4. VLSI Design Flow

4.1 Introduction

Knowledge of the various design processes is vital in understanding testing and design for testability. In this chapter we explore the design process to enhance design testability. For example, appending a DFT structure to the design at its completion may compromise the design goals. Instead, we would like to remove barriers to testing while designing and to make the DFT a natural process in design. Barriers to creating easily testable designs may occur at different levels of the design from the behavioral specifications to the physical layout. Therefore, we examine the design process at each level. Except for a few circuits or subcircuits, the design process is automated in most cases. By *automation*, we mean that software programs are used to perform the tasks. These software programs are known as computer aided design (CAD) tools. The tools perform operations on the design representations, the models that we have discussed in Chapter 3. Any tool is based on an *algorithm* that is developed to perform the task *as accurately as possible within a reasonable amount of time*. In other words, it is not enough to know how to obtain an optimal solution. It is equally important to trade off between the optimality of the outcome and the time to completion.

In this chapter we first discuss the CAD tools and the concept of algorithms. Then we examine the logic design phase and the physical design phase. Both phases of the cycles are shown in Fig. 4.1. Accordingly, we discuss both high-level and logic synthesis as well as the technology mapping, which actually depends on the various styles of design implementations. Often, we refer to these styles as *design methodologies*.

4.2 CAD Tools

Design automation (DA), also known as the use of CAD tools in design, has played a vital role in circuit design by greatly facilitating the various stages of the design process to the extent that they should be part of a design engineer's skill. These tools are a continually evolving set of software programs that have replaced many of the tedious tasks that a designer used to do. They are the main cause of design cycle reduction as illustrated in Fig. 1.1. Originally, these tools were relegated to perform repetitive tasks that would take much longer when executed by

Copyright © 2005 Samiha Mourad

humans. Hence the physical design and layout of the circuit was one of the first design aspects to be handled with CAD tools. However, these days the tools are used in more elaborate tasks. Design automation has evolved into a multimillion-dollar business, and many companies have been founded on one or another aspect of the design for test. Commercial synthesis tools then became very common. In the IC design field, the term *synthesis* is synonymous with automated transformation of a design from one level of abstraction to another of lesser degree of abstraction. Any synthesis tool follows an algorithm that includes two elements: (1) mapping from one design description to another that is closer to the physical viewpoint and (2) Optimizing the design to suit the technology to which it is mapped.

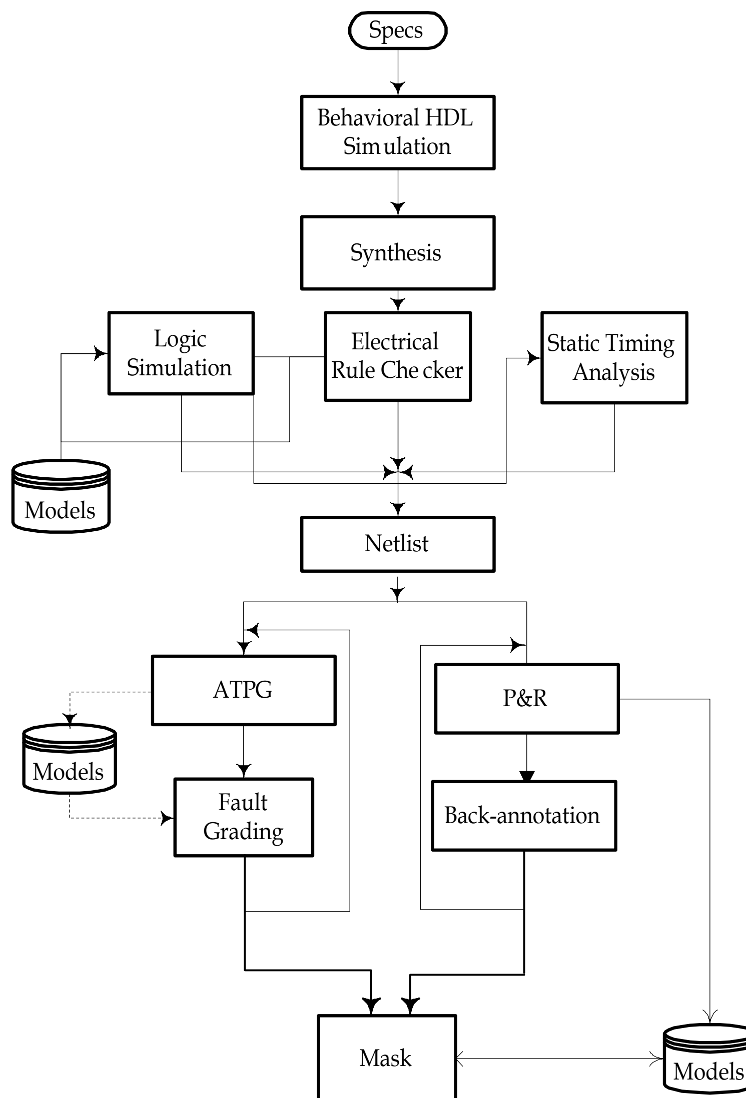

Figure 4.1 Logic and Physical Design

However, CAD tools perform more than synthesis. They involve all steps in the design cycle from

Copyright © 2005 Samiha Mourad

specification entry to parameter extraction from the layout:

- Design entry
- Hardware/Software Codesign
- Simulation (at different levels of design abstraction)
- Synthesis behavioral (to RTL level), logic (to logic gates)
- Embedded test (Scan or BIST) insertion
- Test pattern generation
- Floor planning
- Technology mapping
- Physical design: placement and routing
- Design rule checker (DRC)
- Logic versus schematic (LVS) tool
- Parameter extraction

Design entry tools serve as the interface between users and the other tools. They can be of different styles. *Command modes* were the first means for interaction; however, other modes, such as g*raphic user interface* (GUI), are becoming more sophisticated, due to the unprecedented advancement in graphic design. Also, most CAD programs have dual types of entry. Design entry tools enable users to enter a design hierarchically. Other CAD tools are investigated further in subsequent sections of the chapter, but first we discuss the concept of algorithm.

4.3 Algorithms

Any of the CAD tools mentioned above follows an algorithm that often includes a *heuristic* and a *cost function*. Although informal, an algorithm can be viewed as a recipe. In computer science terminology, an algorithm is “any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output” [Cormen 1992]. It is thus a sequence of data processing steps that operates on the model of the circuit to perform the specific task. The task may be a logic simulation, and the model is a netlist that connects some logic gates, attributes of the gates (such as delays), and a set of stimuli. The outcome of the task is the function or timing of outputs of the circuit. Another task is logic synthesis, which transforms an RTL representation of a circuit into logic gates.

Copyright © 2005 Samiha Mourad

In performing this task, the tool may constantly be guided by a *cost function*. The term *cost* here has no relation to the monetary value of the product. It is a local meaning related to the particular algorithm. In performing the task and selecting between alternatives to realize it, certain parameters are evaluated constantly to assess the results. A cost function in synthesis may be to minimize area or power. An example on the physical level is placement of cells on a chip. The attempt is to find a place on the floor of the chip such that connectivities are minimized. Thus the cost function here is the total length of the interconnect wires. In all these examples, the model for the circuit is a graph with appropriate selection of the vertices and the edges (see Chapter 3).

The complexity of algorithms depends on the problem to be solved. It is measured by the space required in the computer memory and the execution time. Usually, both factors characterize the algorithms. However, as computer memory is becoming bigger, time complexity is more important. The execution time of the software realizing the algorithm is a function of the processing steps followed to solve the problem. This is determined mostly by $n = |V|$, where V is the set of vertices of the graph, $G(V,E,W)$, representing the problem. The time complexity is usually expressed as a function, $f(n)$, such that $f(n) \leq cg(n)$, where c is a constant. This is described as $f(n)$ is of order $g(n)$, $O(g(n))$. To illustrate the concept, consider the multiplication, M, of two n x n matrices, A and B, $M = A$ x B. Any element, $m_i \in M$, is given by $m_{i,j} = \sum_{k=1}^{K=n} a_{i,k} b_{k,j}$. Since there are n^2 elements in M, the complexity of matrix multiplication is $O(n^3)$. The complexity is thus an order of a polynomial in n.

Algorithms with such a complexity are called *tractable*. Most problems encountered in testing are *intractable*. They are characterized as being NP-complete problems. NP stands for nondeterministic polynomial. That is, there is a nondeterministic algorithm that solves the problem in polynomial time. Hence the algorithm includes a heuristic that directs the solution in a more efficient manner. A heuristic is a common sense rule (or set of rules) intended to increase the probability of solving some problem without guaranteeing an optimal solution. A heuristic cannot guarantee that the problem will always have a solution. As we will find out in Chapter 6, test pattern generation is an NP-complete problem [Goel 1980, Fujiwara 1982]. This is also true of partitioning a circuit or placement and routing [Ibarra 1975].

Algorithms developed to solve the same problem often differ in their efficiency in solving the problem. This is for one of several reasons. They may be written for different computer platforms, say a PC versus a workstation, although more and more the difference between the two platforms is narrowing. Also, engineers with

Copyright © 2005 Samiha Mourad

different capabilities or experiences compose them. For the algorithm to be efficient, three main attributes are needed: the right data structure, the right heuristic, and a capable programmer. In addition, the need for an efficient software toolbox can never be overemphasized.

To appreciate the complexity of algorithms and the need for heuristics, assume, for example, that you are developing an algorithm to simulate a human being playing chess. If you try to consider all possible moves and their consequences in every play, a game would not end within a human lifetime. Instead, a heuristic that utilizes the human experience in playing chess may be devised to avoid unnecessary or trivial moves. In this way, the time complexity is reduced.

4.4 Synthesis

Synthesis is the process that transforms the design from one level of abstraction to another that is closer to the physical circuit level in the physical domain. It is the most challenging task to automate because it is more difficult to capture by an algorithm. There are a number of reasons for this: different design constraints and goals, any design representation levels, or several alternatives for design implementation of a given design.

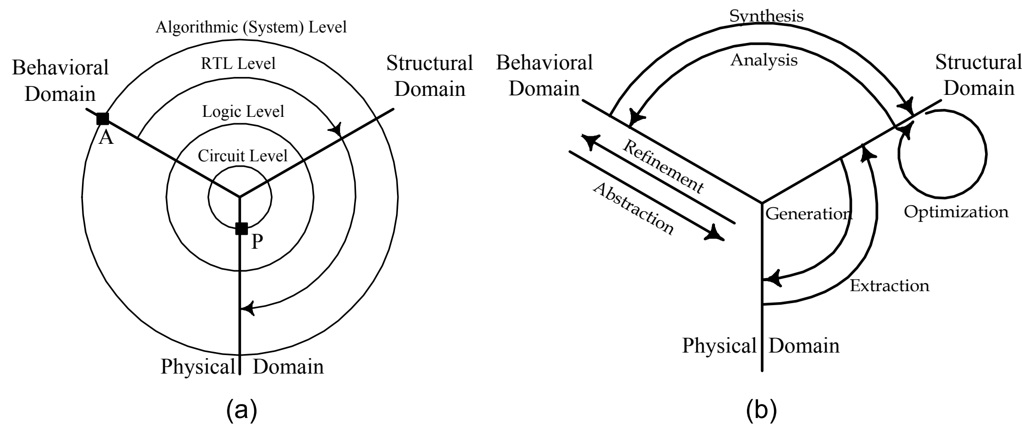

Figure 4.2 Synthesis Processes

According to the taxonomy used in Chapter 3 and illustrated in Figs. 3.1 and 4.2*a*, synthesis transforms a design from a high level of representation (point *A*), the algorithmic code in the behavior domain, to the lowest level in the physical domain (point *P*), the mask. The path from *A* to *P* consists of several translations from one level to another in the same domain and from one domain to another at the same level, and optimization at any point along the path [Gajski 1983, Walker 1985]. These translations are few-to-many, meaning that there are usually few

Copyright © 2005 Samiha Mourad

structural implementations for each behavioral representation and even more physical configurations for each structural model.

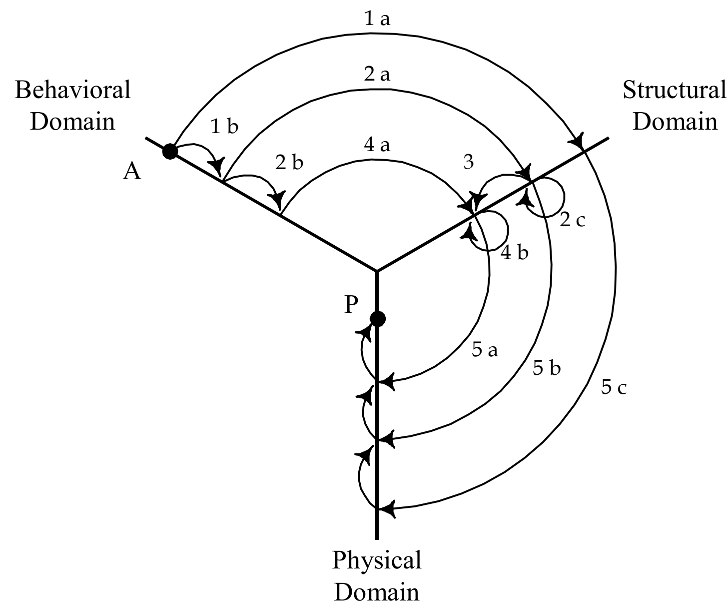

Figure 4.3 High Level Synthesis

The transitions shown in Fig. 4.2*b* represent different tasks performed on the design [Michel 1992]. Based on this model, *synthesis* is a transition from the behavioral to the structural domain. A self-loop is an *optimization* of the design at the same level. A transition from the structural to the physical level results in generating (placing and routing) the mask for the design, while the reverse transition is known as *parameter extraction*. From the layout, electrical parameters are calculated and can be included in the structural domain for accurate simulation of the design. At present, the extraction is also fed back in the design on the behavioral level.

The different synthesis processes are mapped in Fig. 4.3. Step 1a translates the various operations into concrete subsystems, processors, and so on. Step 1b results in an RTL-level description that relates the component from step 1a. The design is represented by operation on the data path by control steps that have been scheduled according to a certain clock. The *RTL synthesis* consists of transitions 2a, 2b, and 2c. This is not a synthesis task proper; it is more of binding the blocks at the system level to specific blocks and relating the different tasks to the clock. Step 2c refines the gate-level design by optimizing on some attribute, number of gates, or interconnect wires. However, step 3 is the technology mapping. Steps 4a and 4b are for the logic synthesis and optimization. Any or all of steps 5a, 5b, and 5c generate the masks, a process that is also known as *physical synthesis*. For a design based

Copyright © 2005 Samiha Mourad

solely on standard cells, step 5a is sufficient. However, a system on a chip that includes cores would also involve step 5c, depending on the type of core, as we define them in Section 4.6.

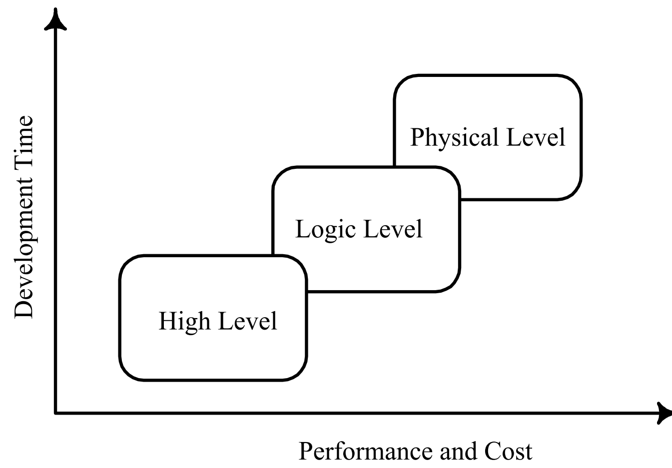

Figure 4.4 Comparing Synthesis Types

Depending on the availability of tools and other factors, the designers may use one or all synthesis types. The more automation that is used, the faster it is to complete the design. A qualitative comparison of the various synthesis types is shown in Fig. 4.4, where the ease of design is given versus performance, cost of development, and completion time. This trend will be clearer as we describe synthesis processes in the remainder of this chapter.

Chronologically speaking, physical synthesis was the first to be automated and is described in detail in subsequent sections of this chapter. Next came logic synthesis, except that it really was more of a transformation of the functional descriptions into optimized, technology-independent logic gates. It is only in the late 1980s that synthesis tools appeared on the market. However, there are companies, such as IBM, that had been using their proprietary tools a decade earlier [Hong 1974]. The first commercial synthesis tools were based on research work done mostly at universities [Brayton 1984, 1987].

Not far behind these logic synthesis tools was the automation process of RTL to gate-level mapping. It has been called *RTL synthesis* to differentiate it from the previous type. The RTL description consists of cycle-by-cycle

Copyright © 2005 Samiha Mourad

behavior, which is still defined in programming-like language in terms of registers, and combinational logic blocks such as arithmetic and logical function.

4.4.1 Behavioral Synthesis

Behavioral synthesis has many advantages. First, it shortens the design cycle and minimizes the chance for errors. As a result, it is possible to hit the market at the appropriate market window for that design. The importance of this market window to product profit was discussed in Chapter 1. Another important factor is that it is possible to explore the design space and select the most appropriate implementation for the product. The design space used to be defined solely by area (hardware) and performance (operating frequency). Thus there is a trade-off between the two attributes, but power and testability are becoming very important parts of the metric for design evaluation.

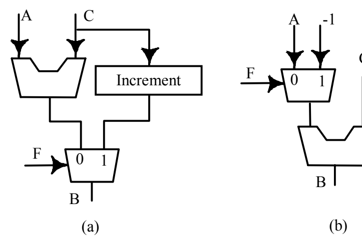

Figure 4.5 RTL Optimization: Alternative Implementations

This type of synthesis consists of three main processes: compilation, scheduling, and allocations, which transform the behavioral model to the RT model shown in Fig. 4.5. The behavioral description given in the left side of the figure is translated into the structure to its right though step 2b in Fig. 4.3*b*. The design seems to require an adder and a decrementer. Since the two operations on the data are mutually exclusive, they must both involve adding *C*. Thus, according to the value of *F*, either *A* or -1 is added to *C,* as depicted by the rightmost structure in the figure. This optimization step is equivalent to Step 2c. The optimization was to reduce the area of the design. Optimization to improve testability is discussed in Chapter 14.

4.4.2 Logic Synthesis

The result of behavioral synthesis is an RTL module that needs to be mapped to the logic or even physical level. The RTL model already includes architecture consisting of several basic modules, such as registers, multiplexers, and functional units. While some of these modules are part of the library, the others are to be designed on the logic level. Logic synthesis consists of two stages: technology independent and technology dependent. The latter stage is also known as *technology mapping*. Steps 4a and 4b represent logic synthesis. The second step is an optimization.

Copyright © 2005 Samiha Mourad

In *technology-independent synthesis*, the network is represented by a model that may be any of the possibilities enumerated in Chapter 3 truth tables, algebraic expressions, BDDs, or other forms of graphs [Bryant 1986]. Multilevel synthesis, which is the approach generally used, organizes the Boolean expressions in tree structures and performs several operations on the tree. The operations are decomposition, extraction, substitution, and collapsing, which are analogous to polynomial division [Bartlett 1986, Brayton 1987]. The following expression is in factored form:

$$X = (AB + B'C)\,[C + D(E\ + AC')] + (D + E)\,(FG) = F_1[C + DF_5\,] + F_3F_4$$

It is represented by the binary tree of Fig. 4.6, where the leaves represent the literals and the other nodes represent the logical operations AND and OR. The main goal is to minimize the number of literals in the Boolean expression. On the tree, these are the edges leading to the leaves and to the five functions F_1 through F_5. In this case the count is 18. These literals represent the interconnections between the gates when the design is implemented. In present technologies, the majority of the IC area is occupied by the wiring connecting the gates.

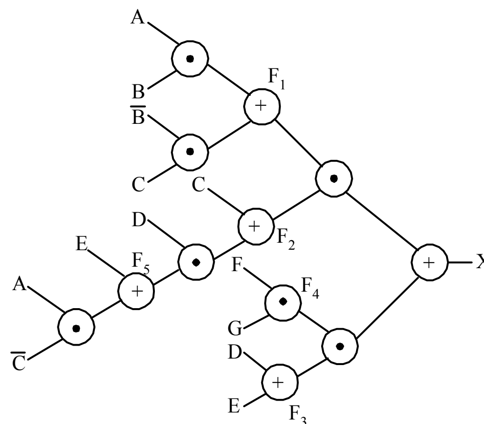

Figure 4.6 Multi-level Logic Synthesis

Technology dependent optimization transforms a Boolean expression into a gate network using the gates in the target library. This is *library binding,* more commonly known as *technology mapping*. The library may consist of only NAND gates or of specific generic gates such as multiplexers. At present, some of the cells in a library are designed to facilitate testing. For example, flip-flops are used for scan-path design or LFSRs for BIST. Both scan path and BIST are DFT techniques that were mentioned in Section 1.10 and are described in detail in Chapters 9 and 11.

Copyright © 2005 Samiha Mourad

4.5 Design Methodologies

Integrated circuits consist of transistors that are placed on the chip and are connected in such a way as to realize the design. Several masks define the locations and connectivity of the transistors. A mask corresponds to one of the silicon compound layers that forms the transistors and the interconnect layers. Circuit implementations may be grouped into two main categories: fully custom and semicustom designs, as illustrated in the hierarchy shown in Fig. 4.7. In fully custom design style, all transistors are handcrafted. Since the designer controls all stages of the chip layout, maximum design flexibility and high performance are possible. Consequently, only highly skilled and competent designers are engaged in such design methodology. Also, development time is long and development costs are extremely high. It is thus more suitable for large volume products. Memory and microprocessor ICs are created using unique masks for all layers during the manufacturing process. The user controls chip density with high utilization. However, the high cost of design and testing can be amortized successfully over the high volume.

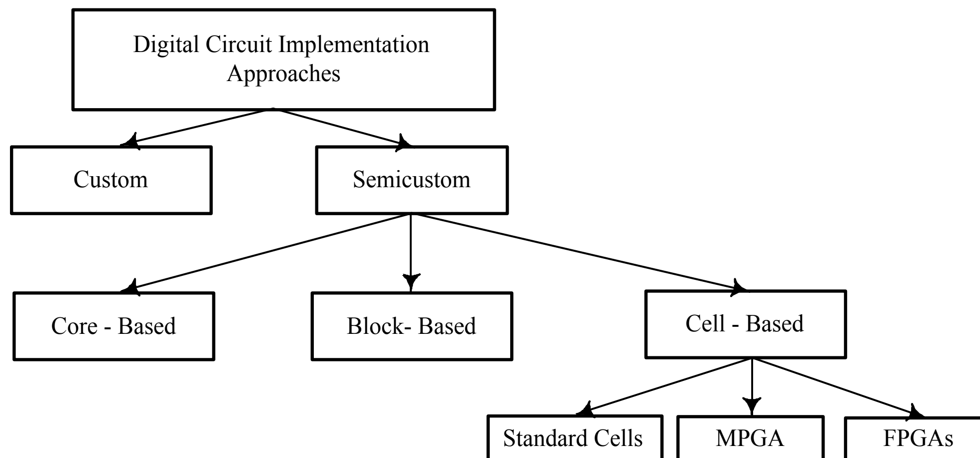

Figure 4.7 Design Methodologies

Semicustom design is facilitated by having CAD tools use a library of predesigned components and sometimes of prefabricated chips such as programmable logic arrays (PLAs) and field programmable gate arrays (FPGAs). It is a restrictive mode of design and results in less compact circuits than that of fully custom design. Its main advantage is shorter time-to-market and rapid prototyping. Therefore, semicustom is more appropriate for moderate and low-volume products. We describe each category in subsequent subsections. A comparison of the various design methodologies using a metric of ease of design and flexibility is shown in Fig. 4.8. The more automated the steps used in the design, the more restricted the designer is and the design time is shorter. However,

Copyright © 2005 Samiha Mourad

the semicustom methodologies are becoming more efficient as we learn in the next section.

4.6 Semicustom Design

Practically speaking, fully custom design is very rare. Even in microprocessors, some components are extracted from a library. Actually, the trend nowadays is toward reuse of already designed and tested cells, usually referred to as *cores* or *intellectual properties* (IPs). This IP terminology is used because the core that is embedded in the design may be acquired from another vendor. The classification of semicustom design here is based on the size and complexity of the components used in the library. The first category is *cell-based*, for which the library components are small gates such as NAND, NOR, and flip-flops. This category is subdivided in turn into standard cells, FPGAs. The second category is *block-based*; these are larger cells such as an ALU or a large multiplier. The last category is *core-based*. The core may be an entire microprocessor or several RAMs.

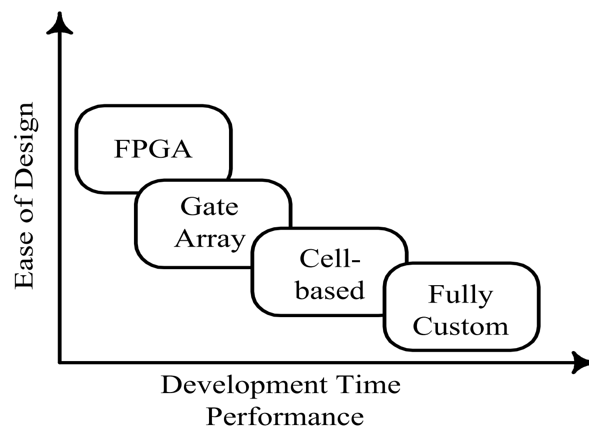

Figure 4.8 Comparing Design Methodologies

Cores come in three types: soft, firm, and hard. A *soft core* is usually described in an HDL language and is, accordingly, very flexible. *Firm* core is already mapped to a library; and a *hard core* is given as a layout that cannot be changed. Core-based design is discussed in Chapter 15. Next, we discuss cell-based design, most of which is also applicable to block-based design.

4.6.1 Standard Cell Design

Under this approach, the logic is mapped into a predesigned library. Depending on the cell type, two types of designs are distinguished: standard cells and macros. The design is called *standard cell* if the cells represent logic gates such as multi-input NANDs, NORs, and so on, and the masks of these gates are of the same height. They are usually saved in a database and designers select the appropriate cells to realize their design. The cells are then placed

Copyright © 2005 Samiha Mourad

in rows and interconnected as illustrated in Fig. 4.9*a*. The space between the rows is called the *channels*. The interconnecting wires are laid along tracks in the channels. This process of interconnecting them is known as *routing*. If the cells to be connected are not in adjacent rows, the wire passes from one channel to another through cell location. This is known as *feed-through*. The width of the channels varies according to the degree of interaction between the cells and the quality of routing. The routing process is discussed in Section 4.7.2. Placement and routing are done automatically, thereby almost removing the designers from the physical design process. Sometimes these cells are larger than simple logic gates; they are then called *macros*. Actually, a macro might be as big as a microprocessor and is referred to as a *core*. However, for macros and cores, the height is not uniform, as illustrated in Fig. 4.9*a*.

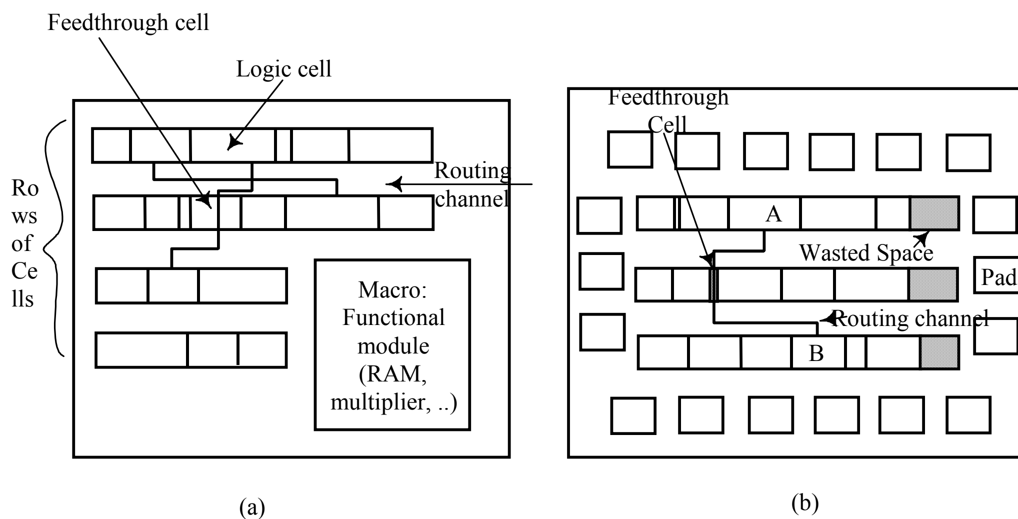

Figure 4.9 Semi Custom Design (a) Standard Cells, (b) Gate-arrays

4.6.2 Mask-Programmable Gate Arrays

A gate array consists of a prefabricated two-dimensional array of transistors that are not connected. Thus there are generic masks for all layers except for the metalization mask. The metalization layer is customized to the user's specifications. Adjacent transistors can be connected to form a two-input NAND gate or a flip-flop as shown in Fig. 4.10. User logic is implemented by patterning these transistors into logic functions and connecting the different modules. Connectivities of the modules are along the channels. Once the connectivities of the cells are finalized, the metal mask is used to finalize the fabrication of the gate arrays. This is the reason they are called mask programmable. The image of the gate arrays appears as shown in Fig. 4.9*b*, which is similar to standard cells except that the channels are of equal width. A cell library, making the designer's expertise less critical than in the case of

Copyright © 2005 Samiha Mourad

the fully custom methodology, usually facilitates the design. For the same reasons, MPGAs offer shorter development time and lower development costs than do standard cells and, a fortiori, fully custom ICs. A special class of gate array is channelless; they are known as *sea of gates*.

4.6.3 Programmable Devices

Programmable devices, which are available in a variety of sizes, architectures, and programmability technologies, have played a key role in digital design for awhile. Programmable devices have the advantage of being available as off-the-shelf parts that can be programmed in a few minutes. Another advantage is that some of them are reprogrammable. However, they are not as versatile as standard cell and gate-array styles of design. Because of their regularity and their programmability, they have particular faults in common as well as particular testing problems.

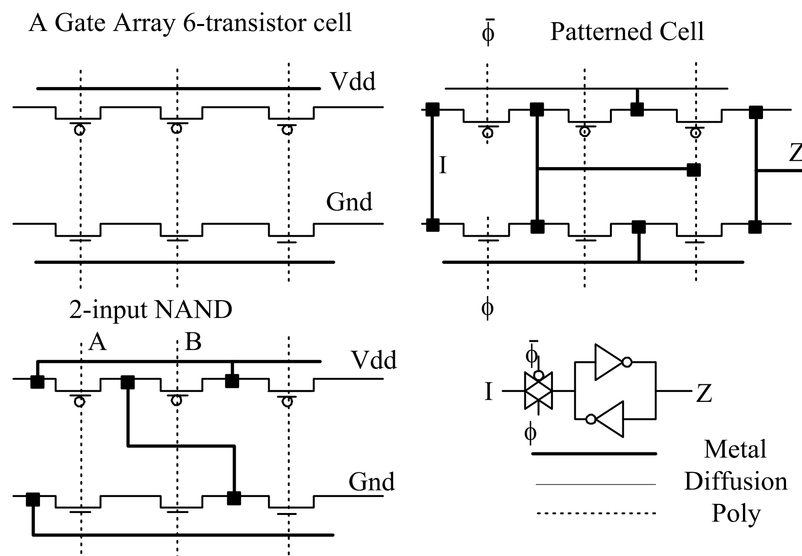

Figure 4.10 Gate Arrays: Patterning Transistors

One group of these devices is characterized by the implementation of any design in a two-level AND/OR structure. This category includes the three types listed in Table 4.1. In addition to failures common in conventional ICs, programmable devices are subject to unique failure modes that we identify as *programmability failures*. A faulty chip or the malfunctioning of the device programmer can cause these types of failures. Like any electronic equipment, the device programmer can fail due to noncalibration. Some devices may be added or omitted unintentionally. To account for these extra and missing devices, DFT techniques have been specially developed [Hong 1980, Fujiwara 1981, 1984].

Table 4.1 Types of AND/OR Arrays

Copyright © 2005 Samiha Mourad

Type	AND Plane	OR Plane	Remark
PROM	Fixed	Program	EPROM and EEPROM
PLA	Program	Program	Are not as used
PLD	Program	Fixed	May include flip-flops

Field programmable gate arrays constitute the second category of programmable devices. Xilinx introduced them in 1985 [Xilinx 1985]. Like MPGAs, these devices consist of uncommitted logic blocks, which are organized in rows or in a matrix form. Unlike MPGAs, there are vertical and horizontal channels in the matrix architecture that usually intersect in a switchbox. Also, the mapping of the design is on the logic level. The blocks can be connected by a general interconnect resource that comprises segments of wires of variable lengths, *segmented tracks*, as illustrated in Fig. 4.11. These wires are organized in specific geometries in such a way as to accommodate channel or switchbox routing, and they are connected by programmable devices. The three main types of programmable devices used are SRAMs [Xilinx 1985], antifuses [Actel 1991, Quicklogic 1993], and electrically erasable connections [Altera 1992]. FPGAs are described in more details in Chapter 13.

Table 4.2 Interconnect Failures: Mapping Failure Modes into Fault Model
Adapted from Chan [1994]

INTERCONNECT FAILURE	FAULT MODEL
Extra connection between independent wires	Bridging Fault
Missing connection between segments of metal	Stuck-open fault or floating gate
Improper programming of the program elements	Delay fault

Similar to PLDs, FPGAs are also subject to programmability failures. However, the magnitude of the problem is increased since the programmable devices are also used in the interconnect resources. This may cause missing and extract connections between the segmented routing resources. In addition, they may produce longer delays than were accounted for at design time. FPGAs are prone to specific physical defects, depending on the underlying architecture and programming technology. For instance, in EPROM-based technology, erasure of programmable connections can occur with exposure to light. The logic cell array's use of volatile static memory makes it prone to problems caused by noise, radiation, and power dropouts. Independent of the technology, failure modes in the interconnection between the segmented metal wires may result in extra and missing connections. These failures result in different possible fault models, as listed in Table 4.2. Testing of FPGAs is one of the main topics in Chapter 13.

Copyright © 2005 Samiha Mourad

4.7 Physical Design

Once the design has been synthesized and mapped to a technology, a netlist is obtained and used in the physical design phase, which involves the following main processes: floor planning, placement, routing, and parameter extraction for logic versus schematic (LVS).

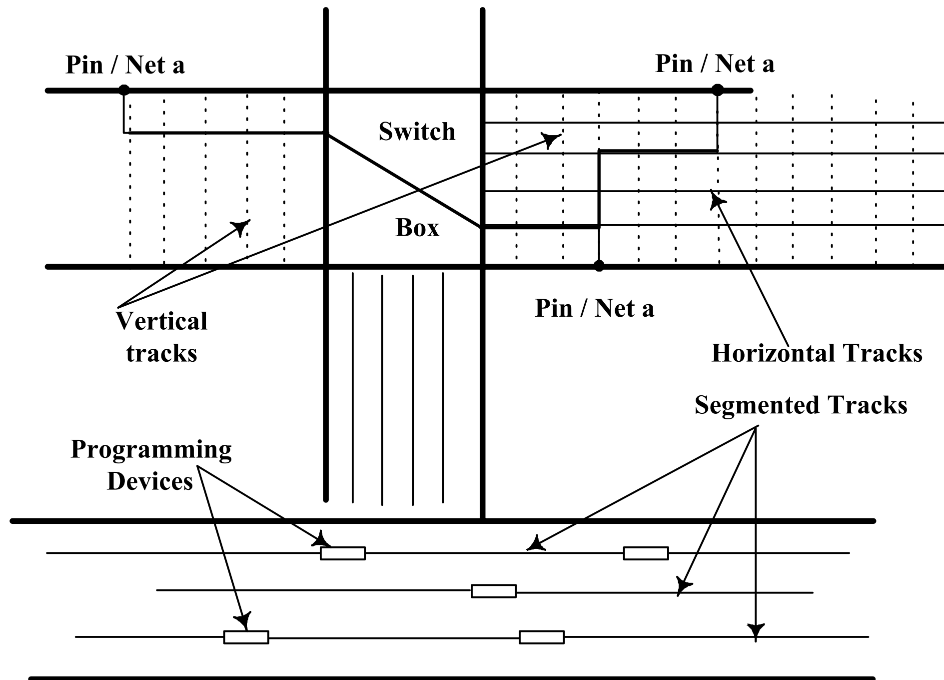

Figure 4.11 General Interconnect Resources in FPGAs

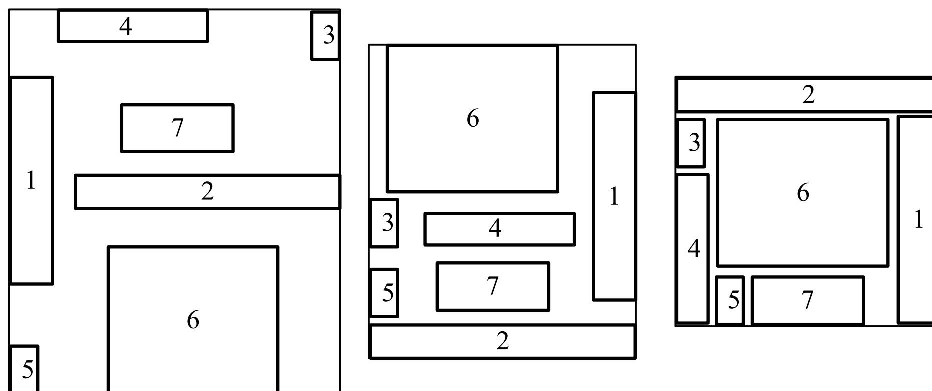

Figure 4.12 Floor Planning

Copyright © 2005 Samiha Mourad

4.7.1 Floor Planning

Floor planning involves assigning locations for the various modules on the chip. Fig. 4.12 shows a progression of floor plans for the same chip. The square shape is a better plan, as it minimizes area. This helps in decreasing delays and hence improves performance. However, it is really late in the design cycle to consider floor planning for the first time at the physical design phase. Today, floor planning is already considered at the logical phase of the design, as illustrated in Fig. 4.13. At the logic level the size of the modules is estimated, and this results in better chip compaction than when done only on the physical level. Floor planning is an essential design, step for hierarchical design methodology. With the emerging paradigm of core-based systems for an SOC, floor planning is considered even at the specification level of the product. Briefly speaking, core-based design relies on incorporating already designed modules, a DSP core, or an entire microprocessor. A floor plan is thus needed to decide about the location on the chip of the reusable cores and their relation to other components yet to be designed. In addition, floor planning is very important to the testing strategy, as outlined in Chapter 1 and illustrated in Fig. 1.15. Core-based design and testing is the topic of Chapter 15.

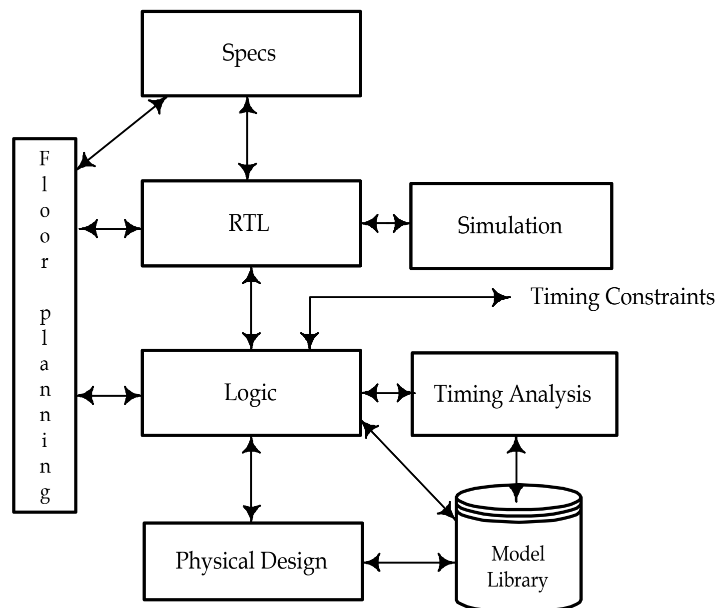

Figure 4.13 Another Perspective of IC Design FlowPlacement and routing (P&R) are very closely related processes. They can be defined as: Given a netlist describing the interconnections of the logic blocks and design rules, it is required to find (1) locations for the logic blocks (placement) and (2)

Copyright © 2005 Samiha Mourad

paths for all interconnections between blocks (Routing). Once the design is placed and routed, then it is important to resimulate it with the actual information about the circuit. For this a parameter extraction is done as indicated symbolically on the Y-chart in Fig. 4.2*b*. The information is used for optimization of P&R or even for resynthesis.

4.7.2 Placement

The placement task is concerned with finding a location on the chip floor for each cell obtained from the technology mapping. It is equivalent to floor planning under very specific circumstances: The exact size of the cells is known and the general organization of the cell is predetermined, say, in rows. The goal of placement is to decide about these locations such that the wiring length of all interconnects is as short as possible. This constraint allows smaller chip and propagation delays. To facilitate placement, the floor is organized in a grid consisting of placeholders, $P = \{p_1, p_2, \ldots p_N\}$ and objects, $M = \{m_1, m_2, \ldots, m_r\}$ to be placed in the holders. Each object, $m_i \in M$, is associated with a set of signals, S_i. For example, in Fig 4.14, we have eight gates (objects) and six nets to be placed. Each gate is connected to a number of these nets varying between one and three nets. Two different placements of these cells are also shown. How can one judge which of the two configurations is better? This is to be assessed by the cost function used by the placement algorithm. To optimize on the cost function, we need to consider all possible alternatives that are $N!/(N - r)!$ The time to accomplish the task will be of order $O(N!)$, where $N \geq r$. Placement is thus an NP-complete problem, and therefore a heuristic must be used. As we mentioned in Section 4.3, a heuristic will yield a near by optimal solution in a reasonable amount of time.

Copyright © 2005 Samiha Mourad

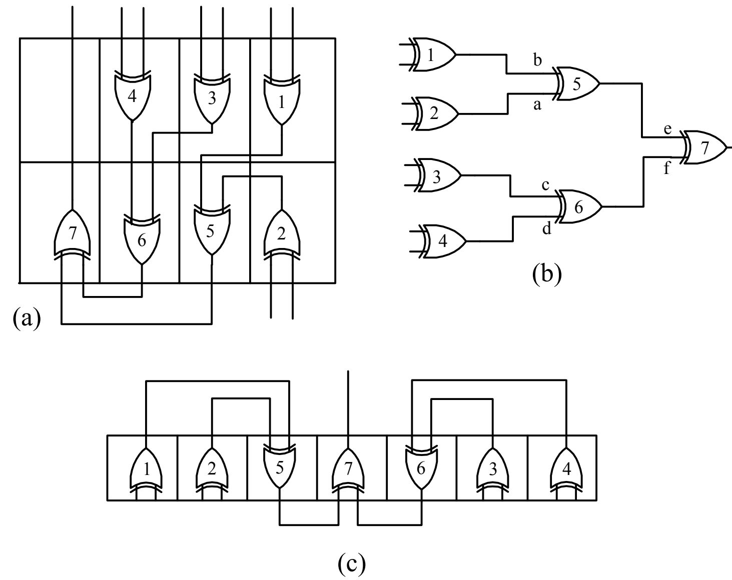

Figure 4.14 The Placement Configurations

Algorithms for placement are of two main types, *constructive* and *iterative improvement*. A commonsense way of doing constructive placement is first to place the most connected module in the middle of the space. Then proceed with one of the lower connectivities. Following this approach regarding the example in Fig. 4.14 resulted in arrangement shown in part (*a*). The other approach starts from an initial placement of the modules, which can be done in a random fashion, proceeds in an iterative improvement, and is performed until a final state is reached such that the wiring length is minimized.

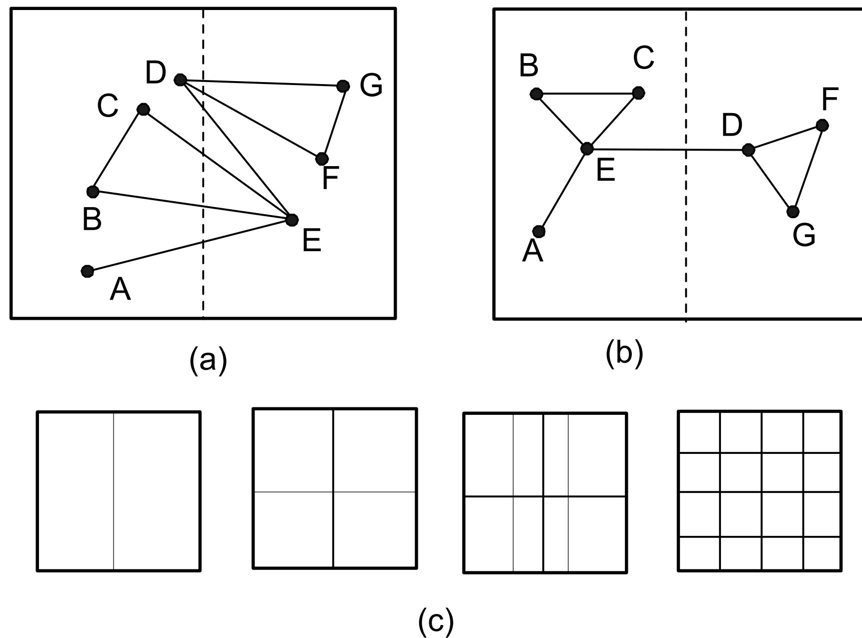

Figure 4.15 Placement: Mincut Algorithm a) Original Placement, b) Final Placement. c) Plan for Iterative Improvement

Copyright © 2005 Samiha Mourad

There are several possible heuristics for iterative improvement. We describe one of them, the min-cut [Lee 1961]. The algorithm operates on a graph model in which the vertices are the cells to be placed and the edges are the interconnect wires between the cell pins. With the initial placement shown in Fig. 4.15*a*, a min-cut approach to improving the placement will separate the floor area in two halves and use a cost function, the number of wires crossing the separation line. The effort of the algorithm is to move some of the vertices (cells) in such a way as to minimize the cost function. If a move of a component, j, results in a cost function $C_j < C_{j-1}$, the move is accepted; otherwise, the acceptance or rejection will depend on the type of strategy followed, as we discussed earlier in Section 3 of this chapter. The two halves of the design are each divided into two parts and the vertices are moved to locations that minimize the cost function. The process is repeated as illustrated in Fig. 4.15c until every object has its place on the floor and no further subdivision of the space is possible. For this example, moving nodes *D* and *E* reduces the cost function from 6 to 1 as illustrated in Fig. 4.15*b*.

4.7.3 Routing

Routing is the process of formally defining the precise conductor paths necessary to interconnect the cells so as to minimize the routing area or the delays. As for placement, routing starts with initial routing, which used to be called *loose routing* and later, *global routing*. This is then followed by a detailed routing which depends on the configuration: *channel routing* or *maze routing*. Before describing some of the routing algorithms, let us list some premises. A *net* is a collection of pins to be connected electrically. Figure 4.16 shows a channel with six pins: *A, B, C, D, E*, and *F*. The conductors connecting the pins are under some electrical constraints, such as a minimum conductor width, w, and minimum spacing between the conductors, s. Both parameters are a function of the technology used. For a generic technology, 2λ, assume that $w = s = 3\lambda$. It is usually convenient to combine the two parameters into a center-to-center spacing of $\delta = w + s$. This will be considered the grid of spacing. All wires will be running along equidistant vertical and horizontal lines of the grid as shown in the upper left corner of Fig. 4.16. All horizontal tracks will be in one plane which is different from that spanned by the vertical tracks. An immediate consequence of this grid is that length of the wire is measured in δ along the vertical and horizontal tracks. Thus two adjacent points along the horizontal or vertical track have a distance of δ. Points *A* and *B* are 4 δ apart, and so are the pair *D*, *E*. This is the actual shortest length of the wire to connect the two corresponding points. This distance is called the *Manhattan distance,* owing to the well-known street configuration of most of Manhattan, one of the New York City boroughs: avenues along the north – south direction and streets perpendicular along the east – west

Copyright © 2005 Samiha Mourad

direction. In addition, it is an easier way of measuring the distance than using the Euclidean distance, along the diagonal.

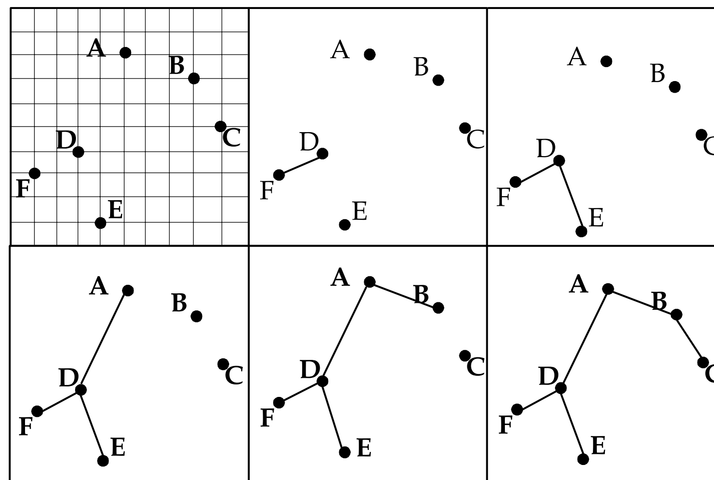

Figure 4.16 Global Routing

Refer to the example in Fig. 4.16, in which the six pins *A, B, C, D, E*, and *F*, of the same net are to be connected. The problem will be modeled with a graph, *G(V*, *E*, and *w*): *V* is the set of pins, *E* is the set of the nets connecting the pins, and w is the set of the distances between the pins. Since every pin is connected electrically to every other pin, the graph would be a complete graph. However, we know very well that if we connect pin *A* to pin *B* and *B* to *C*, then *A* is connected to *C* electrically without having an additional wire connecting these two pins. Thus what is needed is to figure out a tree, T (V_t, E_t, w_t), that spans all pins. In such a case, $V_t = V$, $E_t \subseteq E$, and w comprises the lengths on the edges in *E*. Since we need to minimize the interconnection wire length, it is desirable to obtain a *minimal spanning tree* (MST); that is, a tree for which the sum of the edges is minimal. There are several algorithms to find a minimal spanning tree [Kruskal 1956, Prim 1957]. The simplest algorithm will follow a greedy strategy. First, the edges in the set *E* are sorted in increasing order according to their distances (Manhattan). Start from a vertex that is connected to the shortest edge and place this edge in E_t. In our example, one such vertex is *F*; delete it from *V* and place it in V_t. Select from *E* the shortest edge that connects any vertex in V_t to a vertex in *V*. It is *D* in this case. Place it in V_t and delete it from *V*. Repeat the last step until *V* is empty. The total length of the edges in E_t is the minimal distance. The algorithm is illustrated step by step in Fig. 4.16. The last frame in the figure shows the MST.

Notice also that the global routing is not actually a layout of the wires connecting the pins but is just an

Copyright © 2005 Samiha Mourad

order in which the pins (vertices) will be connected. Once the global routing is completed for all nets, the detailed one follows. Consider connecting two pins that we label as *S* and *T* in Fig. 4.17. The shaded parts of the routing space are obstacles that cannot be used for connecting the two points. The detailed routing finds the specific path of connection. One of the main algorithms finds its roots in applications in operation research [Lee 1961]. Several variations of this algorithm have improved the detailed routing between the two points in such way as to minimize (1) the distance between the two points, (2) the number of turns in the route since, , a via is needed at each turn, and (3) the number of obstacles removed, as this means more space taken for routing another wire with ease.

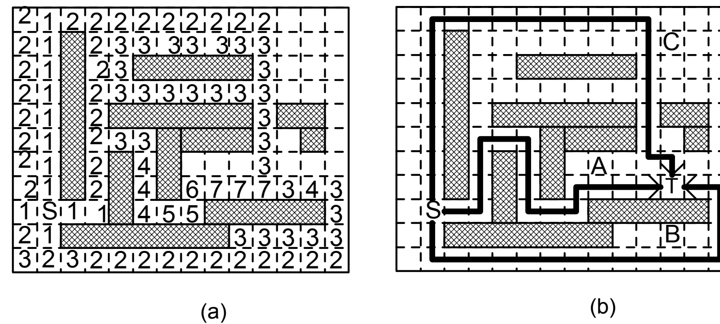

Figure 4.17 Maze Routing

As shown in Fig. 4.17, there are three routes, and a greedy approach would select route A, the shortest. However, there are more corners. Each turn is associated with a via that connects the two segments of wires that are usually on separate layers. Each via contributes to an increase in the resistance of the path. A large number of vias also increases the chance of defects such as opens, as discussed in Chapter 2. Similarly, net *A* can be connected to minimize the corners or to minimize resources, as illustrated in Fig. 4.18*a* and *b*, respectively. All these criteria are important, but they do not include the effects of routing on testing. For example, in the case where we conserved resources, net *A* is routed very closely to net *X*. If the two nets are long enough, their signals might interact and result in crosstalk noise, as mentioned at the end of Chapter 2. This can affect operation of the circuit for deep-submicron technology circuits.

Copyright © 2005 Samiha Mourad

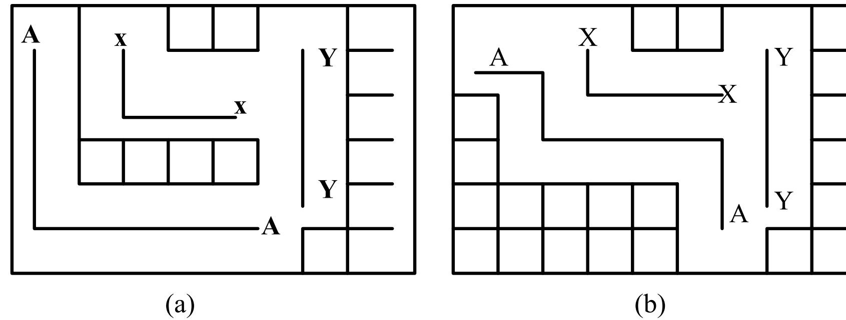

Figure 4.18 Routing Optimization for Noise (a) Space (b)

A concrete example of the importance of placement and routing in testing is the placement of flip-flops in design that includes the scan-path method. Scan-path design is one of the DFT constructs explained in Chapter 9. In this design, some or all of the flip-flops will be connected, at testing time, in one shift register. If placement is done without regard to these design goals, adjacent flip-flops in the same shift register might be placed far away from each other and in any order, causing very long interconnects between them. This affects adversely the performance of the design and deters designers from using this technique. Usually, the placement is completed without concern to this structure and it is important to route them for minimal path. Thus instead of finding the MST as was discussed above, it is important to find minimum spanning path. This problem is addressed in Chapter 9. The importance and complexity of the problem is well explained in [Beenker 1995].

It is preferable to include scan-path during synthesis (one-path design), instead of adding scan-path configuration after synthesis (two-stage design approach), and in this fashion, the P&R process will take testability into consideration in addition to area and performance. This second approach is discussed in Chapter 14.

Copyright © 2005 Samiha Mourad

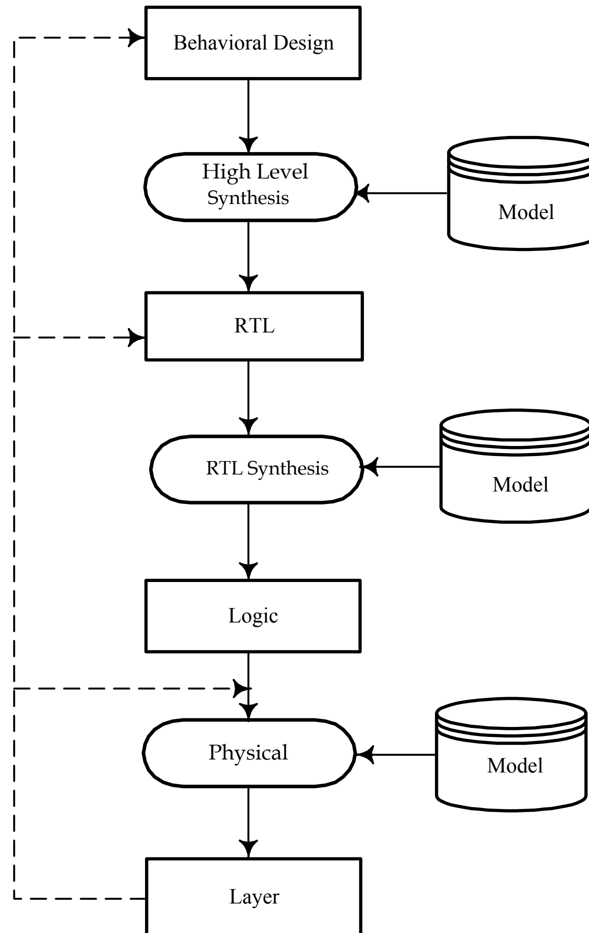

Figure 4.19 Backannotation

4.7.4 Back-Annotation

Back-annotation is the process of extracting information about the circuit from the layout. The layout of ICs, the circuit artwork, consists of a collection of polygons that are connected, overlapped on the same layer and different layers. A circuit extractor estimates various electrical parameters, such as resistance and inductance of lines, capacitances of nodes, and dimensions of the devices. This is accomplished in two phases. The first phase, called *netlist extraction*, consists of the identification of the devices, and the determination of electrically connected regions. The second phase consists of the estimation of the electrical parameters of the devices and the nets.

Although the shapes extracted are regular, they are of varied dimensions and make the extraction process very elaborate, and there are efforts to expedite their operation using parallel algorithms [Banerjee 1994]. In addition to planar capacitance, the extractor should have wall-to-wall capacitances, which are becoming

Copyright © 2005 Samiha Mourad

predominant in present submicron feature sizes. The annotated netlist represents the real circuit and, upon simulation, supplies the actual timing and delays. The information may be fed back to any of the synthesis stages as illustrated in Fig. 4.19. As a result, the circuit may be resynthesized or optimized to meet the specifications. CAD tools used for parameter extraction are usually called *technology CAD* (TCAD), and interfacing them with CAD tools for design and test is pursued vigorously at present.

References

Actel (1991), *ACT Family Field-Programmable Gate Array Databook*, Actel Corporation, Santa Clara, CA.

Altera (1992), *Applications Handbook*, Altera Corporation, San Jose, CA.

Banerjee, P. (1994), *Parallel Algorithms for VLSI computer-aided Design*, Prentice Hall, Upper Saddle River, NJ.

Bartlett, K. et al. (1985), Synthesis and optimization of multi-level logic minimization under time constraints, *IEEE Trans. on Computer-Aided Des.*, Vol. CAD-4, No. 4, pp. 582 – 596.

Beenker, F. P. M., R. G. Bennetts, and A. P. Thijssen (1995), *Testability concepts for Digital ICs*, Kluwer Academic, Norwell, MA.

Brayton, R. K. et al. (1984), *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic, Norwell, MA.

Brayton, R. K. et al. (1987), MIS: *A multiple-level logic optimization system*, *IEEE Trans. on Computer-Aided Des.*, Vol. CAD-6, No. 11, pp.1062 – 1081.

Bryant, R. E. (1986) Graph based algorithms for Boolean function manipulation, *IEEE Trans. on Comput.*, Vol. C-35 No. 8, pp. 677 – 691.

Chan, P. and S. Mourad (1994), *Logic Design with FPGAs*, Prentice-Hall, Upper Saddle River, NJ.

Cormen, C., C. E. Leiserson, and R. L. Rivest (1992), *Introduction to Algorithms*, MIT Press, Cambridge, MA.

Fujiwara, H. and K. Kinoshita (1981), A design of programmable logic arrays with universal tests, *IEEE Trans. on Comput.,* Vol. 30, No. 11, pp. 823 – 828.

Fujiwara, H. (1982), The complexity of Fault Detection for Combinational Logic Circuits, *IEEE Trans. Comput.*, Vol. C-31 No. 6, pp. 555 – 560.

Fujiwara, H. (1986), *Logic Testing and Design for Testability*, MIT Press, Cambridge, MA.

Fujiwara, H. (1984), A new PLA design for universal testability, *IEEE Trans. Comput.*, Vol. C-33, No. 8, pp. 745 - 50.

Gajski, D. D. and R. H. Kuhn, (1983), Introduction: new VLSI tools, *IEEE Computer*, Vol. 6, No.12, pp. 11 – 14.

Goel, P. (1980), Test generation cost analysis and projections, *Proc. 17th Design Automation Conference*, pp. 77 – 84.

Hong, S. J., R. G. Cain, and D .L. Ostapko (1974), MINI: a heuristic approach for logic minimization, *IBM J.l of Res.h and Dev.t*, Vol. 18, pp. 443 – 458.

Copyright © 2005 Samiha Mourad

Hong, S. J. and D. L. Ostapko (1980), FITPLA: A programmable logic array for functional independent testing, *Digest 10th International Symposium on Fault-Tolerant Computing*, pp. 131 –136.

Ibarra, G. H. and S. K. Sahni (1975), Polynomially complete fault detection problems, *IEEE Trans. Comput.*, Vol. C-24, No.3, pp. 242 – 249.

Kruskal, J. B. (1956), On the shortest spanning subtree of a graph and the travelling salesman problem, *Proc. of AMS*, Vol. 1 No. 1, pp. 48 – 50.

Lee C. Y. (1961), An algorithm for path connections and its applications, *IRE Trans. on Elect. Comput.,* Vol. EC 10,No.3, pp. 346 – 365.

Michel, P., U. Lauther, and P. Duzy, Eds. (1992), *The Synthesis approach to digital system design*, Kluwer Academic, Norwell, MA.

Prim, R. C. (1957), Shortest connection networks and some generalizations, *Bell Sys. Tech.l J.*, Vol. 36, No.6, pp. 1389 – 1401.

Quicklogic (1993), *Very High-Speed Programmable ASIC*, Quicklogic, Santa Clara, CA.

Walker, R. A. and D. E. Thomas (1985), A model of design representation and synthesis, *Proc. 22nd Des. Automation Conference*, pp. 453 – 459.

Xilinx (1985), *The XC*4000 *Data Book*, Xilinx, Inc, San Jose, CA.

Problems

4.1 What is the order of complexity of the multiplication of three matrices of order N x M, M x K, and K x L?

4.2 Consider four algorithms with time complexities n, n^3, 2^n, and $n!$, respectively. The time taken to solve a problem of size 10 by the first algorithm is 10 ms. What is the time needed for the other algorithms? What are these times for a problem of size 100?

4.3 Suppose that you are to design a circuit from a behavioral description given in the form of Boolean equations. Outline on the Y-chart in Fig. 3.1 various processes to accomplish your design.

4.4 Change the factored Boolean function given in Section 4.4.2 and illustrated in Fig. 4.6 to a sum of products and determine the number of literals of the obtained expression.

4.5 Put the steps used to develop the MST in Section 4.7.3 in algorithmic form.

4.6 You are to determine a global route for the different components shown in Fig.P4.6. Develop a minimal spanning tree. Is this tree unique? Justify your answer.

4.7 Modify the algorithm developed in Problem 4.6 to obtain the shortest path instead of the spanning tree.

4.8 In which process of the DFT cycle would you recommend that care needs to be taken to avoid crosstalk noise?

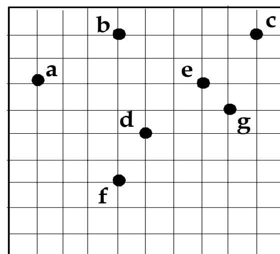

Figure P4.6

Copyright © 2005 Samiha Mourad