Normally, the ring will be centered although the user can add extra space to either the I/O or the core region. By default, the detail router will check for design rule errors before adding the routing and will **NOT** add any routing that would cause a design rule violation. The *Force Insertion* checkbox overrides the checking and will add the power and ground networks even in the presence of design rule errors. By default, the power and ground network is added to the design as pins which may not be modified by the router. If desired, the networks may be added as routing which may be ripped up by the router if needed.

In deep submicron CMOS processes, it is necessary that the router does not create *antennas* which will destroy the gate of MOS transistors during processing. The iTools router has a facility for removing these violations. Use the button below to enable or disable this feature.

icantenna ☒ route ☐ omit step

Now we need to generate the top level routing Tcl script known as the *entire.ddo* file. The *entire.ddo* file controls the detail routing algorithm. EZ will generate a different script based on our choices above. If you change options, you must regenerate the script. You may either allow EZ to generate the script for you or supply your own script:

Generate Entire Script

Select the Top Level Detail Routing Script:

Browse ...

or
Again, it is also possible to customize the detail router algorithm for the top level routing using Tcl. **This is only recommended for advanced users. This script will override all settings in this subsection. It is ok to leave this blank.** Generate the [custom detail routing Tcl](#) and then locate the Tcl script.

Entire.ddo File

☒ tk
☐ vi
☐ emacs

Mode: ☒ Read-Only

Edit File

You can look at the generated *entire.ddo* file or your custom script using the editor of your choice.

4. Routing Output

Finally, we can tell *iroute* to collect the routing data and output the routing data into the *designName.rte* file.

icoutput ☒ output ☐ omit step

5. Generating the *designName.rdo* Script

From the routing options above, EZ can generate a minimal routing script for the top-level routing control program *iroute*. Use the button below, to create the script.

Generate Route Script

After generating the routing script, you may edit it to your liking.

mult_test.rdo File

☒ tk
☐ vi
☐ emacs

Mode: ☒ Read-Only

Edit File

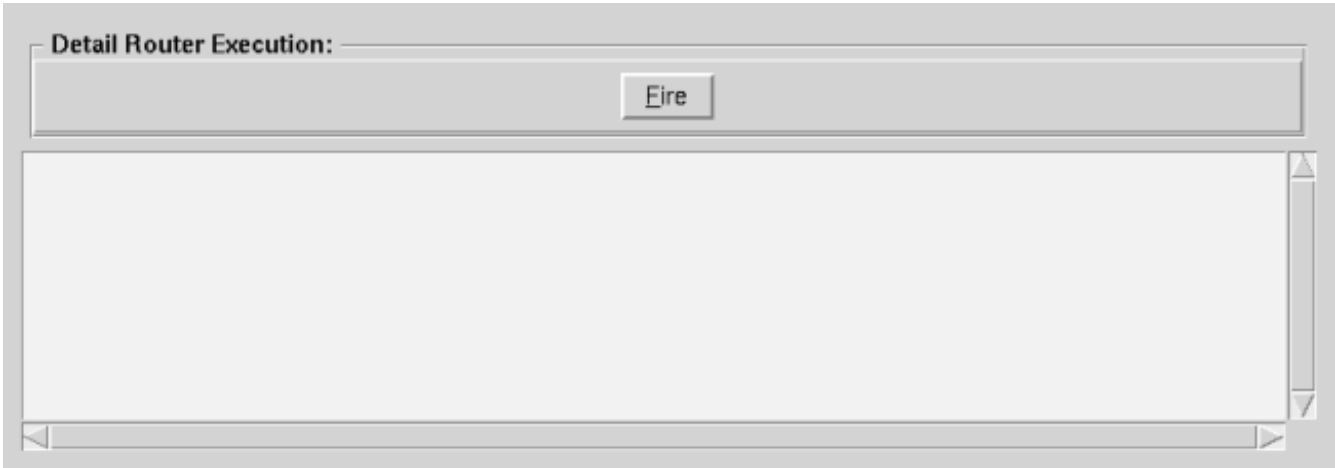
Executing the Detail Router

You may want to edit the parameters which control the execution of the detail


router.  =>



This is the transcript window for the detail routing process. To start the detail router, hit the "Fire" button. You may resize the window by using the first mouse button on any corner of the display window.



Detail Routing Issues

Use the button to view the routing whenever the button is green. 

 =>

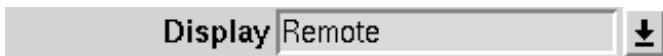
Routeability and Congestion

Routeability Analysis

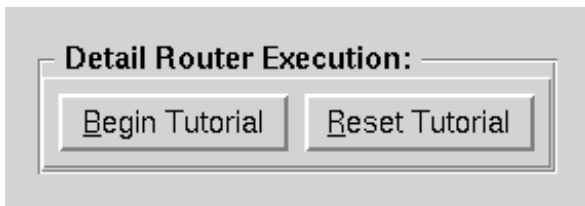
The first step in detail routing is to insure that your design is *routeable*, that is, each net may be routed without any other nets present. A route may not be able to be found because pins are blocked by keepouts or other pins or the pins given are smaller than the minimum design rules. In most cases, this is due to bad input data to the router.

We will now present a series of tutorials designed to explain routeability and congestion. In these tutorials, all steps are automatic, that is, the buttons in the hypertext execute the menus and enter text on the command line for the user. No other user input is required. However, the program is live and users may enter commands as they wish. At any time the user may stop the tutorial and reset it to its initial state using the button provided. The user has the option of either a hypertext or toplevel (remote) window. For small screens, a remote window is recommended.

Do you want an embedded (inline window) or remote (external window) display?



This is the transcript window for the detail routing process. To start the tutorial, hit the "Begin Tutorial" button.



Now that we have loaded the design, let's check to make sure that it is routeable. To do this we enter the *icrouteable* command or choose *Routeability Analysis* under the *Verification* menu which we will can do by clicking on the

button: `icrouteable.` As you can see, nets 1 and 2 are declared not *routeable*. Now let's investigate further. Optionally, the command *icrouteable* may take the net name as an argument in order to discover problems with an individual net. First look at net 1. The itools router assigns a number to each net name. We can find the mapping

between the net number and the name by entering `icnet name 1` on the command line. In this case, net 1 was

mapped to net A. Now let's just look at pins of net A by entering `icdraw net A on; icdraw` Now let's




investigate why A is not routeable. We can enter `icrouteable A -pin_access` and the router states that we can not leave from one of the pins of this net. Something seems wrong. It looks like it is easy to connect these three pins. Next we try to route as much of this net as possible. We do this by entering the command

`icroutenet A -partial -exhaustive` which tries to route from each pin (-exhaustive) while trying to complete as

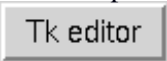
much of the route as possible (-partial). After `icdraw,` we see that two of the pins are connected. What is keeping the last two pins from connecting? We must check to make sure that there are no keepout or unconnected pins

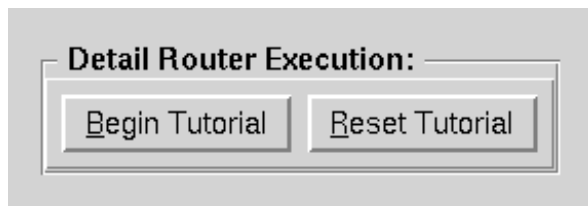
in the way. First turn on the keepouts `Draw Keepouts` (*no problems due to keepouts*) and then turn on the

unconnected pins `Draw Unconnected Pins.` In addition, we display the other nets `icdraw nets on` and we see that an unconnected pin is blocking our path and due to the keepouts no path is possible for this net. It is easier to

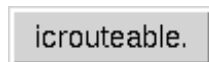
see if we look at the net's *fly lines* which are properly called the *minimum spanning tree* or  for short and look at the  to be connected. Here we see that the net's shortest connections cross and we have a disconnected pin blocking the path. This is a data problem and is a common reason why routes are not completed. We must fix our design and then return to routing. Now it is time to  this tutorial. No need to save anything.

Congestion




Let's edit the files necessary to make this design routeable. We need to remove the unconnected pin from the design. We will do this by commenting out the C1 pin in the *routeable2.lib* file as we have decided to change the cell and remove this pin. We now call the  to accomplish this task. Make sure to *Save* your edit. If you have any problems, you can always use the button below to reset the files of the tutorial. Now we can rerun the tutorial. To start the tutorial, hit the "Begin Tutorial" button.



After loading the design, let's again check to make sure that it is routeable. To do this we enter the *icrouteable* command or choose *Routeability Analysis* under the *Verification* menu which can be done by clicking on the button:

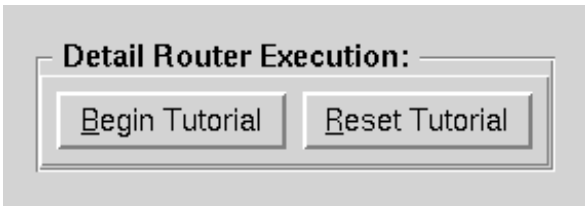


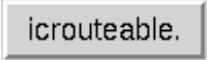

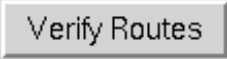

As you can see, nets 1 and 2 are now *routeable*. We would think that we should now be able completely connect all of the signal nets. Unfortunately, this is not necessarily the case. Routeability is a necessary but not a sufficient condition for the router to connect all of the signals. Routeability concerns the ability to find a route for each signal independently. However, when we try to route all of the nets to completion, we may run into problems, namely, *congestion*. Congestion is the result of multiple signal nets competing for fixed routing resources. Simply put, there isn't enough space for all of the wires to exist without design rule violations. Let's try to route this design. We will

start by executing  to connect as many signals as possible. We will then try to use ripup and reroute to connect as many as possible.  You can see in this small case that it is hopeless since we only have one layer for routing and we don't have enough space. Where routeability is usually due to a data error, congestion is normally due to either poor placement or lack of routing resources (not enough layers or space). Congestion problems are much harder to correct. We will fix this design by changing the placement of the pins in the next tutorial. Now we should  this tutorial. No need to save anything.

Routeable Design

Now we will change the placement of the pins in this design. Again to start the tutorial, hit the "Begin Tutorial" button.



After loading the design, let's again check to make sure that it is routeable. Again click the button: . As you can see, the design is *routeable*. Now let us see if we can route all of the signals. Again we execute  to connect as many signals as possible. In this case, all of the routes are connected as we can verify using . You can see that all nets are connected. This concludes our tutorial on routeability and congestion.  No need to save anything.

Customizing the Detail Routing Algorithm

The table below describes additional Tcl commands which are available along with the basic Tcl commands.

Command	Function
icadd_keepin	add fence to keep routing contained
icbeautify	cleanup the routing by rerouting
iccompact_channel	removes tracks from row-based channel
iccompute_boundary	compute routing boundary
icdelete_net	delete routing for a net
icdelete_routing	deletes all routing within an area on a given layer
icglobal	controls routing of special global signals
ichelp	this message
icnet_spacing	sets the net's spacing
icnet_width	sets the net's width
icoutput	output the routing in DEF format
icreport_errors	controls the amount of error reporting
icreroute	reroute a net
icreserve_layer	sets whether layer is used for graph routing
icrestore	restore a routing state
icroute	route the design
icswroute	switchbox routing (under construction)
ictraditional_channel	old technology mode
icverify	verify design rule correctness
icvirtual_channel	sets the bounds of the row-based virtual channel

The following additional commands are available in the graphics mode. Most are available from the top level menus.

Command	Function
icdraw_ctiles	controls drawing of contour tiles
icdraw_routing	controls drawing of routing tiles
icdraw_stiles	controls drawing of space tiles
icedit_inst	edit the properties of an instance
icgraph_longest_path	display longest path in graph
icgraph_max_density	display area of maximum density
iclayers	controls display of layers
icmove_inst	move an instance using the mouse
icobject	return the object under the mouse
icpick_point	returns coordinate under mouse
icpick_tile	alternate form of tile information
icset_3d_camera	controls 3D drawing
ictell_tile	information about tile under mouse

Routing Scripts

The default script for all designs is shown below.

```
icvirtual_channel -top pin -bottom pin

icroute

icoutput

update
```

This script is appropriate for row-based designs. Let's look at each of the commands in turn. The command **icvirtual_channel** creates the routing keepin or fence for a row-based design. The boundary of the channel in the y direction is specified as the topmost and bottommost pins. This command also allows the position to be specified relative to the center of the rows as well as absolute position. Type "**icvirtual_channel -help**" to find out all of the forms of this command. The next command **icroute** is the primary routing command. The primary routing command will first perform graph-based prerouting (if applicable), then perform maze routing, and finally perform channel compaction if row-based. The **icoutput** command saves the routing state to a DEF file. Finally, the Tcl command **update** refreshes the screen.

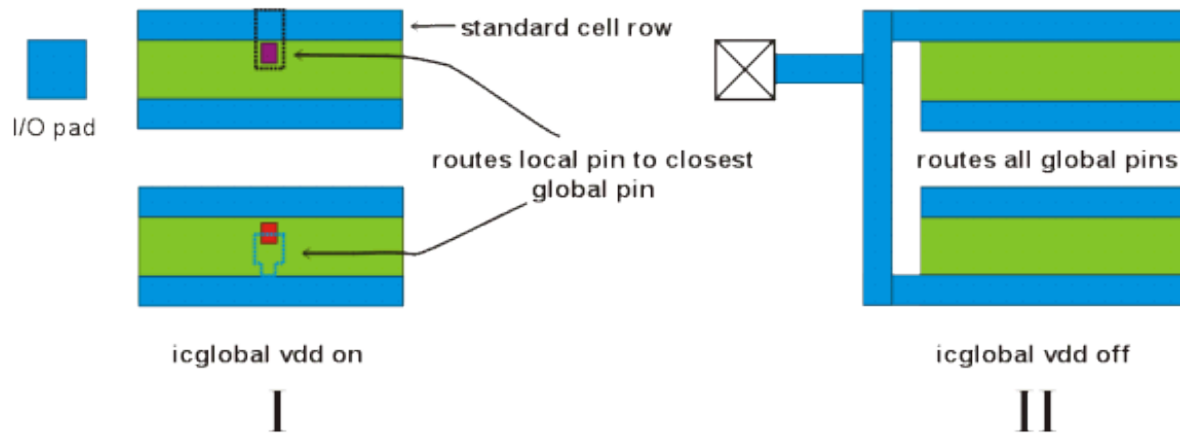
Routing Global Signals

The detail router has the ability to handle global and special signals in a hierarchical manner. This is accomplished through the **icglobal** command. The format of the command is:

```
icglobal <net> [on | off]
```

This command enables and disables the global net properties. When enabled the global net acts as a connected net, non global pins may validly connect to the closest global pin. A global pin is a pin which has type POWER, GROUND, or SPECIAL and belongs to a net which has been constrained to be GLOBAL. Non global pins are pins which do not have pin type POWER, GROUND, or SPECIAL and thus a member of the set {INPUT OUTPUT BIDIRECTIONAL FEED UNKNOWN}. The global pins are not required to be connected while the global property is enabled. When disabled this special property is not enforced, that is, all global pins must be connected. Disabling this global property allows the global net to be connected among themselves.

This routine returns ERROR if the net is not a global net or is not a valid net. Otherwise, the routine returns the current state. If the third argument is supplied, the current state is modified.



In the picture above, the global net property is enabled in frame I. Here, you see that all non-global pins are connected to the closest global pin. In the second frame, the global property is disabled and all pins of the net must be connected. In addition, the width of the net at each stage may be different.

An example of routing power signals is given in the [power.tcl](#) script found in the routing Tcl directory. In this script, the non-global pins are connected first at the minimum widths. This partial routing is then stored. Next, the width is increased to the desired size at the global level and the global property is turned off. At this point the entire net is routed. The script then subtracts the initial partial routing from the completed routing. The result is the wide routing at the top level of the hierarchy without any of the connections to the non-global pins. This wide routing is then turned into prerouting pins which becomes a permanent part of the net. The net is then returned to a minimum width net. This entire process is performed to allow ripup of the local non-global connections. Otherwise, these local connections may block completion of other nets in the vicinity.

Parallel Routing

Enabling Parallel Routing

The first step to perform parallel routing is to create a [parallel host](#) file. Next, start the icdaemons on the remote hosts either [manually](#). Third, enable the parallel option for iroute control program. The default is to run ITOOLS in a single serial process. Each routing region will be executed in turn serially. In the **multi** mode, multiple regions will be executed simultaneously reducing the run time.

ICRT*parallel:

 off  multi

Finally, update the parameter file to reflect the changes.

Update Parameter File

Expected Run time

The expected run time for the routing process currently depends on the number routing regions. In serial mode, the total run time is given by

$$\text{total_time} = \text{time_entire_route} + \text{time_region_1} + \text{time_region_2} + \dots \text{time_region_n}$$

In parallel mode, the expected execution time is given by

$$\text{total_time} = \text{time_entire_routine} + \#_parallel_sets * \max \{ \text{time_processor_1}, \dots \text{time_processor_n} \}$$

where $\#_parallel_sets = \text{number of regions} / \text{number of processors}$.

Itools Global Router Output Files

The CircuitName.pin File

The global routing information is presented in output files named *designName.pin* and *designName.pn2*. The format of both files are the same. The *designName.pin* file describes the routing after the initial simulated placement step. The *designName.pn2* is only present if row evening is performed. In this case, the *designName.pn2* describes the global routing after row evening step.

Each line in these files consist of 10 fields of information concerning an *active* pin. An active pin is a pin which will participate in the minimum area global route of the circuit in the following format:

```
NETNAME GROUPNO INSTANCE MODEL PORT X Y CHANNEL LOC LAYER
```

where the fields are defined as

```
string integer string string string number number integer integer integer
```

Pins which are *inactive*, that is, pins which are not used in the minimum area global route, are not included in this file.

The fields for each active pin are now presented. The first field represents the name of the net attached to the pin. The second field is an integer which represents the group number for the pin. The *group* numbers are globally unique. That is, each net has its constituent pins broken down into groups such that the pins with a common group number are to be interconnected. Two pins belonging to the same net but with different group numbers are not to be interconnected. Hence, a channel or detailed router is not to be given the net name for a pin, but rather the group number in order to achieve the minimum area layout. For clarity, the user may wish to give both the net name and the group number (in the fashion: netName_groupNumber) so that the label for a pin produced on a plot contains the net name. In any event, the group number must be given to the channel or detailed router in order to achieve the routing density reported by **itools**.

The third field is a string representing the name of the instance to which the pins belongs. All instance names are unique within a design. All pseudo cell pins will be suffixed with a number to insure uniqueness. All explicit feed through instances will begin with the prefix ICFD_.

The fourth field is a string representing the model name of the cell to which the pin belongs. The fifth field is a string which represents the port name. (Remember, port names are unique with respect to the cell model). The sixth and seventh fields are a pair of numbers representing the x and y coordinates of the location of the pin. These numbers, if possible are expressed as integers; otherwise, the output number is in hundreds of microns.

The eighth field represents the channel number to which the pin belongs. Horizontal core region channels are numbered (starting at 1) from the channel below the first row. The last channel is numbered (numRows + 1). All I/O channels have zero assigned to their channel field.

The ninth field is a location field. It is an integer which takes on one of four possible values (– 2, – 1, 1, 2). A net, which must leave a horizontal channel and enter a vertical channel does so by means of a pseudo pin at either the left end or right end of a horizontal channel. A pseudo pin at the left end of a horizontal channel has this location field set to – 2 and a pseudo pin at the right end of a horizontal channel has this location field set to 2. The name of a pseudo pin is: PSEUDO_PIN and it is said to lie on a cell named: PSEUDO_CELL. A pseudo pin will be included in two different group sequences – a core subnet and a I/O subnet. The unique pseudo instance name allows for matching between these two sequences. This simplifies connectivity checking on nets, which contain many pseudo pins. This is

furnished as an aid for detailed routers.

If a pin is located at the top of a channel the field is set to 1. If a pin is located at the bottom of a channel the field is set to -1.

The tenth and final field is an integer indicating the layer assigned to this pin.

In summary, the ten fields for an active pin port are:

1. Name of net to which the port belongs.
2. Group number of the port.
3. Cell name to which the port belongs.
4. Model name to which the port belongs.
5. The name of the port.
6. The x-coordinate of the port location.
7. The y-coordinate of the port location.
8. The channel to which the port belongs.
9. Whether the port is at the top (1), bottom (-1), left end (-2), or right end (2) of the channel.
10. Layer.

Notchfilling

Notchfilling

Notch filling is a post processing step used to deal with an artifact of detail routing. In order to minimize the number of design rule checks during routing and to allow a better chance to complete the routing, many routers including the itools detail router allow samenet spacing violations on the routing layers. For example, we can see that in Figure 1 below the second metal layer (magenta) geometries have insufficient spacing as shown in the yellow region in Figure 2. Note that all geometries connect the same net or signal and should all be at equipotential.

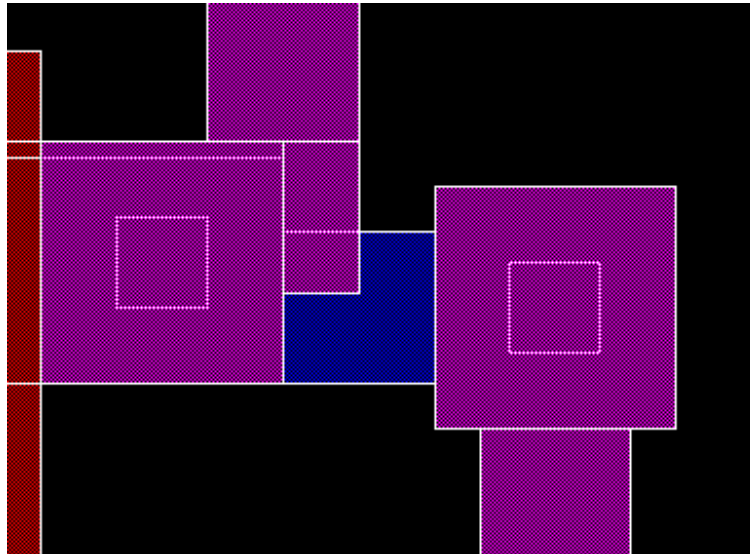


Figure 1. Routing with intranet spacing violations

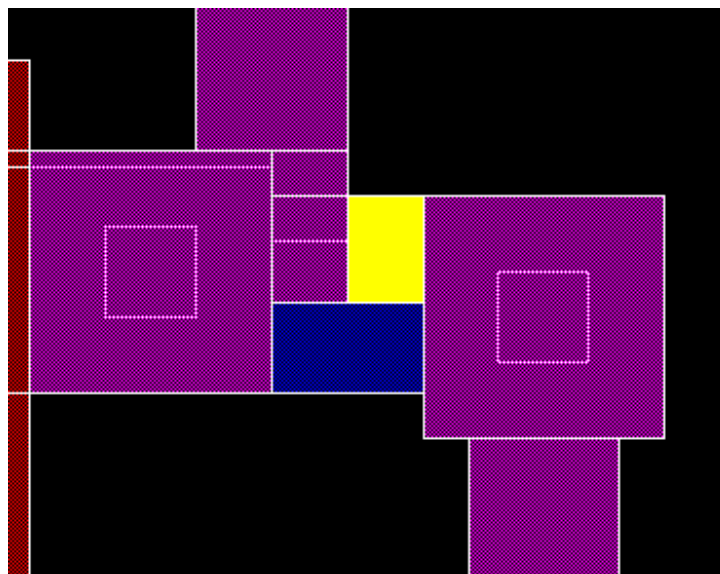


Figure 2. Intranet spacing violation shown in yellow.

In order to correct this problem, we have two choices. Either we 1) reroute the net and move the second metal layer apart or 2) we notch fill the net as shown in Figure 3.

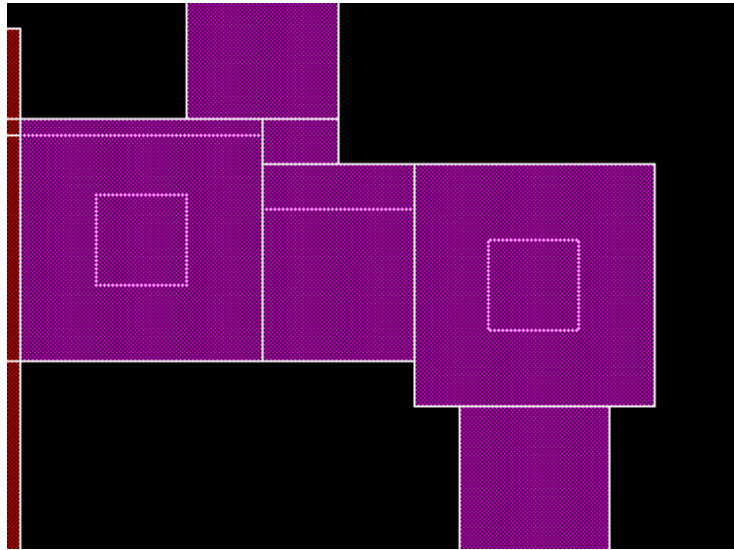


Figure 3. Notchfilling closes the intranet spacing violations.

As you can see the intranet spacing violation is eliminated by filling in the notchs or areas where the spacing between routing on the same layer is less than the design rule. This is executed as a post processing step. Notch filling is automatically performed when outputting the detail routing. In addition, it may be called directly using the Tcl command : *icnotchfill* or may be executed thru the **Reroute Nets With Notch Errors** command found under the **Route** menu.

However, from Figure 3, you can also see a disadvantage of notch filling – the creation of large wires. In more aggressive modern processes, the design rule requires that larger wires require more space between them. If the filled in notch creates a *large* wire then it is possible that the larger spacing may create a design rule violation with other nets. So by fixing the intranet spacing violation we may create a *large* wire and an *internet* spacing violation. Furthermore, the requirements for large wire are often more stringent than the intranet spacing requirements. In this case, rerouting the net using more conservative samenet cut layer spacing would be more advantageous. If samenet cut layers are sufficiently far apart, then no notch can occur in the first place. If the user puts the keyword **ICDR*paranoid_samenet_rules on** in the parameter file, the program will attempt to calculate samenet rules which will minimize notchfilling. This is given as a convenience function; it is recommended that the user calculate the samenet cut spacings for best results.

Itools Documentation

```
#!/usr/local/bin/wish -f
# Program: power/ground routing.
# $Id: power.tcl,v 1.5 2003/02/17 13:06:25 bills Exp $
# $Log: power.tcl,v $
# Revision 1.5 2003/02/17 13:06:25 bills
# Added icroute_power_net.
#
# Revision 1.4 2002/08/21 06:29:03 bills
# Added missing return.
#
# Revision 1.3 2001/12/11 13:03:19 bills
# Added debug and -depth options.
#
# Revision 1.2 2001/05/25 16:32:23 bills
# Now check to make sure power/ground pad exists.
#
# Revision 1.1 2001/03/09 12:03:30 bills
# Initial revision
#
#
#
proc icroute_single_metal_grid {pwr gnd pwr_pad gnd_pad pwr_restrict gnd_restrict pwr_width gnd_width ar

    # First get possible arguments
    set debug false
    set route_depth 0
    set num_args [llength $args]
    for {set i 0} {$i <$num_args} {incr i} {
        set arg_el [lindex $args $i]
        if {$arg_el == "-debug" } {
            set debug true
        } elseif {$arg_el == "-depth" } {
            incr i
            if {$i <$num_args} {
                set route_depth [lindex $args $i]
            } else {
                icmessage errmsg ic_reroute_dv_errors "Missing argument to -depth\n"
                return ;
            }
        } else {
            icmessage errmsg ic_single_metal_grid "Unknown arguments: $args\n"
            icmessage errmsg null "Supported arguments:-debug, -depth \n"
            return ;
        }
    }

    if {$pwr_pad != ""} {
        icpin global $pwr_pad on
    }
    if {$gnd_pad != ""} {
        icpin global $gnd_pad on
    }
    icroutenet -depth 1 $gnd
    icroutenet -depth 1 $pwr
    if {$debug} {
        puts stderr "at first route..."
        icwait
    }

    set partial_pwr_routing [icnet routing $pwr -object]
    set partial_gnd_routing [icnet routing $gnd -object]
```


Itools Documentation

```
# puts stderr "partial power:$partial_pwr_routing"
# puts stderr "partial ground:$partial_gnd_routing"

icglobal $pwr off
if {$pwr_restrict != ""} {
    icnet layers $pwr $pwr_restrict
}
icnet width $pwr $pwr_width
if {"$pwr_pad" != ""} {
    icnet star $pwr $pwr_pad
    icroutenet -partial -source $pwr_pad $pwr -depth $route_depth
} else {
    icroutenet -partial $pwr -depth $route_depth
}

icglobal $gnd off
if {$gnd_restrict != ""} {
    icnet layers $gnd $gnd_restrict
}
icnet width $gnd $gnd_width
if {"$gnd_pad" != ""} {
    icnet star $gnd $gnd_pad
    icroutenet -partial -source $gnd_pad $gnd -depth $route_depth
} else {
    icroutenet -partial $gnd -depth $route_depth
}

if {$debug} {
    puts stderr "after large route..."
    icwait
}

set full_pwr_routing [icnet routing $pwr -object]
set full_gnd_routing [icnet routing $gnd -object]
# puts stderr "full power:$full_pwr_routing"
# puts stderr "full ground:$full_gnd_routing"

$full_pwr_routing -= $partial_pwr_routing
$full_gnd_routing -= $partial_gnd_routing
$partial_pwr_routing free
$partial_gnd_routing free

icnet prerouting $pwr -object $full_pwr_routing
icnet prerouting $gnd -object $full_gnd_routing
$full_pwr_routing free
$full_gnd_routing free

# With prerouting now we can allow some ripup of these nets.
# icnet ripup $pwr off
# icnet ripup $gnd off

# Restore the power/ground default widths so we can reroute in the cell
# area.
icnet width $gnd default
icnet width $pwr default

icglobal $pwr on
icglobal $gnd on

# Restore to original state.
if {$gnd_restrict != ""} {
    icnet layers $gnd all_layers
```

```

}
if {"$gnd_pad" != ""} {
    icnet star $gnd delete
}
if {"$pwr_restrict" != ""} {
    icnet layers $pwr all_layers
}
if {"$pwr_pad" != ""} {
    icnet star $pwr delete
}

icmessage imsg icroute_single_metal_grid "removing conflicts with power/ground grid...\n"
ic_reroute_dv_errors -conflicting_with_net $pwr -depth 1
ic_reroute_dv_errors -conflicting_with_net $gnd -depth 1
}

proc icadd_single_metal_mirrored_supply_grid {pwr gnd layer width start {fflag ""} } {
    # First some error checking.
    if {"$start" != "$pwr" && "$start" != "$gnd"} {
        icmessage errmsg icadd_single_metal_mirrored_supply_grid "start node must be either $pwr or $gnd\n"
        return
    }

    # First bottom row
    if {"$start" == "$pwr"} {
        set boun [icrow boundary 1]
        set row_l [lindex $boun 0]
        set row_b [lindex $boun 1]
        set row_r [lindex $boun 2]
        set row_t [lindex $boun 3]
        set pin [icpin create]
        $pin net $pwr
        $pin layer $layer
        $pin type POWER
        $pin rect "$row_l $row_b $row_r [expr $row_b+$width]"
        $pin end -primary $fflag
    } else {
        set boun [icrow boundary 1]
        set row_l [lindex $boun 0]
        set row_b [lindex $boun 1]
        set row_r [lindex $boun 2]
        set row_t [lindex $boun 3]
        set pin [icpin create]
        $pin net $gnd
        $pin layer $layer
        $pin type GROUND
        $pin rect "$row_l $row_b $row_r [expr $row_b+$width]"
        $pin end -primary $fflag
    }

    set numrows [icrow number]

    # Now the middle rows
    if {"$start" == "$pwr"} {
        set need_power false
    } else {
        set need_power true
    }
    set wid2x [expr 2 * $width]

    for {set row 1} {$row < $numrows} {incr row} {
        set boun [icrow boundary $row]

```

```

set row_l [lindex $boun 0]
set row_b [lindex $boun 1]
set row_r [lindex $boun 2]
set row_t [lindex $boun 3]
set pin [icpin create]
if {$need_power} {
    $pin net $pwr
    $pin layer $layer
    $pin type POWER
    $pin path -width $wid2x "$row_l $row_t $row_r $row_t"
    $pin end -primary $fflag
    set need_power false
} else {
    $pin net $gnd
    $pin layer $layer
    $pin type GROUND
    $pin path -width $wid2x "$row_l $row_t $row_r $row_t"
    $pin end -primary $fflag
    set need_power true
}
}

# Now add the top row
set boun [icrow boundary $numrows]
set row_l [lindex $boun 0]
set row_b [lindex $boun 1]
set row_r [lindex $boun 2]
set row_t [lindex $boun 3]

set pin [icpin create]
if {$need_power} {
    $pin net $pwr
    $pin layer $layer
    $pin type POWER
    $pin path -width $width "$row_l $row_t $row_r $row_t"
    $pin end -primary $fflag
} else {
    $pin net $gnd
    $pin layer $layer
    $pin type GROUND
    $pin path -width $width "$row_l $row_t $row_r $row_t"
    $pin end -primary $fflag
    set need_power true
}
return ;
}

proc icroute_row_bus {net type width layer {fflag ""} } {
    set numrows [icrow number]
    for {set row 1} {$row <= $numrows} {incr row} {
        set pin [icpin create]
        $pin net $net
        $pin layer $layer
        $pin type $type
        $pin instance [icrow name $row]
        set boun [icrow boundary $row]
        set row_l [lindex $boun 0]
        set row_b [lindex $boun 1]
        set row_r [lindex $boun 2]
        set row_t [lindex $boun 3]
        if {"$type" == "POWER"} {

```

```

    set top [icrow power $row]
} else {
    set top [icrow ground $row]
}
if {"$top" == "1"} {
    set pin_info [icpin detect -pintype $type -row $row -top]
    set pin_info [lindex $pin_info 0]
    set pin_data [lindex $pin_info 1]
    set pos [lindex $pin_data 3]
    $pin rect "$row_l [expr $pos - $width] $row_r $pos]"
} elseif {"$top" == "0"} {
    set pin_info [icpin detect -pintype $type -row $row -bottom]
    set pin_info [lindex $pin_info 0]
    set pin_data [lindex $pin_info 1]
    set pos [lindex $pin_data 1]
    $pin rect "$row_l $pos $row_r [expr $pos + $width]"
} else {
    puts stderr "ERROR:incorrect return from icrow $type"
}
$pin end -primary $fflag
}
}

proc icroute_power_net {pwr pwr_pad pwr_restrict pwr_width args} {

    # First get possible arguments
    set debug false
    set noripup false
    set route_depth 0
    set num_args [llength $args]
    for {set i 0} {$i < $num_args} {incr i} {
        set arg_el [lindex $args $i]
        if {$arg_el == "-debug" } {
            set debug true
        } elseif {$arg_el == "-noripup" } {
            set noripup true
        } elseif {$arg_el == "-depth" } {
            incr i
            if {$i < $num_args} {
                set route_depth [lindex $args $i]
            } else {
                icmessage errmsg ic_reroute_dv_errors "Missing argument to -depth\n"
                return ;
            }
        } else {
            icmessage errmsg ic_single_metal_grid "Unknown arguments: $args\n"
            icmessage errmsg null "Supported arguments:-debug, -depth \n"
            return ;
        }
    }

    if {$pwr_pad != ""} {
        icpin global $pwr_pad on
    }
    icroutenet -depth 1 $pwr
    if {$debug} {
        puts stderr "at first route..."
        icwait
    }

    set partial_pwr_routing [icnet routing $pwr -object]

```

```

icglobal $pwr off
if {$pwr_restrict != ""} {
    icnet layers $pwr $pwr_restrict
}
icnet width $pwr $pwr_width
if {"$pwr_pad" != ""} {
    icnet star $pwr $pwr_pad
    icroutenet -partial -source $pwr_pad $pwr -depth $route_depth
} else {
    icroutenet -partial $pwr -depth $route_depth
}

if {$debug} {
    puts stderr "after large route..."
    icwait
}

set full_pwr_routing [icnet routing $pwr -object]

$full_pwr_routing -= $partial_pwr_routing
$partial_pwr_routing free

icnet prerouting $pwr -object $full_pwr_routing
$full_pwr_routing free

# With prerouting now we can allow some ripup of these nets.
if {$noripup} {
    icnet ripup $pwr off
}

# Restore the power/ground default widths so we can reroute in the cell
# area.
icnet width $pwr default

icglobal $pwr on

# Restore to original state.
if {$pwr_restrict != ""} {
    icnet layers $pwr all_layers
}
if {"$pwr_pad" != ""} {
    icnet star $pwr delete
}

if {$debug} {
    puts stderr "before drc violation..."
    icwait
}

icmessage imsg icroute_single_metal_grid "removing conflicts with power/ground grid...\n"
ic_reroute_dv_errors -conflicting_with_net $pwr -depth 1
}

```

Iroute Commands

Iroute command line invocation:

iroute [options] *circuitname* The following command line options are supported:

-n | No graphics. By default **iroute** enters a graphical user interface.

-do script | Execute the Tcl *script* command file.

Like other iTool programs **iroute** is driven by a Tcl/Tk command interpreter. The commands can be supplied in the do command script, or typed interactively if the graphical user interface is used. The following commands are supported:

icregion_router icdr

iroute has the capability to support detailed routers from a number of vendors. At present only iTool's detailed router, **icdr**, is recommended.

icstyle otc

By default **iroute** will not route over standard cells. Setting the style to "otc" allows over-the-cell routing.

iroute regions | pads | region #

The "regions" option is used to route all channels between standard cell rows. The "pads" option is used to route from the core to the pads. The "region #" option is used to route a particular region number.

icexpand_region #

This command is used to expand the allowed routing space for particular routing regions. Normally these regions are only those outside the standard cell core. The standard cell core regions are normally comapacted and expanded automatically to use the minimum area. Regions outside the core are normally fixed and size changes must be explicit.

iroute_rect netname region layername xmin ymin xmax ymax

Adds a rectangle of geometry to the nominated region, or "pads" if it is part of pad routing. This geometry should be added before the region is routed. It is then treated as an obstacle by the detailed router when the region is routed.

icnumstdrows

Returns the number of rows in the standard cell core. This number can be used in Tcl scripts for power or clock pre-routes. For example the following is a simple power routing function **Please note that it is recommended that the power and ground routing be performed in the detail router itself and not in the routing control program.** This example is furnished for completeness and capabilities in the detail router have deprecated this method :

```
proc route_power {} {
    set numrows [icnumstdrows]
    puts "Power will be routed to $numrows rows"
    for {set i 1} {$i <= $numrows} {incr i} {
        set rowpos [icregion_info [expr $i*2]]
        set x [lindex $rowpos 2]
        set y [lindex $rowpos 3]
        icroute_rect hi pads metall [expr $x-50] [expr $y-5] [expr $x+150] $y
    }
    set rowpos [icregion_info 2]
    icroute_rect hi pads metall [expr $x+140] [expr [lindex $rowpos 3]-5] [expr $x+160] $y
}
```

Post-Routing Translation and Verification

Translating using EZ-CAD

After placement and/or routing is performed, it is necessary to return the design data to users choice of netlist format. It is important that the user understand the [data dependencies of CAD systems](#). The translator separates the physical representation from the netlist representation. Hence, different output formats are possible. You must select a desired physical implementation. The netlist output is optional. However, Cadence users will want to pick both LEF and DEF. Use the pulldown menu to set both output formats. When you change translation modes, please wait while EZ enables the correct instructions for the chosen mode.

The tutorial has set all of the buttons for you. You only need to generate the translation script and execute the translation.

Physical Format:	LEF
Netlist Format:	DEF

Please use the control below to locate the Cadence LEF file. The LEF file contains the technology and a physical description of the standard cells, macro cells, and I/O cells which make up the design. The file does not need to exist.

Select the LEF File:		Browse ...
/Users/bill/.itools/tutorials/gridded/stdcell_final.lef		

Now use the control below to locate the Cadence DEF file. The DEF file contains the netlist description, design constraints, and prerouting. The file does not need to exist.

Select the DEF File:		Browse ...
/Users/bill/.itools/tutorials/gridded/stdcell_final.def		

Generating a Translation Script

Now we have all of the information necessary to generate a translate script to control the translation process.

Generate Translate Script	Edit Translate Script
---------------------------	-----------------------

Now we can execute the translation program with the script that we just



generated.

It is possible to view the generated LEF/DEF with the iTools translator. Click below to open the viewer to the final output.

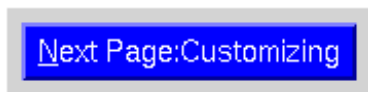


Generating a Grand.do Script

Now that we have gone through the itools flow, iTools can [create a *grand.do* script](#) which will allow you to rerun the entire process without using EZ. You can even run the entire process in batch mode without graphics using the script. This is only recommended for more advanced users.

Verification

Under constructions



=>

Creating a Grand Do Script

Generating the Script

Now we have all of the information necessary to generate a master script or *grand.do* file which we will direct the *ittranslate* program to build the necessary input files and call the appropriate programs to rerun this design without using EZ. This allows advanced users to run the design in batch mode without graphics.

If the user supplied a custom input translation script (which wasn't generated by EZ), you should enter its path here. Otherwise, if left blank, EZ will attempt to regenerate a translation script for you. The program will use the parameters entered in the *Inputs/Translation* section.

Select the Input Translation Script:

Browse ...

Use the button to regenerate the *region.ddo*, *entire.ddo*, and *circuitName.rdo* files if desired. Normally, the scripts already exist after an execution of EZ so it is unnecessary to regenerate them.

Regenerate Scripts:

☐ yes
 ☐ no

Use the button below to generate the *grand.do* file

Generate Grand.do Script

You can look at the generated *grand.dofile* or your custom script using the editor of your choice.

Grand.do File

☒ tk
 ☐ vi
 ☐ emacs

Mode: ☒ Read-Only

Edit File

Now that we have generated a *grand.do* file, we can exit EZ and enter a command shell of your choice. After sourcing the appropriate iTools initialization file (*.icrc*, *.icrc.sh*, *.icrc.ksh* for the CShell, Bourne Shell and Korn Shell respectively), the user needs to enter the following command to run the translator

- `ittranslate -v -do grand.do <designName>`

How Does it Work, and How Do I Change it?


Customizing the Itools Tool Suite

This section describes how to change the behavior of **itools**.


Adding Hot Keys

When a **itools** graphical program is initialized, the program will execute the Tcl script in `~/itools/hotkeys.ini` if it exists. The user is free to add Tcl/Tk binding for their personal use. The full power of TclTk is available at your disposal. Click here to see an example of the [hotkeys.ini file](#).

1. **Tcl** scripts. Many **itools** tools read Tcl scripts to specify and control complex inputs. For example the translation tools can be completely customised by changing the translation script.

As a convenience to the user, the hot key definitions may be manipulated using the hot key editor by clicking the mouse button on the hot key icon  found in the set of itools icons. You will be presented by a dialog which looks similar to:


Defined Hotkeys:

Hot key	Function
<Control-Key-i>	icinvoke_menu Draw {Tile Info}
<Control-Key-z>	iczoom in
<Key-Return>	ickey \$icinputG
<Shift-Key-Z>	iczoom out
 f	icfullview
z	iczoom

Focus: ☐ on click ☒ on entry

Hot Key:

Function:

With this dialog the user can add, delete, and edit hot key definitions. The hot key is defined as a Tcl event. Please see the Tk  man page for a discussion about available Tk events or if your in EZ just click the button. The hot key editor also allows the user to select the focus model.

Graphics Customization

There are a number of X resources that may be specified in the `~/.Xdefaults` or `~/.Xresources` file. The table below shows the parameters which affect the program's graphical display. The most frequently used parameters are **font**, **geometry**, and **reverse**. The **logo** option allows the user to skip the drawing of the itools logo during graphics initialization when the program announces its name. This is useful if you are on a very slow network connection and want to see only minimal graphics output. The use of the **stipple** command is deprecated. You should not need to use the **bw** or **three_button_mouse** parameters unless you are using ancient hardware. The **timeout** option controls the amount of time the program will wait for the graphics to appear on the screen. Normally, graphics should appear immediately but on slow networks the program must wait for the graphics to be *mapped* or appear on the screen. The timeout is in seconds. If the graphics fails to appear on the screen after the timeout you will see the following error message: **ERROR[ICinitGraphics]: Cannot find window for symbolic name: .frame5.frame7.frame1.canvas2.** Either increase the timeout or determine the cause of the slow response. The default timeout should be sufficient in all cases.


Parameter	Function	Default	Example
itools*bw:	black and white display	off	on
itools*font:	graphics font	fixed	-* <i>-courier</i> -* <i>-r</i> -* <i>-*</i> -l4-* <i>-*</i> -* <i>-*</i> -* <i>-*</i> -*
itools*geometry:	controls window geometry	window manager default	650x725+1+20
itools*logo:	controls presentation of itools logo	on	off
itools*reverse:	reverse video	off	on
itools*stipple:	stipple mode	off	on
itools*sync:	X debuggin mode	off	on
itools*three_button_mouse:	enables three button mouse	on	off
itools*timeout:	graphics timeout	120	20

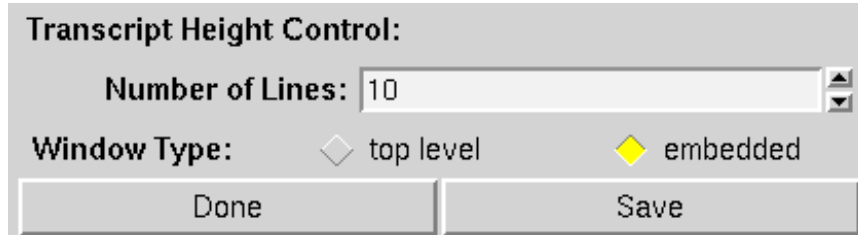
These settings can all be changed using the `xrdb` program found with your X server. For example, to turn off the logo you would enter:

- `echo "itools*logo : off" | xrdb -nocpp`
on the command line. The also commonly stored in your `~/.Xdefaults` or `~/.Xresources` file. See your X man page for more details.

Transcript Window Customization

The placement and height of the transcript window can be adjusted either graphically through the transcript height

icon  found on icon bar or textually using the `~/tools/transcript.ini` Tcl script. If you single click on the transcript height icon using the mouse, you will be presented with the following dialog box:

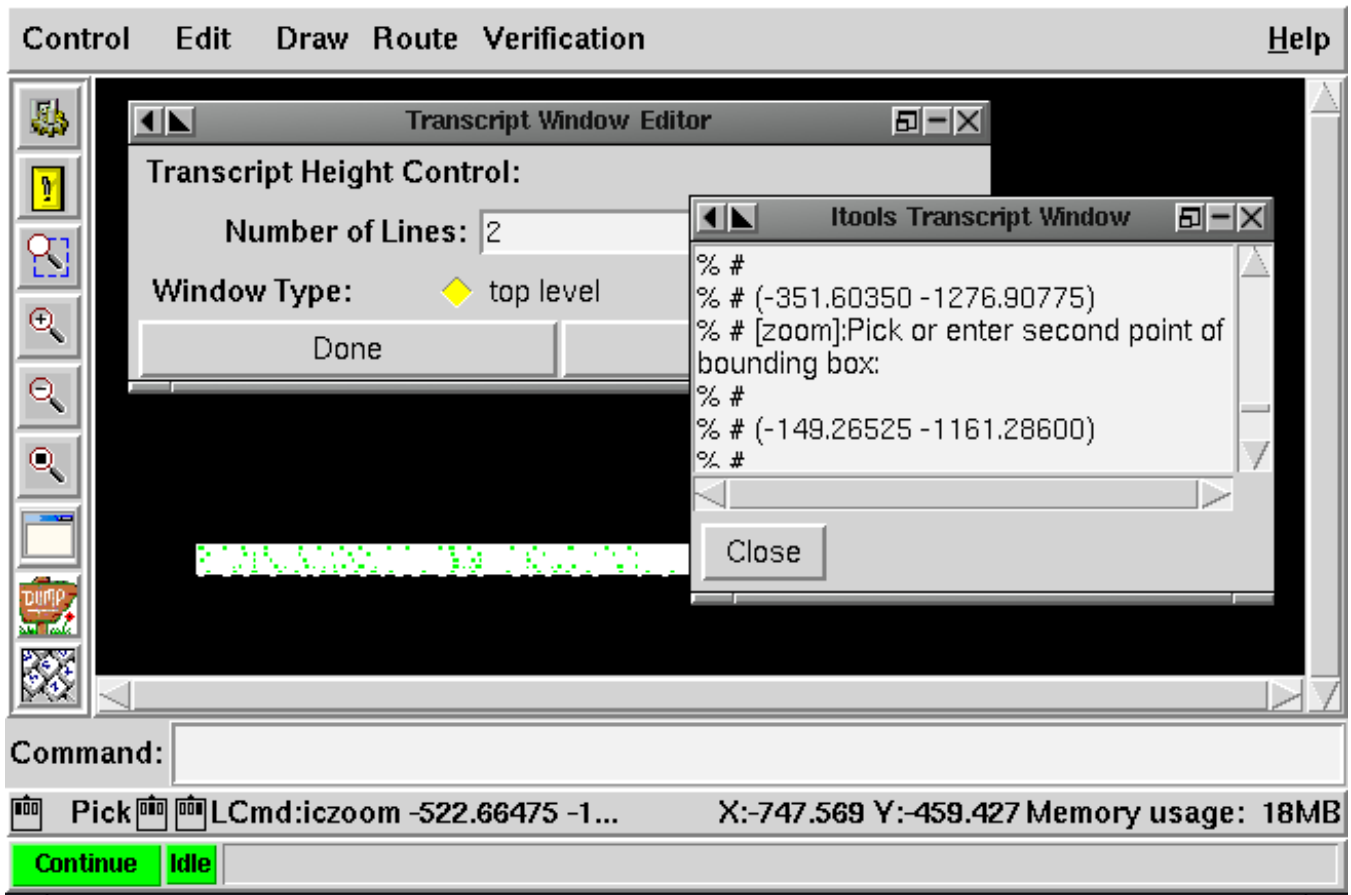


Transcript Height Control:

Number of Lines:

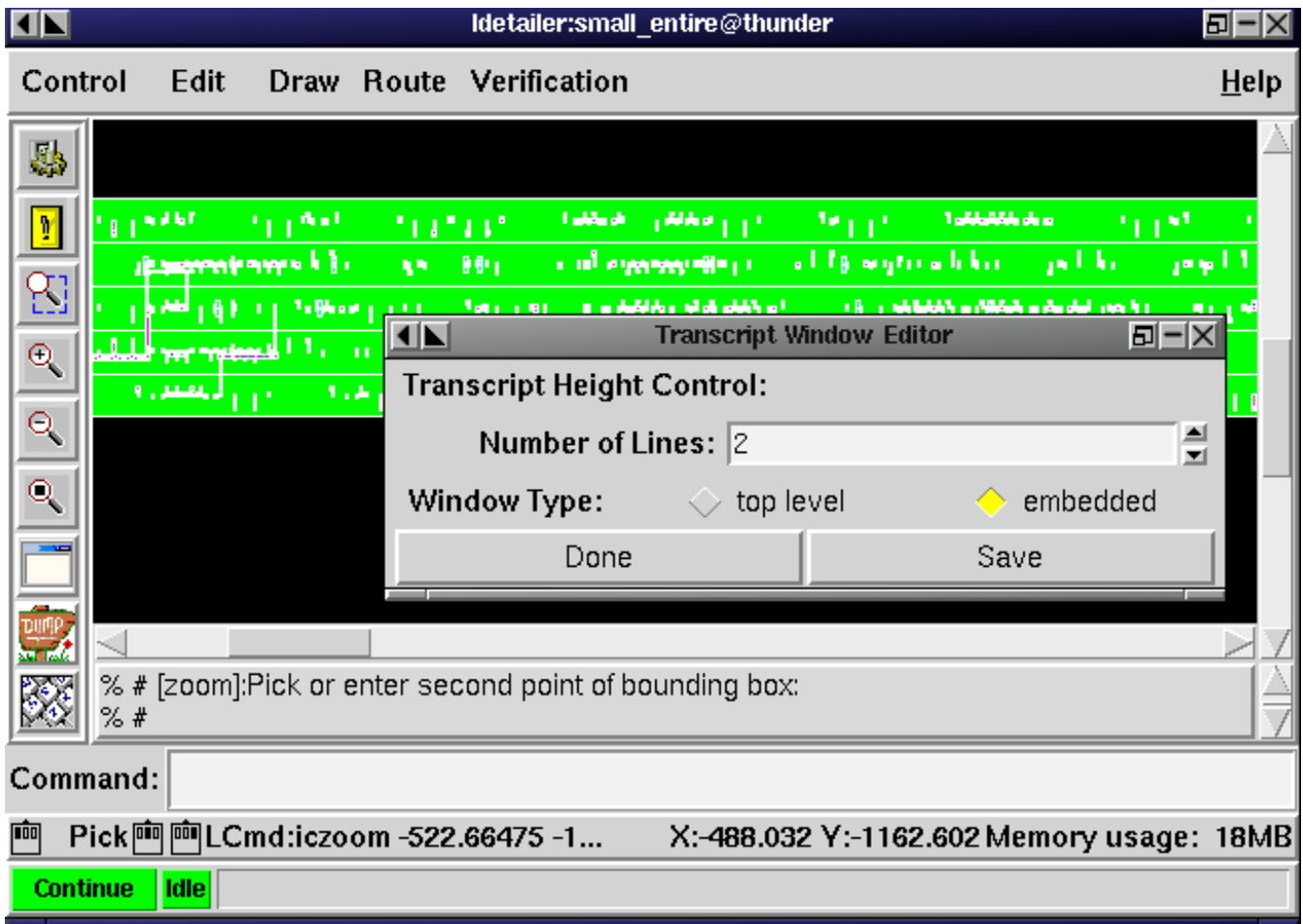
Window Type: ☐ top level ☒ embedded

This dialog box presents you with two choices : the number of lines to be displayed in the transcript window and the window type. All program output is displayed in the transcript window. The transcript window has two modes : *toplevel* and *embedded*. If you choose a top level window, a second window (known as a *top level window*) will be created as shown:



The top level window transcript window has **Close** button which will revert the layout to the default embedded layout as shown below. In the top level mode, the number of lines requested are ignored. When the embedded layout is requested, the use may vary the number of lines of text displayed in the transcript window. In this example, the

number of displayed lines is set to two.



Notice that the transcript height dialog box also has a **Save** button. This will save the current settings to the `~/itools/transcript.ini` file. Upon graphics initialization, itools graphical programs will search for this file and execute it if it exists.

An alternative method is to enter the Tcl command `::ictranscript::configure <numlines> <0 | 1>` where 0 invokes the embedded layout and 1 invokes the top level layout. For example, the command `::ictranscript::configure 2 0` will create the embedded layout as shown above.

Security Issues

The **itools** plugin for Netscape Navigator contains a general-purpose Tcl interpreter. This interpreter is used to configure and control activation of the **itools** tools. It needs to have complete ability to create and delete files on the user's system. This ability can be invoked from any web page loaded while the **itools** plugin is active.

There are two security issues to resolve. The first is to prevent outside web pages from starting the **itools** plugin, then using it to modify users files. For this reason the plugin can only be started from a local file, not from a remote page loaded by the `http:` protocol. This prevents outsiders from starting the plugin and use it to modify your file system.

The second issue is to ensure that the plugin is removed after completion of its use for EZ-CAD. Normally Netscape Navigator will remove the plugin after the last page referencing it is unloaded. However, EZ-CAD users are advised to exit from Netscape Navigator after using it for EZ-CAD. Browsing of pages outside your control is not

recommended while the EZ-CAD plugin is loaded.

Files

Files used in EZ-CAD:

~/itools/ez.ini Contains initializations for Tcl variables used by EZ-CAD applications. This file is automatically written when EZ-CAD variables are changed so that the Tcl state of the plugin should be restored after a restart.

icsearch.js This file is dynamically modified by the Search facility in EZ-CAD. It contains pointers to the pages containing search keywords. This file is included in the search page.

[Next Page: Tips/FAQ](#)



Itools Documentation

```
# A Sample itools graphics initialization file.

# One can set up hot keys and even make them conditional based on a program.
set pname [icprogram]

bind all <CONTROL-F> {icfullview}
bind all <CONTROL-Z> {iczoom}

if {$pname == "genrows"} {
    bind all <CONTROL-T> {icmenu edit-tile}
}
```

Problems with itools

Sending Design Data

Invariably problems do arise in **itools** due to the complex nature of the problem and the algorithm solutions that are applied. Rather than deny the problems exist, we at **itools** want to minimize the impact of any problem that may occur.

First we must gather the design information and get it ready for shipment. Use the buttons below to bundle the design directory. You have two options. You may either send the minimum set of design data : (*design.ckt*, *design.con*, *design.lib*, *design.log*, *design.par*) or send the entire directory.

Gather design (mult_test) data in /Users/bill/itools/tutorials/gridded for shipment

Status:



Make sure you have permission to send the design data. InternetCAD, Inc. is willing to sign any necessary NDA (non-disclosure agreements).

Now send the information to InternetCAD, Inc. via email.

Send information to InternetCAD.com, Inc.

Status:

Frequently Asked Questions

Problem: What is the itools root directory?

Solution: The itools root directory is the top of the itools tree. The pathname will normally contain the version number of the itools program. For example,

```
/Users/bill/programs/itools.2.0.0
```

A directory listing of the root directory should look similar to:

```
>>>ls -a
./      defaults/  EZ/        .icrc      lib/       tcl/
../     DISCLAIMER flows/     .icrc.ksh  license/   test/
bin/    doc@       htmldoc/   .icrc.sh   README    WHATSNEW
```


Notice the presence of the itools initialization scripts: *.icrc*, *.icrc.ksh*, and *.icrc.sh*

Problem: How do I initialize itools?

Solution: Ittools depends on the existence of the following environment variables: **ICDIR ICOS DISPLAY** and the appropriate dynamic library path environment variable (**LD_LIBRARY_PATH, LIBPATH, DYLD_LIBRARY_PATH, or SHLIB_PATH** depending on the Unix system). Ittools supplies a scripts for initializing these variable properly based on operating system and shell type. Currently, three shells are supported: (Bourne/Bash shell, C-shell and Korn Shell). Here is how to use each of these script. While in the [itools root directory](#) type

```
. ./icrc.sh
```

for a Bash or Bourne shell

```
source .icrc
```

for a C-shell or

```
. ./icrc.ksh
```

for a Korn shell. Of course, the user can assign the variable in their profile or initialization scripts. The itools scripts are furnished as a convenience. **It is imperative that these variable be set to the proper values; otherwise, itools will not work properly.**

Problem: I tried to execute the Solaris version of **itools** but I am missing the libucb.so.1 library. Where do I find it? I got the following:

```
itools -v test
ld.so.1: itools: fatal: libucb.so.1: can't open file: errno=2
Killed
```

Solution: You need to add /usr/ucblib to your LD_LIBRARY search path. Use

```
setenv LD_LIBRARY_PATH /usr/ucblib
```

or the appropriate syntax for your current working shell. **Note: this problem has been entirely eliminated in versions 1.3.4 and later.**

Problem: I pulled the dynamically linked version of **itools**. When I try to run it I get the following:

```
itools
/lib/dld.sl: Can't open shared library: /home/itools/version1/itools/lib/HP-UX/libstdmem.sl
/lib/dld.sl: No such file or directory
Abort(coredump)
```

Solution: You need to set the LD_LIBRARY path environment variable to include \$ICDIR/lib/\$ICOS where \$ICDIR is the **itools** root directory and ICOS is the current operating system. On HP-UX, this variable is SHLIB_PATH. The *.icrc*, *.icrc.sh*, and *.icrc.ksh* initialization scripts found in the root directory perform this function for you when in a C shell, Bourne shell, and Korn shell respectively. Look in these files to see how it works.

Question: How do you start the license server?

Answer: The license server should start automatically but may be started [manually](#).

Question: How do you generate timing constraints from Synopsys?

Answer: Use the Synopsys command

```
write_constraints -format sdf-v2.1 -cover_nets -output "xxx.sdf"
```

or

```
write_constraints -format sdf-v2.1 -cover_design -output "xxx.sdf"
```

The write_timing command which also generates sdf does not produce the proper constraints.

From the synopsys documentation for write_constraints:

If the `-cover_design` option is specified, then just enough paths will be generated so that the worst path through every driver-load pin pair is covered. The `-from`, `-to`, `-through`, and `-nworst` options are ignored when using this option. Constraint generation is then based on this 'covering' path set. The `-max_paths` option should not typically be used with `-cover_design`, because the point of the `-cover_design` option is to obtain a path set that covers the entire design. The `-cover_nets` option is similar to `-cover_design` in that it attempts to cover the entire design, but it only generates the worst path for each net instead of each driver-load pin pair. This is most useful for layout tools which are going to convert the path based constraints into net based constraints.

Question: What is the best way to design row-based cells?

Answer: The [best designed cells](#) are very porous cells which are gridded.

Question: How do you use the graph-prerouter of the detail router?

Answer: The [graph-based prerouter](#) expects the ports of row-based designs to occur on a vertical layer within the center of the row.

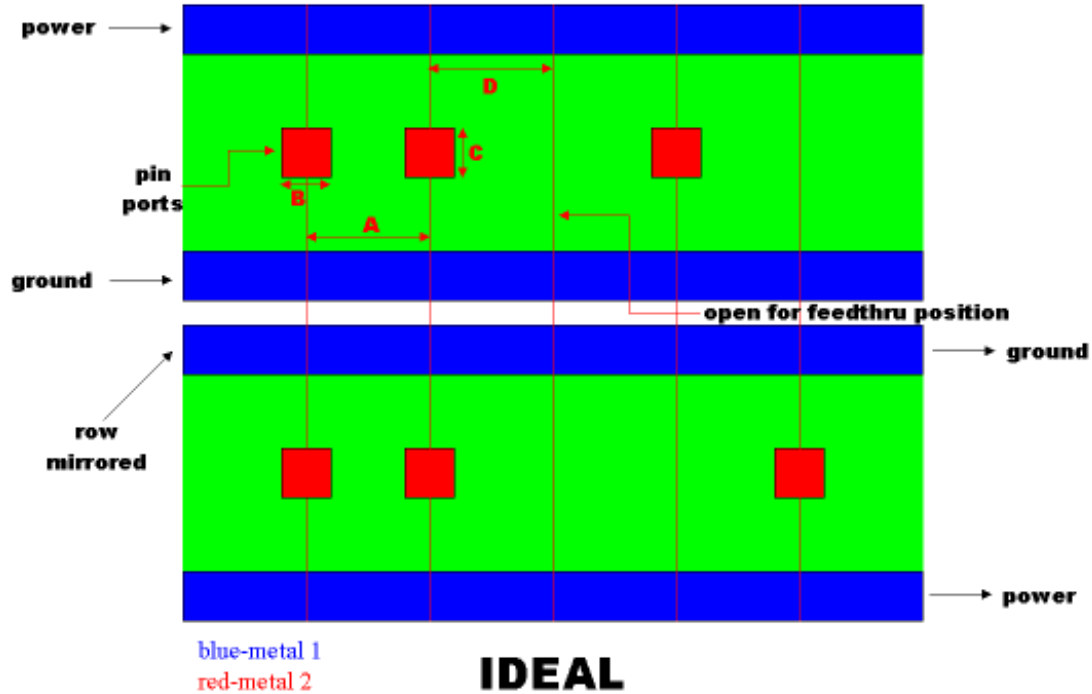
Question: Does itools support critical nets?

Answer: Yes, itools support [critical nets](#) in all phases of physical design – placement, global routing, and detail routing.

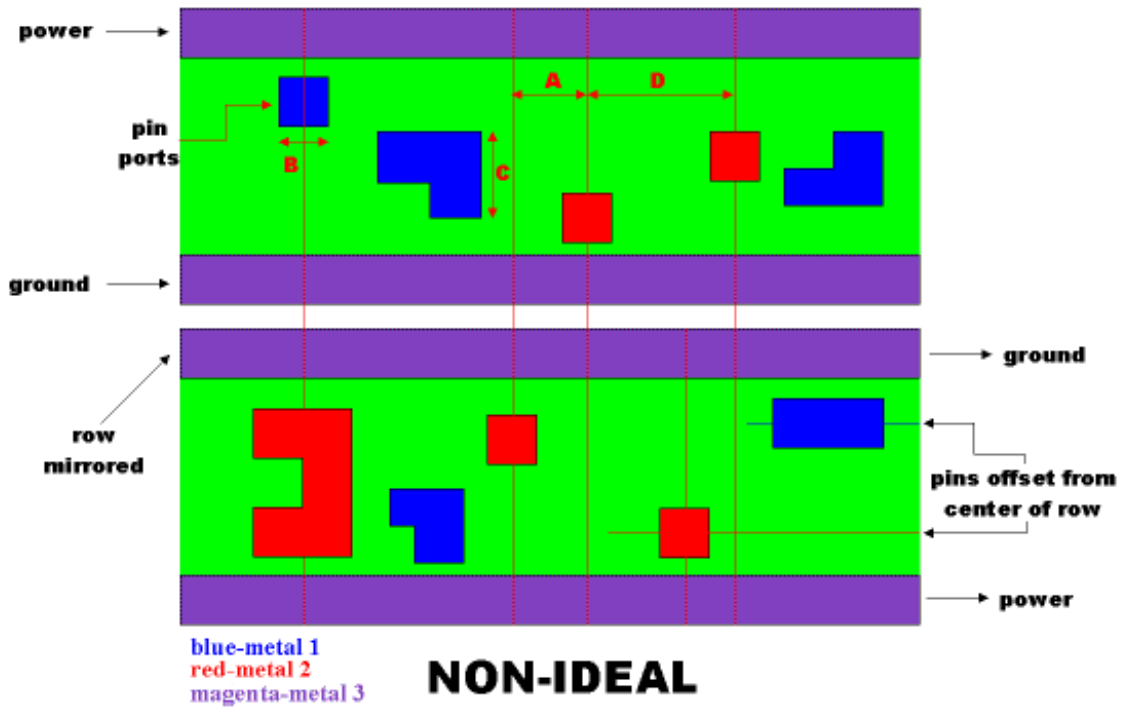
Question: What are the ECO or Engineering Change Order capabilities of itools?

Answer: Itools supports [Engineering Change Orders](#) in placement, global routing, and detail routing.

Designing a standard cell library



The picture above shows a row-based design which is ideal for automatic placement and routing. From the picture, it can be seen that the library is gridded ($A=D$), and that the ports are on the vertical layer and centered in the y direction. Each port is a square and $B=C$ =minimum wire width of the vertical layer. Notice that every other row is mirrored about the x-axis so that power and ground can be shared. Having all of the ports at the center of the row allows maximum decomposition of the routing; channels become independent and can be routed in parallel. Since power and ground are on the bottommost horizontal layer, all upper horizontal layers (layer 3 and on) are available for over the cell routing with no constraints. Any increase in area due to the power and ground on metal 1 is more than compensated by easier detailed routing and an overall area reduction is obtained. It is also extremely important that the cells be porous. The empty grid positions allow routing in the vertical direction and are necessary for crossing the rows. Such designs are easily optimized because the place and global router is aware of the routing resources, that is, none of the resources are ambiguous.

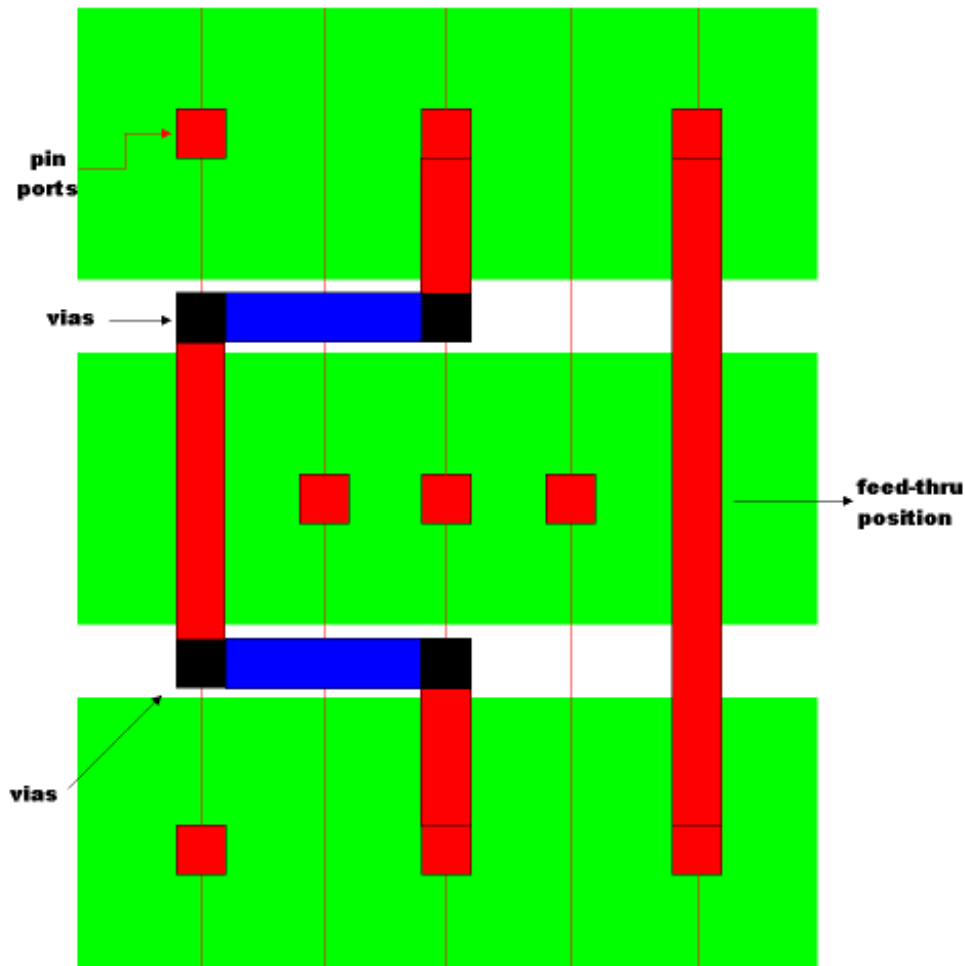


The second picture above shows a typical row-based design. It is far from ideal with respect to placement and routing. From the picture, it can be seen that the library is not gridded ($A \neq D$), and that the pin ports are on both horizontal and vertical layers. In addition, ports may assume any rectilinear shape in general ($B \neq C$). The ports occur at any y coordinate in the cell which impacts the decomposition of the routing; dependencies may exist between different channel. Since power and ground are on the third metal layer, metal 3 now has constraints which limits the amount of detailed routing optimization. This style does mirrored the rows about the x-axis so that power and ground can be shared.

Ittools will route this design by converting it as close as possible to the ideal case. **Idetailer** is a non-gridded router so gridding is not a concern. However, moving the pin ports to the center of the cell and to the proper vertical layer does help the decomposition of the routing.

Vertical Resources Matter

Now let us take a closer look on why vertical resources are extremely important in obtain high density layouts. The picture below shows a portion of a row-based design. We have omitted the power and ground rail for clarity. Here we see a single net which needs to connect from the bottommost row (row 1) to the topmost row (row 3). We assume a 3-layer design: metal 1 and metal 3 run horizontally and metal 2 runs vertically. Notice that an obstacle on metal 2 blocks the path of the route and a detour is necessary.

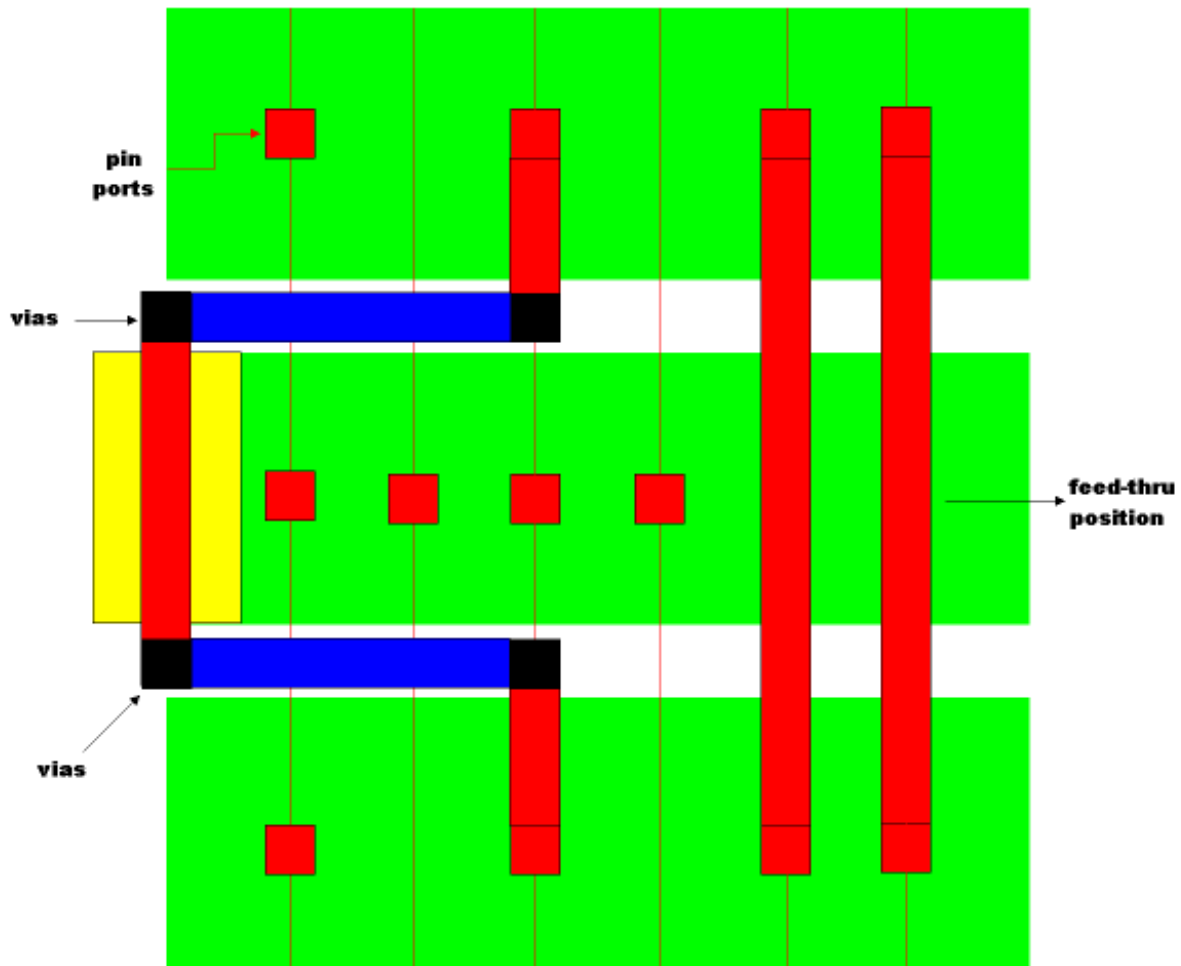


Effects of Blockages on Routing

blue-metal 1

red-metal 2

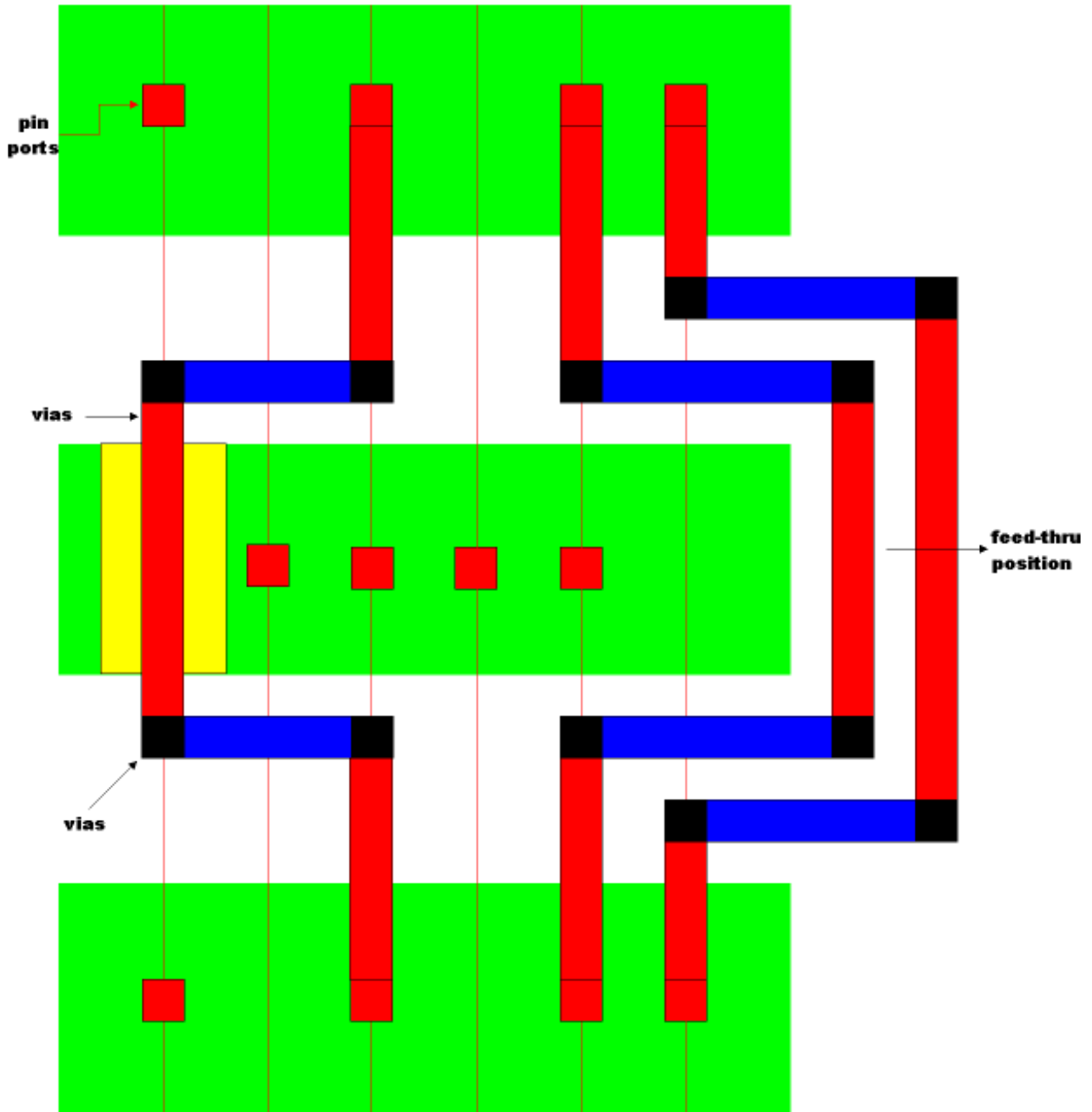
While the previous case seems bad enough, the situation worsens if a route can not find a place to cross over the cell as in the figure below. Here many vertical obstacles prevented the route from crossing over the row. In order to complete the route, a gap or feedthru cell was inserted to allow it to complete the route. However, feedthru cells can only be inserted between the individual cells which comprise the row. This increases the width of the row by one vertical pitch. If the row is the longest row, the area of the chip increases.



Adding an Explicit Feed-Thru

blue-metal 1
red-metal 2

In the picture above, the feedthru cell was depicted as being at the end of the row. However, in general, it may occur anyway in the row. The picture below shows the ramifications of adding feedthru in a design which is deficient in vertical resources. The addition of the feedthru on the left requires a shift to the right of all the cells in the row. Notice that the routes that were aligned vertically are now all adversely affected. Now these routes as well must be rerouted around obstacles. As you can see, when resources are tight each detour may result in two additional horizontal tracks and one vertical track. This can result in dramatic increases in area. Designers have traditionally been neglect in providing sufficient vertical resources. Providing adequate vertical resources is a prerequisite to excellent placement and routing. This requirement is independent of the placement and routing tools used and due to the layout geometry.



blue-metal 1
red-metal 2

**Lack of Vertical Resources
Can Result in Huge Area Increase**

Critical Nets

Itools supports the concept of *critical* nets. Critical nets are nets which have some attribute such as wire length which must be minimized for the proper operation of the circuit. One must be careful in defining critical nets as the relative weight between the nets can radically change the behaviour of the place and route algorithms leading to radically different circuits. We will look at the various options in each of the stages of physical design.

Placement

In placement, there are several ways of defining critical nets by entering constraints in the *circuitName.con* file. The first and recommended way is to define either [timing constraints](#) for the net. These constraints have the form:

- **PATH** net1 [net2]... : lowerBound upperBound [**MONITOR**]
- **PATHCONSTRAINT** {string... | (string string string string)...} (float:float:float) (float:float:float) [**MONITOR**]
- **PATHCONSTRAINT [PATHPINPAIR]** {string... | (string string string string)...} (float:float:float) (float:float:float) [**MONITOR**]

These constraints are very robust and very forgiving of inconsistencies.

The second method of defining critical nets is the use of net weighting. Due to customer insistence, we allow the user to enter net weights. However, we generally frown on its use because the final quality of the result is very dependent on the proper relative choice of the weights and choosing these weights is a very difficult task. The format of the net weight constraint is as follows:

- **NET** netName **WEIGHT** float
- **NET** netName **WEIGHT** float float

In the first form of the command, the specified net will be weighted in both the x and y directions by the value following the **WEIGHT** keyword. In the second form, the user may control the x and y weight factors independently. The x weighting factor is followed by the y weighting factor. Weights must be non-negative. If a net is not specified, the net weight is assumed to be the default value of 1.0.

Global Routing

The global router supports the constraints delineated in the placement section above in addition to the net [PRIORITY](#) constraint. This enables the user to set the routing order so nets which have higher priority are routed first. Critical nets prioritized in this manner will have access to routing resources before other nets are routed which will result in the critical nets have less wirelength and feedthrus.

- **NET** netName **PRIORITY** integer

Detail Routing

The detail router also supports the net [PRIORITY](#) constraint described above. The constraints described in the placement section do not come into play here as the detail router does not change the placement. However, net ordering is important in detail routing and net priority is supported thru the **iccritical** Tcl command and namespace. The **iccritical** Tcl namespace can be found described in [\\$ICDIR/tcl/routing/critical.tcl](#). This namespace references the **iccritical** Tcl command performs the definition and enumeration of the prioritized nets. Returning to our priority example, we have define three nets with priorities.

NET A PRIORITY 2

NET B PRIORITY 2

NET C PRIORITY 1

We may manipulate the priorities with the **iccritical** Tcl command in a routing script. With the net's priorities defined as above in the *circuitName.con* file, the **iccritical** command entered into the idetailer command window would return:

```
>iccritical
A B C
```

In the case, the nets would be routed in order A, B, C. We could also achieve the same result by typing

```
>iccritical A B C
A B C
```

It is important to remember that critical nets are always routed first during a **icbeautify** command. You can achieve further control by using the commands of the **::iccritical** namespace. These commands work only on the critical nets. All non-critical nets are ignored. This is useful at the beginning of a routing script to route only the critical nets. For example, the following Tcl script sequence will define a set of critical nets and path length limits for them as well. Only these nets of the design will be routed.

```
set class1 {A B C}
iccritical $class1
foreach net $class1 {
    icnet pathlimit $net {{METAL1 50t} {METAL2 100t}}
}
iccritical::beautify
iccritical::route_unconnected 1
iccritical::route_unconnected 2
iccritical::route_unconnected 3
iccritical::strong_beautify
iccritical::status
```

Here we put nets A, B, and C as the critical nets. We define the variable **class1** as the Tcl list {A B C}. We then set a path limit on each of these nets as an example of other constraints that may be added as well. The command **iccritical::beautify** routes the nets in the order specified : A, B, C. The **iccritical::route_unconnected** commands will route any unconnected nets at the specified depth with the constraint of maintaining the routing order {A B C}. The **iccritical::strong_beautify** command routes only the critical nets but in reverse order and allows each net to ripup previously routed nets. In essence, the last net routed has the highest priority and is able to ripup all other nets. The **iccritical::status** command returns the connection status of just the critical nets defined.


```
#!/usr/local/bin/wish -f
# Program: idetailer
# $Id: critical.tcl,v 1.3 2005/10/06 07:52:44 bills Exp $
# $Log: critical.tcl,v $
# Revision 1.3 2005/10/06 07:52:44 bills
# Added exists and lock_no_ripup commands in the namespace.
#
# Revision 1.2 2002/11/09 03:02:51 bills
# Added strong_beautify and lreverse commands.
#
# Revision 1.1 2002/11/08 18:21:21 bills
# Initial revision
#
#
proc lreverse {_list} {
    upvar $_list the_list
    for {set i [expr {[llength $the_list] - 1}]} {$i >= 0} {incr i -1} {
        lappend reverse_list [lindex $the_list $i]
    }
    set the_list $reverse_list
    unset reverse_list
}

namespace eval iccritical {
    proc define { net_list } {
        iccritical $netlist
    }

    proc exists { } {
        set critical [iccritical]
        if {$critical != ""} {
            return true
        } else {
            return false
        }
    }

    proc status { } {
        set first 1
        set net_list [iccritical]
        foreach net $net_list {
            set unc [icnet unconnected $net]
            if {$unc > 0} {
                if {$first} {
                    icmessage "Unconnected nets:"
                    set first 0
                }
                icmessage " $net"
            }
        }
        if {$first} {
            icmessage "All nets connected\n"
        } else {
            icmessage "\n"
        }
    }

    proc beautify { } {
        icbeautify -critical_only
    }

    proc strong_beautify { } {
```

```

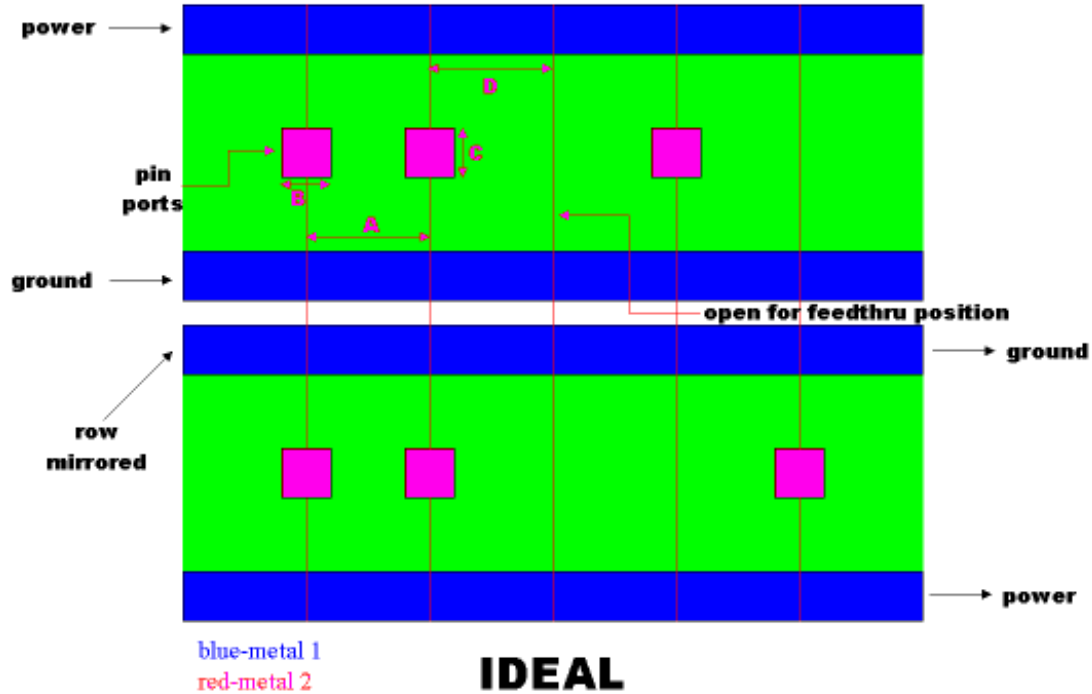
set critnets [iccritical]
lreverse critnets
foreach net $critnets {
    icmessage "routing critical net:$net...\n"
    icroutenet $net -depth 1
}
}

proc route_unconnected { depth } {
    set net_list [iccritical]
    foreach net $net_list {
        set unc [icnet unconnected $net]
        if {$unc > 0} {
            icmessage "routing net:$net at depth:$depth...\n"
            icroutenet $net -depth $depth
        }
    }
}

proc lock_no_ripup { } {
    set critical [iccritical]
    if {$critical != ""} {
        foreach net $critical {
            if {[iccritical noripup $net]} {
                # See if net is routed.
                set num_unc [icnet unconnected $net -native]
                if {$num_unc == 0} {
                    icnet ripup $net off
                    icmessage imsg lock_no_ripup "locking critical net:$net...\n"
                } else {
                    icmessage errmsg lock_no_ripup "can't lock critical net:$net as $num_unc pins aren't connected"
                }
            }
        }
    }
}
}
}
}

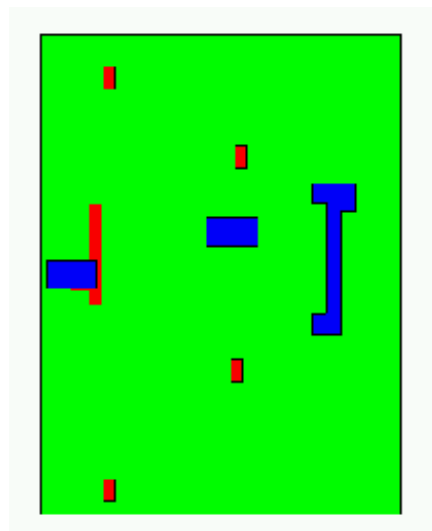
```

Graph-Based Prerouting

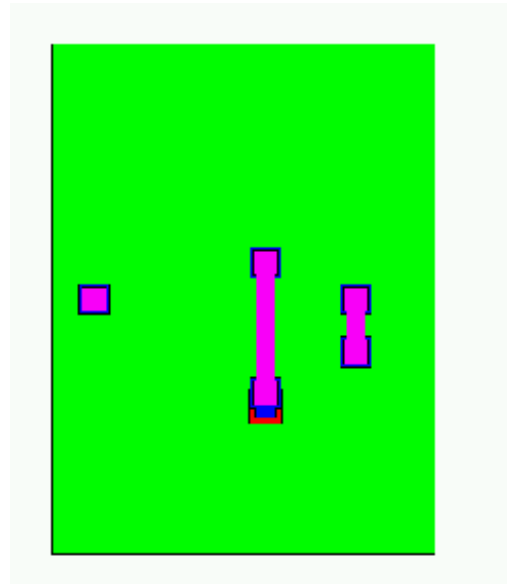


The picture above shows a row-based design which is ideal for the graph-based prerouting phase of the detail router. From the picture, it can be seen that the library ports are on the vertical layer and centered in the y direction. Each port is a square and $B=C$ =minimum wire width of the vertical layer. Having all of the ports at the center of the row allows maximum decomposition of the routing; channels become independent and can be routed in parallel.

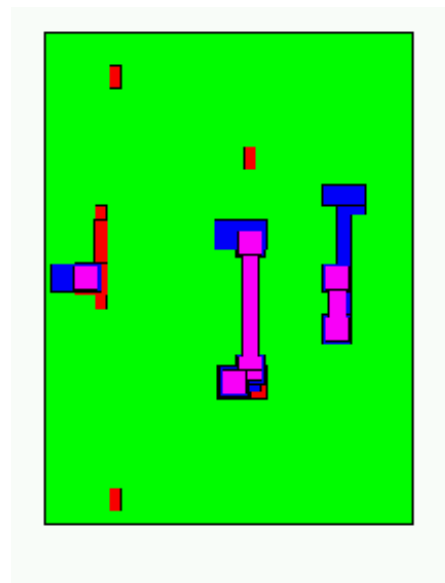
However, not all libraries are ideal. The picture below shows a row-based design which is not ideal for the graph-based prerouter. In the picture, we see that the ports are on poly and metal1. While poly is a vertical layer, most of the poly ports are far from the center of the cell. None of the ports are on the desired layer – metal 2.



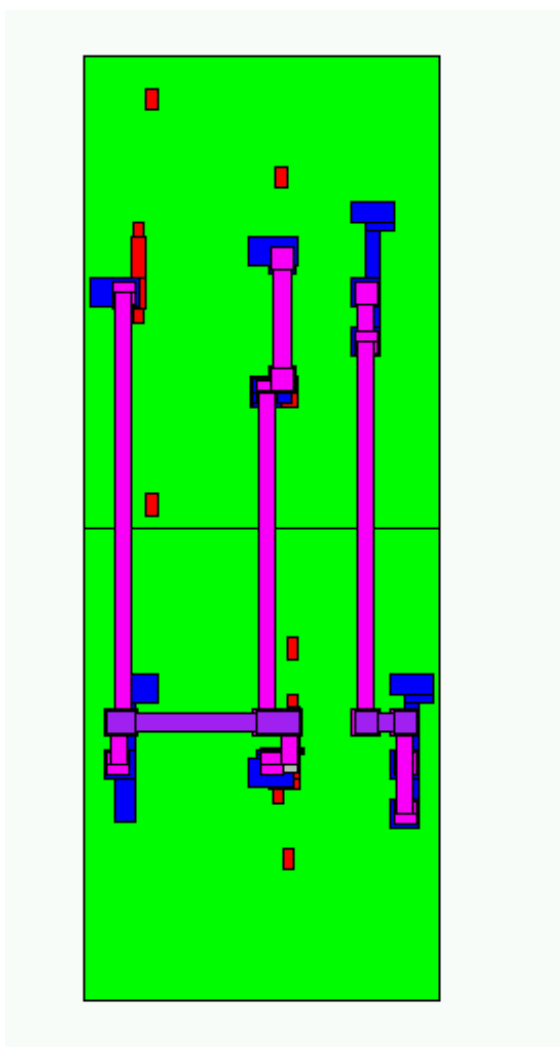
The solution is to create a cover cell which contains routing to bring the ports to the desired vertical layer. This can be accomplished thru the use of the **iccreate_abstractions** command of the translator. For the above example, this results in the following cover cell:



And so the detail router is presented the sum of the origin library cell and the constructed cover cell. The cover cell routing is treated as a suggestion to the detail router; it allows the use of the graph-based prerouter while it allows the maze router to remove unnecessary routing. .



Below we see the output of the graph-based prerouter. We see that the prerouter is a strict Manhattan router; vertical segments exist only on vertical layers whereas horizontal segments exist only on horizontal layers. Notice that the router ignores the ports on the poly and metal1 layers.



But the detail router is not constrained to only use the metal 2 ports. During the execution of the maze router, any port may be selected and the unused ports of the cover cell are deleted. Notice that the net on the left utilizes the closer poly ports whereas the net on the right uses the cover cell ports.

