Chapter 1

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 on strictly 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 less likely that a single chip will intersect that defect. 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.

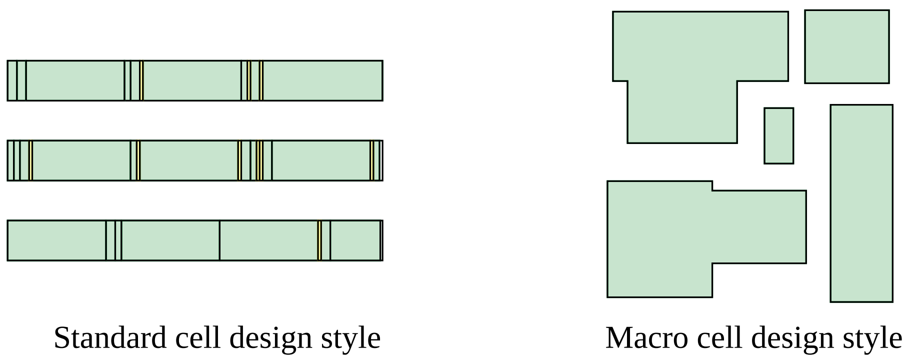


Figure 1.1 Standard cell and macro cell methodologies.

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*), has 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.

1.1 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

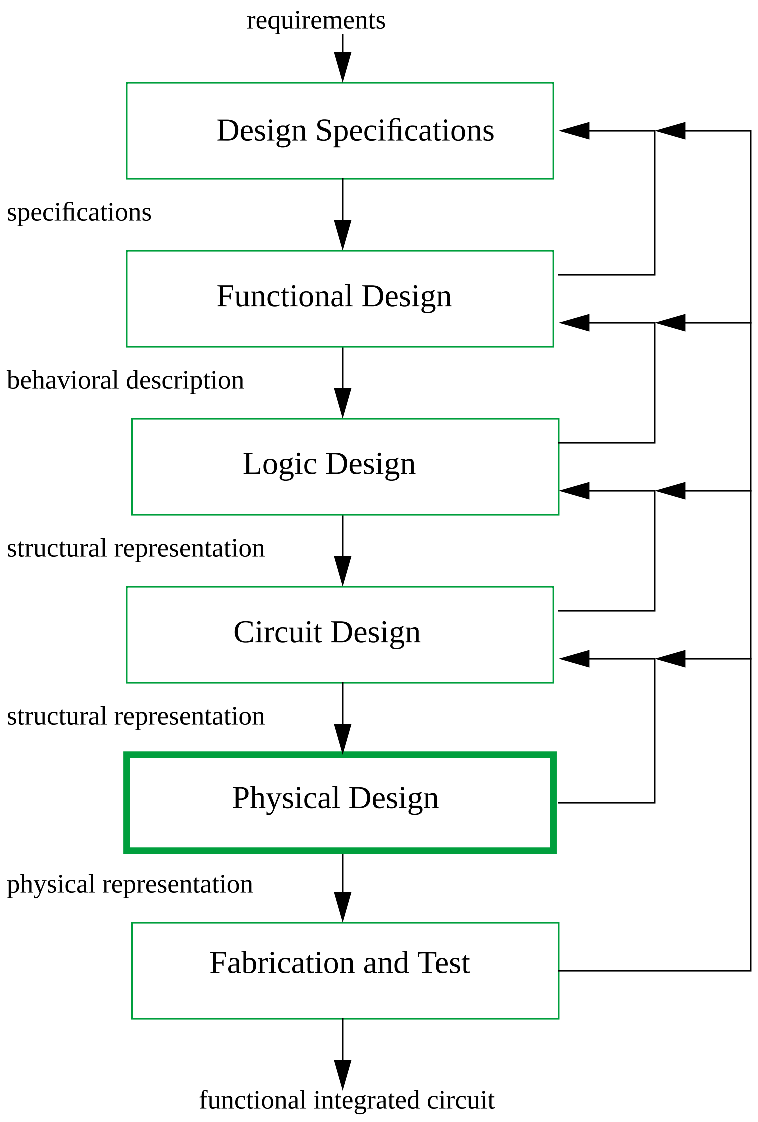
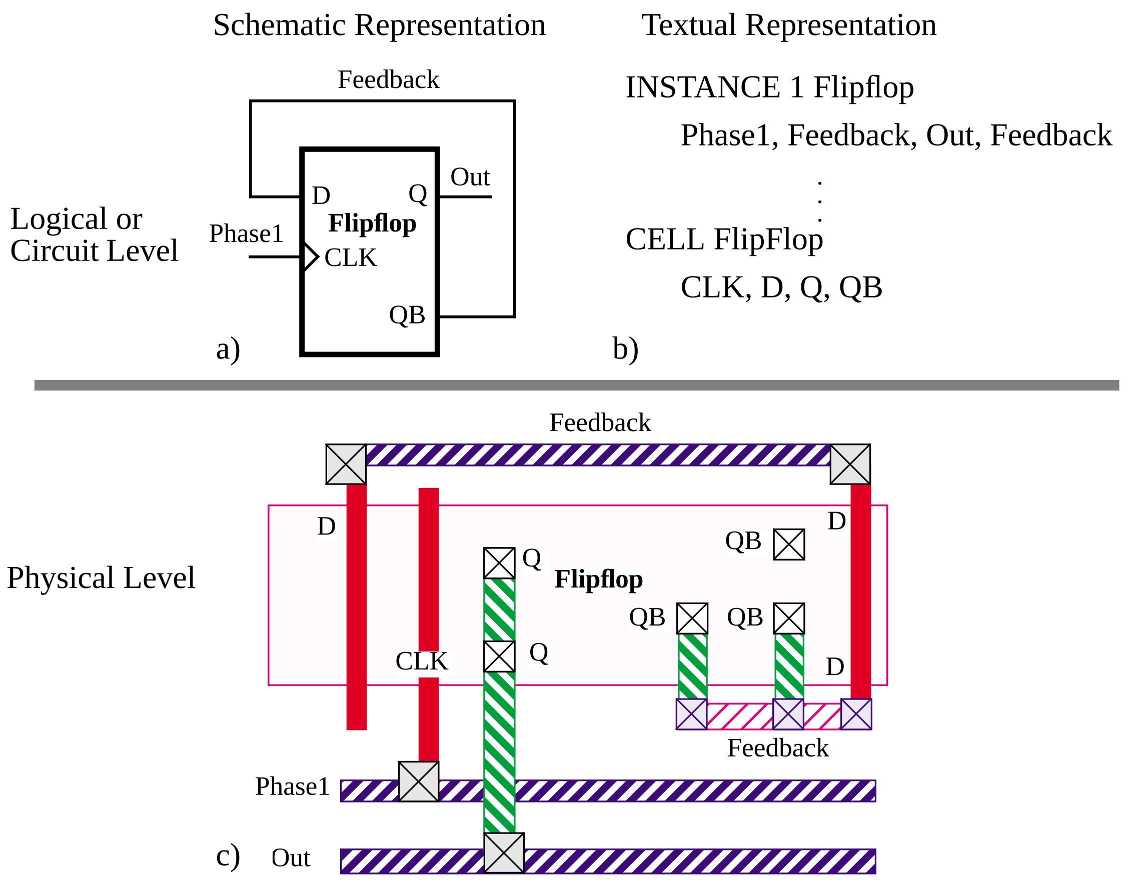


Figure 1.2 Phases of electronic system design. Adapted from [170].

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, or 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.

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.



```
INSTANCE 1 Flipflop
      Phase1, Feedback, Out, Feedback
                  .
                  .
                  .
CELL FlipFlop
      CLK, D, Q, QB
```

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.

1.1.1 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

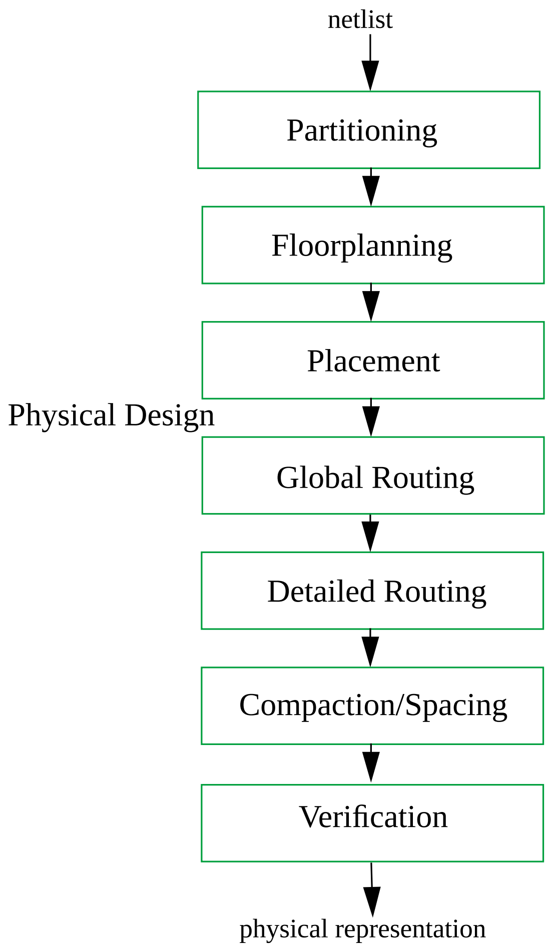


Figure 1.4 Physical design stages.

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].

complexity	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
$000nlgn^*$	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^{lgn}	0.04 sec	1.1 hr	220 days	125 cent	5×10^8cent	
$2^{n/3}$	0.0001 sec	0.1 sec	2.7 hr	3×10^4cent		
2^n	1 sec	35 yr	3×10^4cent			
3^n	58 min	2×10^9cent				

Table 1.1 Running times as a function of input size. One step takes one microsecond. Asterisks denote polynomial time algorithms [222].

time complexity	1 sec	10^2sec (1.7 min)	10^4sec (2.7 hr)	10^6sec (12 days)	10^8sec (3 yrs)	10^{10}sec (3 cent)
$1000n^*$	10^3	10^5	10^7	10^9	10^{11}	10^{13}
$000nlgn^*$	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^{lgn}	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

Table 1.2 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

$$O(g(n)) = \{f(n) \text{ : there exist positive constants c and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} . \tag{1.1}$$

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

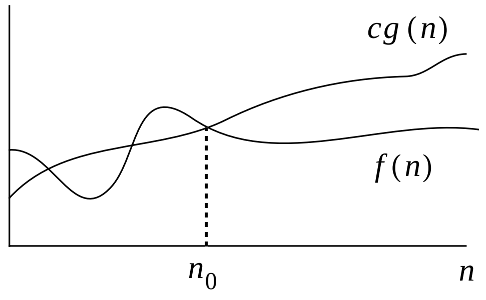


Figure 1.5 *O*-notation gives an upper bound for a function within a constant factor. We write $(n) = O(g(n))$ if there are positive constants n_0 and *c* such that to the right of n_0, the value of $f(n)$ always lies on or below $g(n)$ [42].

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

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

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

2. An algorithm or language *L* is NP complete if

a) $L \in NP$, and

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

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.

1.1.2 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

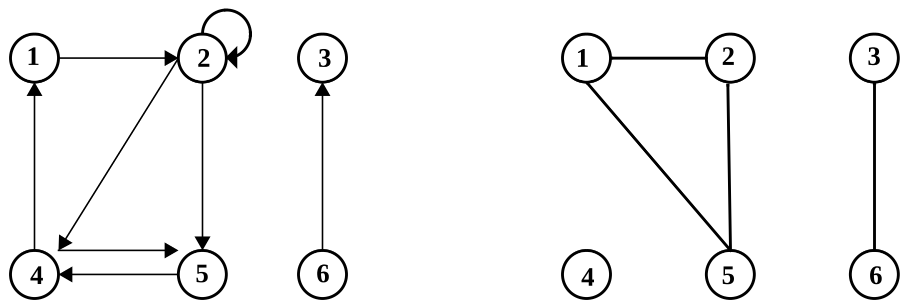


Figure 1.6 Examples of directed and undirected graphs. (a) A directed graph $G = (V,E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2),(2,2),(2,4),(2,5),(4,1),(4,5),(5,4),(6,3)\}$. Edge (2,2) is a self-loop. (b) An undirected graph $G = (V,E)$, where $V = \{1,2,3,4,5,6\}$ and $E = \{(1,2),(1,5),(2,5),(3,6)\}$.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

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 $(v_1, v_2, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$ for i = 1, 2,..., k-1. The path contains the vertices $v_1, v_2, \ldots, v_k$ and the edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$. 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 $v_1, v_2, \ldots, v_{k-1}$ are all distinct. In an undirected graph, a path forms a *cycle* if $v_1 = v_k$ and $v_1, v_2, \ldots, v_{k-1}$ are distinct. A graph with no cycles is *acyclic*. Each edge of a graph may be given a property known as a *weight*.

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) $f{:}V \to V'$ 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.

1.2 Phases of Physical Design

1.2.1 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

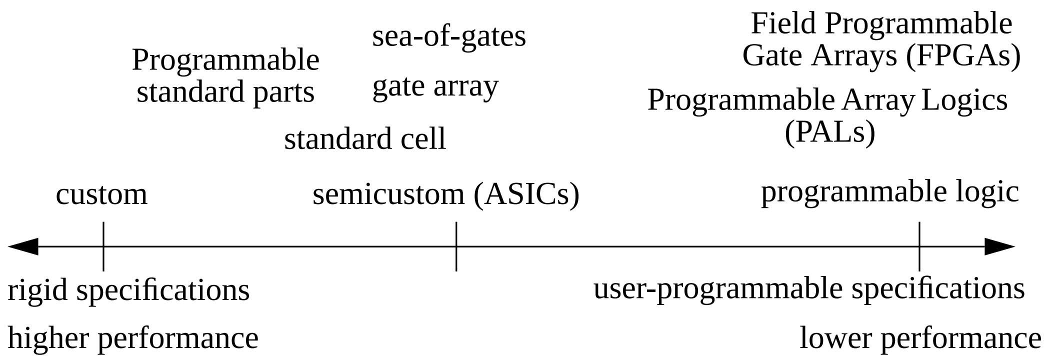


Figure 1.7 Spectrum of design methodologies for integrated circuits.

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.

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 inefficiency[1]. 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

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

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

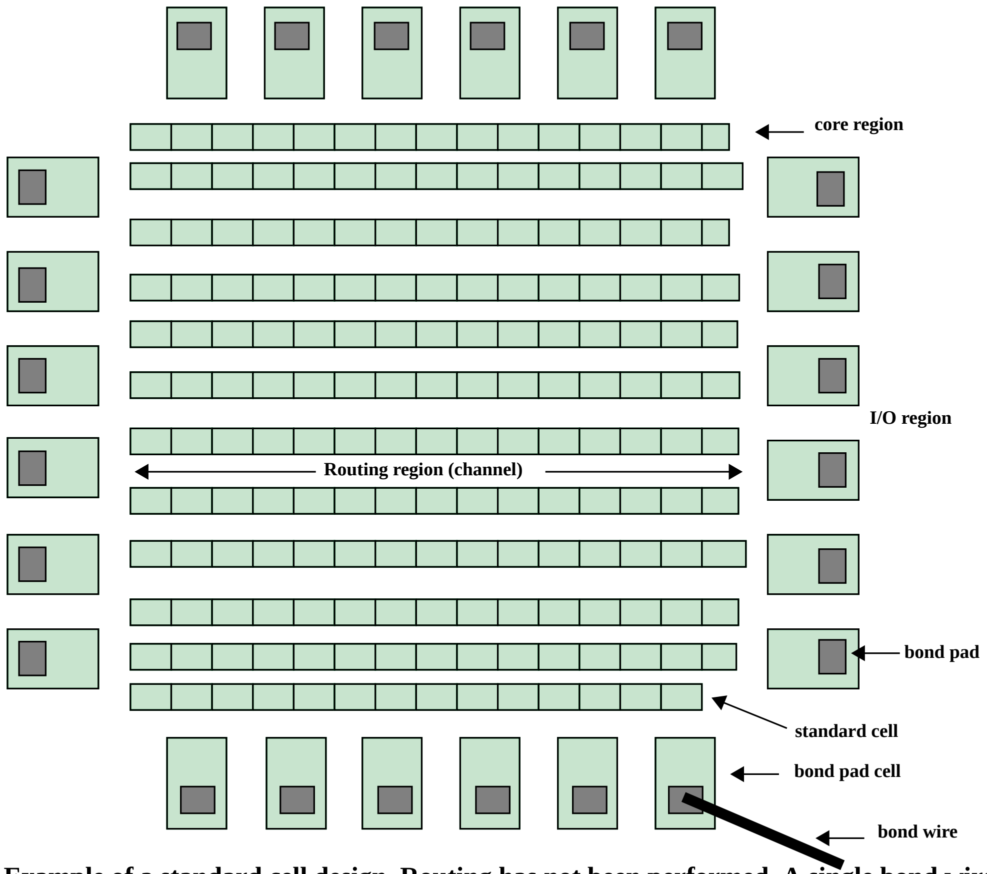
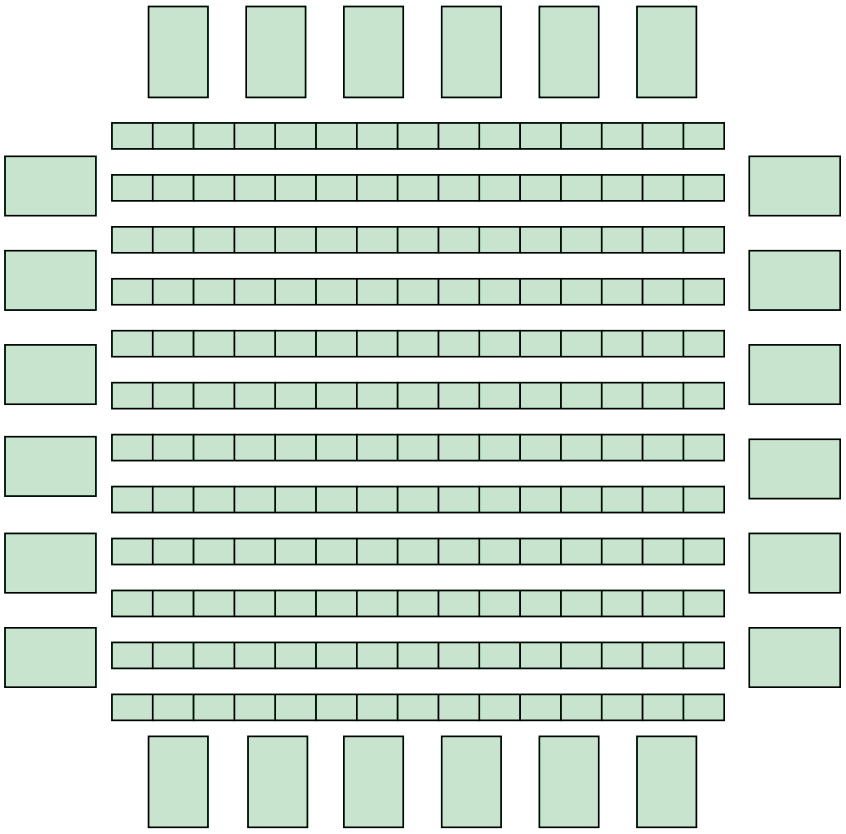


Figure 1.8 Example of a standard cell design. Routing has not been performed. A single bond wire is shown.

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.

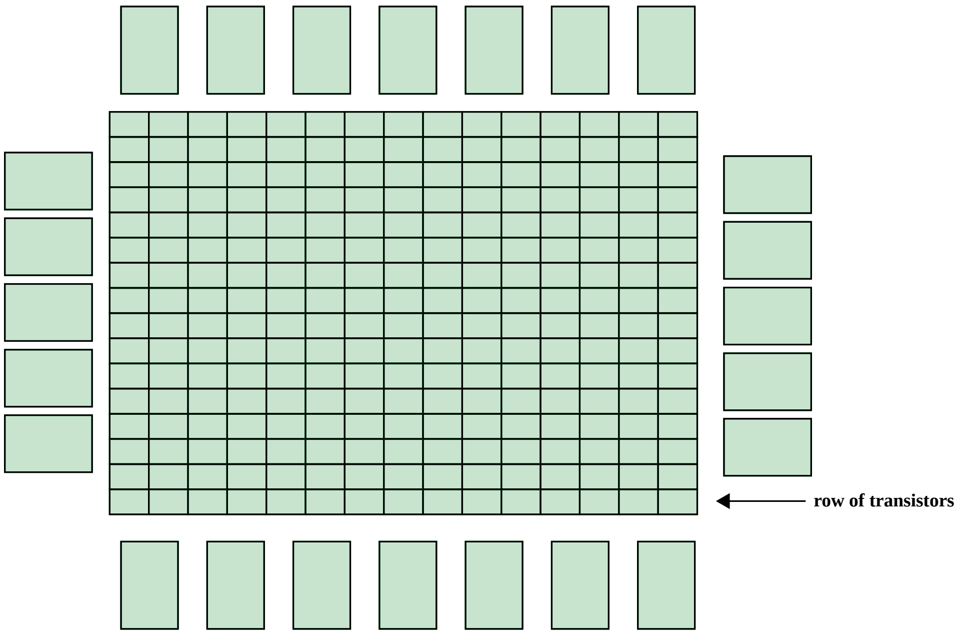
One disadvantage of the standard cell methodology is cost. Performing all of the photolithography steps costs time and money. While programming all layers leads to design flexibility, it requires unique photomasks for each design. None of these photomasks can be used for any other future designs. In addition, any equipment developed to test the IC cannot be shared over designs. The gate array methodology tries to alleviate this problem


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

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.

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



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

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.

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.

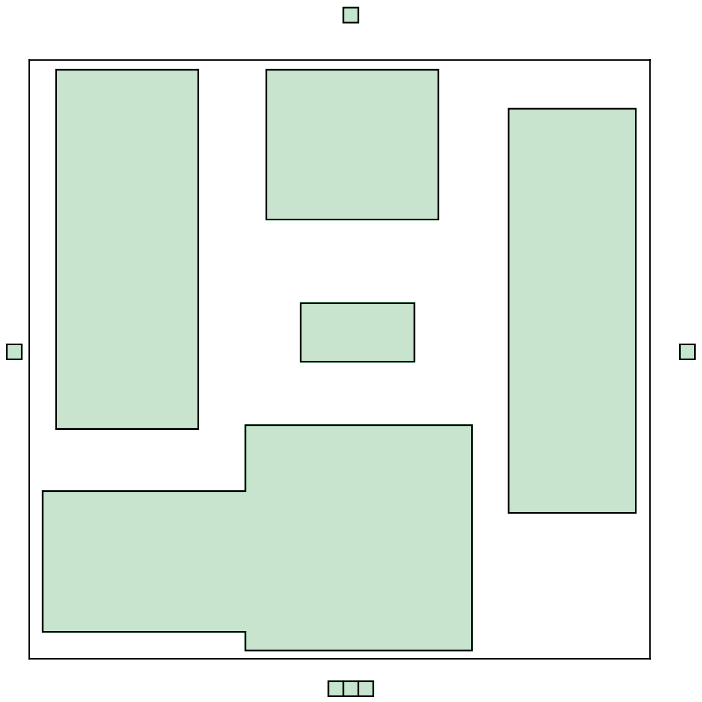


Figure 1.11 Example of a macro cell design.

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 algorithms, however, limit the shapes to rectangles or rectilinear figures. Macro cells are optimized for area and performance, but routing 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.

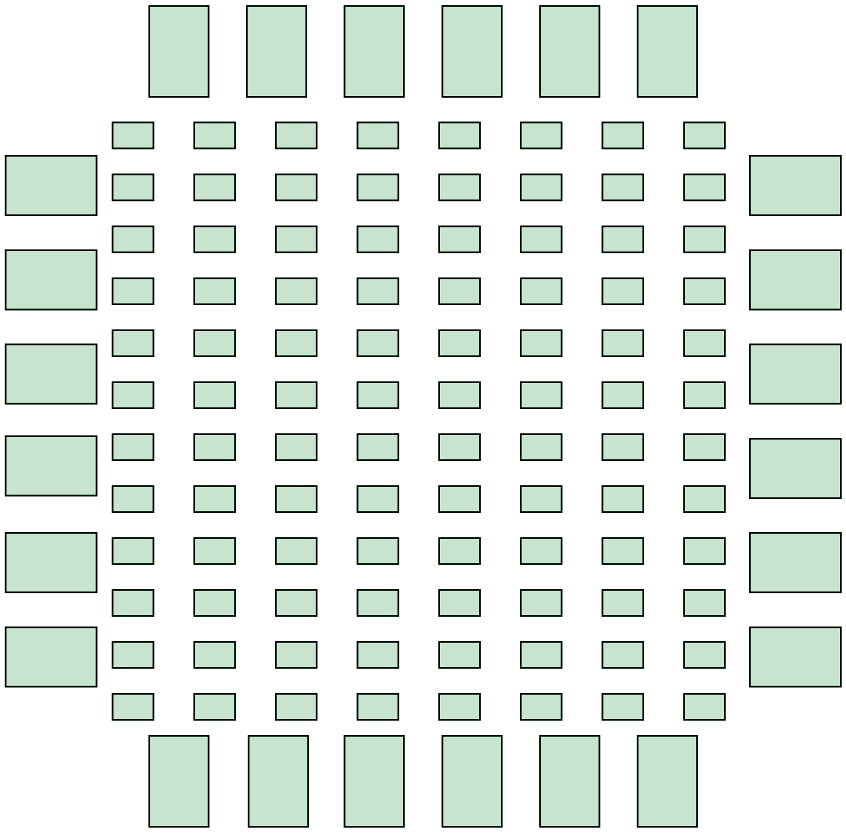


Figure 1.12 Example of an island-style gate-array.

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.

1.2.2 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.

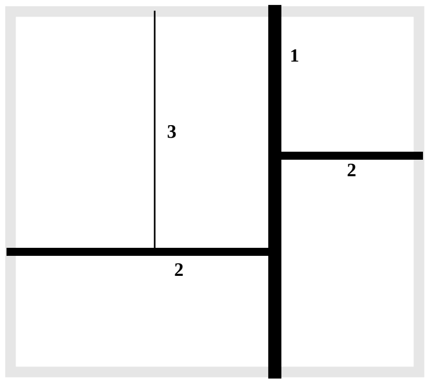
Partitioning a netlist into two pieces by minimizing the number of nets interconnecting the two pieces can be computed in $O\left(n^3\right)$ using a variant of the Ford-Fulkerson algorithm [66]. When size restrictions are imposed on the two pieces, partitioning becomes NP-complete.

1.2.3 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 *nonslice-*



a)

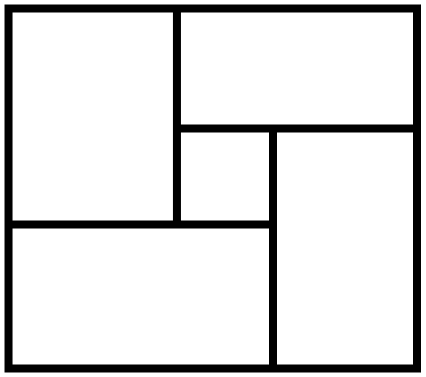

b)

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

able 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.

1.2.4 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.

1.2.5 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.[1] Figure 1.14 shows an example of

1. Formally, the Steiner tree problem in graphs is defined as follows: Given a graph $G = (V,E)$, a weight $w(e) \in Z_0^+$ 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]

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.

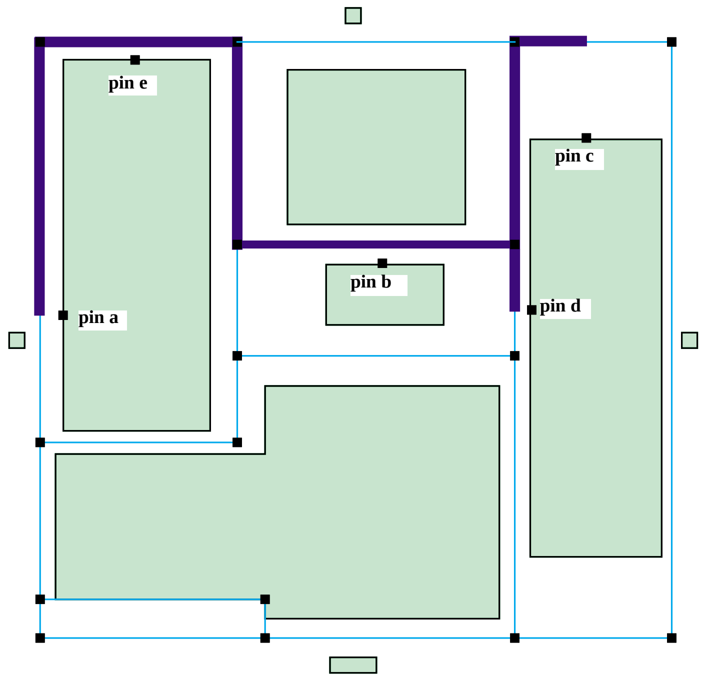
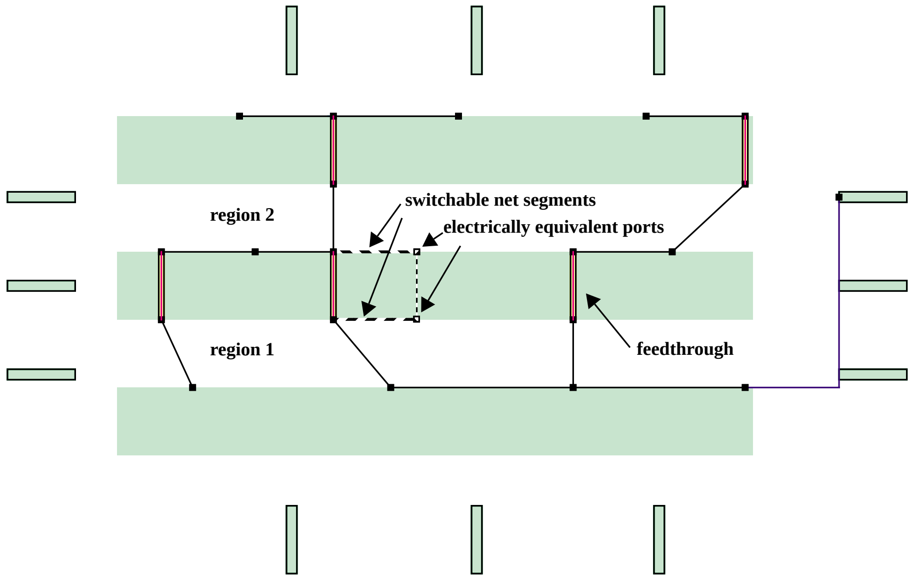


Figure 1.14 Example of graph-based global routing for a 5 pin net. The thick line shows the Steiner tree for this net.

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.



igure 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*.

1.2.6 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*[1] 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].

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

1. 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.

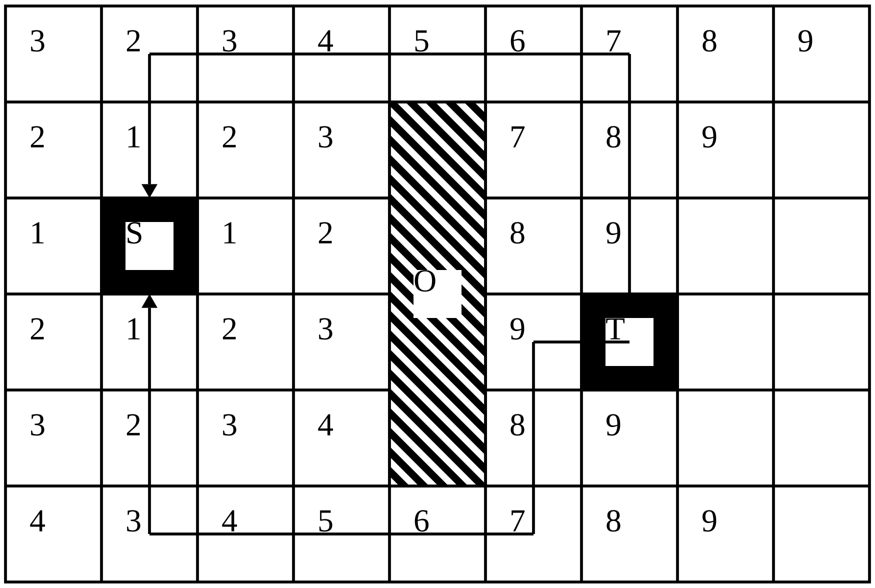


Figure 1.16 An example of one layer maze routing. The width or height of each square equals the grid size. The algorithm labels the grids in a breadth-first manner. Each grid was visited at the step given by the label. There are several paths from the source *S* to the target *T* avoiding obstacle *O*.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

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.

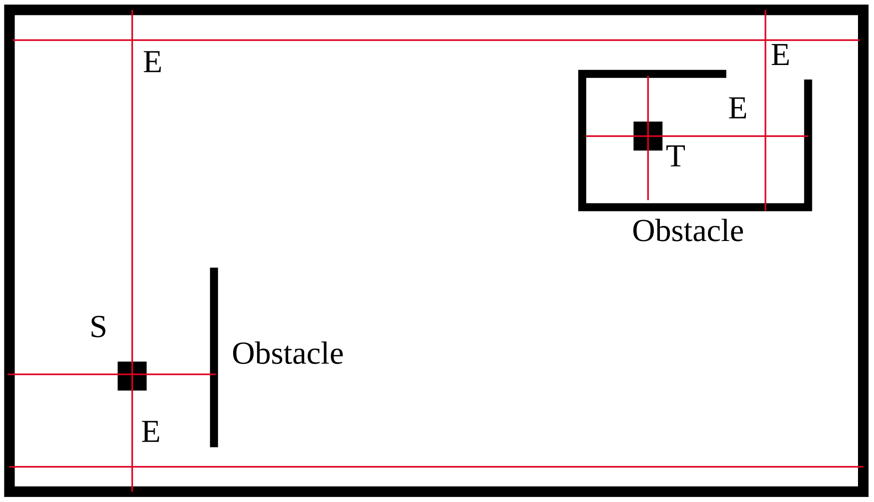


Figure 1.17 Example of a line-probe router [89]. Escape points are labeled E.

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 verti-

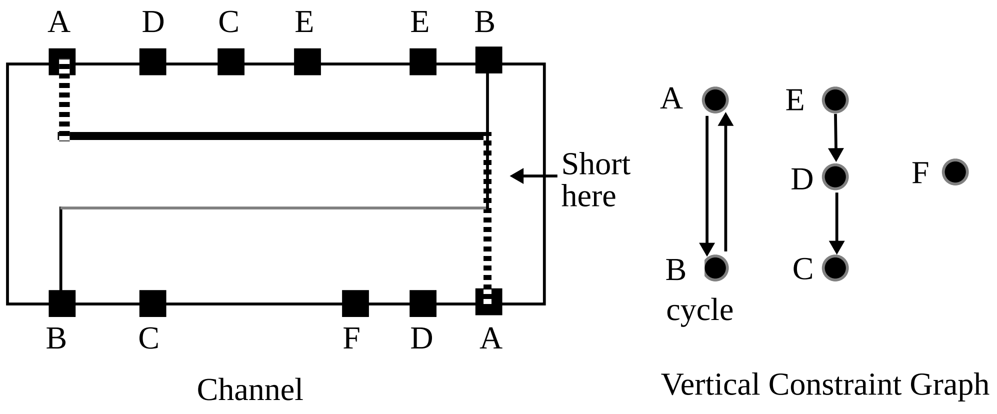


Figure 1.18 Example channel and corresponding vertical constraint graph.

cal 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.

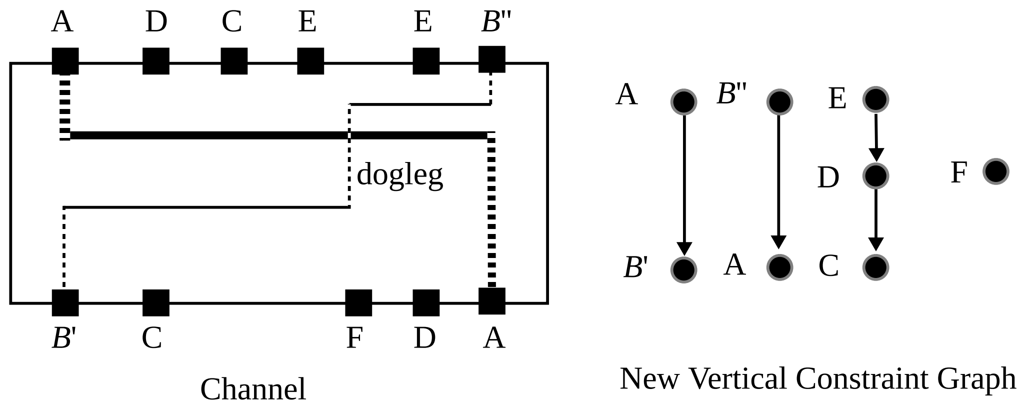


Figure 1.19 Addition of a dogleg to the example channel. The new vertical constraint graph is shown.

Many channel routing algorithms have been proposed. The dogleg channel router breaks vertical constraints by splitting tracks into sections and connecting them with dog-legs [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.

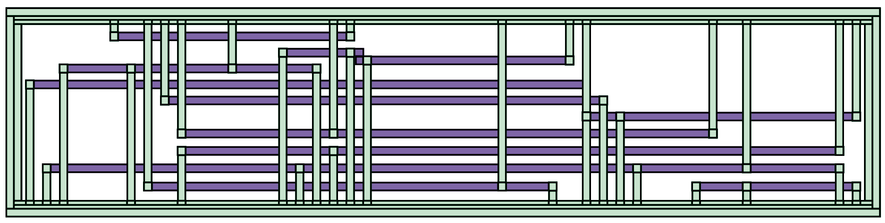


Figure 1.20 A channel routing example using two layers. This is not a left-edge algorithm[234].

Another type of router is the area or switchbox router [37][80][102][207]. An area

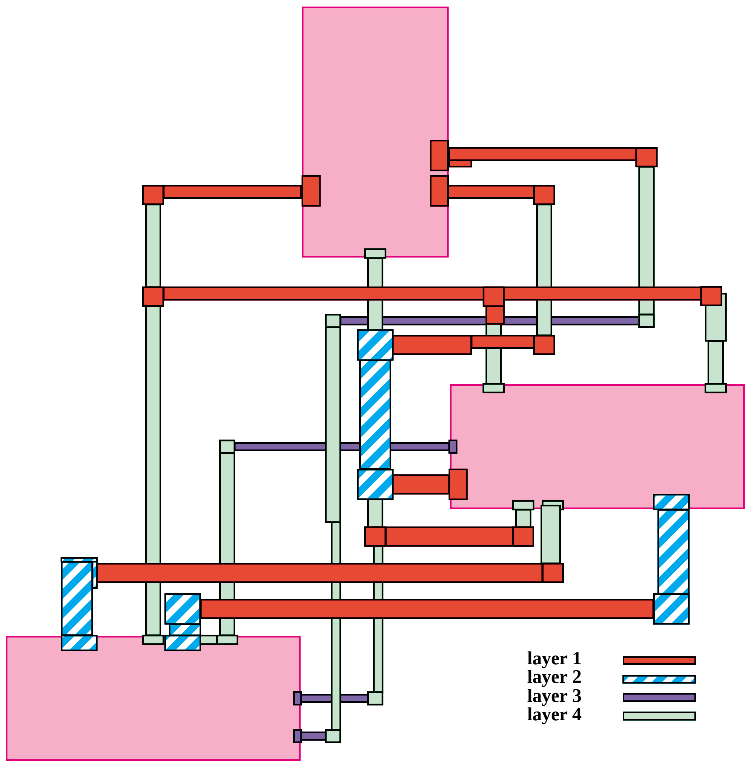


Figure 1.21 An area route for a macro cell example using four layers.

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.

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 $5 \times 10^5 \mathrm{A} \cdot \mathrm{cm}^{-2}$ [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,

$$R = n \cdot R_s \quad , \tag{1.2}$$

where n is the number of squares of a conductor and R_s is the sheet resistance of the conductor in ohms per square[1].

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].

1.2.7 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

1. 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.

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.

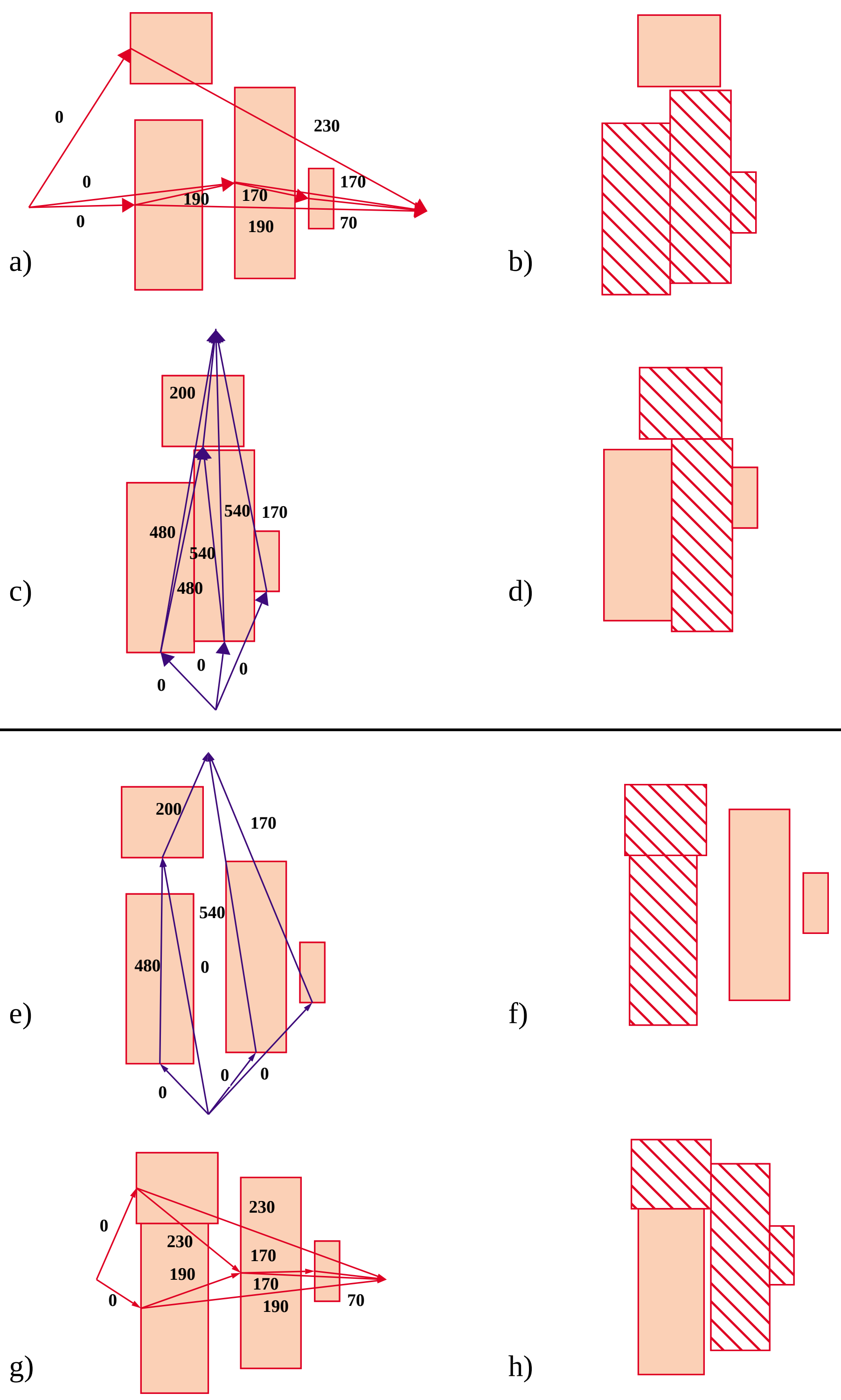


Figure 1.22 Order differences in 1D compaction. Steps a-d show *x*-compaction before *y*-compaction while steps e-h show *y*-compaction before *x*-compaction. The final topology is different in the two cases. The hatched areas are on the critical path. The labels denote the constraints. A centering move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

1.2.8 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]. Checkers 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.

1.3 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.