

Chapter 2

Design Flow

1 INTRODUCTION

The aim with this chapter is to give a brief introduction to design flow in order to motivate the discussion on modeling granularity in the following chapters.

The pace in the technology development leads to an extreme increase of the number of transistor in a system (See *Introduction* on page 1). However, there is a gap between what is possible to design according to what the technology development allows to be designed and what is possible to design in terms of cost (time, money, manpower). The so called productivity gap, which is increasing, is the difference between what is possible to design using available technologies and what is reasonable to accomplish in reasonable design time. The productivity is defined as [120]:

$$P = \frac{\text{total gate count}}{\text{development time} \times \text{number of designers}}$$

It means that if productivity keeps stable, the design time would increase drastically (keeping the design team size constant). An alternative is to increase the size of the design teams in order to keep design time constant. Neither of the approaches are viable. Instead, the evolution in EDA (Electronic Design Automation) will provide productivity improvements. That means that better modeling and optimization techniques have to be developed.

In order for the designers to handle the complexity of the designs due to the technology development, the system can initially be modeled at a high abstraction level. Semi-automatic synthesis and optimization steps (Computer-Aided Design (CAD) tools guided by designers) on the high-level specification transforms the design from the abstract specification to the final circuit (see Figure 4). System modeling at an high abstraction level, means that less implementation specific details are included compared to a system model at lower abstraction levels. Gajski *et al.* [64] view the design flow using a Y-chart where the three direction behavior, structure, and geometry.

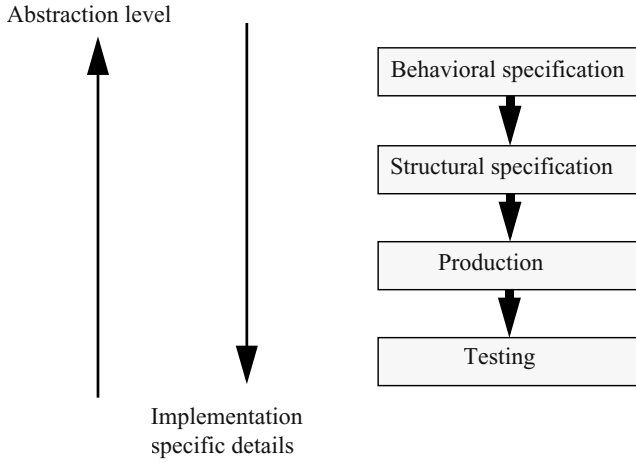


Figure 4. Design flow.

2 HIGH-LEVEL DESIGN

The advantage of a top-down design flow, specifying the design a high abstraction level with less implementation specific details, and through synthesis move to a model with a high degree of implementation specific details, is that design exploration, where design alternatives easily can be explored, is eased. A model at high abstraction level includes fewer details and therefore the handling of the design becomes easier, and more optimization steps can be allowed, which means a more thorough search in the design space.

In the design flow, the system is modeled. The purpose of modeling is to understand the function of the system as well as adding implementation specific details. A number of techniques have been developed. On initiative by VHSIC (Very High Speed Integrated Circuits) programme the description language for digital circuits called VHDL (VHSIC Hardware Description Language) was developed [120]. VHDL became an IEEE standard in 1987, and the latest extension took place in 1993 [268]. Graphical descriptions are useful for humans to more easily understand the design. Examples of graphical descriptions are Block Diagrams, Truth Tables, Flow Charts, State Diagrams (Mealy and Moore state machine), and State Transition Diagrams. Netlists are used to enumerate all devices and their connections. The SPICE (Simulation Program with Integrated Circuit Emphasis) is an example of a device oriented structure [265]. The EDIF (Electronic Design Interchange Format) was developed to ease transport of designs between different CAD systems. And, for capturing delay formats in the design, the SDF (Standard Delay Format) was developed [252].

3 CORE-BASED DESIGN

There are a number of disadvantages with a top-down design flow. One obvious is that at high abstraction level implementation details are less visible. It makes discussion on certain detailed design aspects impossible at high abstraction level. The problem with high abstraction levels where less information is captured has become a problem in sub-micron technology where technology development has lead to high device size miniaturization that requires knowledge of transistor-level details. Another fundamental problem is that in a top-down design flow, there is a basic assumption that the system is designed for the first time. In reality, very few systems are designed from scratch. In most cases, there exists a previous system that is to be updated. It means that small parts or even large part of the previous system can be reused for the new system. Also, other previously designed blocks that have been used in a completely different system can be included in the new system. It can also be so that blocks of logic, cores, are bought from other companies. For instance, CPU cores can be bought from different companies and used in the design.

In the core-based design environment, blocks of logic, so called cores, are integrated to a system [84]. The cores may origin from a previous design developed at the company, it may be bought from another company, or it can be a newly designed core. The *core integrator* is the person that selects which cores to use in a design, and the *core test integrator* is the person that makes a design testable - designs the *test solution*. The core integrator selects core from different *core providers*. A core provider can be a company, a designer involved in a previous design, or a designer developing a new core for the system.

A core-based design flow is typically a sequence that starts with core selection, followed by test solution design, and after production, the system is tested (Figure 5(a)). In the core selection stage, the core integrator selects appropriate cores to implement the intended functionality of the system. For each function there are a number of possible cores that can be selected, where each candidate core has its specification on, for instance, performance, power consumption, area, and test characteristics. The core integrator explores the design space (search and combines cores) in order to optimize the SOC. Once the system is fixed (cores are selected) the core test integrator designs the TAM and schedules the tests based on the test specification for each core. In such a design flow (illustrated in Figure 5(a)), the test solution design is a consecutive step to core selection. In such a flow, even if each core's test solution is highly optimized, when integrated as a system, the system's global test solution is most likely not highly optimized.

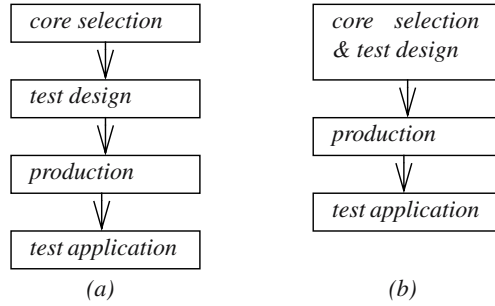


Figure 5. Design flow in a core-based design environment (a) traditional and (b) proposed.

The design flow in Figure 5(b), on the other hand, integrates the core selection step with the test solution design step, making it possible to consider the impact of core selection when designing the test solution. In such a design flow (Figure 1(b)), the global system impact on core selection is considered, and the advantage is that it is possible to develop a more optimized test solution.

The design flow in Figure 1(b) can be viewed as in Figure 6, where the core type is floor-planned in the system but there is not yet a design decision on which core to select. For each position, several cores are possible. For instance, for the `cpu_x` core there are in Figure 6 three alternative processor cores (`cpu1`, `cpu2` and `cpu3`).

In general, the cores can be classified as:

- soft cores,
- firm cores, or
- hard cores.

A soft core is given as a specification that has to be synthesized, while a hard core is already a fixed netlist. A firm core is somewhere between a soft core and a hard core. A soft core is therefore the most flexible type of core;

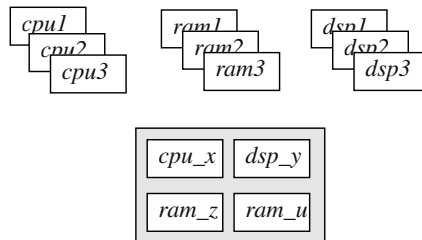


Figure 6. System design.

however, it requires the tools, effort and time involved in the work of synthesizing it to a gate-level netlist. A hard core, on the other hand, is already fixed and ready to be used in the design. Hence, less effort and time are required for the optimization of the core. The difference between a soft core, firm core, and a hard core is also visible when it comes to testing. For soft core there is a higher degree of freedom when determine the test method compared to hard cores where the test method and test vectors are more fixed.

3.1 Network-on-Chip

A problem when the SOC designs grow in complexity is that the on-chip interconnections, such as buses and switches, can not be used anymore due to their limited scalability. The wires are simply becoming too long. An alternative is to use networks on chip (NOC) [14, 121].

3.2 Platform-Based Design

The design flow must not only take the development of the hardware into account but also the development of software. Systems of today include an increasing number of programmable devices. The software (operating system and application programs as well as compilers) must be developed. In order to fix the architecture, the concept of platform-based design has been developed. Sangiovanni-Vincentelli [237] compare it with the PC (personal computer):

- The X86 instruction set makes it possible to re-use operating systems and software applications,
- Busses (ISA, USB, PCI) are specified,
- ISA interrupt controller handles basic interaction between software and hardware,
- I/O devices, such as keyboard, mouse, audio and video, have a clear specification.

The PC platform has eased the development for computers. However, most computers are not PCs but embedded systems in mobile phones, cars, etc [266]. Hence, a single platform-based architecture for all computer systems is not likely to be found.

4 CLOCKING

Clocking the system is becoming a challenge. Higher clock frequencies are making timing issues more important. The storage elements (flip-flops forming registers) in a system have to be controlled. For instance, Figure 7(a) shows a finite state machine (FSM), and a pipelined system is in Figure 7(b); the storage devices are controlled by the clock. Most systems are a combination of FSM and pipelined systems.

The input of a register (flip-flop) is commonly named D , and the output named Q . Figure 8 illustrates the following:

- the *setup time* (T_s) - the time required before the edge until the value is stable at the D input,
- the *hold time* (T_h) - the time required after the edge in order store the value in the register,
- the *clock-to- Q delay* (T_q) - the time required after the edge in order to produce the value on Q ,
- the *cycle time* (T_c) - the time between two positive edges (from 0 to 1).

The registers can be design for instance as a *level-sensitive latch*, *edge-triggered register*, *RS latch*, *T register*, and *JK register* [277].

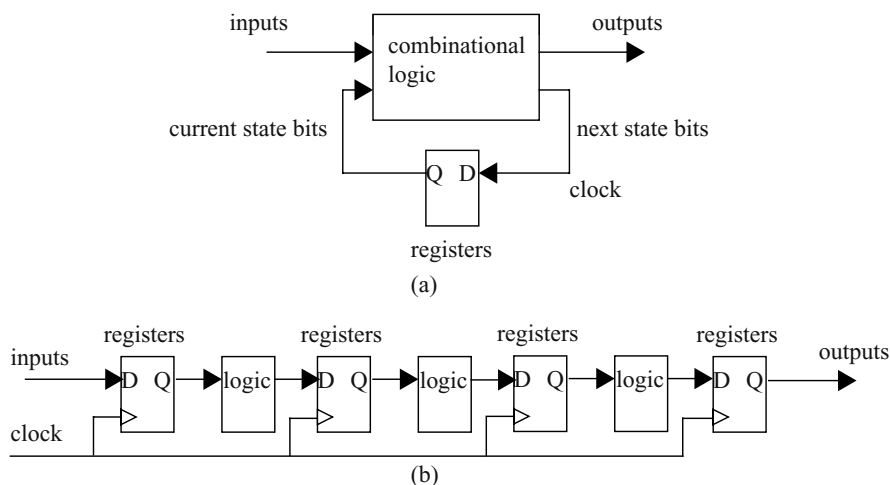


Figure 7. Clocked systems. (a) a finite state machine (FSM); (b) a pipelined system.

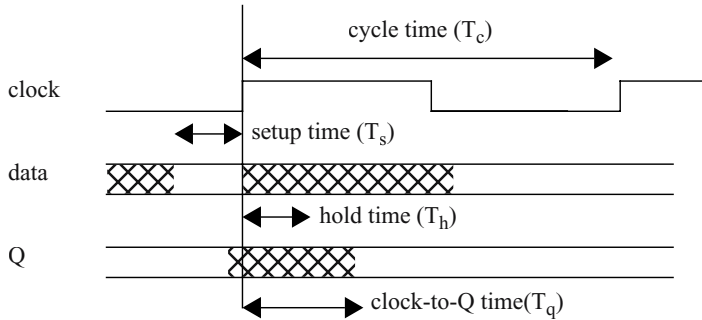


Figure 8. Single phase clocking and its parameters.

4.1 System Timing

Latches and registers can be used in different ways to implement a clocked system [277]. It is important that the clock signal is delivered to the registers correctly. Long wires may introduce delays in the clock distribution leading to clock skew - the clock signal is not delivered at the same time to all clocked elements.

The cycle time (T_c) is given by:

$$T_c = T_q + T_d + T_s$$

where T_q is the clock-to-Q delay, T_s is the setup time, and T_d is the worst-case delay through the combinational logic blocks.

4.2 Clock Distribution

Clock distribution is a problem. For instance, the total capacitance in the system that has to be driven can be above 1000 pF [277]. Driving such capacitance at high repetition rate (clock speed) creates high current (see example in Figure 9).

$$\begin{aligned}
 V_{DD} &= 5V \\
 C_{reg} &= 2000 \text{ pF (20K register bits @ .1pF)} \\
 T_{clk} &= 10 \text{ ns} \\
 T_{rise/fall} &= 1 \text{ ns} \\
 I_{peak} &= C \times \frac{V_{DD}}{T_{rise/fall}} = 2000 \times 10^{-12} \times \frac{5}{1 \times 10^{-9}} = 10A \\
 P_d &= C \times V_{DD}^2 \times f = 2000 \times 10^{-12} \times 25 \times 100 \times 10^{-6}
 \end{aligned}$$

Figure 9. Example illustrating high current and power [277].

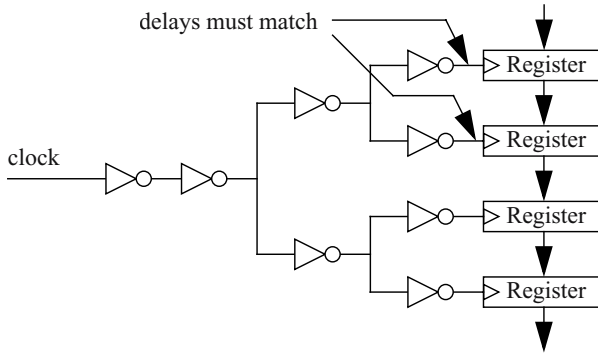


Figure 10. Distributed clock-tree where the logic is omitted.

The clock can be distributed using [277]:

- a single large buffer (*i.e.* cascaded inverters) that is used to drive all modules, or
- a distributed-clock-tree approach (Figure 10).

4.3 Multiple Clock Domains

In a core-based design, a set of cores are integrated to a system. Each core might require its dedicated clock speed. A design with multiple clock domains can present challenges. A typical problem is at the clock-domain borders where one clock frequency is meeting another clock domain. Problems that can appear are, for instance, data loss and metastability. Data loss can appear when data generated by clock1 is not captured by clock2 until it has already been changed. Such problems appear if the destination clock is running at a higher speed. Metastability is due to that there is no timing relation between source and destination clock domain, and it can happen that data and clock signals reach the flip-flops at the same time. Metastability can cause reliability problems.

4.3.1 Phase Locked Loop (PLL)

A phase locked loop (PLL) is used to generate internal clocks in the system for two main reasons [277]:

- Synchronize the internal clock with an external clock.
- The internal clock should operate at a higher frequency than the external clock.

The increasing on-chip clock speed has lead to clock skew problems. A PLL allows an internal clock to be generated that is in phase with an external clock.

4.3.2 Globally-Asynchronous Locally-Synchronous

The advantage of making a system synchronous is that it is:

- a proven technique, and
- wide-spread EDA tools are available.

However, it is important to note that

- clock speed limited by slowest operation,
- clock distribution a problem,
- hard to design systems with multiple clock domains, and
- energy is consumed for idle operation.

An alternative is to make the system asynchronous. Instead of synchronize based on a clock, the system is controlled through communication protocol (often hand-shaking). The advantages with such a system are that is:

- "average-case" performance,
- no clock signal to distribute, and
- no energy for idle operations.

The disadvantages with the asynchronous design approach are that:

- limited EDA supports, and
- extremely difficult to test.

Chapiro [36] proposed a scheme named Globally-asynchronous locally-synchronous (GALS) where there is an asynchronous wrapper around each core/block and where each core itself is synchronous. FIFO queues are inserted at the boundaries between the asynchronous and the synchronous parts.

The advantages with GALS are:

- easy to design systems with multiple clock domains,
- no global clock to distribute,

- power efficient - system only operates when data is available,
- asynchronous circuits (and related problems) limited to the wrapper, and
- main functional blocks designed using conventional EDA tools.

Among the disadvantages is the area overhead for self-timed wrapper. Implementations has shown about 20% power saving at 10% area increase.

5 OPTIMIZATION

Optimization problems can be mapped to well-known problem. The advantage by doing that is that if it is known that the well-known problem is NP-complete, the mapped problem is also NP-complete. The *Knap-sack problem* and the *fractional Knap-sack problem* are interesting for scheduling problems. The Knap-sack problem where given items of different values and volumes, the problem is to find the most valuable set of items that fit in a knapsack of fixed volume. The decision problem, if an item should be included in the knap-sack or not, is NP-hard [42]. However, the *fractional Knap-sack problem* where given materials of different values per unit volume and maximum amounts, find the most valuable mix of materials which fit in a knapsack of fixed volume. Since we may take pieces (fractions) of materials, a greedy algorithm finds the optimum. Take as much as possible of the material that is most valuable per unit volume. If there is still room, take as much as possible of the next most valuable material. Continue until the knapsack is full and the result is an optimal solution [42].

The test scheduling problem for systems where all tests are given and all are assigned a fixed testing time and the objective is to minimize the test application time is in the case when sequential testing is assumed trivial. The optimal test application time $\tau_{application}$ is for a system with N tests each with a test time τ_i ($i=\{1..N\}$) given by:

$$\tau_{application} = \sum_{i=1}^N \tau_i \quad (2.1)$$

The assumptions that only one test at a time can be active at any moment, and that all tests have to be applied, means that any order of the tests is optimal. An algorithm iterating in a loop over all tests is required and at each iteration one test is selected and given a start time. The computational complexity of such an algorithm depends linearly on the number of tests, $O(n)$ - n is the number of tests, hence the algorithm is polynomial (P). Most problems cannot be solved, that is an optimal solution is found, in polynomial time. The



Figure 11. Gantt chart.

problems are then said to be *NP-complete* (non-deterministic polynomial time). Until today, there is no algorithm that in polynomial time guarantees to find the optimal solution for NP-complete problems. At present, all known algorithms for NP-complete problems require time which is exponential in the problem size [42]. Therefore, in order to solve an NP-complete problem for any non-trivial problem size, a heuristic can be used.

Optimization is the search for a solution that minimizes a given cost function where the ultimate goal is to find the *optimal solution* (no better solution can be found). An algorithm that guarantees to find the global optimum is an exact algorithm. In many cases, the problems are NP hard and a heuristic algorithm finds a solution that locally, not globally, optimizes the cost function. A heuristic *can* find the optimal solution, however, there is *no guarantee* that the optimal solution is found. A heuristic is often a trade-off between computational cost (CPU time) and computational quality (how far is the solution from the optimal solution).

In practice, the obvious problem with a heuristic is to evaluate the quality, that is how close is the solution produced from the heuristic compared to the optimal solution (the optimal solution is only guaranteed with exhaustive search, which is not possible due to time constraint). A lower bound can help a designer for instance in giving a feeling on how close to a lower bound a certain solution is. Often a resource limits a solution, and knowledge about the resources can guide the evaluation of the quality of a solution. A way to model resource utilization is to use a Gantt chart [23]. A Gantt chart can either be job (task)-oriented or resource oriented (figure 11). In figure 11(b) task_A and task_B both use resource_1 and it is obvious that resource_1 is used most, and hence, will most likely be the reasons for a bottleneck. A Gantt-chart can be used to define a lower bound on the test application time of a system, for instance. A lower bound is the lowest possible cost of a solution. Note that defining a way to compute a lower bound does not mean it is a way to find a solution corresponding to that the lower bound is found.

Above it was shown that it is trivial to develop an optimal test schedule in respect to test time for a sequential architecture when all tests are given a fixed test time and the objective is to minimize the test application time. In sequential testing, only one test at a time can be active, and that means that no constraint can limit the solution. In concurrent test scheduling, where more

than one test can be applied at a time, conflicts often limits the solution. The problem to minimize the test application time using concurrent test scheduling for a system where all tests are given a fixed testing time *under no constraint* is trivial. All tests are simply started at time point zero. The optimal test application time $\tau_{application}$ is for a system with N tests each with a test time τ_i ($i=\{1..N\}$) given by:

$$\tau_{application} = \max\{\tau_i\} \quad (2.2)$$

The concurrent test scheduling problem, more than one test can be executed at the same time, is in general not *NP*-complete. However, the concurrent test scheduling problem *under constraint* is *NP*-complete. We will discuss this in more detail below.

5.1 Optimization Techniques

The search for a solution can be done using exhaustive search or using some heuristics. The advantage of exhaustive methods is that an optimal solution is found. The problem is that most problems are complicated (*NP*-hard), which means that the computational effort (CPU time) is unacceptable for larger instances of the problem.

The most straight forward optimization approach is exhaustive search. It means that every single solution in the search space is evaluated and the best solution is reported. Below are a selection of such approaches, namely: *backtracking*, *branch-and-bound*, *integer-linear programming*. Iterative algorithms that iteratively search for an optimized solution but cannot guarantee that optimal solution is found are heuristics such as for instance: *local search*, *simulated annealing*, *genetic algorithms* and *tabu search*. These heuristics are iterative, that is from an initial solution an optimized solution is created through iterative modifications (transformations). A heuristic can be constructive - the solution is constructed from scratch. A constructive heuristic is used in an iterative transformation-based heuristic. The computational cost of a constructive algorithm is often in the range of N^x where $x=2, 3, 4$.

5.1.1 Backtracking and Branch-and-bound

The idea when using backtracking for an exhaustive search is to start with an initial solution where as many variables as possible are left unspecified [67]. The back-tracking process assigns values to the unspecified values in a systematic way. As soon as all variables are assigned to valid values, the cost of the feasible solution is computed. Then, the algorithm back-track to a partial solution assigning next value to the unspecified variables.

Often it is not needed to visit all solutions that are produced in the back-tracking process. Therefore the solution space is in back-tracking evaluated in a search tree. The idea in branch-and-bound is to not traverse sub-trees where the cost is higher than the best cost so far. If the cost of the solution of a partial specified solution is worse than the best solution, the sub-tree can be killed or pruned, which means that the sub-tree is not traversed. The search for a solution can be done in a depth-first or breath-first manner.

5.1.2 Integer Linear Programming

Integer Linear Programming (ILP) is a way to transform combinatorial optimization problems into a mathematical format. A high number of problems can relatively easily be reduced to ILP [67]. However, from a computational point of view, this does not help since ILP is NP-complete itself. Nevertheless, there exists a number of general ILP-solvers, which means that if the optimization problem can be modeled as an ILP instance, the software (program) solves the problem. The main advantage of using an ILP-solver is that an exact solution is found. The draw-back is, obviously, the computational cost for larger problems.

ILP is a variant of Linear programming (LP). LP problems can be solved the polynomial-time *ellipsoid algorithm*, however, it is in practice outperformed by the *simplex algorithm* which has a worst-case time complexity [67]. ILP is a variant of LP where variables can only be integers [67].

5.1.3 Local Search

Local search is an iterative method that search a *local neighborhood* instead of searching the whole search space when creating the new solution [67]. The idea is to search the neighborhood and by using modifications called move or local transform the current solution is improved. The best solution, found using exhaustive search in the local neighborhood (*steepest decent*), is used as the next solution. In practice the search in the local neighborhood can be terminated at *first improvement*. The first improvement makes use of less computational cost compared to the steepest decent. However, in steepest decent each iteration creates a better solution, which can lead to less required iterations, hence less computational cost, compared to using first improvement. In order to avoid getting stuck at local optimum, a larger neighborhood can be used. However, if a larger neighborhood is to be considered, higher computational cost is required to search the neighborhood. Local search does not allow moves out of local optimum, so called uphill moves, as allowed in Simulated Annealing and Tabu Search.

5.1.4 Simulated Annealing

Simulated Annealing (or statistical cooling) proposed by Kirkpatrick *et al.* [131] is an optimization technique that is analogous to a physical process [67]. Initially, the material is heated up so that the molecules can move freely (the material is liquid). The temperature is slowly decreased to cool down the material. The freedom of molecules to move is decreased with temperature. At the end, the energy of the material is minimal provided that the cooling speed was very slow.

The pseudo-code for Simulated Annealing is in Figure 12. An initial solution is created and in each iteration a random modification of the solution is allowed. Initially, at high temperatures, major modifications are allowed while as the temperature decreases smaller and smaller modifications are allowed. If a better solution than the previous is created, it is kept. And at a certain probability, in order to avoid local optimum, worse solutions are accepted.

```

Construct initial solution,  $x^{now}$ ;
Initial Temperature:  $T=TL$ ;
while stop criteria not met do begin
  for  $i = 1$  to  $TL$  do begin
    Generate randomly a neighboring solution  $x' \in N(x^{now})$ ;
    Compute change of cost function  $\Delta C = C(x') - C(x^{now})$ ;
    if  $\Delta C \leq 0$  then  $x^{now} = x'$ 
    else begin
      Generate  $q = \text{random}(0, 1)$ ;
      if  $q < e^{-\Delta C/T}$  then  $x^{now} = x'$ 
    end;
  end;
  Set new temperature  $T = \alpha \times T$ ;
end;
Return solution corresponding to the minimum cost function;

```

Figure 12. Simulated Annealing algorithm.

5.1.5 Genetic Algorithms

Genetic algorithm (pseudo-code in Figure 13) is inspired by nature and the theory of evolution [197]. An initial set of valid solutions called the population is created and in an iterative process, new solutions are created. The current population is replaced by a new population. In order to create a new solution, called a child, two solutions are selected, called parent1 and parent2. The selected parents create in a crossover process a child, which will belong to the new generation. The idea is that by combining features from the best solutions (selected parents) in current population, new better solutions (children) can be created. Each solution is characterized by its chromosome and the chromosome from two parent solutions creates a child solution. In order to avoid local optimum, mutation is allowed in the crossover process. A muta-

tion is, as in nature, the insertion of a small modification in the creation of the child solution.

```

Create an initial population pop
Do begin
    newpop=empty
    For i=1 to size_of_population Begin
        parent1=select(pop)
        parent2=select(pop)
        child=crossover(parent1, parent2)
        newpop=newpop+child
    End
    pop=newpop;
End;
Return solution corresponding to the minimum cost function;

```

Figure 13. Genetic algorithm.

5.1.6 Tabu Search

Tabu search is an optimization technique where the solution is improved through search in the neighborhood. An initial solution is created and a set of modifications are defined. The initial solution is through the defined modifications improved in n iterations. In order to avoid having a modification followed by a re-modification, a tabu list is used. The tabu list of length k keep track on the k last moved and prohibit cycles of length $< k$.



<http://www.springer.com/978-0-387-25624-5>

Introduction to Advanced System-on-Chip Test Design
and Optimization

Larsson, E.

2005, XX, 388 p.,

ISBN: 978-0-387-25624-5