CHAPTER 1
at a glance

- Introduction to Genetic Algorithms (GAs)
- GA terminology
- Genetic operators
  - Crossover
  - Mutation
  - Inversion
- EDA problems solved by GAs

Chapter 1

INTRODUCTION

The Genetic Algorithm (GA) was invented by Prof. John Holland [95] at the University of Michigan in 1975, and subsequently it has been made widely popular by Prof. David Goldberg [79] at the University of Illinois. The original GA and its many variants, collectively known as genetic algorithms, are computational procedures that mimic the natural process of evolution. The theories of evolution and *natural selection* were first proposed by Darwin to explain his observations of plants and animals in the natural world [48]. Darwin observed that, as variations are introduced into a population with each new generation, the less-fit individuals tend to die off in the competition for food, and this *survival of the fittest* principle leads to improvements in the species. The concept of natural selection was used to explain how species have been able to adapt to changing environments and how, consequently, species that are very similar in adaptivity may have evolved.

Much has been learned about genetics since the time of Charles Darwin. All information required for the creation of appearance and behavioral features of a living organism is contained in its chromosomes. Reproduction generally involves two parents, and the chromosomes of the offspring are generated from portions of chromosomes taken from the parents. In this way, the offspring inherit a combination of characteristics from their parents. GAs attempt to use a similar method of inheritance to solve various problems, such as those involving adaptive systems [95]. GAs have also been applied to optimization problems [54], and the applications that we will address in this book fall into this category. The objective of the GA is then to find an optimal solution to a problem. Of course, since GAs are heuristic procedures, they are not guaranteed to find the optimum, but experience has shown that they are able to find very good solutions for a wide range of problems.

GAs work by evolving a population of individuals over a number of generations. A fitness value is assigned to each individual in the population, where the fitness computation depends on the application. For each generation, individuals are selected from the population for reproduction, the individuals are *crossed* to generate new individuals, and the new individuals are mutated with some low mutation probability. The new individuals may completely replace the old individuals

in the population, with distinct generations evolved [79]. Alternatively, the new individuals may be combined with the old individuals in the population [95]. In this case, we may want to reduce the population in order to maintain a constant size, e.g., by selecting the best individuals from the population. Both of these approaches have been used for applications that will be described in this book, and both approaches have yielded good results. The choice of which approach is used may depend on the application. Since selection is biased toward more highly fit individuals, the average fitness of the population tends to improve from one generation to the next. The fitness of the best individual is also expected to improve over time, and the best individual may be chosen as a solution after several generations.

GAs use two basic processes from evolution: *inheritance*, or the passing of features from one generation to the next, and competition, or *survival of the fittest*, which results in weeding out the bad features from individuals in the population. The main advantages of GAs are:

- they are adaptive, and learn from experience;
- they have intrinsic parallelism;
- they are efficient for complex problems; and
- they are easy to parallelize, even on a loosely coupled Network Of Workstations (popularly known as NOW), without much communication overhead.

Various different representations and operations have been used in genetic algorithms, and many of these variations are described in detail in existing texts [79], [95]. Our goal in this chapter is to provide an overview of the approaches and operations that we have found to be useful for problems in VLSI design and test automation. We begin with a quick introduction to GA terminology. The two basic GA approaches are presented next, first the *simple GA*, also called the *total replacement algorithm*, and then the *steady-state algorithm*, which is characterized by overlapping populations. Detailed descriptions of various genetic operators used are given next, followed by an example illustrating the use of a GA for test generation. We conclude with a brief discussion about the application of GAs to problems in VLSI design and test automation.

1.1 Introduction to GA Terminology

All genetic algorithms work on a *population*, or a collection of several alternative solutions to the given problem. Each individual in the population is called a *string* or *chromosome*, in analogy to chromosomes in natural systems. Often these individuals are coded as binary strings, and the individual characters or symbols in the strings are referred to as *genes*. In each iteration of the GA, a new *generation* is evolved from the existing population in an attempt to obtain better solutions.

The *population size* determines the amount of information stored by the GA. The GA population is evolved over a number of generations.

An *evaluation* function (or fitness function) is used to determine the *fitness* of each candidate solution. The fitness is the opposite of what is generally known as the *cost* in optimization problems. It is customary to describe genetic algorithms in terms of fitness rather than cost. The evaluation function is usually user-defined, and problem-specific.

Individuals are *selected* from the population for reproduction, with the selection biased toward more highly fit individuals. Selection is one of the key operators on GAs that ensures survival of the fittest. The selected individuals form pairs, called *parents*.

Crossover is the main operator used for reproduction. It combines portions of two parents to create two new individuals, called *offspring*, which inherit a combination of the features of the parents. For each pair of parents, crossover is performed with a high probability P_C, which is called the *crossover probability*. With probability $1 - P_C$, crossover is not performed, and the offspring pair is the same as the parent pair.

Mutation is an incremental change made to each member of the population, with a very small probability. Mutation enables new features to be introduced into a population. It is performed probabilistically such that the probability of a change in each gene is defined as the *mutation probability*, P_M.

Inversion is a genetic operator which does not change the solution represented by the chromosome, but rather changes the chromosome itself, or the (binary) representation of the solution. The *inversion probability* is denoted by P_I.

The *generation gap* is the fraction of individuals in the population that are replaced from one generation to the next and is equal to 1 for the simple GA.

A *schema* is a specific set of values assigned to a subset of the genes in a chromosome. It is a partial solution and represents a set of possible fully specified solutions. For example, consider a cell placement problem, in which the solutions are encoded as triples consisting of the cell identifiers and their assigned x- and y-coordinates. A schema in this context represents a subplacement in which a subset of the cells are assigned to specific positions. The remaining cells may be placed in various different positions. A schema with m specified elements and *don't-cares* in the rest of the $n - m$ positions (such as an m-cell subplacement in an n-cell placement problem) can be considered to be an $(n - m)$-dimensional hyperplane in the solution space. All points on that hyperplane (i.e., all individuals that contain the given subplacement) are *instances* of the schema. Note here that a subplacement does not have to contain physically adjacent cells, such as a rectangular patch of the chip area.

For a given problem, various genes may be linked, and specific values may be required for groups of genes in order to obtain a good solution. These *schemata* represent the various features of the candidate solutions. GAs implicitly operate upon the various schemata in parallel, which is why they are so successful in solving complex optimization problems. The genetic operators create a new generation of individuals by combining the schemata of parents selected from the current generation. Due to the stochastic selection process, the fitter parents, which are expected

to contain some good schemata, are likely to produce more offspring. At the same time, the bad parents, which contain some bad schemata, are likely to produce fewer offspring. Thus, in the next generation, the number of instances of good schemata tends to increase, and the number of instances of bad schemata tends to decrease. The fitness of the entire population is therefore improved.

In a typical, binary-coded GA, where the chromosomes are bit strings, each string in the population is an instance of 2^L schemata, where L is the length of each individual string. The number of different strings or possible solutions to the problem is also 2^L, and the total number of different schemata contained in all these strings is 3^L, since each gene in a schema may be 0, 1, or don't care, x. Thus, the population represents a very large number of schemata, even for relatively small population sizes. By evaluating a new offspring, we get a rough estimate of the fitness of all of its schemata. The numbers of these schemata present in the population is thus adjusted according to their relative fitness values. This effect is known as the intrinsic parallelism of the GA. As more individuals are evaluated, the relative proportions of the various schemata in the population reflect their fitness values more and more accurately. When a better schema is introduced into the population through one offspring, it is inherited by others in the succeeding generation, and thus, its proportion in the population increases. It starts driving out the less fit schemata, and the average fitness of the population improves.

1.2 The Simple GA

The simple GA (also referred to as the total replacement algorithm) is illustrated in Fig. 1.1 and also in Fig. 1.2 in flowchart form. The simple GA is composed of

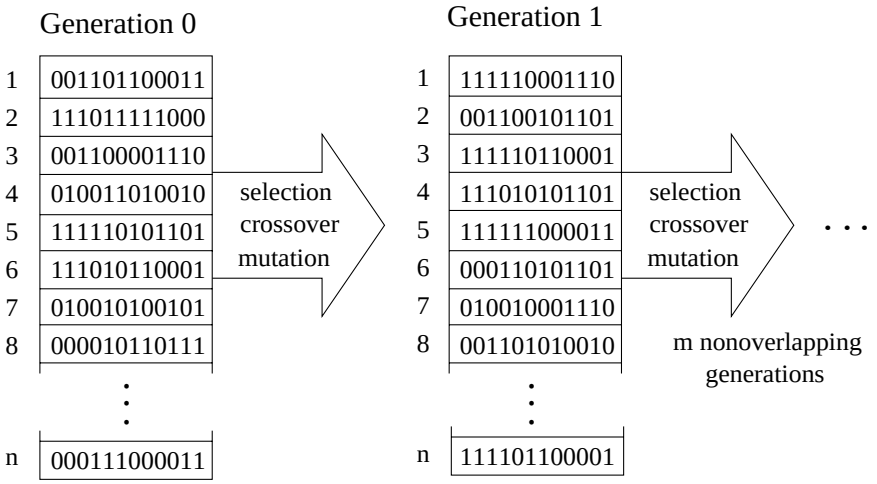


Figure 1.1. The simple genetic algorithm

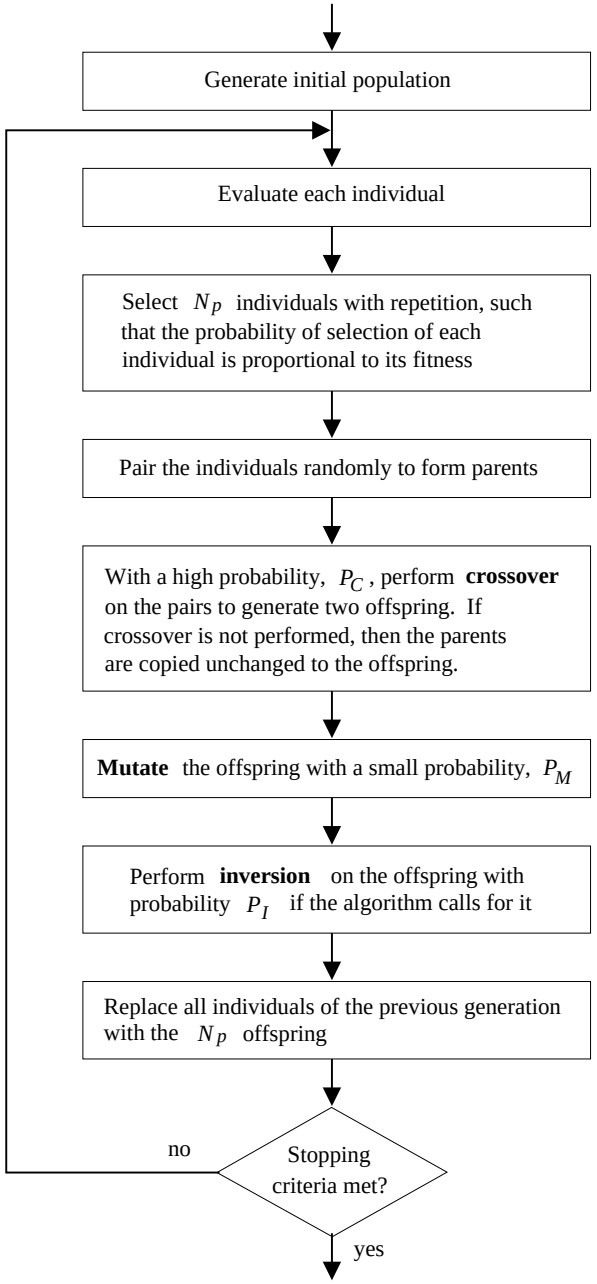


Figure 1.2. Flowchart of the simple genetic algorithm

populations of strings, or *chromosomes*, and three evolutionary operators: *selection, crossover,* and *mutation* [79]. The chromosomes may be binary-coded, or they may contain characters from a larger alphabet [69], [80]. Each chromosome is an encoding of a solution to the problem at hand, and each individual has an associated fitness which depends on the application. The initial population is typically generated randomly, but it may also be supplied by the user. A highly fit population is evolved through several generations by *selecting* two individuals, *crossing* the two individuals to generate two new individuals, and *mutating* characters in the new individuals with a given mutation probability. Selection is done probabilistically but is biased toward more highly fit individuals, and the population is essentially maintained as an unordered set. Distinct generations are evolved, and the processes of selection, crossover, and mutation are repeated until all entries in a new generation are filled. Then the old generation may be discarded. New generations are evolved until some stopping criterion is met. The GA may be limited to a fixed number of generations, or it may be terminated when all individuals in the population converge to the same string or no improvements in fitness values are found after a given number of generations. Since selection is biased toward more highly fit individuals, the fitness of the overall population is expected to increase in successive generations. However, the best individual may appear in any generation.

1.3 The Steady-State Algorithm

In a GA having overlapping generations, only a fraction of the individuals are replaced in each generation [55], [236]. The steady-state algorithm is illustrated in Fig. 1.3. In each generation, two *different* individuals are selected as parents, based on their fitness. Crossover is performed with a high probability, P_C, to form offspring. The offspring are mutated with a low probability, P_M and inverted with probability P_I, if necessary. A duplicate check may follow, in which the offspring are rejected without any evaluation if they are duplicates of some chromosomes already in the population. The offspring that survive the duplicate check are evaluated and are introduced into the population only if they are better than the current worst member of the population, in which case the offspring replaces the worst member. This completes the generation. In the steady-state GA, the generation gap is minimal, since only two offspring are produced in each generation.

Duplicate checking may be beneficial because a finite population can hold more schemata if the population members are not duplicated. Since the offspring of two identical parents are identical to the parents, once a duplicate individual enters the population, it tends to produce more duplicates and individuals varying by only slight mutations. Premature convergence may then result. Duplicate checking is advantageous under the following conditions:

- the population size is small;
- the chromosomes are short; or
- the evaluation time is large.

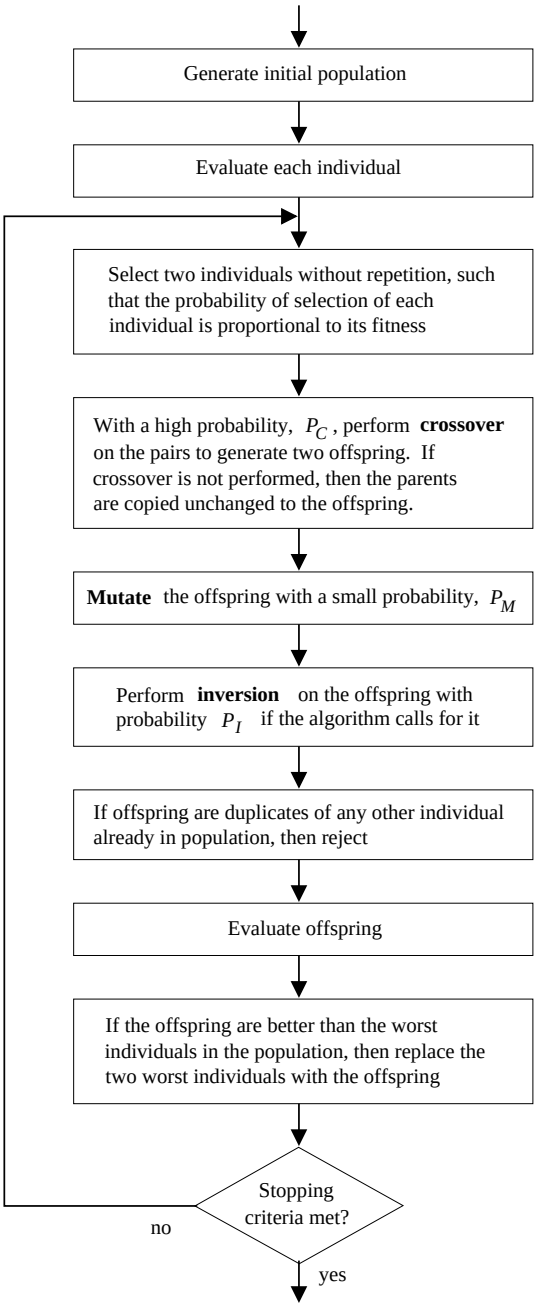
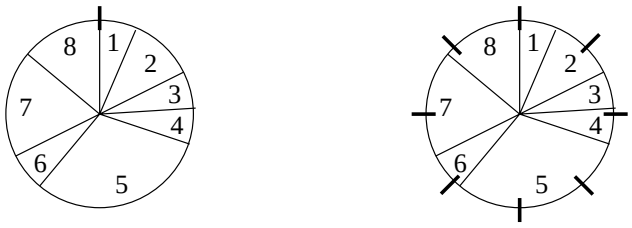


Figure 1.3. Steady-state genetic algorithm



(a) Roulette Wheel Selection (b) Stochastic Universal Selection

Figure 1.4. Proportionate selection schemes

Each of the above conditions reduces the duplicate checking time in comparison to the evaluation time. If the duplicate checking time is negligible compared to the evaluation time, then duplicate checking improves the efficiency of the GA.

The steady-state GA is susceptible to stagnation. Since a large majority of offspring are inferior, the steady-state algorithm rejects them, and it keeps making more trials on the existing population for very long periods of time without any gain. Because the population size is small compared to the search space, this is equivalent to long periods of localized search.

1.4 Genetic Operators

The genetic operators and their significance can now be explained. The description will be in terms of a traditional GA without any problem-specific modifications. The operators that will be discussed include selection, crossover, mutation, fitness scaling, and inversion.

1.4.1 Selection

Various selection schemes have been used, but we will focus on *roulette wheel selection, stochastic universal selection,* and binary *tournament selection* with and without replacement [13], [81]. As illustrated in Fig. 1.4(a), roulette wheel selection is a proportionate selection scheme in which the slots of a roulette wheel are sized according to the fitness of each individual in the population. An individual is selected by spinning the roulette wheel and noting the position of the marker. The probability of selecting an individual is therefore proportional to its fitness. As illustrated in Fig. 1.4(b), stochastic universal selection is a less noisy version of roulette wheel selection in which N equidistant markers are placed around the roulette wheel, where N is the number of individuals in the population. N individuals are selected in a single spin of the roulette wheel, and the number of copies of each individual selected is equal to the number of markers inside the corresponding slot. In binary tournament selection, two individuals are taken at random, and the

better individual is selected from the two. If binary tournament selection is being done without replacement, then the two individuals are set aside for the next selection operation, and they are not replaced into the population. Since two individuals are removed from the population for every individual selected, and the population size remains constant from one generation to the next, the original population is restored after the new population is half-filled. Therefore, the best individual will be selected twice, and the worst individual will not be selected at all. The number of copies selected of any other individual cannot be predicted except that it is either zero, one, or two. In binary tournament selection with replacement, the two individuals are immediately replaced into the population for the next selection operation.

The objective of the GA is to converge to an optimal individual, and *selection pressure* is the driving force which determines the rate of convergence. A high selection pressure will cause the population to converge quickly, possibly at the expense of a suboptimal result. Roulette wheel selection typically provides the highest selection pressure in the initial generations, especially when a few individuals have significantly higher fitness values than other individuals. Tournament selection provides more pressure in later generations when the fitness values of individuals are not significantly different. Thus, roulette wheel selection is more likely to converge to a suboptimal result if individuals have large variations in fitness values.

1.4.2 Crossover

Once two chromosomes are selected, the *crossover* operator is used to generate two offspring. In *one-* and *two-point crossover,* one or two chromosome positions are randomly selected between one and $(L-1)$, where L is the chromosome length, and the two parents are crossed at those points. For example, in one-point crossover, the first child is identical to the first parent up to the crossing point and identical to the second parent after the crossing point. An example of one-point crossover is shown in Fig. 1.5. In *uniform crossover,* each chromosome position is crossed with some probability, typically one-half.

Parent 1:	*1 0 1 1 0 1 1 0 1*	*1 1 1 0 0 1 1 0 0*
Parent 2:	0 0 1 1 0 1 1 0 0	1 0 0 1 0 1 0 0 0
Offspring 1:	*1 0 1 1 0 1 1 0 1*	1 0 0 1 0 1 0 0 0
Offspring 2:	0 0 1 1 0 1 1 0 0	*1 1 1 0 0 1 1 0 0*

Figure 1.5. One-point crossover

Crossover combines the schemata or building blocks from two different solutions in various combinations. Smaller good building blocks are converted into progres-

sively larger good building blocks over time until we have an entire *good* solution. Crossover is a random process, and the same process results in the combination of bad building blocks to result in poor offspring, but these are eliminated by the selection operator in the next generation. An example of this progressive growth of schemata is given in Fig. 1.6. In the figure, let us assume that the fitness function is such that consecutive strings of 1's result in a higher fitness. Thus, schemata of the form xx111xx are good. The figure shows the offspring from one crossover operation taking part in further crossovers. Although the sequence of events shown may occur over many generations, it illustrates how strings of consecutive 1's can be gradually accumulated.

	Chromosome	Schema
Parent 1:	*01011* | *0110100*	xxx*11*xxxxxxx
Parent 2:	00110 | 1110010	xxxxx111xxxx
Offspring:	*01011* | 1110010	xxx*11*111xxxx
Parent 1:	*01011111* | *0010*	xxx*1111*xxxx
Parent 2:	10011000 | 1101	xxxxxxxx11xx
Offspring:	*01011111* | 1101	xxx*1111*11xx

Figure 1.6. Advantage of crossover: Schema growth

At the same time, schema destruction also results from crossover, as illustrated in Fig. 1.7. Any schemata spanning the cut point of the crossover operation are destroyed; i.e., they are not inherited by either of the offspring. This destruction introduces new random schemata instead. The destruction probability of a schema is proportional to its length. Since long schemata span a large part of the solution string, they are cut by any random cut point with a high probability. Holland's schema theorem analyzes these two processes and concludes that the net result is a gain of good schemata [95].

	Chromosome	Schema
Parent 1:	*11010* | *0010011*	*11*xxxxxxxx*11*
Parent 2:	10110 | 1110000	
Offspring:	*11010* | 1110000	

Figure 1.7. Destruction of long schemata during crossover

The end result of crossover is that schemata from the two parents are combined. If the best schemata from the parents can be combined, the resulting offspring may be even better than the two parents. Since, highly fit individuals are more likely to be selected as parents, the GA examines more candidate solutions in good regions of the search space and fewer candidate solutions in other regions. The optimum

solution may therefore be discovered relatively quickly. The performance of the GA depends to a great extent on the performance of the crossover operator used.

The amount of crossover is controlled by the *crossover probability*, which is defined as the ratio of the number of offspring produced in each generation to the population size. A higher crossover probability allows exploration of more of the solution space and reduces the chances of settling for a false optimum. A lower crossover probability enables exploitation of existing individuals in the population that have relatively high fitness.

1.4.3 Mutation

As new individuals are generated, each character is mutated with a given probability. In a binary-coded GA, mutation may be done by flipping a bit, while in a nonbinary-coded GA, mutation involves randomly generating a new character in a specified position. Mutation produces incremental random changes in the offspring generated through crossover, as shown in Fig. 1.8. When used by itself, without any crossover, mutation is equivalent to random search, consisting of incremental random modification of the existing solution, and acceptance if there is improvement. However, when used in the GA, its behavior changes radically. In the GA, mutation serves the crucial role of replacing the gene values lost from the population during the selection process so that they can be tried in a new context, or of providing the gene values that were not present in the initial population.

Before Mutation:	110100010011
After Mutation:	110000010011

Figure 1.8. Mutation operator

For example, say a particular bit position, bit 10, has the same value, say 0, for all individuals in the population. In such a case, crossover alone will not help, because it is only an inheritance mechanism for existing gene values. That is, crossover cannot create an individual with a value of 1 for bit 10, since it is 0 in all parents. If a value of 0 for bit 10 turns out to be suboptimal, then, without the mutation operator, the algorithm will have no chance of finding the best solution. The mutation operator, by producing random changes, provides a small probability that a 1 will be reintroduced in bit 10 of some chromosome. If this results in an improvement in fitness, then the selection algorithm will multiply this chromosome, and the crossover operator will distribute the 1 to other offspring. Thus, mutation makes the entire search space reachable, despite a finite population size. Although the crossover operator is the most efficient search mechanism, by itself, it does not guarantee the reachability of the entire search space with a finite population size. Mutation fills in this gap.

The mutation probability P_M is defined as the probability of mutating each gene. It controls the rate at which new gene values are introduced into the population. If it is too low, many gene values that would have been useful are never tried out. If it is too high, too much random perturbation will occur, and the offspring will lose their resemblance to the parents. The ability of the algorithm to learn from the history of the search will therefore be lost.

1.4.4 Fitness Scaling

If a proportionate selection scheme is used, the GA selects individuals with probability proportional to their fitness. If the raw fitness values are used for this probabilistic selection, without any scaling or normalization, then one of two things can happen. If the fitness values are too far apart, then it will select several copies of the good individuals, and many other *worse* individuals will not be selected at all. This will tend to fill the entire population with very similar chromosomes and will limit the ability of the GA to explore large amounts of the search space. On the other hand, if the fitness values are too close to each other, then the GA will tend to select one copy of each individual, with only random variations in selection. Consequently, it will not be guided by small fitness variations and will be reduced to random search. Fitness scaling is used to scale the raw fitness values so that the GA sees a *reasonable* amount of difference in the scaled fitness values of the best versus the worst individuals. Thus, fitness scaling controls the *selection pressure* or discriminating power of the GA.

The following fitness scaling algorithm applies to evaluation functions that determine the *cost*, rather than the *fitness*, of each individual. From this cost, the fitness of each individual is determined by scaling as follows.

A reference *worst cost* is determined by

$$C_w = \overline{C} + S\sigma$$

where $\overline{C}$ is the average cost of the population, S is the user-defined sigma scaling factor, and σ is the standard deviation of the cost of the population. In case C_w is less than the real worst cost in the population, then only the individuals with cost lower than C_w are allowed to participate in the crossover.

The fitness of each individual is determined by

$$F = \begin{cases} C_w - C & \text{if } C_w > C \\ 0 & \text{otherwise.} \end{cases}$$

This scales the fitness such that, if the cost is $\pm k$ standard deviations from the population average, the fitness is

$$F = (S \pm k)\sigma.$$

This means that any individuals worse than S standard deviations from the population mean ($k = S$) are not selected at all. The usual value of S reported in the literature is between 1 and 5.

The cost vs. fitness is illustrated for three possible cost distributions in Fig. 1.9. If S is small, the ratio of the lowest to the highest fitness in the population increases, and the algorithm becomes more selective in choosing parents (Fig. 1.9(a)).

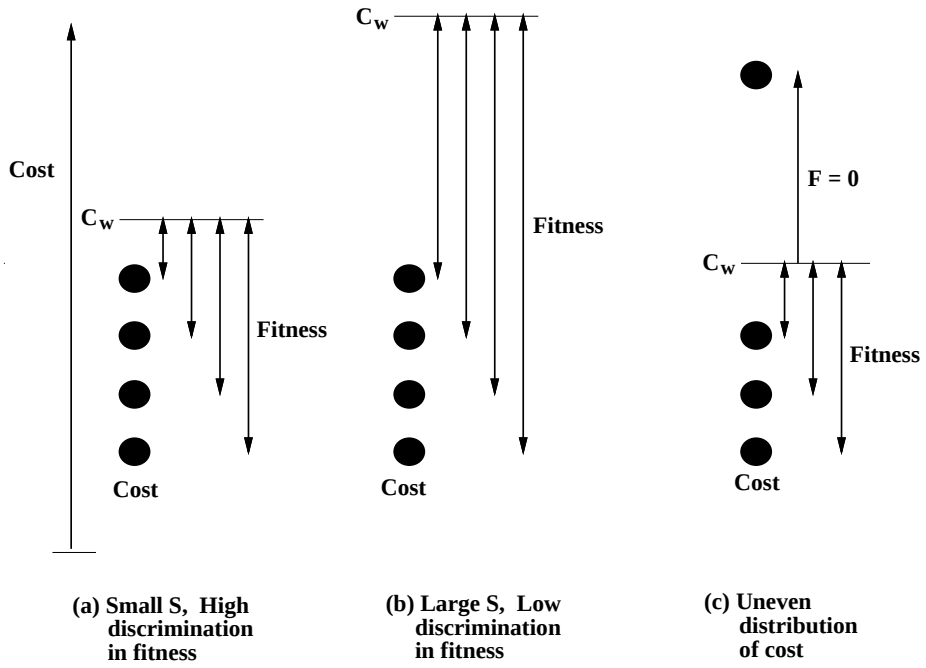


Figure 1.9. Effect of sigma scaling on fitness

On the other hand, if S is large, then C_w is large, and the fitness values of the members of the population are relatively close to each other (Fig. 1.9(b)). This causes the difference in selection probabilities to decrease and the algorithm to be less selective in choosing parents. Fig. 1.9(c) shows a case where one individual is much worse than the rest of the population. The fitness of this individual is set to zero, and the algorithm still has good discrimination between the rest of the individuals.

An alternative way to control the selection process is to use *ranking* [79]. Individuals are sorted according to their fitness values, and the number of copies of each individual selected for the next generation depends on its rank. Ranking is used in the GAs for macro cell routing, as will be described in Chapter 4.

1.4.5 Inversion

The inversion operator takes a random segment in a solution string and inverts it end for end (Fig. 1.10). This operation is performed in a way such that it does

not modify the solution represented by the string. Instead, it only modifies the *representation* of the solution. Thus, the symbols composing the string must have an interpretation independent of their position. This can be achieved by associating an identification number with each symbol in the string and interpreting the string with respect to these identification numbers instead of the array indices. When a symbol is moved in the array, its identification number is moved with it, and therefore, the interpretation of the symbol remains unchanged.

For example, Fig. 1.10 shows a chromosome. Let us assume a very simple evaluation function such that the fitness is the binary number consisting of all bits of the chromosome, with bit 0 being the least significant, and bit-9 being the most significant. Since the bit identification numbers are moved with the bit values during the inversion operation, bit 0, bit 1, etc., still have the same values, although their sequence in the chromosome is different. Hence, the fitness value remains the same. The inversion probability is the probability of performing inversion on each individual during each generation. It controls the amount of group formation.

Before	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
Inversion:	1	1	0	0	0	0	1	1	0	0
After	B_9	B_8	B_7	B_2	B_3	B_4	B_5	B_6	B_1	B_0
Inversion:	1	1	0	1	1	0	0	0	0	0

Figure 1.10. Inversion operator

Inversion changes the sequence of genes randomly, in the hope of discovering a sequence of linked genes placed close to each other. That is, for long schemata consisting of many *don't-cares* in the middle, it tries to resequence the genes to bring them to neighboring locations. This operation, illustrated in Fig. 1.11, has the effect of shortening the schema length. Longer schemata have a higher probability of being cut and destroyed by the one-point crossover operator, and hence, long schemata with *don't-cares* embedded will not be explored efficiently. By reducing their lengths, and grouping genes of a schema in close proximity in the chromosome, the inversion operator greatly improves the efficiency of the one-point crossover operator.

Before Inversion:	11xxxx11xx
After Inversion:	11x11xxxxx

Figure 1.11. Shortening of schema by inversion

However, inversion is a random process, and where there is a probability of reducing schema length, there is also a probability of increasing it. Such offspring with increased schema length will prove to be less fit in the long run and will be weeded out by the selection algorithm. Hence, the useful effect of inversion will

dominate. If the genes are arranged in the chromosome such that related genes are close together from the beginning, then the good schemata will be short without the need for inversion. Such a genetic coding will be highly beneficial. In problems where such a genetic coding can be derived, the inversion operator is not used.

1.5 GA Example

Many of the optimization problems encountered in VLSI design, layout, and test automation can be solved with high-quality results using genetic algorithms. Let us consider the problem of test generation for a very simple circuit and observe how a GA behaves when applied to this problem. Suppose that we want to generate a test to detect a fault that causes node Y to be stuck at logic 0 (denoted Y s-a-0) in Fig. 1.12. To excite the fault, node Y must be driven to logic 1 in the fault-free

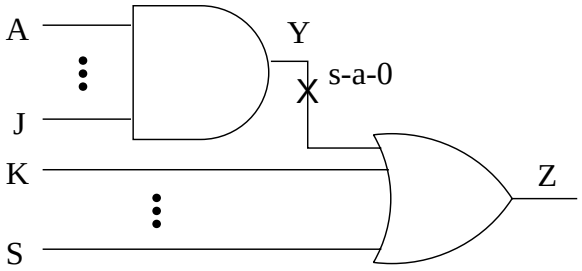


Figure 1.12. GA-based test generation example

circuit, so that there will be a difference between the fault-free circuit and the faulty circuit. Nodes A through J must therefore be set to 1. To propagate the fault effects to output node Z, nodes K through S must be set to 0.

The required test vector is nearly impossible to find using a random approach. Specific values are required on 19 circuit inputs, and it is very unlikely that the correct combination will be generated randomly. Consider how the problem could be solved with a GA. If the GA fitness function simply indicates whether or not the fault is detected, the test is very unlikely to be found. However, if the fitness function favors setting nodes A through J to 1 and setting nodes K through S to 0, the test has a good chance of being found.

A simple GA was implemented with the following fitness function to solve this problem:

$$\begin{aligned}\text{fitness} = \;& (1 \text{ if fault is excited or } 0 \text{ otherwise}) + \\ & (\text{fraction of inputs A–J set to } 1) + \\ & (\text{fraction of inputs K–S set to } 0)\end{aligned}$$

A population size of 32 was used, and the GA was run for 16 generations. Tournament selection, uniform crossover, and crossover and mutation probabilities of 1.0

Table 1.1. Best Individual by Generation for GA Example

Generation	Vector	Fitness
0	0011010111000001000	1.5
1	0101011111000000000	1.7
2	1111111111101000000	2.8
3	1111111111100001100	2.7
4	1111111111000000011	2.8
5	1111111110000000000	1.9
6	1101111011000000000	1.8
7	1111111111000000000	3.0

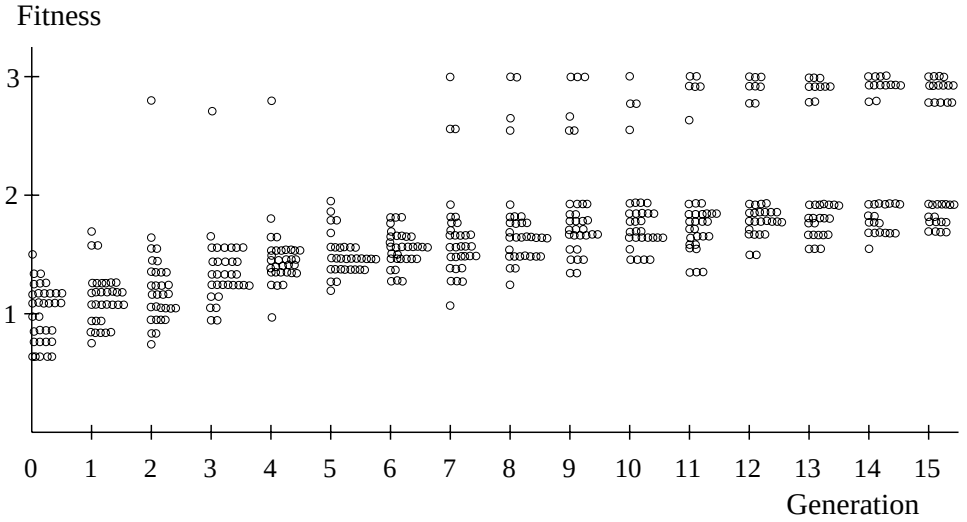


Figure 1.13. Fitness distribution by generation for GA example

and 0.03 were used. The best individuals found in the first eight generations are shown in Table 1.1, and the distribution of individuals in each generation is shown in Fig. 1.13. The GA finds some good individuals as early as generation 2, but the best individuals are lost in generation 5. The solution is finally found in generation 7. For this circuit, 256 individuals must be evaluated in order to find the solution; this is far fewer than the number of random vectors that would be required. The average fitness of the population gradually improves from 1.0 in generation 0 to 2.3 in generation 15. Once the solution is found in generation 7, it appears in every subsequent generation, and by generation 15, it is replicated four times in the population.

The actual fitness distribution by generation for a given GA will depend on a number of factors, including type of GA, selection and crossover schemes, GA parameters, and fitness function used. A properly designed GA is expected to converge to the solution if it is allowed to run for a sufficient number of generations. Whether or not the GA is allowed to run until convergence depends on the application. If solution quality is critical, the GA should be allowed to run for a large number of generations. On the other hand, if execution time is a limiting factor, a suboptimal solution may be selected after only a small number of generations.

1.6 GAs for VLSI Design, Layout, and Test Automation

Due to the complexity of VLSI designs, Electronic Design Automation (EDA) tools have become a necessity. EDA tools exist to automate the design process so that even novice designers can produce good quality designs in a reasonable amount of time. One example is an automatic placement and routing package. Other EDA tools simply facilitate the design process, leaving the important decisions up to the designer. For example, a full custom layout tool enables the designer to specify the exact placement of transistors and metal interconnections for a design, but the mundane tasks of generating the actual mask specifications for wafer fabrication are left to the tool. Even when the automatic tools are used, the manual tools may still be necessary. If the automatic routing tool is unsuccessful in routing some of the interconnections, the routing must be performed manually by the designer.

The key steps in the VLSI design process are illustrated in Fig. 1.14. Design entry is the first step in the design flow. It usually starts with a high-level design specification, e.g., written in English. From the high-level design specification, a behavioral description may be created using a hardware description language such as Verilog or VHDL. If a high-level behavioral description is available, behavioral synthesis tools may be used to generate a Register Transfer Level (RTL) design. Otherwise, the designer enters the circuit specification manually at the register transfer level. In this case, a Verilog or VHDL description may be entered using a text editor, or a block diagram schematic editor may be used. Logic synthesis tools are commonly used to generate gate-level descriptions from RTL descriptions, although designs can also be entered directly at the gate level, either using a text editor and a hardware description language or a graphical schematic capture tool. As the design is entered, it must be verified for correct functionality at every level. Functional simulation has been used for this purpose, combined with electrical-level simulation of low-level components and timing verification. Recently, formal verification tools have become available to perform some of the verification tasks. For designs that require minimal power consumption, power analysis must also be performed.

Following the design entry and verification is the task of layout generation, which provides all the information necessary for generating masks for wafer fabrication. The layout may be generated automatically, using automatic placement and routing tools, or it may be generated manually, as mentioned previously. Layouts must be

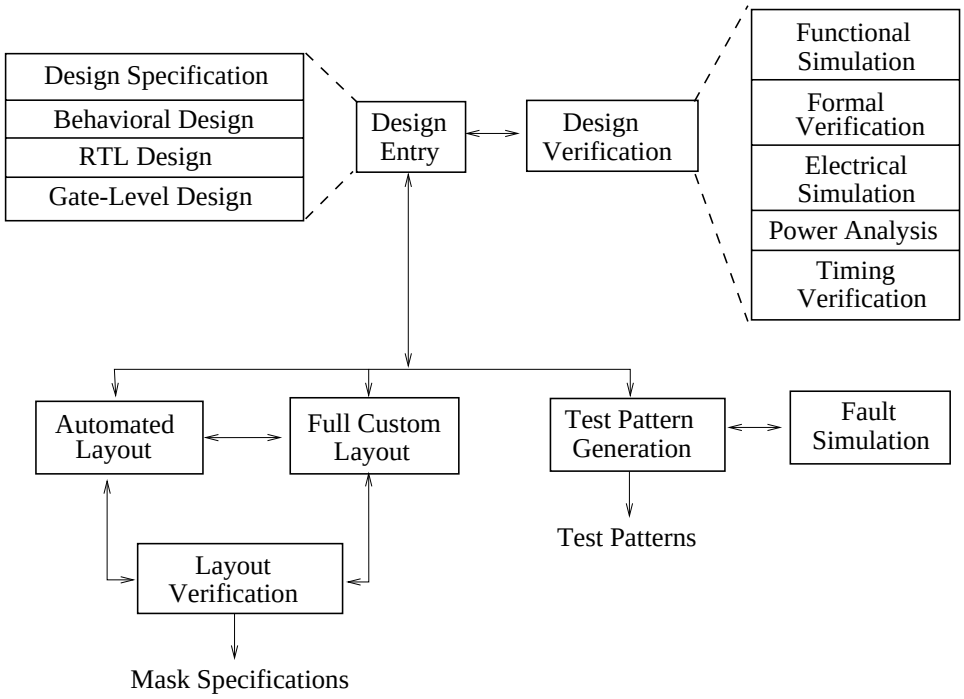


Figure 1.14. VLSI design process

verified to ensure that they conform to design rules before the masks are generated. Test patterns are necessary to test the chips once they are fabricated, and these tests may be generated once the design entry is completed. An Automatic Test Generator (ATG) may be used for this purpose; alternatively, the designer may generate tests manually. In either case, a fault simulator is used to evaluate the quality of the tests generated in terms of fault coverage.

Several of the tasks involved in the VLSI design process involve optimization problems. For example, an automatic placement tool must decide the optimal position in which to place each component. The specific problems are usually NP-complete; therefore, heuristic techniques have been used to obtain solutions. However, even if adequate approaches have been devised in the past, design complexity continues to increase with the continuing improvements in technology. Therefore, new approaches may be warranted over time, and GAs are often a good choice. Much research has been done in applying GAs to various tasks in the VLSI design process. Some of the techniques developed are being integrated into commercial EDA tools. The rest of this book will provide details about some of the EDA applications where GAs have been used. These applications include partitioning, automatic placement and routing, technology mapping for FPGAs, automatic test

generation, and power estimation. One chapter is devoted to each of these topics, and an additional chapter is included covering parallel implementations for two of these applications, namely, automatic placement and automatic test generation. Indeed, a key feature of GAs is that they are easily parallelized, and parallelization may be necessary for handling large designs in a reasonable amount of time for many EDA applications. Our objective is to provide examples where GAs have been successfully applied in the past so that the reader will be able to apply similar techniques in solving his or her own problems. We will now proceed to the EDA problems to which GAs have already been applied, concluding the chapter with a brief description of each.

1.6.1 Partitioning

Partitioning is used in various EDA applications, e.g., as a tool for placement algorithms or for assigning circuit elements to blocks that can be packaged separately. In VLSI design applications, partitioning algorithms are used to achieve various objectives.

1. Circuit Layout: A class of placement algorithms, *min-cut placement*, is based on repeated partitioning of a given network, so as to minimize the size of the cut set at each stage, where the cut set is the set of nets that connect two partitions. At each partitioning stage, the chip area is also partitioned, e.g., alternately in the vertical and horizontal directions, and each block of the network is assigned to one region on the chip. This process is repeated several times, until each block consists of only one cell. The resulting assignment of cells on the chip gives the final layout.

2. Circuit Packaging: Semiconductor technology places restrictions on the total number of components that can be placed on a single semiconductor chip. Large circuits are partitioned into smaller subcircuits that can be fabricated on separate chips. Circuit partitioning algorithms are used to obtain such subcircuits, with a goal of minimizing the cut set, which determines the number of pins required on each chip. This technique is gaining renewed importance these days for Field Programmable Gate Arrays (FPGAs), which require automatic software for partitioning and mapping large circuits on several FPGA chips for rapid prototyping.

3. Circuit Simulation: Partitioning has also been used to split a circuit into smaller subcircuits which can be simulated independently. The results are combined to study the performance of the overall circuit. This speeds up the simulation process by several times and is used in relaxation-based circuit simulators. This process is also used for simulating circuits on multiprocessors.

Partitioning algorithms are broadly divided into two classes: constructive and iterative improvement. Constructive algorithms may start from empty initial partitions, and they generally grow clusters of well-connected components around one or

more seed nodes or components, selected on the basis of some user-defined criteria, namely, number of fanin and fanout lines associated with a node. The quality of solutions generated by this class of partitioning algorithms is quite poor and degrades significantly as the partitioning problem size increases. The main advantage of constructive algorithms is that they are very fast and they usually scale well with the problem size. Therefore, these algorithms are frequently used to create an initial partitioning that may be further improved by applying other types of algorithms.

Iterative improvement algorithms, in contrast to constructive techniques, start with an initial partitioning, accomplished through some user-defined methods or randomly, and the algorithms tend to incrementally refine the initial partitioning through successive iterations; i.e., at any stage, a complete solution is always available. The algorithms tend to terminate when no more improvements can be found. Thus, these algorithms often terminate at local optima that are closely tied with the initial partitioning. Certain stochastic algorithms may give better results than deterministic algorithms. Iterative algorithms are widely used for high-quality two-way as well as multiway partitioning.

Chapter 2 gives a short overview of popular partitioning algorithms before discussing the genetic-based partitioning algorithms. In general, graph algorithms are highly successful in solving partitioning problems. The original graph approach, proposed by Kernighan and Lin [114] for solving telephone routing problems, was later adapted by Schweikert and Kernighan [205], to solve circuit partitioning by introducing a new representation, known as the net-cut or hypergraph model. This early version of circuit partitioning used a sorting algorithm to identify which pair of nodes (one from each block) should be swapped so that the cut-set size is minimized. Since the average run time for sorting n items is $O(n \log n)$, the overall complexity of the algorithm is worse than quadratic to the problem size, and it is unacceptable for large circuits. Fiduccia and Matthyeses [70] later developed an efficient bucket data structure to reduce the complexity of the partitioning algorithm, and by carefully modifying the original Kernighan-Lin algorithm, they developed a linear time circuit partitioning algorithm. However, this algorithm often gives solutions of inferior quality, especially if the circuit netlist is such that multiple ties occur during selection of the cell to be moved. Krishnamurthy [125] improved the performance of the Fiduccia-Mattheyses-type partitioning algorithm by adding a multilevel lookahead cell gain mechanism that enables the algorithm to more accurately select the cell which minimizes the cut-set size, considering future movement of cells. Dutta and Deng [63] pursued a more powerful method by adding a probability-based heuristic to the Fiduccia-Mattheyses algorithm and further improved the quality of solutions. Sanchis [200] has extended these approaches to perform multiway partitioning. Far superior results are achieved for recursive or repeated applications of the Kernighan-Lin or Fiduccia-Mattheyses algorithms, as was originally suggested by Kernighan and Lin in their seminal paper [114].

Besides the graph algorithms, a number of other combinatorial optimization techniques for circuit partitioning have been described in the literature, e.g., stochastic (simulated annealing [115]), adaptive (evolution methods [198]), neural [243], and

spectral [8], [9]. In Chapter 2, we demonstrate how genetic algorithms can be designed to formulate partitioning problems. Two different types of GAs are discussed to show how genetic algorithms can be used to perform high-quality bipartitioning and multiway partitioning. Location-based GA encoding has been used in which the chromosome is a string of binary bits, where a 0 represents that a cell is in the left partition and a 1 represents that the cell is in the right partition. Both breadth-first and depth-first orderings are discussed in these two different implementations. The first version uses the classical cut-set minimization criterion for dividing the cells in two (for bipartitioning) or more (for multiway partitioning) partitions. The second version of the GA uses the ratio-cut metric, which tends to detect the clusters in a circuit netlist (owing to the hierarchical nature of real-world electrical circuits), and divides the cells into two or more blocks based on the cell clusters. This version of the GA employs a Fiduccia-Mattheyses-type graph-based local improvement algorithm in order to obtain fast solutions to large problems. Ratio-cut and Fiduccia-Mattheyses-based graph algorithms are compared with these two versions of genetic algorithms, and it is shown that the GA-based partitioning techniques give superior results for bipartitioning and multiway partitioning.

1.6.2 Automatic Placement

Usually in a custom design of chip layout, two different styles of cell placements are adopted: *standard cell placement* and *macro cell placement*. Standard cells are relatively smaller blocks available in a library to implement various functions: from basic NAND, NOR, NOT, XOR, etc., of several fanin capabilities to complex decoders, multibit magnitude comparators, programmable counters, 4-bit ALU functions, etc. These cells are optimized carefully by the cell library vendors and are provided to VLSI designers in the form of a library pertaining to a specific process technology. Standard cells are so called since they have constant cell height, but variable cell width that depends on the functional complexity of a cell. Macro blocks, on the other hand, are parameterizable, have variable dimensions, and often have arbitrary rectilinear shapes. Since the designer compiles these blocks depending on user requirements, the blocks are usually significantly larger than standard cells. The macro cell layout style is the most flexible layout style. It allows inclusion of compiled blocks, such as RAM or ROM arrays, PLAs, etc., in arbitrary sizes, which are difficult to include in the standard cell layout style. A typical VLSI design may include standard cells, macro cells, and full custom logic.

Standard cells are usually aggregated laterally in rows of cells. They are permuted between themselves in such a way that the overall amount of interconnections required to wire them together to satisfy the given netlist will be minimum and also that the overall chip size will be minimum. Standard cells are usually designed so that the power and ground interconnections run horizontally through the tops and bottoms of the cells. When the cells are placed adjacent to each other, these interconnections form a continuous track in each row. The logic inputs and outputs of a module are available at *pins*, or terminals, along the top or bottom edge (or

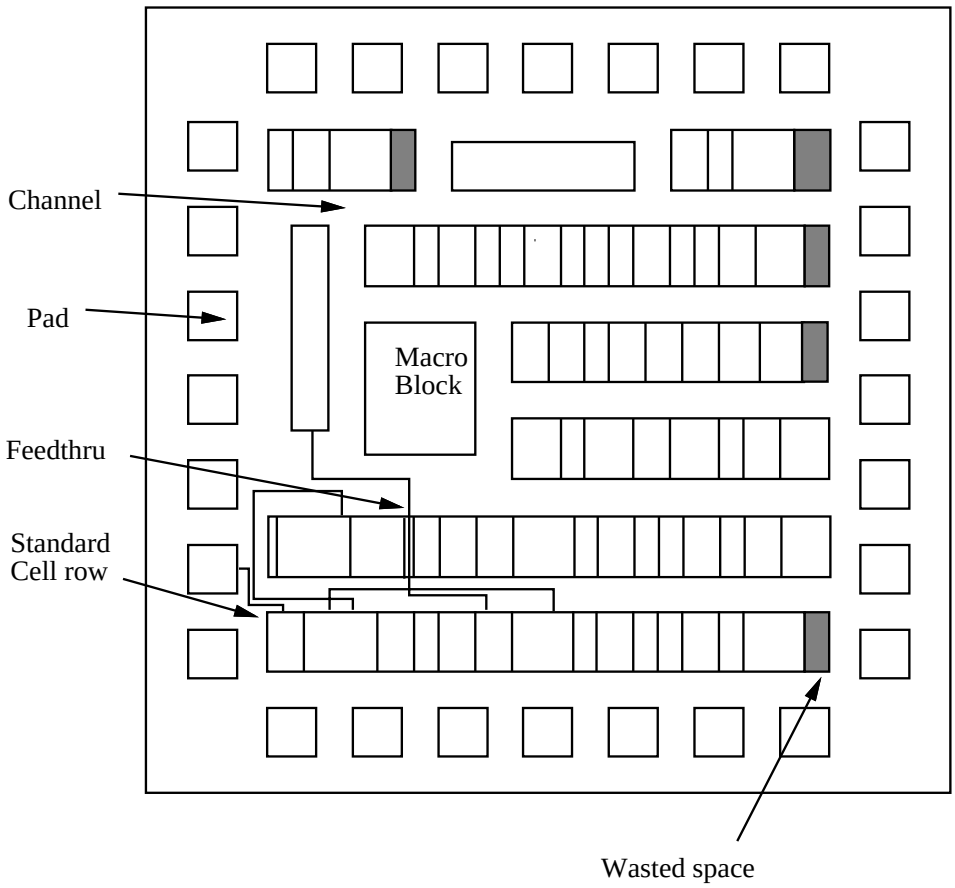


Figure 1.15. Standard cell layout style with macro blocks

both). Earlier generations of standard cell layouts set aside predefined horizontal routing zones, more appropriately called channels, between the rows of standard cells, as illustrated in Figure 1.15. These channels had horizontal tracks which were used for metal interconnection wires corresponding to the nets in a netlist. Connections from one row to another were done either through vertical wiring channels at the edges of the chip or by using feedthrough cells, which are standard height cells with just a few interconnections running through them vertically. At that time, VLSI technologies could support only one or two layers of metal interconnections in addition to polysilicon wires that were mostly used for short local connections. Over-the-cell routing was not possible at that time. Now with the availability of multilayer metal interconnects, over-the-cell routing is possible. The resulting routing styles are not necessarily of the classical channel routing genre where largely

horizontal segments of tracks containing metal wires are used to define the interconnects between different cells. Nevertheless, the main objective of a standard cell placement algorithm remains the same: to permute the cells between themselves and assign them to various rows in such a way that the objective function involving the overall interconnect length and the chip size is minimized without violating the non-overlapping cell requirement; i.e., no two adjoining cells should overlap laterally or between the edges of two successive rows.

Macro cells are usually rectilinear in shape and are much larger than standard cells. Not only can these cells be placed anywhere within the perimeter of a chip layout, but they can also be rotated in various orientations, as long as all rectilinear edges of a macro cell remain strictly parallel to one of the chip axes. Fig. 1.16 shows a chip composed entirely of macro cells with no standard cells. Each cell is surrounded by routing channels. The layout design objectives of standard cell

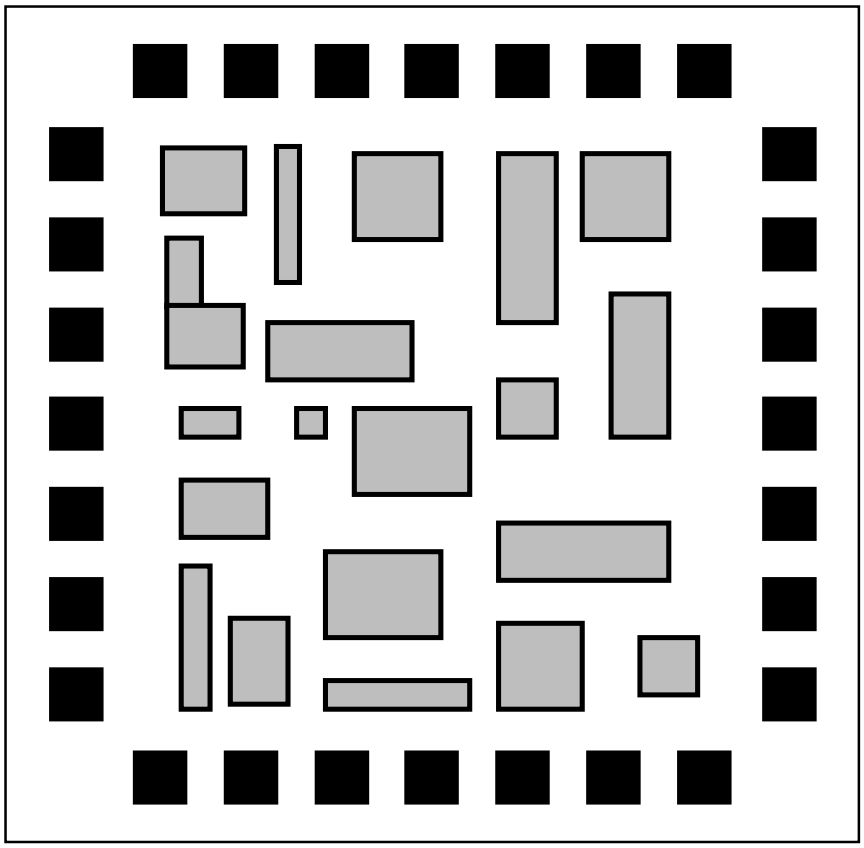


Figure 1.16. Macro cell layout style

placement and the mode of operation of the placement algorithm are significantly different from the layout optimization criteria in a macro cell layout design. The larger macro cells have many terminals that require interconnections. Thus, a macro cell placement tool must account for the additional channel area that is required to place the maze of interconnects around each macro cell. This estimation process is reasonably compute intensive. It involves determination of cell placement and orientation to minimize channel area, overall interconnect length, and chip area. The fact that all macro cells are not strictly rectangular makes the optimal placement of macro cells computationally intractable. These rectilinear shapes may require a major reallocation of positions and orientations of other cells whenever a big non-rectangular cell is rotated.

Cell placement problems can be broadly divided into four different classes based on the optimization techniques applied [212]. Graph-based approaches typically use a partitioning method known as the *min-cut* algorithm, originally proposed by Breuer [21]. By applying an alternating sequence of horizontal and vertical slicings that will partition components such that the cut-set size is minimized (hence, the name min-cut), one can find the correct placement of cells by recursively using a partitioning algorithm. Surais and Kedem [221] have extended this approach to concurrent two-dimensional slicing or sectioning so that the resulting partitioning is divided into four different quadrants. This graph-based approach is called *quadrisectioning* and is found to be very fast. The *second* class of algorithms is based on physical phenomena that are dictated by the laws of physics. Three well-known algorithms belonging to this class are: (i) *simulated annealing*, which mimics the well-known annealing process that blacksmiths apply to manufacture metallic implements [115], (ii) *generalized force-directed placement*, which is based on the well-known Hooke's law in elasticity [86], and iii) power dissipation (i.e., distribution of voltages and currents) in a resistive linear passive network of electrical components [35]. Although these physical phenomena range in different aspects from the Maxwell-Boltzmann distribution of molecules and lattice structures in a solid object to distribution of electrons in an electrical circuit, they all tend to arrange the natural system such that the potential energy of the system is always minimized and the system is in a stable state. In a cell-placement problem, this minimization of potential energy of a physical system is akin to minimizing its cost or objective function and is tantamount to simultaneous minimization of chip area and improvement of chip speed. The *third* class of algorithms is based on numerical optimization, which may involve postulating the cell placement problem as a set of equations to be solved by finding the eigenvalues and eigenvectors of a block diagonal matrix. Eigenvectors corresponding to a nontrivial minimum eigenvalue pertain to the minimized objective function and can denote the cell positions with reference to their $(x,\ y)$ coordinates. Similar to spectral-based partitioning algorithms [8], [9], placement tools developed using numerical optimization techniques have been found to give very high quality solutions [226]. Other algorithms in this class also utilize integer programming and linear programming based optimization techniques. The *fourth* class of algorithms is based on evolution algorithms.

In Chapter 3, genetic-based cell placement algorithms are discussed for both standard cell and macro cell layouts. For standard cell placement, a permutation class of genetic representation is used that is akin to the one used in solving the traveling salesman problem (TSP). In order to eliminate conflicts and inconsistencies that may occur due to application of a simple crossover operator with random cut and paste, three different crossover operators are introduced which systematically generate offspring that represent valid placement solutions. The three crossover operators are called *order*, *PMX*, and *cycle* crossover. A meta-genetic optimization process is used to select the various tuning parameters, such as crossover rate, inversion rate, and mutation rate. Cycle crossover has been found to be superior to other crossover operators. For macro cell placement, a two-dimensional bin-packing chromosomal representation is used, where the cells are arranged in the form of a priority queue. Since macro cell algorithms are very computation intensive, the genetic algorithm is transformed into simulated annealing by decreasing the population size according to some temperature schedule, and mutation operations are increased in the later phase of the algorithm. The mixed genetic algorithm (GA) and simulated annealing (SA) approach, called SAGA, is shown to provide superior results to the conventional optimization techniques used in macro cell placement tools for several benchmark circuits.

1.6.3 Automatic Routing

The objective of an automatic routing tool is to connect the cell pins while meeting any layout constraints. Typical constraints are to minimize the overall area, avoid meandering paths and thereby reduce delays, and minimize the lengths of critical nets. Interconnection wires are assumed to have widths that meet the design rules imposed by the technology used and to be adequately separated from adjacent wires.

Modern high-performance, multimillion transistor VLSI chips with multiple layers of interconnects are extremely dense and often too complex to have interconnects routed in their entirety by a single pass of a grid-based algorithm, such as a maze router, which is commonly used for Integrated Circuit (IC) and Printed Circuit Board (PCB) wire routing. Consequently, due to the high degree of complexity, most VLSI routing algorithms operate in a two-phase mode. First, a *global* or *loose* routing is performed that determines the topologies of nets and their relative positions with respect to objects which act as obstacles to the routing. This is followed by a *detailed* routing phase, where the final interconnection specifications among the module terminals are determined in terms of layers, vias, and track assignments. Examples of detailed routers are channel routers, maze-type routers, river routers, and switchbox routers, and effective heuristics have been developed for such routers. Specialized routers are used for power, ground, and clock routing, which are generally separated from signal layers and are critically routed to meet the constraints of a high-performance chip.

The main task of global routing is to break up a larger, more complex problem into several smaller ones that are more tractable in nature. In this book, we discuss

only the problem of global routing, and in particular, global routing for macro cells. In a simultaneous place-and-route algorithm, often global routing is blended with the cell placement algorithm in an intertwined fashion. Therefore, the genetic-based automatic global routing technique described in Chapter 4 is a natural sequel to Chapter 3, which describes the genetic-based macro cell placement algorithm.

Depending upon the problem complexity, global routing can be broken into two subproblems: (i) global routing graph construction and (ii) wiring the nets within the routing regions. Most global routers perform routing in terms of a global routing graph, which is extracted from the given placement and the routing region definitions. The edges of the graph correspond to future routing regions, while the vertices correspond to intersections of routing regions. While the graph is highly regular for gate arrays and most PCB routing, the graph is irregular for macro cell routing. The routing graph of a simple placement of a macro cell design, assuming suitable routing region definitions, is shown in Fig. 1.17.

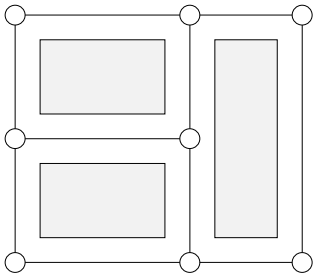


Figure 1.17. Initial global routing graph for a small macro cell problem

Further, the structure of the global graph can also be augmented by information on available feedthroughs. A feedthrough represents a routing area that can pass through a particular cell or macro and is either deliberately introduced as a mechanism for reducing wire lengths of global nets or can be constructed from unused space within a cell. In addition, in order to ease the task of detailed routing, several optimality criteria are used by the global router, such as minimizing total wire length, minimizing total area, minimizing local densities, etc. Every edge, therefore, is assigned one or more cost values typically representing the length of the associated routing region and/or the capacity of the region, i.e., the number of nets which can pass through the region.

To compute a global route for a specific net, vertices representing the terminals are added at appropriate locations, as illustrated in Fig. 1.18. Here, for each terminal, the location of the corresponding terminal vertex is determined by a perpendicular projection of the terminal onto the edge representing the appropriate routing channel. Finding a global route then becomes equivalent to finding a minimum-cost subtree in the routing graph which spans all the terminal vertices. This tree finding can be done using standard shortest-path, spanning-tree, or Steiner-tree algorithms with modifications to account for features peculiar to a certain layout style. Subse-

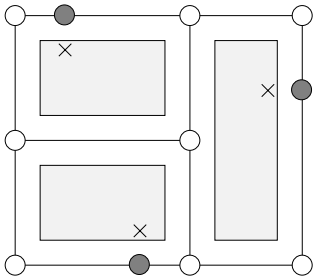


Figure 1.18. Routing graph augmentation due to terminals

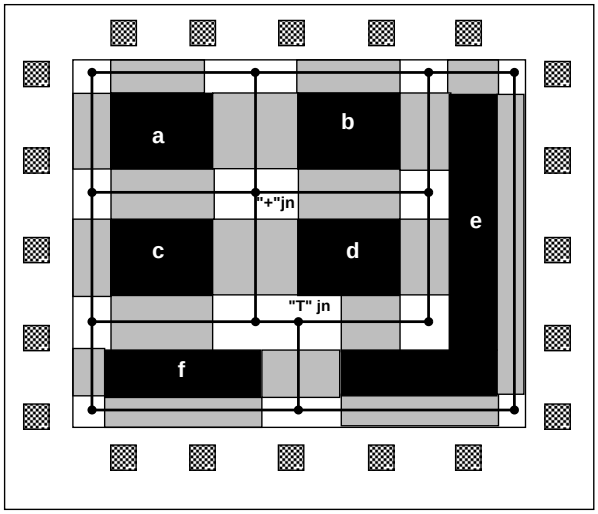
quent to global routing, each net that spans more than one region (edge in the global routing graph) is decomposed into subnets, one per region. Between neighboring regions, pseudo-pins are introduced to indicate the intersection of global paths with the boundaries of the local regions.

In macro cell layout, the routing region definition is more involved. Routing region definitions consist of partitioning the routing area into a set of nonintersecting rectangular regions called channels. Channel definition and ordering are key steps in the overall placement, global routing, and detailed routing. After routing regions are defined, the routing graphs are generated according to one of three different models: (i) the rectilinear adjacency graph model, (ii) the polar graph model, or (iii) the hybrid graph model. The rectilinear adjacency graph and horizontal and vertical polar graphs are illustrated in Fig. 1.19. Global routing can be broadly divided into two main classes: sequential approaches and concurrent approaches.

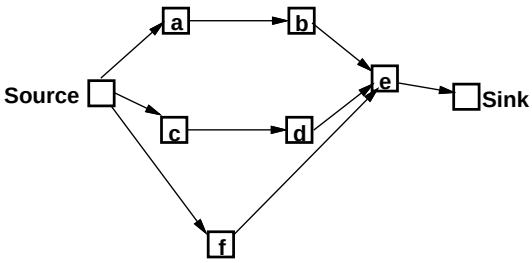
1. **Sequential Approaches:** As the name suggests, these routers construct a complete global routing by considering one net at a time. These can be further categorized based on the number of alternative solutions they consider for each net and the number of passes they perform. The actual algorithms depend on the nature of the global routing graph. For instance, for grid graphs, the maze routing algorithm and its variants are often used. However, for more general routing graphs, particularly for macro cell layout, general shortest-path or Steiner-tree algorithms are required.

2. **Concurrent Approaches:** A main drawback of sequential routers is that the result quality is highly dependent on the order in which nets are routed and in general, it is difficult to devise a good net ordering a priori. Consequently, in concurrent routing, all nets are considered simultaneously and the net-ordering problem is bypassed. Two popular solution techniques that fall into this category include:

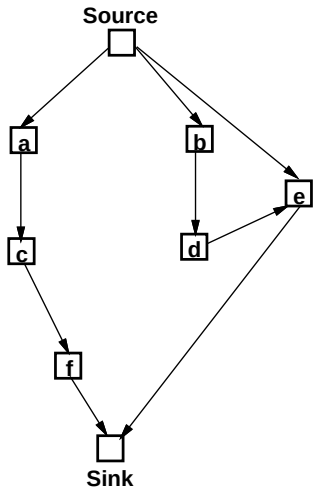
   (a) **Integer Programming:** This method formulates the global routing problem as an integer programming problem. Methods based on relaxation or simulated annealing have been used to help solve the integer programs.



(a) Rectilinear adjacency graph.



(b) Horizontal polar graph.



(c) Vertical polar graph.

Figure 1.19. Routing graphs for 6-cell macro cell layout

(b) **Hierarchical Methods:** These approaches either proceed in a top-down or a bottom-up fashion, obtaining complete solutions at each level of the hierarchy and using the solutions obtained to influence the routing at the next level. Often the problems that arise at any one level of the hierarchy are simple enough to be solved exactly.

Chapter 4 describes a new genetic-based Steiner global routing algorithm. Numerous algorithms of various kinds have been developed for the Steiner Problem in a Graph (SPG) that find a minimum cost subgraph spanning a set of designated vertices on a given graph. Exact algorithms can be found in [11], [14], [40], [60], [90], [140], [216]. However, since the SPG is NP-complete [111], these algorithms have exponential worst case time complexities. Therefore, a significant research effort has been directed toward polynomial time heuristics [11], [121], [172], [184], [222], [239]. Simulated annealing has also been applied to SPG [59]. A first effort to apply genetic algorithms to SPG is due to Kapsalis, et al. [110]. The genetic-based algorithm for SPG described in Chapter 4 differs from [110] in a number of ways: by enforcing constraint satisfaction, the proposed algorithm eliminates the need for imposing penalties for invalid solutions; the new algorithm uses an inversion operator; it also uses a more efficient method for performance evaluation; and unlike in [110], which requires problem-specific parameter settings, a fixed set of parameters are used in the algorithm described in Chapter 4. Consequently, from experimental results, it can be seen that the GA-based global routing algorithm discussed in Chapter 4 outperforms the GA-based algorithm in [110] and an iterated version of the shortest path heuristics proposed by Winter and Smith [239]. It has been demonstrated that the GA is capable of finding optimal solutions in more than 77% of all test cases run and is within 1% of the optimal result in more than 92% of all test runs.

1.6.4 Technology Mapping for FPGAs

Field Programmable Gate Arrays (FPGAs) are devices that can be configured by the user to implement a required logic function. FPGAs are being used extensively for fast system prototyping due to their low cost and the ability to produce the final circuit within a relatively short time. An FPGA can be viewed as a modification of a mask-programmable gate array, where the reprogramming time and cost are drastically reduced, or as a Programmable Array Logic (PAL) device whose size is increased by an order of magnitude. A PAL device contains an array that implements AND/OR logic and is restricted to a small number of equivalent logic gates.

Figure 1.20 shows the structure of a typical FPGA, which consists of logic blocks surrounded by routing resources. Each logic block consists of a combinational logic part and a sequential part (a latch). There are two main classes of FPGA architectures: one is Lookup Table (LUT) based, and the other is multiplexer based. The LUT-based FPGA is widely available commercially through several FPGA manufacturers. In a LUT-based FPGA device, the programmable logic block consists of a K-input lookup table, also called a configurable logic block (CLB), which can implement any Boolean function of up to K variables.

A K-input lookup table is a digital memory in which K inputs are used to address a 2^K by 1-bit memory that stores the truth table of the Boolean function. The XC3000 device from Xilinx is an FPGA device in which the basic programmable

logic block is a K-input lookup table which can implement any Boolean function of up to K variables. The CLB for the XC3000 device is shown in Figure 1.21 [24]. Here, a CLB can represent either a 5-variable function or two 4-variable functions which together depend on at most 5 variables.

The technology mapping problem for a LUT-based FPGA device is to transform a Boolean network into a functionally equivalent network of K-input LUTs (in general, technology mapping is a logic synthesis step in which equations and latches are implemented by choosing gates from a fixed library of primitive components). The objective of the technology mapping tool for FPGAs is to find the best mapping of logic onto FPGA logic blocks. Here, the quality of the mapping is measured in terms of the size of the required FPGA and the routability of the design, as well as other factors, such as delays. The technology mapping problem for the XC3000 device is to map the design onto CLBs in the device.

There are several approaches for solving the FPGA mapping problem. Some approaches [72], [112], [152], [202], [240] map the logic based on algebraic decomposition. Other approaches [130], [168] are based on function decomposition, which is a Boolean optimization method. All of the above approaches to solving the mapping problem map the logic onto LUTs and then pack these LUTs into CLBs. As the number of inputs to a LUT increases, the search space increases exponentially, and these methods, particularly those based on Boolean optimization, do not scale well.

Chapter 5 describes a genetic algorithm for technology mapping to FPGAs. The algorithm can target either the mapping onto LUTs or can directly map onto CLBs.

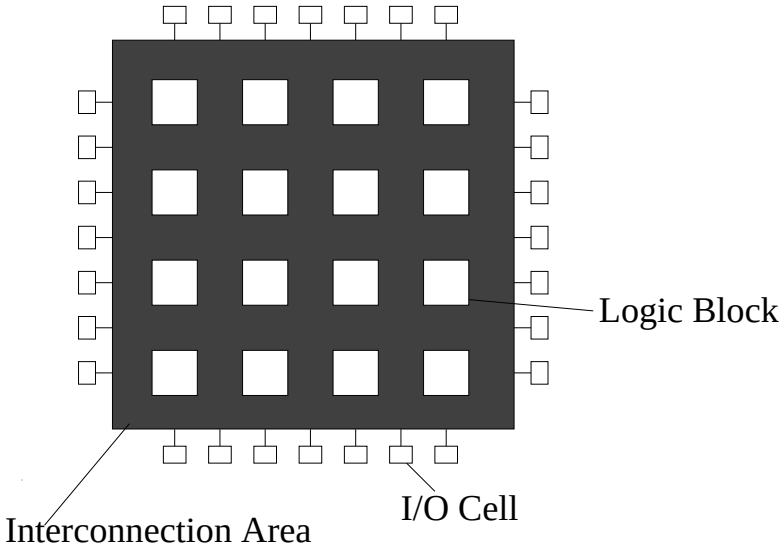


Figure 1.20. FPGA structure

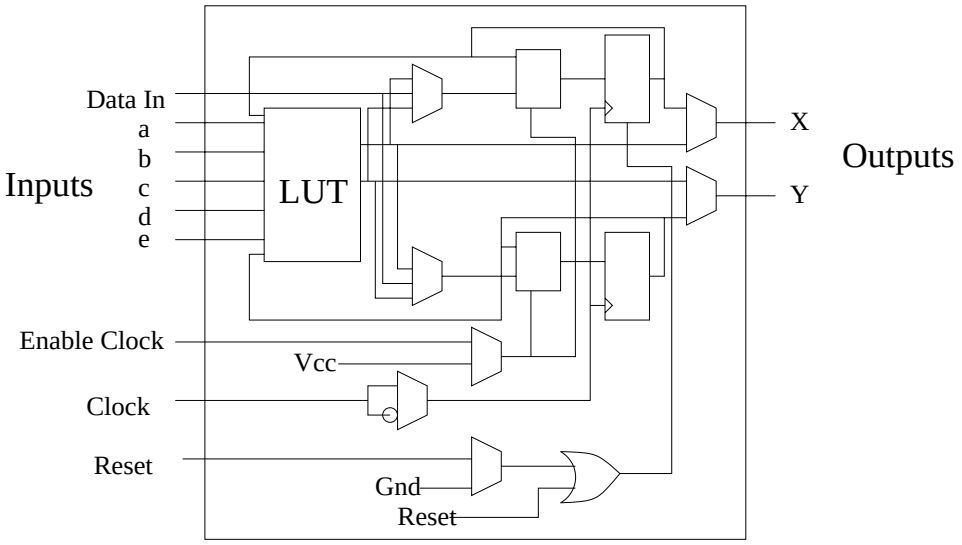


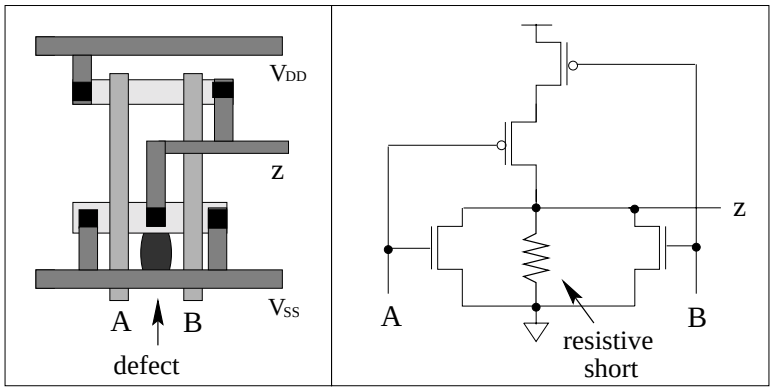
Figure 1.21. Xilinx XC3000 CLB

Designing the GA such that it maps directly onto CLBs leads to far better CLB mappings. Furthermore, as the number of inputs to a LUT increases, the GA-based approach exhibits superior results compared to other approaches.

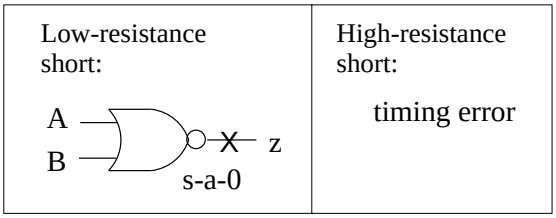
1.6.5 Automatic Test Generation

Once a chip is fabricated, it must be tested to ensure that it functions as designed. The assumption is made that the design is correct. The set of tests applied should be able to uncover all possible defects that could occur in the manufacturing process. That is, the output response of a defective chip should be different from the outputs of a good chip. Then the defective chips can be weeded out so that they are not shipped to customers.

Consider the defect between the output of the NOR gate and V_{ss} as shown in Fig. 1.22(a). A transistor-level representation is shown in Fig. 1.22(b), and two possible logical manifestations of the defect are shown in Fig. 1.22(c). If the resistance is low, the output of the NOR gate will behave as if it is stuck at logic 0. If the resistance is high, a timing error may result. We would like to ensure that the test set contains a test to detect this defect in case it occurs. Obtaining a set of tests that will detect most possible defects is a difficult problem, and this is the objective of an automatic test generator. The approach commonly used is to try to generate a test set that detects all single stuck-at faults in the circuit. After a high fault coverage for single stuck-at faults is achieved, additional test vectors may be generated that target other fault models, such as the delay fault model.



(a) Physical Defect (b) Electrical Fault



(c) Logical Fault

Figure 1.22. Effects of a physical defect

Deterministic test generation algorithms for combinational circuits [85] and sequential circuits [37], [76], [134], [141], [145], [160], [204] have been used in the past, but execution times are often long, due to the large number of backtracks that often occur. Simulation-based approaches have also been used [6], [20], [139], [203], [207], [218], [242] and are effective in reducing execution time. However, fault coverages are often lower. The use of GAs for simulation-based test generation has been shown to provide better fault coverages than deterministic algorithms in some cases and lower execution times [46], [97], [98], [99], [102], [179], [181], [189], [194], [195], [196], [197], [219].

GA-based test generators developed at the University of Illinois in the past few years are described in detail in Chapter 6. A simple GA with nonoverlapping populations has been used, due to the irregular search space. Exploration of a large number of candidate solutions overrides exploitation of known good solutions. Furthermore, in contrast to other applications, the GA is invoked repeatedly for test generation, and the objective with each invocation is to generate a vector or sequence of vectors to aid in detecting one or more faults. Because of the large number of invocations of the GA and the high cost of accurate fitness evaluation, the GA

parameters must be set to reduce the computation time to a reasonable level, at the expense of solution quality. In addition, approximate fitness functions can be used to further reduce the computation time. Of course, tradeoffs in computation time vs. solution quality can always be made, depending on which factor is more important in a particular situation. In fact, experimental results published in the literature clearly illustrate the tradeoffs that have been made in various implementations.

GA encoding for the test generation problem has been straightforward. A simple string representation is typically used in which each gene represents the logic value to be applied to a particular input of the circuit at a specific time. Standard crossover and mutation operators can therefore be used. While the first implementations targeted a large number of faults in the circuit concurrently, the trend has been to separate the tasks of fault excitation and fault effect propagation and to target faults individually. This approach was first used in GATTO [181], [46], and it has resulted in the highest fault coverages ever reported for many of the benchmark circuits in STRATEGATE [99]. Although the simple strategy used initially for test generation in GATEST [189], [194] does not achieve such high fault coverages, it is very effective for test sequence compaction, as discussed in Chapter 6.

1.6.6 Power Estimation

Even though VLSI designs are fabricated in CMOS technology, which is a low-power technology, we still have to worry about power for large chips. Power dissipation in CMOS has two components: static, due to leakage current, and dynamic, due to switching activity. The static power is relatively low and is often neglected in power estimation. Dynamic power dissipation occurs for two reasons. First, the short circuit component is due to the current pulse that flows through the p and n transistors from V_{dd} to ground while they are both conducting simultaneously. This component of the power dissipation is small in a well-designed circuit. Secondly, the capacitive component is due to charging and discharging the various capacitances associated with each gate. This capacitive component depends on the node capacitance, the switching frequency, and the supply voltage as follows:

$$Power = \frac{1}{2}\ t\ C\ V_{dd}^2\ f$$

where V_{dd} is the power supply voltage, C is the node capacitance, f is the clock frequency, and t is the number of toggles between logic 1 and logic 0 that occur within a clock cycle. Estimation of power dissipation for a circuit involves summation of the power dissipated at individual nodes.

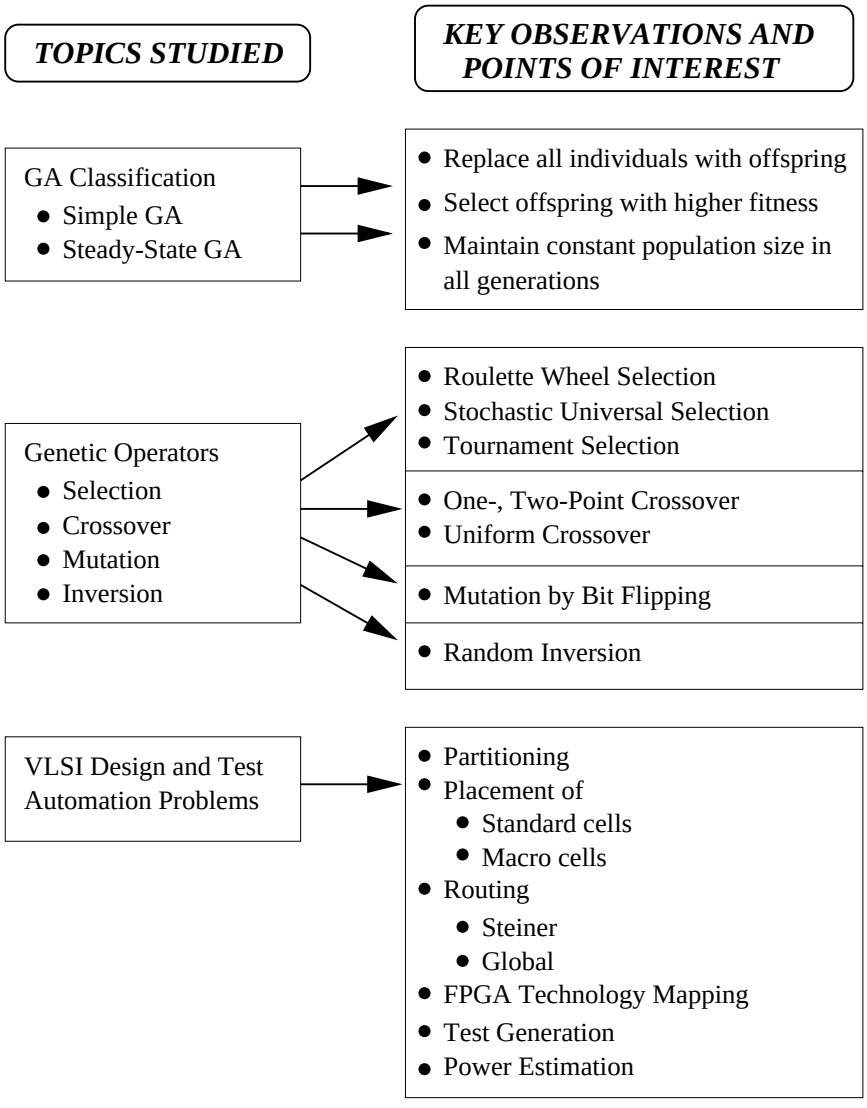
The objective of a power estimation tool is to determine the maximum power dissipation that will occur. If the power dissipation is found to be too high, modifications can be made to the design, or appropriate packaging technology can be used. At the gate level, static techniques that use probabilistic information about circuit activity are often employed for estimating power dissipation, e.g., [42]. However, the average power dissipation estimated using this approach does not provide accurate

boundary conditions for determining the limits of the design. Dynamic techniques that simulate the circuit using typical input sequences are more effective at estimating the peak power dissipation, which provides a limit for circuit operation. However, a method is needed to provide candidate sequences so that a sequence that leads to maximal power dissipation can be found. Deterministic techniques proposed previously for this problem may be unsuitable for large circuits. Chapter 7 describes a GA that solves this problem and is able to handle large circuits while requiring very little computation time. The peak power estimates provided by the GA achieve much tighter lower bounds on the actual peak power than a random approach that requires considerably more execution time.

The GA begins with random sequences and attempts to evolve a sequence with maximum power dissipation. A simple binary encoding is used in which each individual in the GA population represents the initial circuit state and a sequence of vectors to be applied. The fitness function estimates power dissipation by simulating the sequence and keeping track of the toggle count at each node. This toggle count is weighted by the node capacitance to get power dissipation at each node, and the summation of power dissipation over all nodes is computed.

Power dissipation in a circuit is dependent upon delays of internal circuit nodes. Glitches and hazards result in power dissipation. Therefore, the accuracy of the power estimate is dependent upon the delay model used when simulating each sequence. Peak power estimation under four different delay models is explored in Chapter 7, and comparisons are made among the various models. In general, sequences optimized under one nonzero delay model provide good measures for other nonzero delay models. In addition, various durations are studied, and computation of peak single-cycle, n-cycle, and sustainable power dissipation is addressed.

SUMMARY


TOPICS STUDIED
KEY OBSERVATIONS AND POINTS OF INTEREST
GA Classification
Simple GA
Steady-State GA
Replace all individuals with offspring
Select offspring with higher fitness
Maintain constant population size in all generations
Genetic Operators
Selection
Crossover
Mutation
Inversion
Roulette Wheel Selection
Stochastic Universal Selection
Tournament Selection
One-, Two-Point Crossover
Uniform Crossover
Mutation by Bit Flipping
Random Inversion
VLSI Design and Test Automation Problems
Partitioning
Placement of
Standard cells
Macro cells
Routing
Steiner
Global
FPGA Technology Mapping
Test Generation
Power Estimation