3. Design Flow for Configurable Architectures

As FPGAs were the first reconfigurable devices, a number of approaches have been developed for synthesis of circuits onto these devices. Due to the fine granularity of FPGAs, those algorithms work on logic functions and large numbers of processing elements (which are also known as logic blocks for FPGAs). Although coarse grain architectures feature several advantages, which reduce the complexity of the synthesis process, the basic design steps are similar to those for FPGAs. Additionally, several algorithms from FPGA design have been adopted in existing design environments for coarse grain reconfigurable architectures. This chapter gives an overview about the general design methodology and common algorithms. Though, some existing coarse grain architectures examined later on have special architectural features (e.g. linear array topology), which simplifies some steps, enabling the use of proprietary greedy algorithms. Such algorithms are outlined in the examination of the according programming environment rather than in this section.

A coarse overview on the general synthesis process for reconfigurable architectures is given in figure 3-1. After the design is specified in the design entry step, a netlist is generated. On this netlist, optimizations are applied, before a technology mapping step takes place, which results in a technology mapped netlist. This netlist is ready for the mapping onto the architecture, which takes a placement and a routing phase as the main steps. After the placement and routing has been done, the configuration code to be downloaded onto the device can be generated. To verify the design, simulation is used. Typically, after the design entry, the design can be checked by functional simulation, while after the placement and routing a timing simulation is possible. The steps of the design flow will briefly be discussed in the following, taking focus on approaches used later.

3.1 Design Entry

In the design entry step, the designer creates a specification of the problem to be mapped onto the architecture. The result of the design entry phase is an intermediate data structure representing the application. The typical format for this data is a netlist, which resembles a directed acyclic graph, with vertices representing operations and edges describing the data flow between the operations. Netlists derived from a high-level input language, like a common programming language for microprocessors, may as well be reduced to a tree.

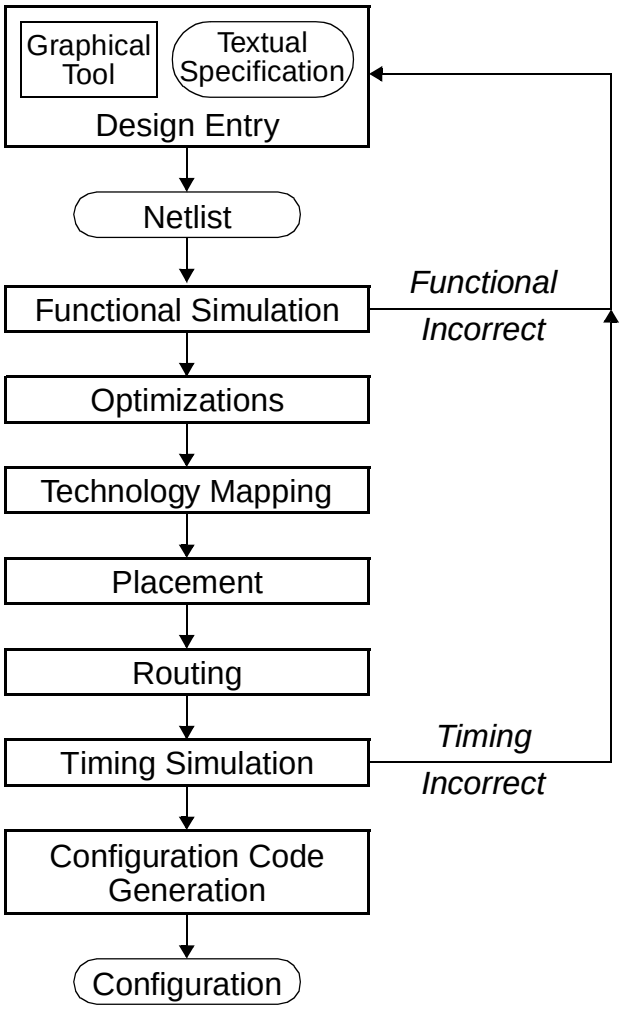

Figure 3-1: General design flow for reconfigurable architectures

There are basically two ways for the input of a design:

- Graphical design entry
- Textual design entry

Both possibilities are present in existing design environments for FPGAs as well as for coarse grained architectures. However, due to the differences between the architectural approaches, the typical ways of design entry differ also. In the following, general aspects of this step for both types of architectures are discussed.

3.1.1 Design Entry for FPGAs

The classic way of design entry for FPGAs is an interactive graphical schematic entry tool, which allows to build logic networks by selecting library elements and connecting them manually. Such tools are also part of VLSI electronic design automation (EDA) frameworks from different vendors. Examples for schematic entry tools are Viewdraw from View Logic Systems [Fawc94], Design Architect from Mentor Graphics [Ment93], or Composer from Cadence Design Systems [Cade91]. Most FPGA design environments, e.g. [Xili98], [Alte98], [Alte99], [Atme99], allow to make use of such existing schematic entry tools, integrating the FPGA design environment into the VLSI design framework.

While the graphical design entry is good for glue logic, it is not appropriate for behavioral designs like state machines or complex designs. For this purpose, a textual specification is more adequate, which describes the design using a hardware description language (HDL). Contemporary FPGA design environments typically support several of these languages. System-level HDLs like Very High Speed Integrated Circuit HDL (VHDL), Verilog, or JHDL [HuBe99] allow the description of a design at several levels of abstraction.

Another similarity between VLSI and FPGA design flow is given by the possibility for hierarchical designs, where the global design is split up into modules, which are themselves described in a sub-design. Hierarchical design is supported by both schematic entry tools and system-level HDLs. This approach provides also a certain degree of reusability of existing modules, which can be collected in libraries.

Due to the fine-grain nature of FPGAs, arithmetic designs like datapaths are still hard to handle. An attempt to ease the design of such circuits are parameterized module generators [Fawc94] provided by some FPGA vendors, e.g. LogiBLOX from Xilinx#[Xili98] or the macro generators from Atmel [Atme99]. These generators accept a generic description of a complex logic function and automatically produce an implementation of this function for the target architecture.

3.1.2 Design Entry for Coarse Grained Architectures

As coarse grained architectures are mostly targeted for computational purposes, existing design environments typically use textual specifications for design entry. In contrast to FPGA design environments using HDLs, the input languages for coarse grain architecture programming frameworks are either high-level programming languages, like C or FORTRAN (e.g. for the Garp architecture or RAW machines), or proprietary languages similar to common programming languages (e.g. DIL for PipeRench, ALE-X for the KressArray, or RaPiD-C). The latter type of languages often contains special constructs to support specific

features of the according architecture. Furthermore, the proprietary languages are often used as an intermediate form in a more general programming environment, which is capable to process a common programming language and extract the parts to be implemented by the reconfigurable architecture. This is typically done by a software partitioning step (see figure 3-1). An example for such an environment is the CoDe-X framework described in [Beck97], [Schm94]. Such environments often apply optimizations known from compiler technology, e.g. loop transformations or vectorization. Although microprocessor-like programming languages are the general approach for coarse grained architectures, some environments rely on HDLs (e.g. for the CHESS array) or generally spoken, a structural description of the algorithm to be mapped (e.g. for the Colt architecture).

If a high-level programming environment is not available for an architecture, the textual description resembles a type of assembler code as known for microprocessors. This level of description compares to the configuration code of FPGAs. Some existing architectures, which are programmed at this level, feature also interactive graphical tools to support the design entry (e.g. MorphoSys or PADDI-2). However, the entry level is higher than the logic level of FPGAs, featuring complex arithmetic operators the architecture can perform. Due to the general high level of programming, concepts like hierarchical design and parameterizable module generators are currently rather uncommon for coarse grained architectures.

3.2 Netlist Optimizations

Before the netlist is processed further on, most design environments apply optimizations to simplify the following mapping task. For FPGAs, those operations are performed on logic level [SGR93], while for coarse grained architectures, the optimizations take place at operator level, like in compiler design [ASU86]. Though, the basic techniques are the same for both approaches. In the following, some common optimizations are sketched briefly.

Common Subexpression Elimination. This optimization tries to identify multiple occurrences of identical subexpressions, where the value of each subexpression does not change during the computation. Instead of computing the subexpression each time it occurs, it is computed only once and the result is used later on. On the logic level used by FPGAs, the elimination of common subexpressions is called extraction, if it spans several expressions, and decomposition, if the subexpression occurs several times in a single expression. An example for common subexpression elimination is shown in figure 3-2.

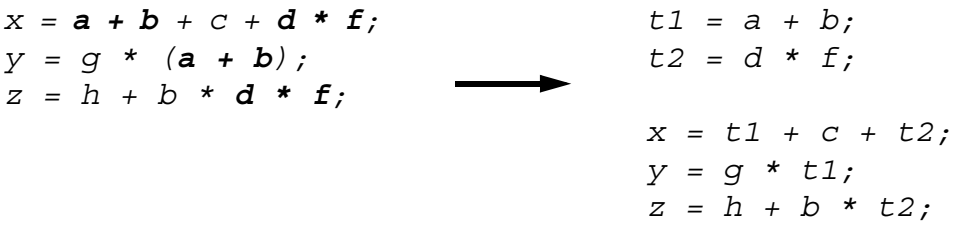

Figure 3-2: Example for the elimination of common subexpressions

Tree-Height Reduction. This optimization tries to exploit inherent parallelism in an expression by allowing several parts to be calculated concurrently, thus reducing the number of calculation steps [Kuck78]. This is done by balancing the according expression tree, as the tree-height corresponds to the number of computing steps. A simple technique uses associative and commutative laws for the operators to reduce the tree height. The effect of this optimization is shown in figure 3-3, with the parentheses around the multiplication for illustrative purposes.

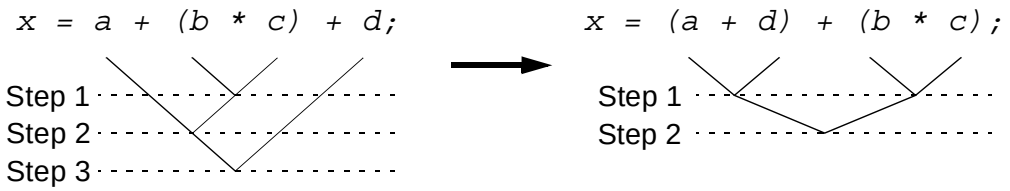

Figure 3-3: Simple tree-height reduction using associativity and commutativity

While the above technique preserves the number of operations, the use of the distributive law for tree-height reduction inflicts an increase of operators. For example, the expression *y = a * (b * c * d + e)* with four operators is computed in four steps. With the distributive law applied, the equivalent expression *y = a * b * c * d + a * e* features five expressions, but requires only three steps. If the distributive law can be applied the other way, both tree-height and operator count may be decreased. This process is called factorization. For example, *y = a * b * c + a * b* can be factorized into *y = (a * b) * (c + 1)*. The techniques mentioned above can also be applied at logic level [SGR93].

Algebraic Simplifications. These class of optimization tries to simplify the computation by identifying parts that can be calculated at compile-time and replacing the resulting obsolete code. For example, the expression *a - a* can be

replaced by the constant value *0*. A special class of simplifications applies to constants. The technique constant folding calculates expressions containing constant operands, using the result in subsequent computation stages. For boolean expressions at logic level, optimizations equivalent to these techniques are the identification of don't care terms and tautology checking [SGR93].

3.3 Technology Mapping

The technology mapping phase solves the problem of transforming the netlist with operators from the source file into another netlist, where the nodes are operators which the target architecture can handle. In the case of FPGAs, these operators are logic blocks, while for coarse grained architectures, the operators are more complex. Also, the netlist in an FPGA design flow will typically contain logic gates at bit-level, while for coarse grained architectures, the nodes in the netlist will be multi-bit operators, like the ones of the target architecture. However, in the last years there has been some research effort to map also coarse grained netlists onto FPGAs. Thus, there are three cases of the technology mapping problem that can be distinguished:

- Mapping logic-level netlists onto fine grained architectures
- Mapping operator-level netlists onto fine grained architectures
- Mapping operator-level netlists onto coarse grained architectures

Although some coarse grained architectures support fine grained operations, the case of mapping logic-level netlists onto coarse grained architectures is not considered, as those architectures are generally designed for computing applications. In the following, each technology mapping problem is discussed briefly.

3.3.1 Mapping Logic-Level Netlists onto Fine Grained Architectures

The mapping of logic-level netlists onto fine grained architectures is the standard case occurring in FPGA design flow. For lookup table (LUT) based FPGAs, each LUT can implement any boolean function with a certain number of input variables. Thus, the mapping includes packing of several netlist nodes into one LUT. There are two major categories for mapping algorithms [SGR93]:

Algorithmic Techniques. These approaches map the netlist into a graph consisting of two-input NAND gates, called the subject graph. The possible functions, which can be performed by the logic blocks are stored in a library, also as graphs of such NAND gates. These library graphs are also called pattern

graphs. The mapping is done by finding the minimum cost cover of the subject graph by the pattern graphs. Both subject and pattern graphs are directed acyclic graphs (DAGs). As the graph covering problem for such graphs is NP-hard, heuristics are used. A widely employed method is to break up the graph into trees, transforming the problem into a tree-covering problem, which can be solved in linear time. This was first used by the DAGON system [Keut87]. Other well-known approaches are MIS [DGR87], as well as Chortle [FRC90], [Fran93] and Chortle-crf [FRC91], which do not use libraries, but map the netlist directly into LUTs.

Rule-Based Approaches. Rule-based mapping techniques traverse the netlist and replace subnets with patterns representing the gates in the library that match the function of the subnet. Such approaches are slower but could yield better results since detailed information about the library gates can be captured and electrical considerations can be taken into account. An example for a system employing a rule-based technique is LSS [DJG84] by IBM.

3.3.2 Mapping Operator-Level Netlists onto Fine Grained Architectures

An approach to do this kind of mapping consists in replacing each complex operator in the netlist by an according logic network from a library and feeding the resulting logic-level netlist into an FPGA design flow. However, this approach performs poorly, as the netlist has to be flattened, resulting in a loss of the regularity information provided by the typical bit-slice layout of the complex operators. In the subsequent placement and routing phase, an irregular placement requiring complex routing will probably result. This can be avoided by mapping each node into a pre-mapped module, which retains the regularity, but may lead to underutilization of LUTs as optimization over module boundaries is not possible. An approach by Koch [Koch96a], [Koch96b] performs a module compaction step after module selection and placement to address this problem. The FAST system [NBK94] identifies groups of operator nodes, which can be merged and maps those groups into optimized modules. The GAMA tool by Callahan et al. [CCDW98] employs a tree-covering algorithm at module level combined with linear programming to preserve regularity information.

3.3.3 Mapping Operator-Level Netlists onto Coarse Grained Architectures

As coarse grained architectures are typically designed for implementing datapaths, the operator repertory often matches the set of possible operators in the netlist. Thus, a direct mapping can be performed. An exception occurs, if an operator in the netlist is too complex to be implemented by a single processing

element. This case is similar to the problem described in the previous section. It is solved by library-based approaches, replacing the complex operator by a net of more simple operators. One existing approach for the PipeRench architecture [BuGo99] uses direct library mapping, without considering the loss of regularity discussed above. However, the library elements are optimized when being instantiated and the placement and routing for PipeRench can be done by greedy algorithms due to the architectural features. For the Garp architecture, a more complex approach [CCDW98] is employed, which addresses the preserving of regularity information and the utilization of the processing elements.

3.4 Placement

The placement phase assigns the nodes of the technology-mapped netlist to physical processing units of the target architecture. These units are logic blocks for FPGAs or processing elements for coarse grained architectures. The placement has a significant impact on the following routing phase. Thus, this process is driven by an objective function, which typically reflects the routability of the placement. As the problem is quite similar for reconfigurable architectures and VLSI design, many algorithms from design for VLSI circuits or mask-programmable gate arrays can be applied with some modifications [SaYo95]. However, while a bad VLSI placement may result in big requirements for routing area and long wire lengths, a bad result of this process for reconfigurable architectures may lead to a placement which is not routable at all, as the number of available routing resources is fixed.

The placement problem for reconfigurable architectures can be defined as follows: Let $U = \{u_1, u_2, ..., u_n\}$ be a set of processing units, $O = \{o_1, o_2, ..., o_m\}$, with $m \eth n$ be a set of operators, represented by nodes of a netlist. The placement problem consists in finding a mapping function $P : O \rightarrow U$, which satisfies the following conditions:

- No two operators in O are mapped onto the same unit.
- The placement P is routable.
- The objective function for P is optimized.

Sometimes, O and U do not resemble the sets of all nodes of the netlist or processing units of the array respectively. Instead, a subset of O may be fixed, i.e. pre-assigned to specific processing units (which in turn are a subset of U). Only the remaining operators are assigned to the remaining units during the placement process.

Objective Functions. The objective function which is aimed to be minimized is typically the total wire length, which generally corresponds directly with the routability and the performance of the placement. Some placement approaches consider also explicitly timing and performance issues [MBR00], [RiEt95], [NaRu95], [SwSe95], [Srin91], [SCK91]. As the actual exact value of the objective function can be determined not before the routing has been done, the objective has often to be estimated. For example, a widely-used measure for the estimated wire length of a net is the size of the minimal bounding box containing all terminals [BeRo97]. For performance-driven placement, the objective may be to minimize the estimated length of the critical paths, while minimizing the total wire length will rather sum up all estimations [SaYo95], [Sher95].

A quite different approach does not rely on estimations for the objective but performs actual (sometimes partial) routing during the placement process [Ebel00], [HaKr95], [NaRu95]. This approach of simultaneous placement and routing has the potential for better solutions than separate phases, but involves typically more computation time, making it not suitable for large numbers of cells to be routed. However, for coarse grain architectures, the time requirements are still moderate, making this approach a viable alternative.

The complexity of the placement problem depends on the target architecture, but for most real-world applications, is known to be NP-hard. Thus, heuristics are used in order to get a good placement in a moderate computation time. The placement algorithms most widely used today can be distinguished into three major classes [BRM99]:

- Analytic algorithms
- Partitioning-based algorithms
- Simulation-based algorithms

Often, actual placers combine several of the above algorithm types in order to counter specific drawbacks. In the following, analytic and partitioning-based algorithms are sketched briefly. Simulation-based algorithms are examined in more detail, as they are most widely used for reconfigurable architectures.

3.4.1 Analytic Placement Algorithms

Analytic approaches try to calculate the ideal position of each cell by transforming the placement problem into a numerical optimization problem, which is then solved by techniques like quadratic programming or non-linear programming. If the objective function is the wire length, the basic idea behind this approach can be modeled by a mechanical equivalent, with the cells to be placed connected by nets resembling bodies connected by springs, which exert attractive forces to other connected bodies. The force applied by each spring is proportional to the distance of the attached bodies. If the mechanical system is left

on its own, the bodies move, until the system achieves an equilibrium of forces. Considering the original placement problem, this equilibrium state resembles the optimal placement for the cells. Approaches for a numerical solution to this problem have been published e.g. in [Quin75], [AJK82]. Other analytic approaches use timing or performance for objective functions rather than the wire length, e.g. [SCK91], [Srin91], [RiEt95]. The described numerical methods can also be combined with other algorithms, e.g. partitioning-based placement approaches [HuKa97], [KSJA91], [SDJ91].

3.4.2 Partitioning-Based Algorithms

This approach tries to group connected cells together. This is done basically by repeated partitioning of the netlist into two sub-netlists. Concurrently, the available array of cell positions is partitioned in a similar way, so each sub-netlist is assigned to a subsection of the array. The process is repeated until each sub-netlist consists of a single cell, which has then a unique place in the array. The different partitioning approaches are distinguished by the way the array and the netlists are partitioned.

For the partitioning of the array, mostly bipartitioning is applied [KSJA91], [SDJ91], often with alternate horizontal and vertical partitioning. Other approaches use a quadrisection of the array [HuKa97]. A common optimization objective is to minimize the number of nets crossing the cut-line between the sub-netlists. This approach is also known as min-cut placement [Breu77a], [Breu77b].

The prototype iterative heuristic for the partitioning of the netlist is the approach of Kernighan and Lin [KeLi70], where a partitioning is optimized iteratively in several passes. In each pass, every cell is moved once between two partitions by swapping pairs of cells in a way that the highest gain for the objective is achieved. After all cells have been moved, the encountered partition with the best value for the objective is restored and used as start for the next pass. The process is repeated until no further improvement of the objective is achieved. A common alternative to this approach is the method of Fiduccia and Mattheyses [FiMa82], which has only linear complexity.

3.4.3 Simulation-Based Algorithms

Simulation-based algorithms mimic a natural process or phenomenon, which is known to produce a state or an output, which is optimized for a given objective. The three approaches discussed here are force-directed placement, simulated annealing, and genetic algorithms.

3.4.3.1 Force-Directed Placement

This algorithm simulates a mechanical system of bodies connected with springs, as has been discussed in the above section 3.4.1 for analytic placement approaches. The analyticl approaches employ numeric optimization to minimize the potential energy of the system, thus considering all cells simultaneously and calculating the ideal placement. In contrast to this, the approaches covered in this section simulate the mechanical system, implementing a constructive placement procedure. Starting with an initial placement, one cell at a time is selected and its optimal location (with zero potential force) is computed. This process is iterated to improve the solution. There are several versions of this approach, which differ in the action taken if the target location for the cell to be moved is occupied. One possible option consists in displacing the cell at the target location, and use the displaced cell directly as the next one to be placed onto its optimal location. Such an algorithm is described in [ShMa91].

3.4.3.2 Simulated Annealing

Simulated Annealing is one of the most well developed placement methods [Sher95]. It has been successfully used in the classic TimberWolf system [SeSa86], [SwSe95], [SuSe95] for VLSI placement, and is also the prevailing method used for placement of reconfigurable architectures [BRM99], [BeRo97], [MBR00], [EMHB95]. One advantage of this algorithm is the possibility to change the objective function easily without having to alter the basic algorithm. Thus, there exist placement tools based on simulated annealing, which can be adapted to different architectures [BeRo97], [MBR00].

Like the name suggests, simulated annealing mimics the process of gradually cooling molten metal. Basically, the metal tends to form a crystal grid, which resembles the state containing the least energy and the desired result of the annealing process. However, if the cooling happens to fast, the metal atoms will rather form a suboptimal structure, as there is no thermal energy for them to leave the suboptimal positions they have reached in the fast cooling process. This situation corresponds to a local optimum in general optimization. The solution to this problem consists in a slow decrease of the temperature, which leaves the atoms enough mobility to leave suboptimal positions and form a globally better structure. The simulation of this process [MRRT53] can be used to solve general optimization problems [KGV83]. A generic algorithm of simulated annealing for placement in pseudo-code [BRM99] is shown in figur e3-4.

The algorithm consists of an outer loop, which realizes the cooling schedule of the process by decreasing the temperature, and an inner loop, which simulates the annealing at each temperature step by generation and eventually acceptance of new placements. A new placement is typically produced by selecting two cells randomly and exchanging their positions. The algorithm starts by determining

```
Algorithm Simulated Annealing
{     /* Initialize variables */
      T = InitialTemperature() ;
      P = InitialPlacement() ;
      R = InitialRange() ;
      /* "Outer loop" */
      WHILE ( ExitCriterion() == FALSE )
      {     /* "Inner loop" */
            WHILE ( InnerLoopCriterion() == FALSE )
            {     /* Generate a new configuration */
                  P_new = GenerateMove( P,R ) ;
                  ΔC = Cost( P_new ) - Cost( P ) ;
                  X = Random( 0,1 ) ;
                  /* Check, if new configuration is accepted */
                  IF ( ΔC < 0 OR X < e^(-ΔC/T) )
                  {     P = P_new ;
                  }
            }     /* End of inner loop */
            /* Update Temperature and Range */
            T = UpdateTemperature() ;
            R = UpdateRange() ;
      }     /* End of outer loop */
}     /* End of algorithm */
```

Figure 3-4: Generic Simulated Annealing algorithm for placement

an initial placement, an initial temperature, and an initial range. The range parameter is used in some approaches to control the generation of new states. It denotes the maximum distance allowed between cells selected for swapping. The range is typically high at the beginning and is decreased during the process to allow only fine-tuning of the placement at the end. The outer loop is controlled by an *ExitCriterion*, which determines the end of the annealing process.

For each temperature step inside the outer loop, a number of new states are generated. This number needs not to be fixed. Instead, the inner loop is controlled by an according *InnerLoopCriterion*. In each step, a new placement is generated. A cost function representing the objective of the placement is applied and the change in quality of the states is measured. If the new placement is better than the old one, it is accepted. If it is worse, it may still be accepted with a certain probability ($e^{-\Delta C / T}$), which depends on the current temperature. When the temperature is high, almost every new state is accepted, while at lower temperatures, the probability to accept a state that makes the placement worse is

also low. Note, that the term "$\Delta C < 0$" is obsolete, as if it is true, also the second condition ("$X < e^{-\Delta C / T}$") holds. It has been included for illustrative purposes. After a number of changes have been tried out in the inner loop, the temperature and the range are updated.

The objective of the annealing is expressed by the cost function ("Cost"). For placement, objective functions as discussed above, like the total wire length, can be used. Unlike other algorithms, simulated annealing allows easy changing of cost functions or the combining of several objectives into one.

The start temperature, the rate at which the temperature is decreased (*UpdateTemperature*), the exit criterion for the outer loop, the number of moves at each temperature (*InnerLoopCriterion*) and the method of generating new states (*GenerateMove*) determine the annealing schedule, which has a significant impact on the quality of the placement. While simple schedules use fixed values for initial values and a simple function for temperature decrease, a number of adaptive schedules has been published, which consider properties of the individual placement problem (like the number of cells) and feedback gathered during the annealing process. As a general guide for adaptive schedules, it was found that it is desirable to keep the rate of accepted moves near the value 0.44, as for this value, the temperature can be decreased most quickly without losing quality of the result [LaDe88], [SwSe90].

For the determination of a suitable start temperature, one approach published by Huang et al. [HRS86] uses an estimate derived from the standard cost derivation of sample moves. Another, more complex method based on measurement of the probability distribution of the changes in the cost function has been published in [RKW90]. Another part of the cooling schedule is the calculation of the next temperature (*UpdateTemperature*) and the *InnerLoopCriterion* determining the number of steps. Different adaptive schedules have been proposed in [HRS86], [LaDe88], [SwSe90]. The first two schedules involve a quite complex calculation for either the *InnerLoopCriterion* or the temperature decrease, but provide a good adaptability to the problem. The last approach features less complexity sacrificing adaptability. A schedule proposed in [BRM99] tries to combine the advantages of the three previous works. This schedule uses a start temperature determined from sample moves like in [HRS86], a number of moves per temperature depending on the number of cells, and decreasing-functions for both the range and the temperature, which consider the achieved acceptance rate for the previous temperature.

Besides the simulated annealing described above, derivative algorithms have been published for general optimization, which show some advantages compared to the original method. These are the Threshold Accepting and the Great Deluge Algorithms, which are in fact simplifications of general simulated annealing. Both algorithms will be discussed briefly in the following.

Threshold Accepting Algorithm. This derivative has been published in [DuSc90]. The pseudocode for the threshold accepting algorithm following the form chosen for the simulated annealing in figure 3-4, is shown in figure 3-5.

```
Algorithm Threshold Accepting
{     /* Initialize variables */
      T = InitialTemperature() ;
      P = InitialPlacement() ;
      R = InitialRange() ;
      /* "Outer loop" */
      WHILE ( ExitCriterion() == FALSE )
      {     /* "Inner loop" */
            WHILE ( InnerLoopCriterion() == FALSE )
            {     /* Generate a new configuration */
                  P_new = GenerateMove( P,R ) ;
                  ΔC = Cost(P_new) - Cost(P) ;
                  X = Random( 0,1 ) ;
                  /* Check, if new configuration is accepted */
                  IF ( ΔC < T )
                  {     P = P_new ;
                  }
            }     /* End of inner loop */
            /* Update Temperature and Range */
            T = UpdateTemperature() ;
            R = UpdateRange() ;
      }     /* End of outer loop */
}     /* End of algorithm */
```

Figure 3-5: Generic Threshold Accepting algorithm for placement

The difference between threshold accepting and simulated annealing lies in the removal of the acceptance probability. Instead, a move is always accepted, if the quality change of the placement lies within a threshold defined by the temperature. Although this algorithm has not yet been applied to placement problems to the author's knowledge, it is claimed to be superior to simulated annealing for other problems, achieving better results according to [DuSc90]. However, this algorithm still requires a good cooling schedule.

Great Deluge Algorithm . This algorithm is a further simplification of simulated annealing, published by Dueck in [Duec93]. The pseudo-code of the great deluge algorithm is shown in figure 3-6.

```
Algorithm Great Deluge
{     /* Initialize variables */
      T = InitialTemperature() ;
      P = InitialPlacement() ;
      R = InitialRange() ;
      /* "Outer loop" */
      WHILE ( ExitCriterion() == FALSE )
      {     /* Generate a new configuration */
            P_new = GenerateMove( P,R ) ;
            C = Cost( P_new ) ;
            /* Check, if new configuration is accepted */
            IF ( C < T )
            {     P = P_new ;
            }
            /* Update Temperature and Range */
            T = T - Constant ;
            R = UpdateRange() ;
      }     /* End of outer loop */
}     /* End of algorithm */
```

Figure 3-6: Generic Great Deluge algorithm for placement

The great deluge algorithm does neither employ a probability for the acceptance nor an inner loop to try several moves for one temperature step. Instead, the total cost of the placement is considered. A new placement is always accepted as long as its total cost lies below the current temperature. This algorithm features the major advantage that no cooling schedule is needed. According to [Duec93], it achieves results which are similar in quality to those from the threshold accepting algorithm described above. However, the great deluge algorithm takes typically twice as much computation time.

3.4.3.3 Genetic Algorithm

The genetic algorithm technique [Holl75] simulates the natural evolution of a population of individuals, which have to adapt to their environment in order to survive. Those individuals will produce offspring, whereby individuals which are better adapted will be preferred. Thus, according to the rule of "survival of the fittest", the population will migrate towards optimal fitness. By transforming a desired optimization objective to a fitness function, the genetic algorithm can be used for general optimization problems.

The genetic algorithm for placement starts with an initial set of random configurations (the population). Each individual in the population resembles a solution to the placement problem. In an algorithm implementation, an individual is represented e.g. as a string of symbols. The symbols are known as genes, and the whole string is termed a chromosome. Thus, a chromosome describes a valid placement solution. Besides such complete solutions, there are also partial solutions possible which are termed schemas. An example for a schema is a set of genes describing the placement for a subset of cells, with no information about the locations of the rest.

The simulation is done by iteratively producing new individuals in generation steps. During each iteration, all individuals in the current generation are evaluated using a cost function (which is also called fitness function) describing the optimization goal. New individuals are created from existing parent individuals by combining their features stored in the chromosomes. The selection of the parents is done using the fitness value resulting from the mentioned evaluation. The higher this value is for an individual, the higher is the probability for it to be selected for reproduction. The actual generation of new offspring chromosomes is done by applying a number of genetic operators mimicking natural effects happening to DNA sequences. Common operators for the genetic algorithm are crossover, mutation and inversion. Crossover combines parts of two parent chromosomes to create one offspring chromosome. Mutation affects a single chromosome, applying a random change to the information. Inversion applies also to a single chromosome, typically mirroring a part of it. It must be noted, that inversion does not change the information resembled by the chromosome (in contrast to mutation), but only its representation. After the offspring has been produced, the new population is generated by selecting some individuals from both the new offspring and the parents, keeping the population size constant. The selection is again based on the fitness of the individuals. Due to the selection process, individuals with bad fitness will tend to be sorted out, while individuals with good fitness will tend to survive. Thus, the fitness of the new generation will tend to improve. This general genetic algorithm is shown as pseudo-code in figure 3-7. In order to apply the genetic algorithm for the placement problem, the representation of the placement, a selection strategy, and suitable genetic operators have to be determined. In the following, example solutions found in the literature are discussed briefly.

Placement Representation. The classic structure of a chromosome is a string of symbols. Approaches employing a direct representation are described in [ShMa91] and in [CoPa86]. Here, each string position describes the corresponding location of the placement in row-major order. Thus, each string position holds the identification of the corresponding cell mapped to this position. A different

```
Algorithm Genetic Algorithm
{     /* Initialize variables */
      Np = Population_Size ;
      Ng = Number_of_Generations ;
      No = Number_of_Offsprings ;
      Pm = Probability_for_Mutation ;
      Pi = Probability_for_Inversion ;
      /* Generate initial population */
      Population = Construct_Population( Np ) ;
      /* Evaluate fitness of each individual */
      FOR i=1 TO Np
      {     EvaluateFitness( Population[ i ] ) ;
      }
      /* Iterate for Ng generations */
      FOR i=1 TO Ng
      {     /* Create No individuals offspring */
            FOR j=1 TO No
            {     /* Choose parents due to fitness value */
                  X = ChooseParents( Population ) ;
                  Y = ChooseParents( Population ) ;
                  /* Generate offspring by applying crossover */
                  Offspring[ j ] = CrossOver( X,Y ) ;
                  /* Apply mutations with probability Pm */
                  /* and inversions with probability Pi */
                  FOR k=1 TO Np
                  {     IF ( Random( 0,1 ) < Pm )
                        {     ApplyMutation( Population[ k ] ) ;
                        }
                        IF ( Random( 0,1 ) < Pi )
                        {     ApplyInversion( Population[ k ] ) ;
                        }
                  }
                  EvaluateFitness( Offspring[ j ] ) ;
            }     /* End of offspring loop */
            Population = Select( Population, Offspring, Np ) ;
      }     /* End of generation loop */
      Solutionindex = FindHighestFitness( Population ) ;
      RETURN ( Population[ Solutionindex ] ) ;
}     /* End of algorithm */
```

Figure 3-7: Generic Genetic Algorithm for placement

representation based on trees has been used in the approaches published in [CHMR91], [Esbe92] and [ScVo97]. These works describe algorithms which can also be used for the more general floorplanning problem [SaYo95], [Sher95].

Selection Method. After the offspring has been generated, a number of individuals from both the parents and the offspring has to be selected in order to keep the population size constant. Three common methods are described briefly in the following.

- In competitive selection, the parents and offspring compete with each other, and the N_p fittest individuals are selected.
- In random selection, as the name implies, the N_p individuals for the next generation are selected randomly from the parents and the offspring.
- Stochastic selection is similar to the process of parent selection described in the general algorithm above. An individual is selected for the next generation with a probability corresponding to its fitness. Thus, this method combines competitive selection and random selection.

Crossover Operator. This genetic operator is used to produce a new chromosome from two parent chromosomes. Assuming a string representation of chromosomes, the basic method consists in placing a cut at a random position on both parent chromosomes and merge the part before the cut of the first parent with the part after the cut of the second parent. However, this direct approach can not be used for the placement problem, as it must be assured to generate only chromosomes which represent a valid placement. The direct approach may result in an invalid placement, as shown in an example in figur e3-8a, with cells B and E appearing twice. Due to this problem, the crossover operators for placement are typically more complex. A variety of such operators have been developed [ShMa91]. An example for an operator giving valid results is shown in figure 3-8b. The order crossover copies the part of parent α before the cut to the offspring. The rest of the offspring chromosome is composed of the missing genes from α, but they are arranged in the order they appear in parent β.

Mutation Operator. Mutation applies small random changes to a single chromosome. In the evolution process, this serves the important role of providing new genetic information which was not present in the original population or which has been lost during the selection process and can be tried in a new context. Mutation is controlled by a corresponding probability P_m denoting the mutation rate. In the genetic algorithm, mutation is typically implemented by random exchange of two genes in the chromosome. An example is shown in figure 3-9.

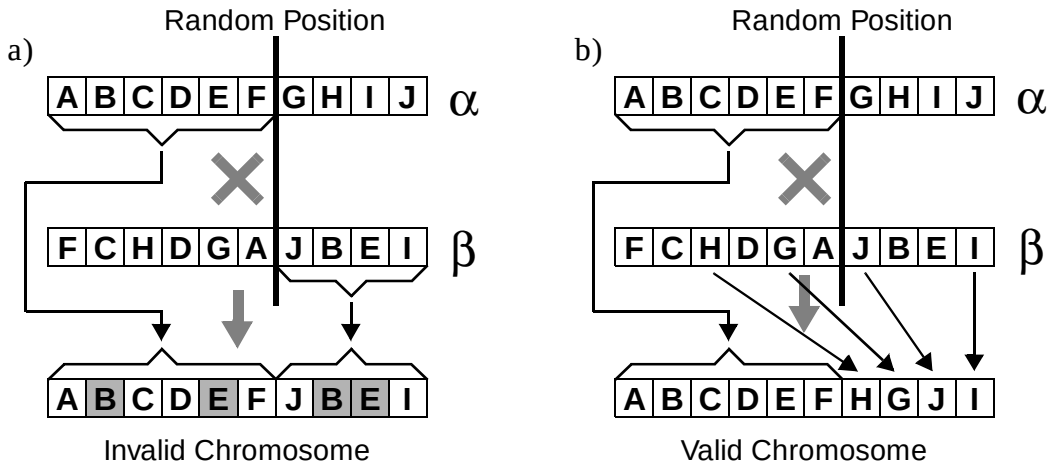

Figure 3-8: Crossover operator for genetic algorithm:
a) direct approach resulting in an invalid placement,
b) order crossover as example for a valid operator

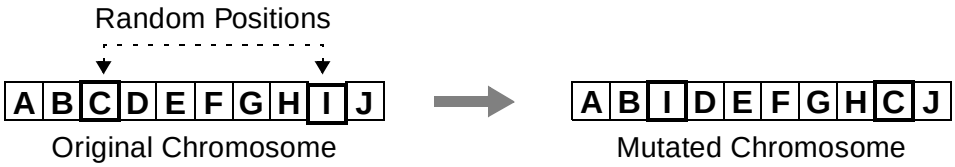

Figure 3-9: Mutation operator for genetic algorithm

Inversion Operator. Like mentioned above, the inversion operator does not change the information of a chromosome, but only the representation of the information. It is not found in every genetic algorithm (e.g. [CoPa86]) and is typically implemented by selecting two cut positions and reverting the segment in-between. The main task of this operation is to allow groups of genes, which stay together in subsequent crossover operations. Similar to mutation, inversion is controlled by an inversion probability P_i. The operator requires a representation, which is independent of the order of the genes, thus a simple placement encoding like the one in figure 3-8 and figure 3-9 can not be used. A possible adequate encoding consists of each gene holding the cell identification as well as the coordinates of the cell in the placement. An example for the inversion operator using this encoding is shown in figure 3-10.

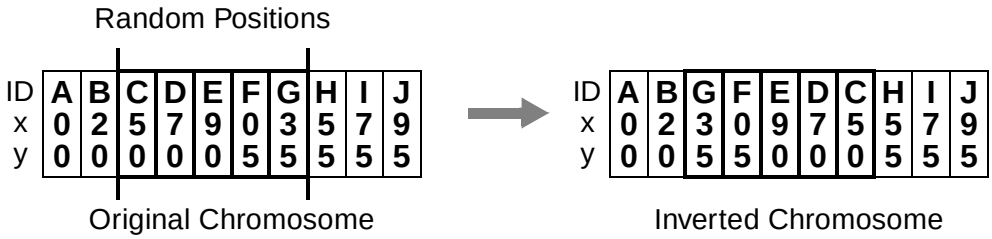

Figure 3-10: Inversion operator for genetic algorithm

3.4.4 Other Placement Algorithms

Besides the common approaches described in the previous subsections, placement algorithms have been developed, which do not belong to one of the above classes. In this section, a variation of the cluster growth algorithm, which has been actually used for placement of reconfigurable architectures, will be sketched briefly.

3.4.4.1 Hierarchical Clustering Algorithm

Hierarchical clustering is a constructive algorithm which tries to build a layout by placing one cell (called the seed cell) and placing other cells, which are heavily connected to the seed, nearby. This way, clusters of cells are formed. In the hierarchical clustering approach published in [SaRo99], the clusters are organized in a predefined number of levels, each level made up of a certain number of subclusters. The clusters are built by randomly selecting seed cells and joining other cells based on their connectivity. The connectivity of a candidate is measured by the number of connections between the candidate and the cluster being constructed, and the number of nets which would be absorbed if the candidate would be added to the cluster. Here, a net is called absorbed, if all cells on that net would lie inside the cluster. The clustering algorithm has a linear execution time, but, being a greedy algorithm, is known to produce suboptimal results. Thus, in the approach described in [SaRo99], the clustering is followed by a simulated annealing.

3.5 Routing

Once the placement of the cells in a reconfigurable architecture is done, the routing phase determines which reconfigurable switches are to be activated to form the connections between the cells described by the netlist. Depending on the architecture properties, this task can have different complexities and is sometimes solved by proprietary algorithms. In this subsection, common routing

approaches developed for two-dimensional reconfigurable architectures are sketched. Most approaches are adaptations of algorithms from VLSI design. However, routing for reconfigurable architectures differs basically from VLSI routing, as the routing resources are fixed and can not be extended, if necessary.

The general routing problem can be described as follows: Given a set of cells with ports on the boundaries, a set of nets describing ports of cells to be connected together, a set of routing resources among the cells, and a placement of the cells, the routing problem consists of finding a set of paths to connect the cells according to the nets. Those paths are to be formed using the routing resources (which e.g. resemble routing area in VLSI design, switchboxes and wires in design for FPGAs). The routing solution is subject to certain constraints and should optimize a given objective function. For reconfigurable architectures, the constraints are given by the fixed amount of routing resources, which imply that none of the existing wires may be occupied by more than one net. There are different possible objective functions. The most common objective is to simply minimize the path length, as longer paths occupy extra routing resources. An additional routing goal is to prefer nets on the critical path to be routed using short paths and fast routing resources. These approaches are called timing-driven, while routers neglecting the timing are called routability-driven.

Another classification for routers is based on the number of steps done. Combined global-detailed routers generate a complete routing in one step. Examples for such algorithms are described e.g. in [EMHB95], [ACGR94], [AlRo95], [LeWu95], [WuMa95]. In contrast to this, two-step routing algorithms perform global routing [Rose90], [CTZW94] and detailed routing [BRV92], #[LBV97] as two subsequent steps. While the global routing determines the channels of wires to be used, the detailed routing does the actual selection of the wires in the channels. As a third category in this classification, some approaches combine the routing phase with the placement step [NaRu95].

Routing algorithms used for reconfigurable architectures are typically those known from VLSI design as maze routers. The classic maze routing algorithm by C. Y. Lee [Lee61] assumes the whole layout to be a rectangular grid. The cells to be wired, the routing resources, and obstacles which are to be avoided, are a subset of grid locations, with already routed paths resembling also obstacles. The Lee algorithm, which is sketched below, finds the shortest path between two arbitrary grid points, given such a path exists. For the general routing problem, the Lee algorithm is applied sequentially to all nets, as only one net at a time is routed.

The basic Lee algorithm has several drawbacks [SaYo95]. One problem for reconfigurable architectures lies in the grid representation of the layout, which is not adequate to represent the often nonuniform routing resources of reconfigurable architectures. This problem can be handled by employing a graph

representation of the routing architecture known as the routing resource graph described below. For this representation, the Lee algorithm is replaced by Dijkstra's algorithm [Dijk59], which finds the shortest directed path between two vertices in a graph with weighted edges.

A more general problem relates to the fact, that the quality of the routing is highly dependent on the order in which the nets are routed. This is caused by the sequential consideration of the nets, which lacks a global view of the routing problem. Thus, early routed nets will probably block resources which would have been needed by nets routed later on. To meet this problem, the algorithms described in [AlRo95], [ACGR95] try to find the optimal sequence by routing the nets one by one until a net cannot be routed. The routing is then started over with the failing net put to the first position to be routed. Another approach consists in ripping up and rerouting nets. A popular algorithm employing this strategy is the Pathfinder router published in 1995 by Ebeling et al. [EMHB95]. This algorithm is also used in several programming environments for coarse-grain reconfigurable architectures, thus it is described in more detail in this chapter. In the following subsections, the concept of the routing resource graph, the grid-based Lee routing algorithm, Dijkstra's algorithm and the Pathfinder algorithm are sketched briefly.

3.5.1 The Routing Resource Graph

The problem of nonuniform routing resources described above is solved by employing a graph representation of the routing architecture known as the routing resource graph [EMHB95], [NaRu95]. This directed graph consists of nodes representing the wires and cell ports, and edges representing possible connections between them. An example in figure 3-11 illustrates this representation. Two cells, *A* and *B*, are connected by wiring as shown. The according routing resource graph comprises a node for each cell as source and sink, one for each cell port, and one for each wire. A directed edge between two nodes describes the possibility to connect the source resource to the target resource, indicating the direction of data flow. Note, that for bidirectional connection possibilities two edges are required. For example, *XwireA* and *YwireA* may be connected to transfer data from *XwireA* to *YwireA* or vice versa. An example path (shaded in the figure) connects cell *A* to cell *B*, leading over the port *Aout*, the wires *XwireA* and *YwireB*, and the port *Bin2*.To model channels with several wires, nodes representing routing resources can feature a capacity value. In order to find the shortest path in the graph representation, Dijkstra‘s Algorithm [Dijk59] can be used, with the necessity to split up multi-terminal nets into several two-terminal nets. Other approaches directly route multi-terminal nets by solving a corresponding Graph Steiner Minimal Tree (GSMT) or Graph Steiner Arborescence (GSA) Problem [AlRo95], [ACGR95].

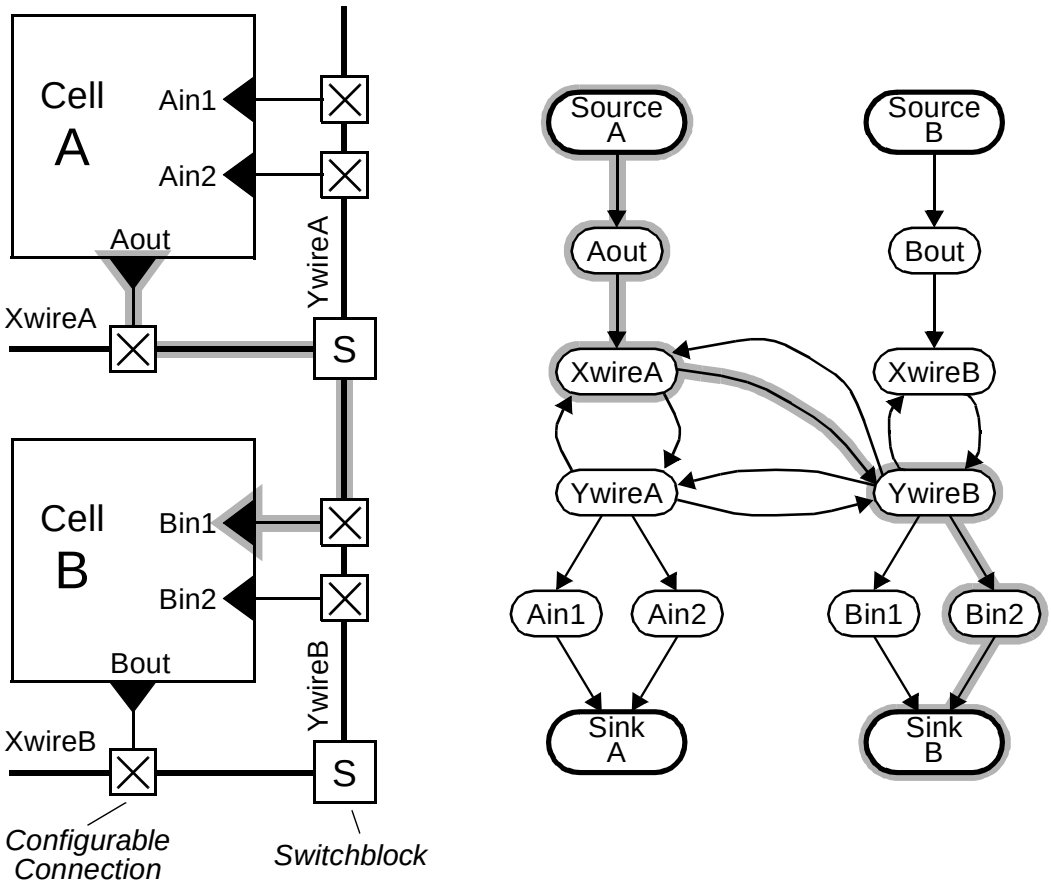

Figure 3-11: Example for a routing resource graph

3.5.2 The Lee Routing Algorithm

The algorithm by Lee [Lee61] assumes the routing area to be a grid, to which all start and end points, as well as obstacles and generated routing paths, are aligned to. The algorithm finds the shortest path between two grid points, if it exists, avoiding obstacles. This path consists of a sequence of adjacent grid points. The connection of two points is not appropriate for multi-terminal nets, for which an extension is needed. In this brief sketch, only the basic algorithm is shown. The algorithm starts with a fill phase, where the unblocked grid positions *i* manhattan distance steps apart from the source point *S* are labeled with the label *i*. Starting from *S*, all adjacent empty locations of *S* are labeled '1'. In each further *i*th iteration, all empty neighbors to the locations labeled *i-1* are labeled *i*, forming a diamond shaped wavefront of increasing labels starting from *S*. The wavefront expansion stops, if the target location *T* is reached, or if there are no empty neighbors left to locations labeled *i-1*, or if *i* reached an upper limit of the path length. An example labeling is shown in figur e3-12a.

a)

4	3	4	5	6	7	8			
3	2	3	4	5	6	7	8		
2	1	2	3	4	5	6	7	8	
1	**S**	1	Obstacle				8		
2	1	2		8					
3	2	3		7	8	**T**			
4	3	4	5	6	7	8			
5	4	5	6	7	8				
6	5	6	7	8					
7	6	7	8						

b)

4	3	4	5	6	7	8			
3	2	3	4	5	6	7	8		
2	1	2	3	4	5	6	7	8	
1	**S**	1	Obstacle				8		
2	1	2		8					
3	2	3		7	8	**T**			
4	3	4	5	6	7	8			
5	4	5	6	7	8				
6	5	6	7	8					
7	6	7	8						

Figure 3-12: Lee routing algorithm:
a) labeling phase
b) retrace phase

After the labeling phase, the shortest path is identified by retracing the labels from the target *T* back to the source *S* after decreasing labels. There are typically several paths with the same total distance possible. For routing purposes, normally the path with the least bendings is chosen. An example is shown in figure 3-12b.

The algorithm can easily be extended so preferences about grid locations can be defined by giving the locations different weights. In the labeling phase, this weight can be added to the label, effectively influencing the path which will be found in the retrace phase.

3.5.3 The Algorithm by Dijkstra

In [Dijk59], Dijkstra published an algorithm to find the shortest path between two given vertices in a graph. Before the algorithm is sketched, the terms used in the following discussion as well as the problem will be defined more formally.

Let $G(V,E)$ be a graph with a set of vertices $V = \{v_1, ..., v_n\}$ and a set of Edges $E = \{e_1, ..., e_m\}$. Let $Inc(e)$ be the incidence function defined as $Inc(e) = \{a, b\}$, $e \in E$, $a,b \in V$, where a and b are the vertices incident to edge e. Let $len(e)$ denote a weight (the 'length') assigned to each edge with $len(e)$ Š 0 for all $e \in E$. This weight is also referred to as the length of the edge. Let s and t be two vertices, $s,t \in V$. The problem consists in finding a path P from s to t, $P = \{p_1, ..., p_m\}$, $p_i \in E$ for $i = 1,..., m$, such that the sum $len(p_1) + ... + len(p_m)$ is minimal. Based on this definitions, the algorithm by Dijkstra can be written as shown in figure 3-13.

```
Algorithm Shortest Path
{     T = V ;
      /* Set the markers for vertices */
      FOREACH x IN T
      {     Marker[ x ] = ∞ ;
      }
      Marker[ s ] = 0 ;
      DO
      {     /* Find vertex u with minimal marker */
            MinMarker = ∞ ;
            FOREACH w IN T
            {     IF ( Marker[ w ] < MinMarker )
                  {     u = w ;
                        MinMarker = Marker[ u ] ,
                  }
            }
            FOREACH e IN E WITH u ∈ Inc( e )
            {     /* Find the vertex v to which e leads */
                  v = Inc( e ) - { u } ;
                  IF ( v ∈ T AND Marker[ v ] > Marker[ u ] + len( e ) )
                  {     Marker[ v ] = Marker[ u ] + len( e ) ;
                  }
            }
            T = T - { u } ;
      }
      WHILE ( u ≠ t AND MinMarker ≠ ∞ ) ;
}     /* End of algorithm */
```

Figure 3-13: Shortest path algorithm by Dijkstra

Dijkstra's algorithm stores for each vertex a marker consisting of the accumulated length of the shortest sub-path from the source to this vertex. When the target vertex has been reached, the path is given by the vertices with the lowest weights. In this regard, Dijkstra's algorithm is analogous to the Lee algorithm, but resembles a generalization due to the possibility to express arbitrary connections between the cells using graphs. A problem for the Lee algorithm can be transformed into one suitable for Dijkstra's algorithm by assuming the grid to be a graph with each grid position resembling a node, and each node being connected to its four nearest neighbors.

3.5.4 The Pathfinder Negotiated Congestion Algorithm

As mentioned above, the sequential application of the Lee or Dijkstra algorithm to route a list of nets may result in a bad solution. The Pathfinder algorithm [EMHB95] tries to improve the routing iteratively by ripping up and rerouting all nets. It is based on an earlier work by Nair [Nair87] and is itself the base for several derivatives, e.g. [BeRo97], [SBR98]. Pathfinder can be used to find a routing meeting a given timing constraint. However, in this section a more simple variant only based on congestion avoidance is presented.

The algorithm iterates, until no routing resource is used more than once. In each iteration, every net is ripped up and rerouted according to a cost function, which is to be kept minimal. During the routing, multiple use of a routing resource is basically allowed. However, resources which are overused are affected with an extra cost, which is increased from iteration to iteration. This way, nets are encouraged to avoid the overused resources and use others instead, except it turns out that a detour would be too expensive. As the cost is increased gradually, the effect achieved is a negotiation of the routing resources between the nets. If there is no overuse of resources left, the routing is finished.

The exact cost to use a routing resource n (which is a node in the routing resource graph) is given as: $Cost_n = (b_n + h_n) * p_n$, where b_n is a base cost of the resource, h_n is a cost related to the history of congestion of the resource during previous iterations, and p_n is a cost related to the current number of other nets using the resource. A pseudo-code description of the Pathfinder negotiated congestion algorithm working on a graph representation is shown in figure 3-14. The algorithm in the inner loop to route the ith net determines a minimum spanning tree, with *Cost* and the derived *PathCost* as the objective functions.

3.6 Configuration Code Generation

After the placement and routing, all information determining a configuration is available. The remaining step consists in the generation of a binary file which is appropriate for downloading to the architecture. The tool for the generation of this file can be compared to an assembler for microprocessors, featuring no special algorithms. The format of the final configuration file is typically proprietary and unique for each architecture.

```
Algorithm Pathfinder
{     /* Let RoutingTree[ i ] be a set of all nodes in the current routing for net i */
      WHILE ( OverusedResourcesExist == TRUE )
      {     /* Loop over all nets */
            FOR i=1 TO NumberOfNets
            {     RipUp( RoutingTree[ i ] );
                  /* Now reroute net i */
                  RoutingTree[ i ] = SourceNode( i ) ;
                  /* Traverse all target nodes (sinks) */
                  FOR j=1 TO NumberSinks( i ) ;
                  {     /* Find path from SourceNode(i) to Sink(i,j) */
                        PriorityQueue = RoutingTree[ i ] at cost 0 ;
                        WHILE ( NotReached( Sink( i, j ) ) == TRUE )
                        {     M = FindLowestCostNode( PriorityQueue ) ;
                              DeleteNode( PriorityQueue, M ) ;
                              /* Traverse all fanout nodes of M */
                              FOR k=1 TO Fanout( M )
                              {     /* Determine kth fanout node */
                                    N = FanoutNode( M, k ) ;
                                    PathCost[ N ]= Cost[ N ] + PathCost[ M ] ;
                                    AddNode( PriorityQueue, N )
                                          at cost PathCost[ N ];
                              }
                        }     /* Now jth sink in net i has been reached */
                        /* Backtrace the path from SourceNode(i) to Sink(i,j) */
                        FOR k=1 TO PathNodes( Source( i ), Sink( i, j ) )
                        {     /* Find kth Node on the Path */
                              N = GetPathNode( Source( i ), Sink( i, j) , k ) ;
                              Cost[ N ] = UpdateCost( N ) ;
                              AddNode( RoutingTree[ i ], N ) ;
                        }     /* Now path Source(i)->Sink(i,j) is routed */
                  }     /* Now net i is routed */
            }     /* Now all nets are routed */
      }
}     /* End of algorithm */
```

Figure 3-14: Pathfinder negotiated congestion routing algorithm

3.7 Simulation

Simulation is used to validate the design. In the FPGA design flow, there are typically two steps of simulation, functional and timing simulation. The functional simulation verifies, that the specification meets the desired function of the circuit. It is typically done after the design entry by a simulation software. At this time, the exact timing behavior of the circuit is not yet known, as this depends on the wire lengths, which are determined by the subsequent placement and routing steps. Thus, a timing simulation is typically done after placement and routing, using additional worst case delay times gained through back annotation from the final mapping information. This simulation models the behavior of the circuit more accurately than the functional simulation and can be used to verify timing constraints. However, some placement and routing algorithms allow to specify such constraints, thus incorporating them in the design flow [EMHB95].

In programming environments for coarse grain architectures, simulation is not that common as for FPGAs. Many existing works rely on HDL models of the architecture, which are used for simulation, employing standard high level design tools. Yet, some environments feature dedicated simulator tools.