Chapter 3
Homogeneous Architectures Exploration Environments

The advancement in process technology has greatly enhanced the capacity of FPGAs and they have become increasingly popular for the implementation of larger designs. Design of large devices implies fundamental and efficient innovation in FPGA architecture to improve density, speed and power optimization of the architecture. It has been observed that most of the designs exhibit a locality in their connections and specific architectures are required to be designed to exploit their locality and improve the architecture efficiency. In this chapter we present an exploration environment for a new tree-based homogeneous FPGA architecture [132] that exploits the locality that most of the designs exhibit. Routability and interconnect area of this architecture depends on switch boxes topology and signals bandwidth (in/out signals per cluster). In tree-based FPGA we use full crossbar switch boxes and we aim at exploiting the available flexibility to reduce signals bandwidth based on suitable partitioning approaches.

It is well established that the quality of an FPGA based implementation is largely dependant on the accompanying flow that is used to map different applications on the FPGA architecture. Benefits of an otherwise well designed, feature rich FPGA architecture might be impaired if the CAD tools can not take advantage of the features that FPGA provides. In this book we have designed an environment for tree-based architecture that exploits the features of this architecture. The environment is based on a mixture of generalized and specifically developed tools. These tools are used to map different application on the tree-based architecture. A reference mesh-based architecture and its associated exploration environment is also presented in this chapter and the two architectures are compared using the results that are obtained through the exploration environments of both architectures.

U. Farooq et al., *Tree-Based Heterogeneous FPGA Architectures*,
DOI: 10.1007/978-1-4614-3594-5_3,

3.1 Reference FPGA Architectures

This section gives basic overview of the two FPGA architectures that are used in this book. A brief overview of generalized mesh and tree-based architectures is already presented in Chap. 2. Here, we present further details and also present the customized software flow that we have developed for both architectures. Although two architectures are comprised of similar logic and routing resources (i.e. configurable logic blocks, multiplexors, configuration memory etc), it is the arrangement of these resources that differentiates the two architectures. In a tree-based architecture logic and routing resources are arranged in hierarchical manner while in mesh-based architecture resources are arranged in an island-style.

3.1.1 Mesh-Based FPGA Architecture

A mesh-based FPGA is represented as a grid of equally sized slots which is termed as slot-grid. The reference mesh-based FPGA is a VPR-style (Versatile Place & Route) [22] architecture, as shown in Fig. 3.1. It contains Configurable Logic Blocks (CLBs) arranged on a two dimensional grid. Each CLB contains one Look-Up Table with c_{in} inputs and $c_{out} = 1$ output and one Flip-Flop (FF). In mesh-based FPGA, input and output pads are arranged at the periphery of the slot-grid as shown in Fig. 3.1 and a CLB is surrounded by a single-driver unidirectional routing network [77]. The routing network is arranged in the form of uniform horizontal and vertical routing channels where each channel contains a fixed number of routing tracks which is termed as the channel width of the architecture.

A mesh-based FPGA is divided into "tiles" that are repeated horizontally and vertically to form a complete FPGA. A CLB, along with its surrounding routing network, forms the tile of the architecture which is repeated horizontally and vertically to form the complete FPGA architecture. A single FPGA tile, surrounded by its routing network is shown in Fig. 3.2. In this figure CLB contains a LUT which has 4 inputs and 1 output. Each of the 4 inputs of a CLB are connected to 4 adjacent routing channels. The output pin of a CLB connects with the routing channel on its top and right through the diagonal connections of the switch box (highlighted in the bottom-left switch box shown in Fig. 3.2). The switch box uses unidirectional, disjoint topology to connect different routing tracks together. The connectivity of a routing track incident on a switch block with routing tracks of other routing channels that are incident on the same switch block, termed as switch block flexibility (Fs), is set to be 3. The connectivity of routing channel with the input and output pins of a CLB, abbreviated as Fc(in) and Fc(out), is set to be maximum at 1.0. The channel width is varied according to the netlist requirement but remains in multiples of 2 [77].

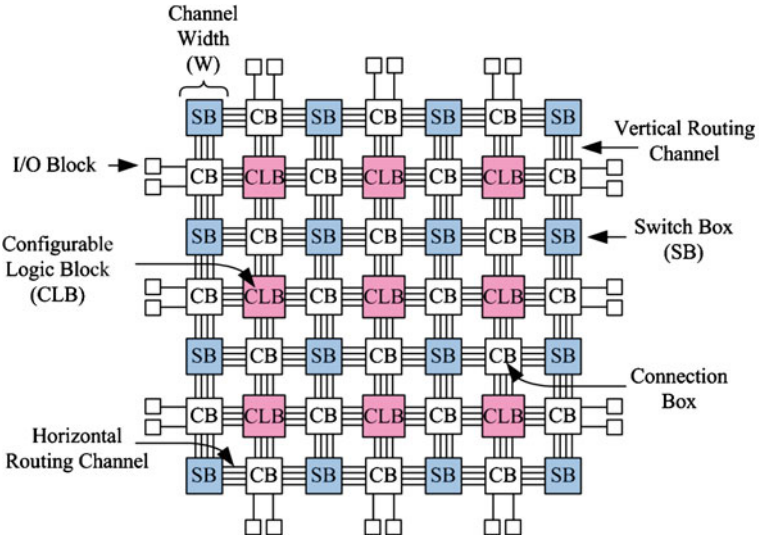

Fig. 3.1 Generalized mesh-based FPGA architecture

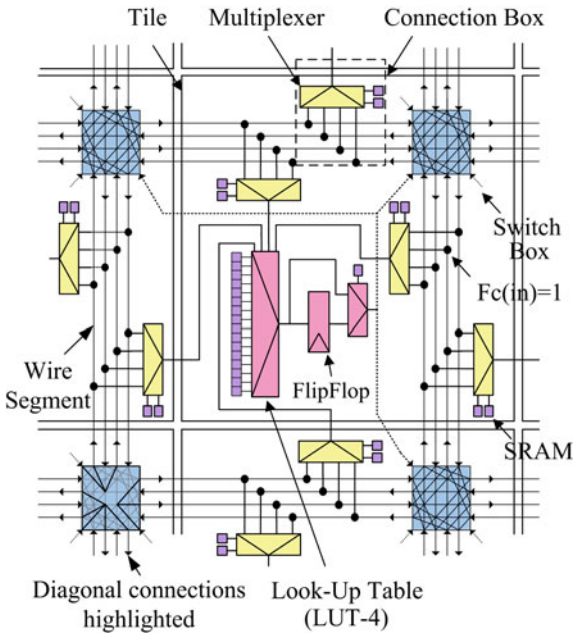

Fig. 3.2 Detailed interconnect of a CLB with its surrounding routing network

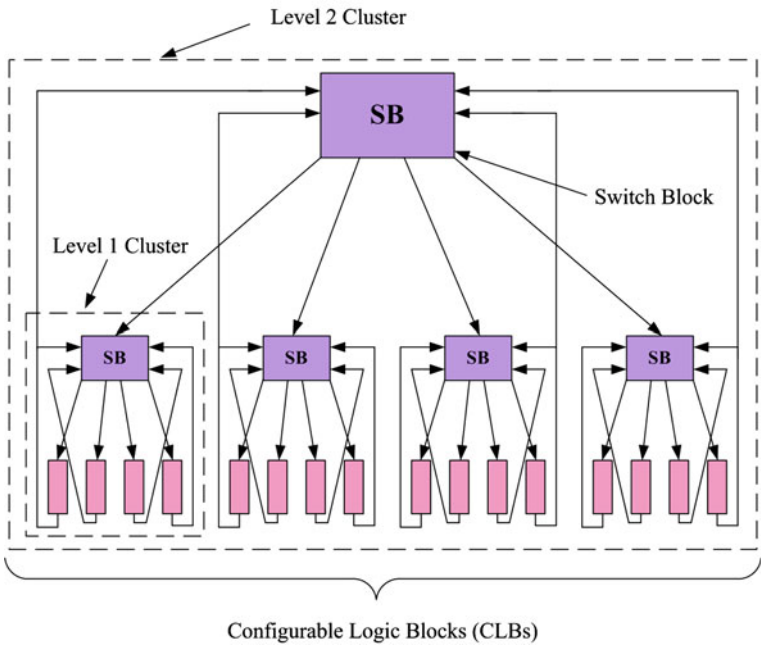

Fig. 3.3 Generalized tree-based FPGA architecture

3.1.2 Tree-Based FPGA Architecture

In a tree-based architecture logic and routing resources are partitioned into a multi-level clustered structure where each cluster contains sub-clusters and signals coming in and out of clusters communicate with other clusters using switch blocks. Generalized example of a two level, arity 4 tree-based architecture is shown in Fig. 3.3 where level 2 cluster contains 4 level 1 sub-clusters and each level 1 sub-cluster contains 4 CLBs. Since each cluster contains 4 sub-clusters in this architecture, it is termed here as arity 4 architecture. Tree-based architecture uses single driver unidirectional wires as bidirectional wires introduce considerable routing and area overhead [77]. Just like [3, 4], for tree-based architecture, a fully hierarchical interconnect is built where inter level signal bandwidth grows according to Rent's rule [25]. As it can be seen from Fig. 3.3 that contrary to [4], our tree-based architecture is divided into two unidirectional routing networks: the downward network and upward network. The downward network is inspired from SPIN [97] and it uses butterfly fat tree topology [33] to connect different signals using switch boxes and unidirectional wires. The upward network uses hierarchy and connects different signals using linearly populated switch boxes and unidirectional wires.

3.1.2.1 Architecture Interconnect Details

In a tree-based architecture, each configurable logic block (CLB) contains one Look-Up-Table (LUT) with c_{in} inputs and $c_{out} = 1$ output, followed by a bypass Flip-Flop. CLBs are grouped into k sized clusters and interconnect is organized into levels. For example in Fig. 3.3, the cluster size is 4, architecture has two levels and it supports 16 CLBs. Let $nb\ell$ denote the number of levels of a given Tree containing N leaves ($nb\ell = log_k(N)$). In each level ℓ we have $\frac{N}{k^\ell}$ clusters; C is the set of clusters in all levels. A cluster with index c belonging to level ℓ is noted by $cluster(\ell, c)$. A cluster switch block is divided into separate Downward Mini Switch Boxes (DMSBs) and Upward Mini Switch Boxes (UMSBs). DMSBs are responsible for downward interconnect and UMSBs are responsible for upward interconnect. DMSBs and UMSBs are combined together to route different signals of the netlists that are mapped on the architecture. These DMSBs and UMSBs are unidirectional full cross bar switches that connect signals coming into the cluster to its sub-clusters and signals going out of a cluster to the other clusters of hierarchy. Each $cluster(\ell, c)$ where $\ell \geq 1$ contains a set of inputs $N_{in}(\ell)$, a set of outputs $N_{out}(\ell)$, a set of downward and upward switch boxes and k sub-clusters. The inputs and outputs of cluster $cluster(\ell, c)$ are divided equally among its DMSBs and UMSBs which are used to connect these inputs and outputs to its sub-clusters and to other clusters of hierarchy. Sub-clusters of $cluster(\ell, c)$ are $cluster(\ell - 1, k.c + i)$ where $i \in \{0, 1, 2, \ldots, k - 1\}$. k is called $cluster(\ell, c)$ arity.

Each cluster in level 0 is denoted $cluster(0, c)$ or $leafcluster(c)$ and corresponds to the Configurable Logic Block (CLB) and contains c_{in} inputs, 1 output, no MSBs and no sub-cluster. Each $cluster(\ell, c)$ where $\ell < nb\ell - 1$ has an owner in level ℓ', where $\ell' > \ell$, denoted $cluster(\ell', c \div k^{(\ell' - \ell)})$. We define for each $cluster(\ell, c)$ a position inside its owner in level $\ell + 1$ (direct owner) by the following function:

$$\begin{aligned} pos: \quad C &\longrightarrow \{0, 1, 2, \ldots, k - 1\} \\ cluster(\ell, c) &\longmapsto c \ \ mod \ \ k \end{aligned}$$

Two clusters belonging to level ℓ and with the same owner at level $\ell + 1$ have two different positions. To get the cluster owner in level ℓ' of $cluster(\ell, c)$ ($\ell < \ell' \leq nb\ell - 1$) we define the function:

$$\begin{aligned} owner: \quad C \times \mathbb{N} &\longrightarrow C \\ (cluster(\ell, c), \ell') &\longmapsto cluster(\ell', c \div k^{\ell' - \ell}) \end{aligned}$$

3.1.2.2 Downward Network

Figure 3.4 shows a sparse downward network based on unidirectional DMSBs. The downward interconnect topology is similar to the butterfly fat tree. Each DMSB of a $cluster(\ell, c)$ where $\ell > 1$ is connected to each sub-cluster through one and only

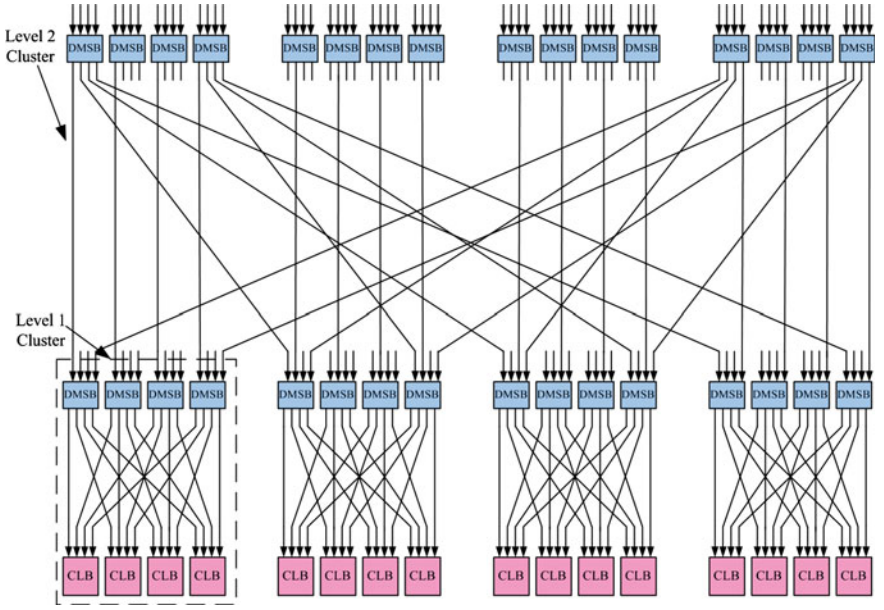

Fig. 3.4 Detailed downward interconnect of a tree-based architecture

one input pin. Thus, the DMSBs number in a cluster situated in level ℓ is equal to the input number of a cluster situated in level $\ell - 1$: $nbDMSB(\ell) = N_{in}(l - 1)$. For example in Fig. 3.4, the number of DMSBs at level 2 are equal to the number of inputs of a sub-cluster at level 1 (i.e. 16).

We name $DMSB(\ell, c, m)$ as the successor of $DMSB(\ell', c', m')$ where $0 < \ell < \ell'$ if there is a downward directed path from $DMSB(\ell', c', m')$ to $DMSB(\ell, c, m)$. The path between a DMSB and its successor is unique. Thus each $DMSB(\ell', c', m')$ has a successor in each sub-cluster belonging to level ℓ $DMSB(\ell, c, m)$ where $0 < \ell < \ell'$.

3.1.2.3 Upward Network

Figure 3.5 shows the upward network of a tree-based architecture that uses UMSBs that allow LBs outputs to reach a larger number of Downward MSBs (DMSBs). The UMSBs are organized in a way that allows CLBs belonging to the same "owner cluster" (at level 1 or above) to reach exactly the same set of DMSBs at each level. The interconnect offers more routing paths to connect a net source to a given sink. In this case we are more likely to achieve highly congested netlists routing. This gives an efficient solution for mapping netlists since instances may have different fanout sizes. For example in Fig. 3.5, a CLB output can reach all 4 DMSBs of its owner cluster at level 1 and all the 16 DMSBs of its owner cluster at level 2.

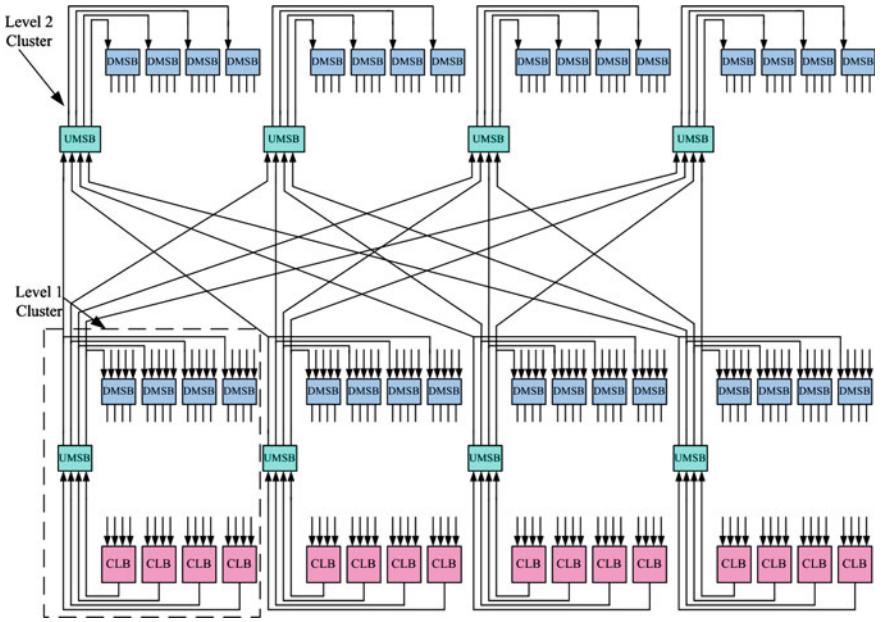

Fig. 3.5 Detailed upward interconnect of a tree-based architecture

3.1.2.4 I/O Connections

Figure 3.6 shows the combined downward and upward interconnect of tree-based architecture. Also it can be seen from this figure that output and input pads are grouped into specific clusters. The cluster size and the level where it is located can be modified to obtain the best design fit. Each input pad is connected to all UMSBs of the upper level. In this way each input pad can reach all CLBs of the architecture with different paths.

Similarly, output pads are connected to all DMSBs of the upper level; in this way they can be reached from all CLBs through different paths. The flexibility of I/O pads is kept higher than those of CLBs to ensure the routing of highly congested netlists.

3.1.2.5 Interconnect Depopulation

Although the use of DMSBs and UMSBs gives the architecture a great amount of flexibility in terms of the number of paths that can be used to route different signals on the architecture, it increases the number of switches in the architecture which can increase the area of the architecture. This can be compensated by reduction of in/out signals bandwidth of clusters at every level. In fact Rent's rule [25] is easily adapted to tree-based structure:

$$IO = c.k^{\ell.p} \tag{3.1}$$

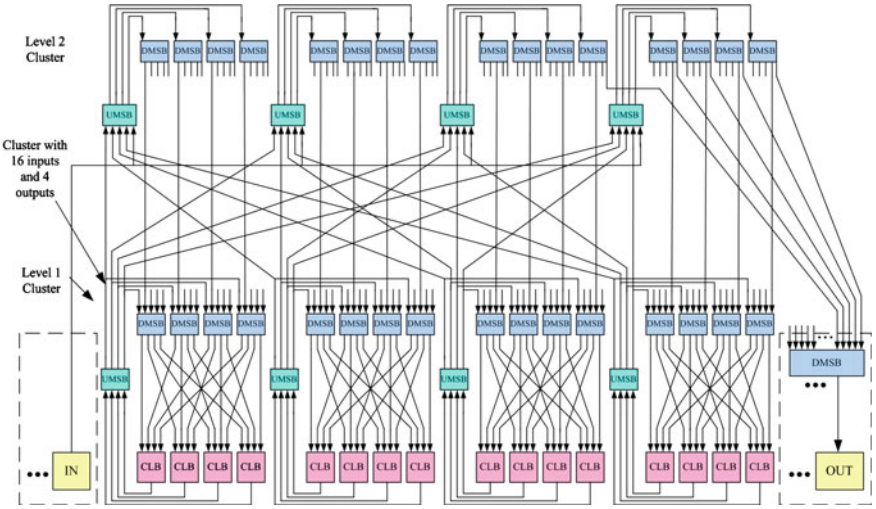

Fig. 3.6 Tree-based architecture with detailed *downward* and *upward* networks

where ℓ is a Tree level, k is the cluster arity, c is the number of in/out pins of a logic block and IO the number of in/out pins of a cluster situated at level ℓ. Intuitively, p represents the locality in interconnect requirements. If most connections are purely local and only few of them come in from the exterior of a local region, p will be small. In Tree-based architecture, both the upward and downward interconnects populations depend on this parameter. We can depopulate the routing interconnect by reducing the value of p which in turn reduces the signal bandwidth of the architecture. By doing so the architecture routability is reduced too. An example of a depopulated tree-based interconnect is shown in Fig. 3.7. Compared to the example shown in Fig. 3.6, number of inputs of each cluster at level 1 are reduced from 16 to 10 and the number of outputs are reduced from 4 to 3. By reducing the inputs and outputs the number of switches are reduced by 21% and value of p is reduced from 1 to 0.79. Although, this reduction improves the area of the architecture, it reduces its flexibility too. Thus we have to find the best tradeoff between interconnect population and logic blocks occupancy. Dehon showed in [4] that the best way to improve circuit density is to balance logic blocks and interconnect utilization. In tree-based architecture, the logic occupancy factor is controlled by N, the leaves (CLBs) number in the Tree. N is directly related to the number of levels and the clusters arity k. In most cases N is larger than the number of netlists instances. This means that in these cases we have a low logic utilization. This is not really penalizing since it can be compensated by a depopulated interconnect. In other words, the area overhead due to unused CLBs is compensated by congestion spreading and interconnect reduction.

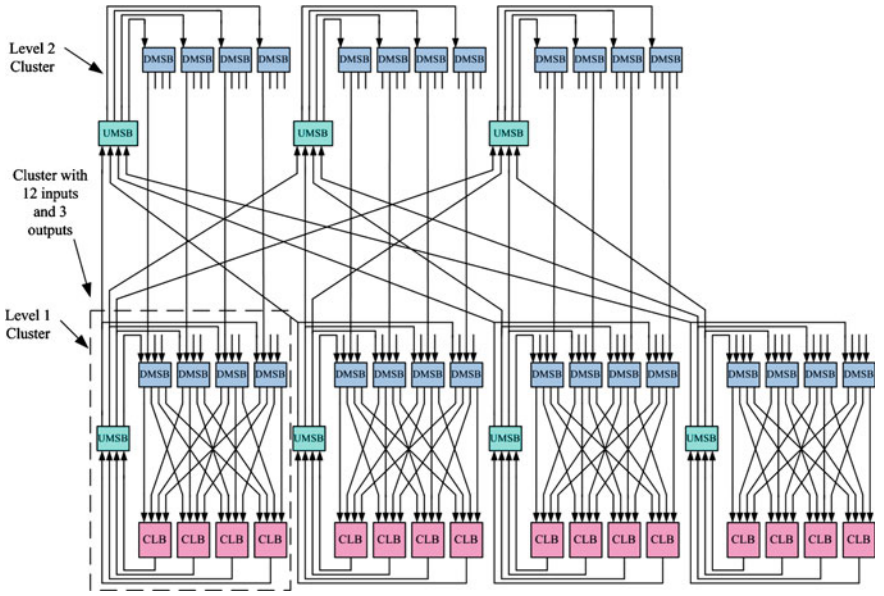

Fig. 3.7 Depopulation of tree-based architecture using Rent's rule

3.1.2.6 Rent's Rule Based Model

Based on Rent's rule presented in Eq. (3.1), we evaluate the Tree architecture switches requirement to connect LBs.

Switches requirement

We model upward and downward networks separately:

Downward network:

We note:

- $N_{in}(\ell)$ the number of inputs of a cluster located at level ℓ.
- $N_{out}(\ell)$ the number of outputs of a cluster located at level ℓ.
- c_{out} the number of outputs of an LB.
- c_{in} the number of inputs of an LB.
- k clusters arity (size).

Clusters located at level ℓ contain $N_{in}(\ell - 1)$ DMSB with k outputs and $\frac{N_{in}(\ell)+kN_{out}(\ell-1)}{N_{in}(\ell-1)}$ inputs. As we assume that the DMSB are full crossbar devices, we get $k(N_{in}(\ell) + kN_{out}(\ell - 1))$ switches in the switch box of a level ℓ cluster. Since we have $\frac{N}{k^{\ell}}$ clusters in level ℓ, we get a total number of switches, related to the downward network, given by:

$$\sum_{\ell=1}^{\log_k(N)} k \times N \times \frac{N_{in}(\ell) + kN_{out}(\ell-1)}{k^{\ell}}$$

$N_{out}(0) = c_{out}$ is the number of outputs of a Basic Logic Block. Following Eq. (3.1), we get $N_{in}(\ell) = c_{in}.k^{\ell.p}$ and $N_{out}(\ell-1) = c_{out}.k^{(\ell-1)p}$. The total number of switches used in the downward network is:

$$N_{switch}(down) = N \times (k^p c_{in} + kc_{out}) \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

Upward network:

Clusters located at level ℓ contain $N_{out}(\ell-1)$ UMSB with k inputs and k outputs. As we assume that UMSB are full crossbar devices, we get $k^2 \times N_{out}(\ell-1)$ switches in the switch box of a level ℓ cluster. As we have $\frac{N}{k^{\ell}}$ clusters at level ℓ we get the total number of switches, related to the upward network:

$$\sum_{\ell=1}^{\log_k(N)} \frac{k^2 \times N}{k^{\ell}} \times N_{out}(\ell-1)$$

$N_{out}(0) = c_{out}$ is the number of outputs of a Basic Logic Block. Following (3.1), we get $N_{out}(\ell-1) = c_{out}.k^{(\ell-1)p}$.
The total number of switches used in the upward interconnect is:

$$N_{switch}(up) = N \times k \times c_{out} \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

The total number of Tree-based interconnect switches is

$$N_{switch}(Tree) = N_{switch}(down) + N_{switch}(up)$$

$$N_{switch}(Tree) = N \times (k^p c_{in} + 2kc_{out}) \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

The number of switches per Logic Block is:

$$N_{switch}(LB) = (k^p c_{in} + 2kc_{out}) \times \sum_{\ell=1}^{\log_k(N)} k^{(p-1)(\ell-1)}$$

$$N_{switch}(LB) = \begin{cases} (k^p c_{in} + 2kc_{out}) \times \frac{1-N^{p-1}}{1-k^{p-1}} & \text{if } p \neq 1 \\ (k^p c_{in} + 2kc_{out}) \times \log_k(N) & \text{if } p = 1 \end{cases}$$

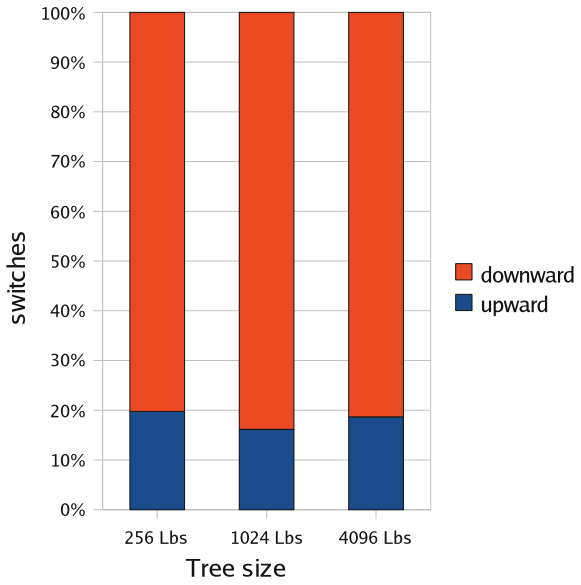

Fig. 3.8 Interconnect switches distribution

$$N_{switch}(LB) = \begin{cases} O(1) & \text{if } p < 1 \\ O\left(\log_k(N)\right) & \text{if } p = 1 \end{cases} \tag{3.2}$$

The cost of adding the upward can be compensated by reducing the architecture Rent's parameter. In addition we notice that the number of the upward network switches is smaller than the switches number in the downward network:

$$\frac{N_{switch}(down)}{N_{switch}(up)} = \frac{k^p c_{in} + k c_{out}}{k \times c_{out}}$$

With $p = 1, k = 4, c_{in} = 4$ and $c_{out} = 1$ this ratio is equal to 5. In Fig. 3.8, we show the distribution of interconnect resources between the upward and the downward networks for different Tree sizes (we include in/out pads connections).

Wiring Requirements

At each level ℓ of the hierarchy, every switching node has $n_{in}(\ell)$ inputs and $n_{out}(\ell)$ outputs. This makes the bisection width equal to $(c_{in} + c_{out})k^{\ell.p}$. Since $\forall \ell \in \{1, \ldots, \log_k(N)\}$ $k^{\ell.p} \leq N$, the bisection width is $O(N^p)$. For a 2-dimensional network layout this bisection width must cross the perimeter out of the subarray. Thus the perimeter of each subarray is $O(N^p)$. The areas of the subarray will be proportional to the square of its perimeter, making: $A_{subarray} \propto N^{2p}$. The required area per logic block (LB) based on wiring constraints, is therefore evaluated by:

$$A_{LB} \propto N^{2p-1}$$

In tree-based architecture, we can control bisection bandwidth in each level based on Rent's parameter ($p < 1$). Consequently, physical layout generation may be much optimized since wiring is no more dominant.

3.1.3 Comparison with Mesh Model

Concerning switches per logic block growth, it was established in [4] that in the Mesh architecture:

$$N_{switch}(LB) = O(N^{p-0.5}) \tag{3.3}$$

Equations (3.2) and (3.3) show that in the tree-based architecture, switches requirement grows more slowly than in common mesh-based architecture. These results are encouraging for constructing very large structures, especially when p is less than 1. But this does not mean that our Tree-based topology is more efficient than mesh-based architecture, since they do not have the same routability. The best way to check this point is through experimental work. Based on benchmark circuits implementation, we compare the resulting areas in the case of tree-based and mesh-based FPGA.

3.2 Architectures Exploration Environments

Since we are exploring two different architecture topologies, an effort is required to ensure transparency for comparison between two architectures. For this purpose, we have designed separate exploration environments for the two architectures. Some parts of two exploration environments are shared while rest of them are designed specifically to meet the needs of two architectures. Exploration of each architecture starts with its definition which is done through an appropriate architecture description mechanism. Once the architecture is defined, netlists are separately placed and routed on the two architectures. Although separate placement and routing tools are developed for two architectures, these tools are based on generic algorithms. These algorithms are adapted appropriately to the needs of the two architectures. Once the placement and routing of the netlists is performed, the area of the architectures is calculated which eventually leads to the termination of architecture exploration. Details of different steps that are involved in architecture exploration are explained in the following sections.

3.3 Architecture Description

3.3.1 Architecture Description of Tree-Based Architecture

In exploration environment of tree-based architecture, an architecture description mechanism is used to define different parameters of a tree-based architecture. The architecture description starts with the number of levels of the architecture. Then the level of I/O pads clusters and the number of I/Os per cluster are defined. Later the parameters of clusters located at all levels of the architecture are defined. These parameters include the arity and signal bandwidth of clusters that are located at that particular level. After that, architecture optimization is either set to be true or false. In case this parameter is set to be true, a binary search algorithm is applied to search the best signal bandwidth for the clusters that are located at different levels of hierarchy. Otherwise a fixed routing interconnect based on the initially defined cluster bandwidth values is built and no optimization is performed. Finally the parameters of CLBs are defined that include their level, area, inputs, outputs and details of their pins.

3.3.2 Architecture Description of Mesh-Based Architecture

The exploration environment of mesh-based architecture used in this book is based on the environment presented in [82]. In this environment, architecture description of mesh-based architecture starts with the definition of height and width of the slot-grid. Then Channel_Type of the architecture is defined which can be either a unidirectional mesh [77] or a bidirectional mesh [120] routing network. The channel width of the routing network is then either set to a constant value, or a binary search algorithm searches a minimum possible channel width. In case of unidirectional mesh, the channel width remains in multiples of 2. Finally the parameters of CLBs are defined which include a name, a size (number of slots occupied), area, number of inputs/outputs and the detail of their pins.

3.4 Software Flow

Once the architecture is defined, different netlists are placed and routed on the architecture using a software flow. However, before being placed and routed on an architecture, a netlist is required to pass through certain number of processes so that it might be converted from hardware description to a format that can be placed and routed on the FPGA architecture (.net format). Complete software flow illustrating these processes along with the placement and routing modules is shown in Fig. 3.9. As it can be seen from the figure that a certain part of software flow is shared by

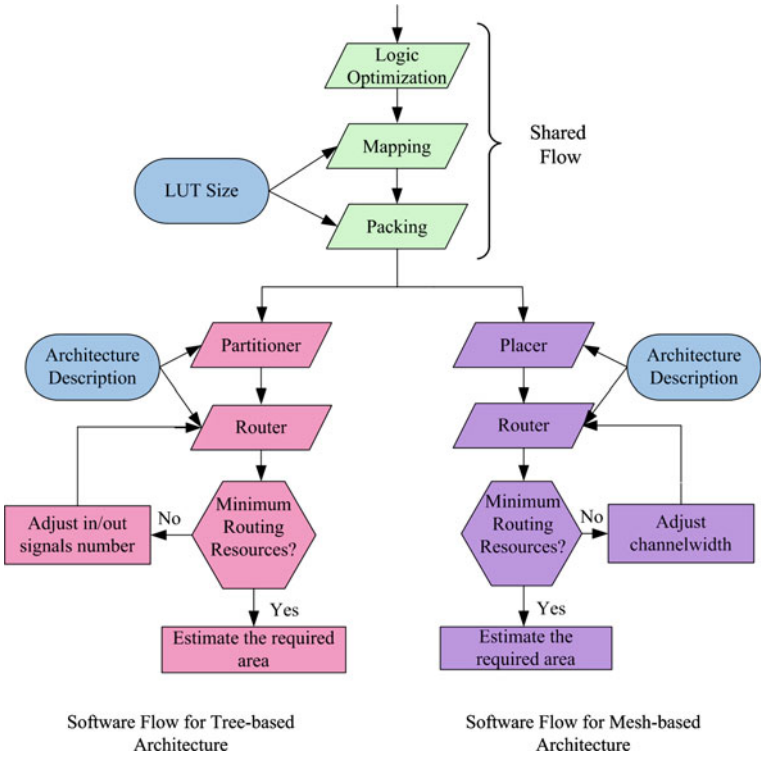

Fig. 3.9 Software flow

both mesh-based and tree-based architectures. This part involves the transformation of netlist from hardware description to .net format (i.e. synthesis of netlist) and once the netlist is converted into .net format, the two architectures place and route the netlist separately using their appropriate tools. A brief description of different tools involved in the software flow is given as follows:

3.4.1 Logic Optimization, Mapping and Packing

The input to the software flow is the hardware description of the netlist. First of all the netlist is synthesized/logically optimized using a tool called SIS [102]. This is a process in which circuit description is converted into gate level presentation. SIS is an open source tool and it can be replaced by any commercial synthesis tool.

After logic optimization, mapping of the netlist is performed using SIS. Mapping is a process that converts gate level representation into K-input LUT and flip-flops. This process takes LUT size as its input and converts logic expressions into given

LUT size netlist. We have used "Flow Map" [64] mapper for our experimentation which is included in SIS package. This mapper uses timing and area as its objectives and netlists produced using this mapper produce good results in terms of area and delay.

After mapping, packing is performed using T-VPACK [14] that packs registers together with K-input LUTs and converts the netlist into .net format. A netlist in .net format contains CLBs and I/O instances that are connected together using nets. The size of a CLB is defined as the number of LUTs contained in it and in this book this size is set to be 1 for both mesh-based and tree-based architectures. Once netlist is obtained in .net format, it is placed and routed separately on tree-based and mesh-based FPGA.

3.4.2 Software Flow for Tree-Based Architecture

3.4.2.1 Partitioning

In recent FPGA architectures, interconnect is organized in multiple hierarchical levels. Hierarchy becomes an interesting feature to improve density, to reduce run time effort (divide and conquer) and to consider local communication. In the case of a Tree-based interconnect we get multiple hierarchical levels. Levels number depends on the total number of CLBs and clusters size (arity). Basically if 2 signals are within the same hierarchy level, it does not really matter where they are within that hierarchy. Similarly, geometrically close cells incur greater delay to get to other locations outside their hierarchical boundary than to distant cells within their hierarchical boundary. Thus, unlike flat or island style device, a hierarchical architecture uses a natural placement algorithm based on recursive partitioning.

Multilevel hierarchical organization is considered in our CAD flow and netlists instances are partitioned between architecture clusters in the best possible way, reducing the desired objectives. There are two main partitioning approaches: bottom-up (clustering) and top-down. The choice between both approaches depends on levels number, clusters size, clusters number at each level and problem constraints. In [133], authors proposed to use hMetis [49] which uses an FM algorithm [47] based top-down partitioning approach for tree-based architecture. In fact top-down approaches based on FM refinement heuristics are efficient when we target a small number of clusters (parts) of important size (balance constraint). To investigate partitioning approaches, we used a multilevel hypergraph data structure called *Mangrove*. It provides a development framework for efficient modeling of hypergraph nested partitions. It offers a compact C++ data structure and a high level API. As illustrated in Fig. 3.10, this structure is organized as follows:

- *ClusteringHierarchy*: holds a vector of nested partitions called *Clustering Level*, and refers to a unique enclosing cluster *TopLevelCluster*,

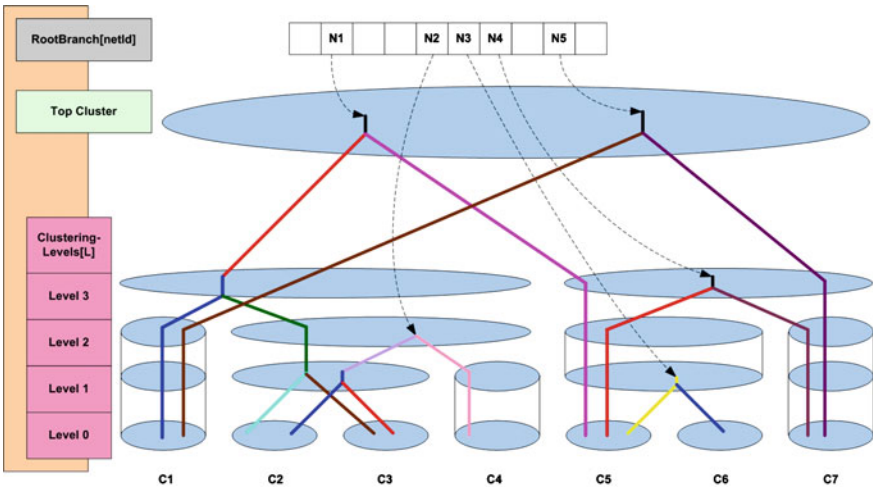

Fig. 3.10 Mangrove data structure: multilevel clustered hypergraph

- *ClusteringLevel*: corresponds to the set of clusters at the partitioning at a given level. A *clusteringLevel* corresponds to an hypergraph where nodes are clusters located at this level,
- *Cluster*: Aggregates sub-clusters belonging to a lower *ClusteringLevel* (unless leaf one). A *Cluster* may cross multiple levels and has *UpperLevel* and *LowerLevel* identifiers,
- *Net*: presents a tree of branches,
- *Branch*: represents the net (signal) crossing point of a cluster boundary. Branch bifurcates within a cluster if the net crosses at least 2 sub-clusters.

Since in Mangrove a *clusteringLevel* can be added at any level, this structure can be used in different partitioning approaches: Bottom-up and top-down. The combination of both approaches leads to an efficient multilevel partitioner where first multilevel bottom-up coarsening is run and then top-down multilevel refinement is applied. In Fig. 3.11, we show different steps of recursive netlist partitioning based on a multilevel approach. The netlist is first partitioned into 2 parts (first level) and then instances inside each part are partitioned into 2 fractions. In each partitioning phase we apply a multilevel coarsening followed by a multilevel refinement. Finally, we obtain the partitioning result corresponding to each level. The final result describes how instances are distributed between clusters of the Tree-based topology. Recursive partitioning is also interesting to reduce run time since it allows to avoid applying FM heuristics directly on a large number of parts, which can dramatically increase partitioning run time according to [75].

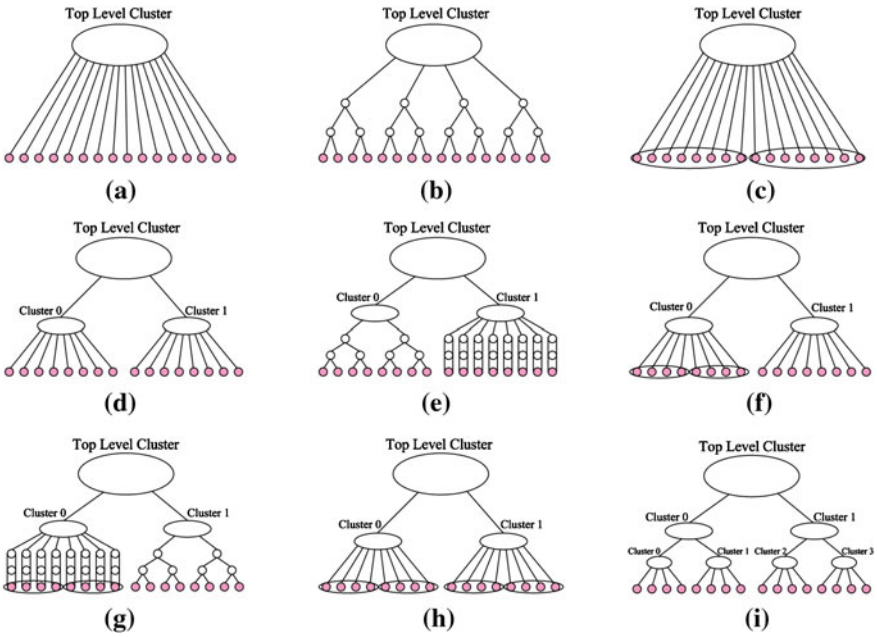

Fig. 3.11 2-levels recursive bi-partitioning steps. **a** Leaves at level 0 hypograph partitioning, **b** coarsened hypergraph, **c** bi-partitioned hypergraph, **d** clustering according to partitioning, **e** restricted coarsening in cluster 0, **f** bi-partitioning in sub hypergraph (cluster 0), **g** restricted coarsening in cluster 1, **h** bi-partitioning in sub hypergraph (cluster 1), **i** clustering according to partitioning

3.4.2.2 Routing

Once partitioning is done, placement file is generated that contains positions of different blocks on the architecture. This placement file along with netlist file is then passed to another software module called router which is responsible for the routing of netlist. In order to route all the nets of netlist, routing graph is first constructed that contains the details about all the routing resources of the architecture. The construction of routing graph is mainly dependant upon the cluster bandwidth information which is extracted from the architecture description file. Once the routing graph is constructed, routing resources of the architecture are later assigned to respective blocks of the netlist that are placed on the architecture. These routing resources are modeled as directed graph abstraction $G(V, E)$ where the set of vertices V represents the in/out pins of different blocks and the routing wires in the interconnect structure and an edge E between two vertices, represents a potential connection between the two vertices. Router is based on PathFinder [80] routing algorithm that uses an iterative, negotiation-based approach to successfully route all nets in a netlist.

3.4.3 Software Flow for Mesh-Based Architecture

3.4.3.1 Placement

For mesh-based architecture, the netlist obtained in the .net format is placed using a placement algorithm. The placement algorithm determines the position of block instances of a netlist on their respective block types on FPGA architecture. The main goal is to place connected instances near to each other so that minimum routing resources are required to route their connections. The placer uses simulated annealing algorithm [37, 105] to achieve a placement having minimum sum of half-perimeters of the bounding boxes of all the nets.

3.4.3.2 Routing

After the placement of netlist on the FPGA architecture, the exploration environment constructs routing graph for the architecture. Few architecture description parameters required for the construction of routing graph are taken from architecture description parameters. These parameters mainly include the type of routing network (unidirectional or bidirectional), channel width, I/O rate, block types and their pin positions on the block. After the construction of routing graph, the PathFinder routing algorithm [80] is used to route netlists on the routing architecture.

3.4.4 Timing Analysis

Timing analysis evaluates performances of a circuit implemented on an FPGA (mesh or tree) in terms of functional speed. Thus, once an application is completely placed and routed we estimate the minimum feasible clock separately for mesh and tree-based architectures. To achieve timing analysis we need 2 different graphs:

- Routed graph: Describes the way netlist instances are routed using architecture resources. This graph allows to evaluate routing delays between netlist instances connections. A path connecting two instances crosses several wires and switches. The connection delay is equal to the sum of resources delays.
- Timing graph: It is a direct acyclic graph generated from the netlist hypergraph. Nodes correspond to instances pins and edges to connections. Based on the resulting routed graph, each edge is labeled with the corresponding routed connection delay. The minimum required clock period is determined via a breadth-first traversal applied on this graph.

Only the routed graph is architecture dependent. Timing graph generation and critical path extraction depend only on netlist to implement.

3.4.5 Area and Delay Models

Once the netlists are placed and routed on the architecture, area and performance of the architecture is calculated using respective area and delay models. These models are generic in nature and they are applicable to both mesh-based and tree-based architectures. This section describes the generic area and delay models that we have used for both architectures.

3.4.5.1 Area Model

The area model is used to compute the areas of two architectures under consideration and these architectures are then compared using the area values calculated through their respective models. As mentioned in [22], discussions with FPGA vendors have revealed that transistor area, and not wiring density, is the area limiting factor. The use of directional wires in Virtex-I also suggests that routing area is transistor-dominant and must be reduced. As it was explained by DeHon [5], large area of switches compared to wires is one of the key reasons why we have to care about the number of switches required by a network. If the wire pitch is 5–8λ, the area of a wire crossing is 25–64λ^2. The area of static memory cell used to configure a switch is roughly 1,200λ^2. A switch transistor size is 2,500λ^2. In this case the ratio switches area/wires area can reach the value of 40. This ratio increases if we want more than just a pass gate for the switch. We may want to rebuffer the switch or even add a register to it. Such switch can easily be 5–10Kλ^2. The large area ratio means that we definitely need to take much care about switch count in the interconnect.

Area of any FPGA architecture can be basically divided into two parts: logic area and routing area. Logic area is a small part of total area and it comprises of the area of logic cells (i.e. CLBs) of the architecture. Routing area, on the other hand, comprises of the area of switching cells used by the routing network of the architecture and it can take up to 90% of the total area of the architecture. Routing area of the architecture includes area of configuration memory, multiplexors and buffers etc. An example showing how these switching cells are combined to construct a routing interconnect is shown in Fig. 3.12. Area of SRAMs, multiplexors, buffers and Flip-Flops is taken from a symbolic standard cell library (SXLIB [9]) which works on unit Lambda(λ). Area of different cells used for the area calculation is shown in Table 3.1.

3.4.5.2 Delay Model

The delay through the routing network may easily be dominant in a programmable technology. Care is required to minimize interconnect delays. The 2 following factors are significant in this respect:

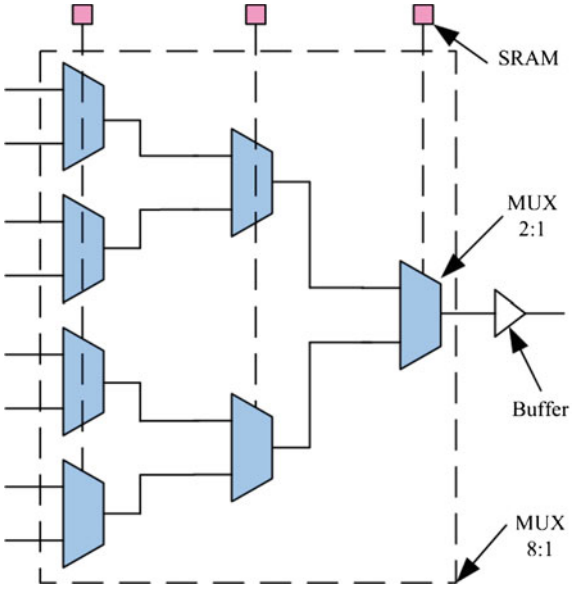

Fig. 3.12 An example showing the use of switching cells

Table 3.1 Area of different cells

Block name	Inputs	Outputs	Block size (λ^2)
clb	4	1	58,500
sram	–	–	1,500
buffer	1	1	1,000
flip-flop	1	1	4,500
mux 2:1	2	1	1,750

- Wires delay: Delay on a wire is proportional to distance and capacitive loading (fanout). This makes interconnect delay roughly proportional to distance run. Consequently short signals runs are faster than long signals runs.
- Switches delay: Each programmable switche in a path (crossbar, multiplexer) adds delay. This delay is generally much larger than the propagation or fanout delay. Consequently, one generally wants to minimize the number of switch elements in a path, even if this means using some longer signals runs.

Wire length and switches delays depend respectively on physical layout and cells library.

Table 3.2 Description of circuits used in experiments

Index	Circuit name	No of inputs	No of outputs	No of 4-input LUTs
1	pdc	16	40	3,832
2	ex5p	8	63	982
3	spla	16	46	3,045
4	apex4	9	19	1,089
5	frisc	20	116	2,841
6	apex2	38	3	1,522
7	seq	41	35	1,455
8	misex3	14	14	1,198
9	elliptic	131	114	2,712
10	alu4	14	8	1,242
11	des	256	245	1,506
12	s298	4	6	1,091
13	bigkey	229	197	1,147
14	diffeq	64	39	1,161
15	dsip	229	197	1,145
16	tseng	52	122	953

3.5 Experimentation and Analysis

Exploration environments described in the previous section are used to place and route different netlists on the two architectures and results are later compared to evaluate them. We have used 16 largest MCNC benchmarks for our experimentation. Details of these benchmarks are shown in Table 3.2. Name and I/Os of the circuits under consideration are shown in first three columns of the table whereas the size of each benchmark in terms of the number of 4 input LUTs used by it is shown in the last column of the table.

3.5.1 Architectures Optimization Approaches

The benchmarks shown in Table 3.2 are individually placed and routed on the two architectures under consideration. Specifically developed optimization approaches are used for both architectures to get the optimized area and delay results for both architectures. An overview of these optimization approaches is given below.

3.5.1.1 Tree-Based Architecture Optimization

Optimization of the tree-based architecture is dependant upon the information given in the architecture description file. If the optimization flag is false, the routing of the

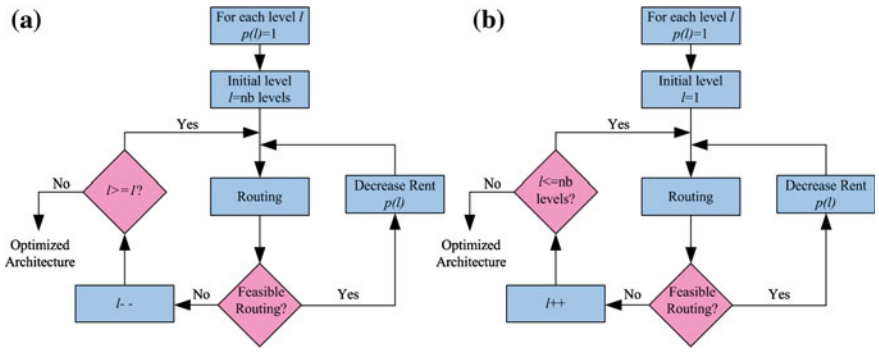

Fig. 3.13 Tree-based architecture optimization flow. **a** Top-down architecture optimization, **b** bottom-up architecture optimization

netlist is performed with given signal bandwidth and the exploration of the architecture is terminated after the calculation of area. However, if the optimization flag is true, a binary search algorithm is used to find the minimum signal bandwidth required to route the netlist. As explained in Sect. 3.1.2 the optimization of a tree-based architecture is dependant upon N and Rent's parameter p, the optimization algorithm that we employ is used to optimize the value of p for each level of the tree-based architecture. We apply binary search algorithm on each level to determine the minimum I/O bandwidth required at that particular level and this process continues until all the levels of the architecture are optimized. The value of p is then averaged across all the levels to determine the p for architecture. Although the clusters situated at different levels of hierarchy may have different values of p, clusters located at same level have same value of p. Based on the level order, we have explored three different types of optimization approaches for tree-based FPGA architecture. A brief overview of these approaches is described as follows:

1. Top-down approach: As shown in Fig. 3.13a, we start by optimizing the top level down to the lowest one. At each level we apply binary search to determine the smallest input/output signals number allowing to route the benchmark circuit.
2. Bottom-up approach: As shown in Fig. 3.13b, we start by optimizing the lowest level up to the highest one. At each level we apply binary search to determine the smallest input/output signals number allowing to route the benchmark circuit.
3. Random approach: All levels are optimized simultaneously. We choose a level randomly, we decrease its input/output signals number, depending on the previous result obtained in this level; then we move to an other level. In this way we move randomly from a level to another until all levels are optimized.

The 3 approaches have the same objective and aim at reducing clusters signals bandwidth for every level. The difference is the order in which levels are processed. In Table 3.3, we show architecture Rent's parameter (in each level) obtained with each technique. The first column of the table shows Rent's parameters, at each level, obtained after circuits partitioning. Results correspond to averages of all 16 circuits.

Table 3.3 Levels Rent's rule parameters

Level	Circuits partitioning	Architecture top-down	Architecture bottom-up	Architecture random
1	0.64	0.98	0.79	0.88
2	0.55	0.88	0.74	0.79
3	0.50	0.80	0.77	0.76
4	0.49	0.75	0.86	0.73
5	0.45	0.59	0.87	0.7

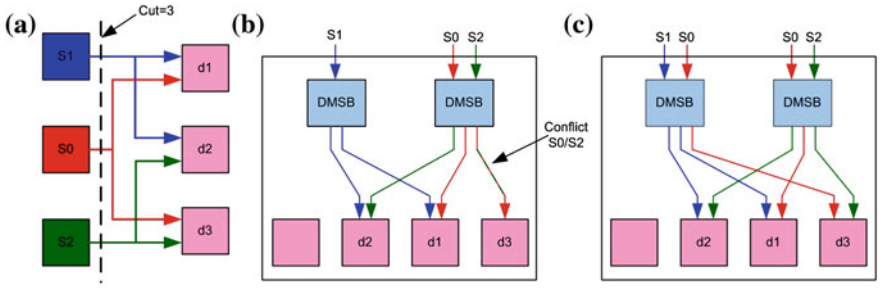

Fig. 3.14 A netlist routing example. **a** Partitioned netlist, **b** routed netlist with conflict, **c** routed netlist with no conflict

We notice that in all cases, architecture Rent's parameters are larger than partitioned circuits Rent's parameters. This is due to the depopulated switch boxes topology. In fact, to solve routing conflicts, a signal may enter from 2 different DMSB to reach 2 different destinations located at the same cluster. In Fig. 3.14 we show an example of a partitioned netlist to place and route on an architecture with LBs inputs number equal to 2 (2 DMSBs in each cluster located at level 1) and clusters size equal to 4. As shown in Fig. 3.14, if each signal enters from only one DMSB, we cannot solve conflicts. To deal with such problem we propose to enter the signal driven by S0 from two different DMSBs. Thus, the resulting architecture cluster degree is equal to 4, whereas the corresponding part degree is equal to 3 (number of crossing signals).

In Fig. 3.15, we show the average overhead between partitioning and architecture Rent's parameters with each optimizing approach. We notice that in the case of the top-down (bottom-up) approach, overhead increases when we go down (up) in the Tree. This was expected since the top-down (bottom-up) approach first optimizes high (low) levels. With the random approach, we notice that levels overheads are balanced.

We compared the resulting architectures (with the 3 approaches) in terms of area and speed performance. Average results are shown in Table 3.4. We notice that with the random approach we obtain the smallest area (22% less than top-down and 8% less than bottom-up). This means that optimizing levels simultaneously allows avoiding local minima and obtaining a balanced congestion distribution over levels. The bottom-up approach provides a smaller area than the top-down one. Nevertheless,

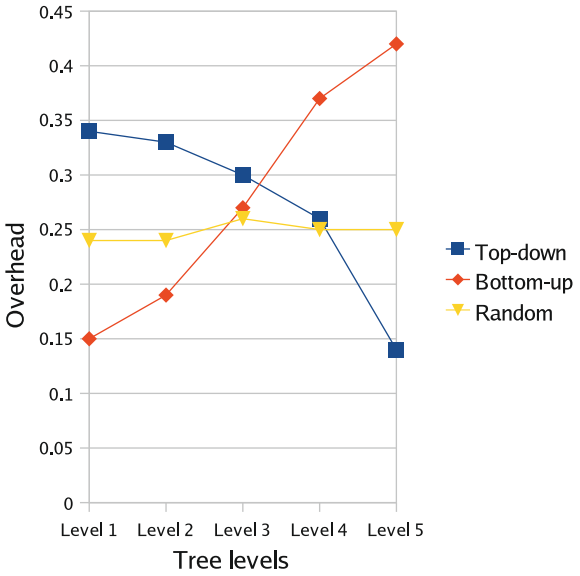

Fig. 3.15 Overhead between architecture and partitioned netlist Rent's parameters (16 benchmark avg.)

Table 3.4 Area and performance comparison between various optimizing approaches

Optimizing approach	Area (λ^2) $\times 10^6$	Critical path switches
Top-down	1,498	98
Bottom-up	1,326	106
Random	1,221	101

it is penalizing in terms of critical path switches number (8% more switches than top-down approach). In fact starting by optimizing low levels means that local routing resources are intensively reduced and signals are routed with resources located at higher levels. Consequently, signals routing uses more switches in series.

To reduce the gap between circuit and architecture Rent's parameters, we must improve the partitioning tool and especially the objective function in order to reduce congestion and resources (clusters inputs) required to route signals.

3.5.1.2 Mesh-Based Architecture Optimization

Like tree-based architecture, the optimization of mesh-based architecture is also dependant on the information given in the architecture description file. If the binary search flag is false, routing of the netlist is performed using a given value of channel width and the experimentation is terminated with the area calculation. However if the binary search flag is true, routing graph is constructed for varying channel widths;

routing is tried for each channel width until a minimum channel width is found. This optimization approach is similar vpr-based optimization approach used in [22].

3.5.2 *Effect of LUT and Arity Size on Tree-Based FPGA Architecture*

Before we start with the comparison between two FPGAs under consideration, effect of LUT (K) and arity (N) size is first explored for tree-based FPGA architecture. Many studies in the past several years have been carried out to see the effect of LUT and cluster size on the density of mesh-based FPGA architecture. The work in [8] compiles a very detailed study regarding the effect of LUT and cluster size on the density and performance of FPGA architecture. In [8] the authors have shown that LUTs with sizes 4 to 6 and clusters with sizes 3 to 10 give the most efficient results in terms of area-delay product for an FPGA. The work in [71] demonstrated that LUT size of 4 is most area efficient in a non clustered context. But all the work previously done in this context focusses on the mesh-based architecture and no prior work has been done yet for tree-based architectures. In this work, we first start our experimentation by exploring the effect of LUT and arity size on a tree-based architecture. This exploration is significant in the sense that an appropriate combination of K and N plays an important role in the overall efficiency of the architecture. In order to perform the exploration we use 16 MCNC [108] benchmarks shown in Table 3.2. These benchmarks are generated with different LUT (K) and arity sizes (N) and then they are placed and routed on the tree-based architecture using the flow described in Sect. 3.4. For these benchmarks, LUT size is varied from 3 to 7 while arity size is varied from 4 to 8.

Effect of LUT (K) and arity size (N) on area and performance of tree-based FPGA architecture is shown in Figs. 3.16 and 3.17 respectively. It can be seen from Fig. 3.16 that for almost all arity sizes, there is a reduction in total average area from LUT-3 to LUT-4. However, as the LUT size is increased beyond LUT-4, the total area of the architecture increases. It can be concluded from this figure that $K = 4$ with $N = 4$ gives best overall area results.

The second key metric is the effect of K and N on the performance of tree-based architecture. Since we do not have accurate wire lengths, only results regarding number of switches crossed by critical path are presented here. The extraction of number of switches crossed by critical path is explained in Sect. 3.4.4. Figure 3.17 shows the effect of varying K and N on critical path performance of the architecture. Results presented in this figure correspond to average number of switches crossed by 16 benchmarks under consideration. It is clear from this figure that an increase in K or N decreases the average number of switches that are crossed by critical path. This is because of the fact that an increase in K or N decreases the architecture size and there are smaller number of switches on the critical path. However, an increase in K or N increases the size of these switches; hence resulting in an increase in the

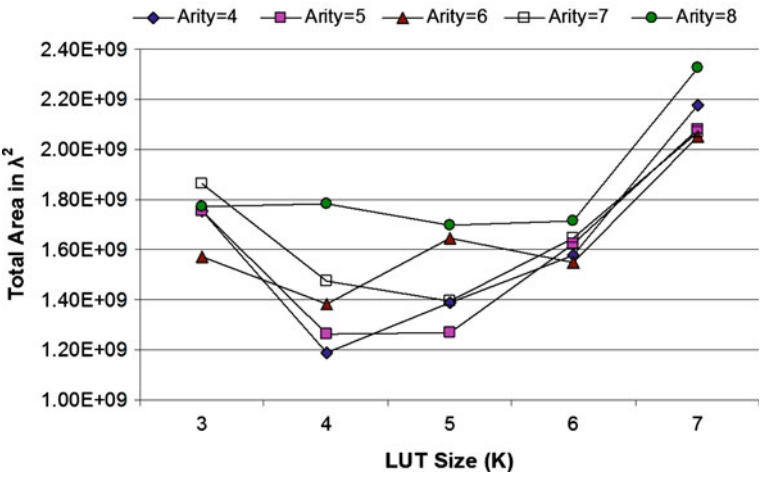

Fig. 3.16 Effect of LUT and arity size on total area of tree-based FPGA architecture

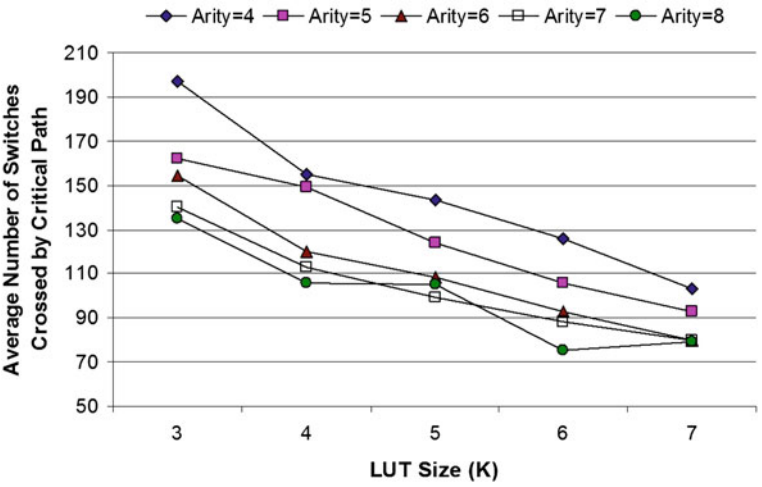

Fig. 3.17 Effect of LUT and arity size on critical path of tree-based FPGA architecture

intrinsic delay of the switch. So from this figure we can conclude that with a increase in K and N there will be decline in the critical path delay provided that the increase in internal delay of switch due to its increased size does not overshadow the reduction in number of switches. Further details regarding the effect of K and N on tree-based architecture can be found in [46] where we have come up with the conclusion that overall $K = 4$ with $N = 4$ provide the best overall results and this is the combination that will be used for tree-based FPGA architecture in this work.

Table 3.5 Architecture characteristics

Circuit name	Mesh-based architecture			Tree-based architecture		
	Architecture N×N	Occupancy (%)	Channel width	Architecture levels	Occupancy (%)	Rent's p
pdc	64 × 64	93	20	4 × 4 × 4 × 4 × 4 × 4 × 4	93	0.98
ex5p	32 × 32	95	16	4 × 4 × 4 × 4 × 4 × 4	96	1.00
spla	57 × 57	93	18	4 × 4 × 4 × 4 × 4 × 4 × 4	74	0.90
apex4	34 × 34	94	16	4 × 4 × 4 × 4 × 4 × 4 × 2	53	0.83
frisc	55 × 55	94	14	4 × 4 × 4 × 4 × 4 × 4 × 4	69	0.86
apex2	40 × 40	95	14	4 × 4 × 4 × 4 × 4 × 4 × 2	74	0.91
seq	39 × 39	96	16	4 × 4 × 4 × 4 × 4 × 4 × 2	71	0.89
misex3	36 × 36	92	14	4 × 4 × 4 × 4 × 4 × 4 × 2	58	0.84
elliptic	53 × 53	97	12	4 × 4 × 4 × 4 × 4 × 4 × 4	66	0.80
alu4	36 × 36	96	14	4 × 4 × 4 × 4 × 4 × 4 × 2	60	0.83
des	40 × 40	94	10	4 × 4 × 4 × 4 × 4 × 4 × 2	73	0.91
s298	34 × 34	94	12	4 × 4 × 4 × 4 × 4 × 4 × 2	53	0.77
bigkey	35 × 35	93	8	4 × 4 × 4 × 4 × 4 × 4 × 2	56	0.74
diffeq	35 × 35	95	10	4 × 4 × 4 × 4 × 4 × 4 × 2	56	0.72
dsip	35 × 35	93	8	4 × 4 × 4 × 4 × 4 × 4 × 2	56	0.74
tseng	31 × 31	99	8	4 × 4 × 4 × 4 × 4 × 4	93	0.88
Average	–	94	14	–	68	0.85

3.5.3 Comparison Between Homogeneous Mesh and Tree-Based FPGAs

In [132], a comparison between mesh-based and tree-based FPGAs is performed and it is shown that on average tree-based FPGA is 56% better than mesh-based FPGA in terms of area. However, the reference mesh-based architecture used in [132] is a bidirectional FPGA architecture and in [77], authors have proposed to replace the bidirectional interconnect with the unidirectional interconnect as the later gives better results compared to the former. So in this chapter we have changed the reference mesh-based architecture from bidirectional to unidirectional and we have re-evaluated the tree-based architecture. For tree-based architecture the LUT size is set to be 4 while arity size is set to be 4 too as it gives best overall results for it and for mesh-based architecture the LUT size is also set to be 4 and CLB size is set to be 1 for both architectures.

Experimental results of the two architectures are shown in Tables 3.5 and 3.6 respectively. Experiments are performed for individual netlists where an appropriate architecture is defined for each netlist and architecture is optimized using the optimization algorithm described in previous section. Although individual optimization approach has been, at times, controversial because most engineers think that FPGA is a fixed device and it does not vary in response to individual circuits that are being mapped on it. However, this more refine approach is usually used as it is necessary

Table 3.6 Comparison results between mesh-based and tree-based architectures

Circuit name	Mesh-based architecture			Tree-based architecture			Gain		
	Area $\times 10^6\lambda^2$	SRAMs $\times 10^3$	MUXs $\times 10^3$	Area $\times 10^6\lambda^2$	SRAMs $\times 10^3$	MUXs $\times 10^3$	Area (%)	SRAMs (%)	MUXs (%)
pdc	2,756	425	810	1,344	289	404	51	32	50
ex5p	567	84	162	296	62	86	47	25	46
spla	1,994	304	577	1,078	211	305	46	30	47
apex4	639	95	183	441	83	114	30	12	37
frisc	1,483	221	415	965	185	264	34	16	36
apex2	787	118	220	525	106	146	33	9	33
seq	839	124	240	480	93	128	42	24	46
misex3	639	95	179	417	77	105	34	19	40
elliptic	1,208	183	329	812	154	201	32	15	38
alu4	639	95	179	433	80	113	32	15	36
des	572	93	158	523	85	144	8	8	9
s298	500	76	136	378	66	92	24	12	32
bigkey	364	56	96	348	56	81	4	0.54	16
diffeq	432	70	119	330	56	74	23	20	38
dsip	364	56	96	347	56	81	4	0.64	15
tseng	281	43	74	242	41	54	13	5	26
Average	879	134	248	559	106	150	29	15	34

to evaluate the quality of an architecture in a more precise manner [7]. Different architectural parameters for the two architectures (architecture size, occupancy and signal bandwidth) are shown in Table 3.5 where individual architecture is defined and optimized for each of the netlist under consideration. It can be seen from the table that, compared to the occupancy of mesh-based architecture, tree-based architecture has a smaller average occupancy. This smaller occupancy of tree-based architecture is due to its hierarchical nature and compared to mesh-based architecture the logic resources of the tree-based architecture are under utilized. However, poor logic utilization is remedied by spreading the congestion of interconnect resources (congestion spreading effect is explained below) which eventually leads to better area results compared to mesh-based architecture.

Area results of the two architectures are shown in Table 3.6. It can be seen from the table that tree-based architecture gives better area results for all the netlists and on average it gives 29% area gain compared to unidirectional mesh-based architecture. One of the reasons of this area gain is the ability of tree-based architecture to simultaneously control the logic occupancy and interconnect population. It can be seen from Table 3.5 that generally the netlists with higher occupancy have a higher Rent's p (e.g. netlist named ex5p) and the netlists with smaller occupancy have a smaller Rent's p. This confirms that we can balance the interconnect and logic utilization by decreasing the logic occupancy and spreading the congestion. In fact, for tree-based architecture, we use a high-interconnect/low-logic utilization which is in

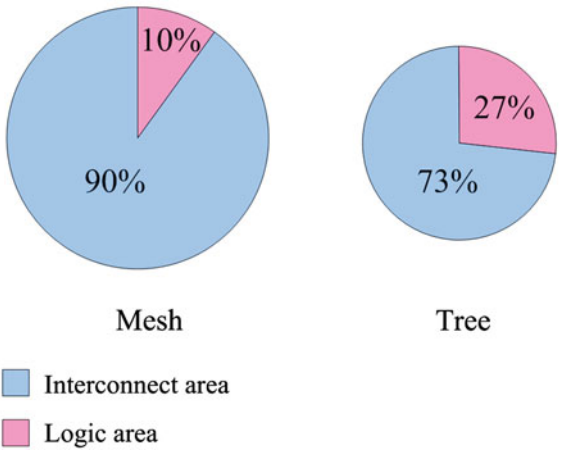

Fig. 3.18 Area distribution between interconnect and logic area for mesh-based and tree-based architectures

contrast to the mesh architecture's high-logic utilization approach. It can be seen from Fig. 3.18, unlike mesh-based architecture where interconnect occupies 90% of total area, in tree-based architecture interconnect occupies 73% of total area. This is because of the fact that in tree-based architecture, lower logic occupancy leads to higher interconnect depopulation which ultimately leads to better area results for tree-based architecture.

3.6 FPGA Hardware Generation

This section presents an automated method of generating hardware description of mesh-based and tree-based FPGA architectures. For both mesh-based and tree-based architectures, the FPGA hardware generator is integrated with their exploration environment. By doing so, all FPGA architectural parameters that are supported by the exploration environment are automatically supported by the VHDL model generator. Different size of FPGAs, with varying signal bandwidths, different variety of blocks, modification in connection patterns are automatically supported by this VHDL model generator. The VHDL model is passed to Cadence Encounter to generate layout of FPGA for 130 nm 6-metal layer CMOS process of ST Micro-Electronics. Different steps involved in the FPGA hardware generation are described in the sections that follow where the hardware generation of both mesh-based and tree-based architectures is detailed. Although this chapter considers only the generation of VHDL model of two FPGA architectures, an approximate layout scheme is taken into account to generate efficient VHDL model for two architectures. This scheme considers the efficient distribution of logic and routing resources of the architecture and it tries to distribute them in a uniform manner; hence eventually leading towards a less

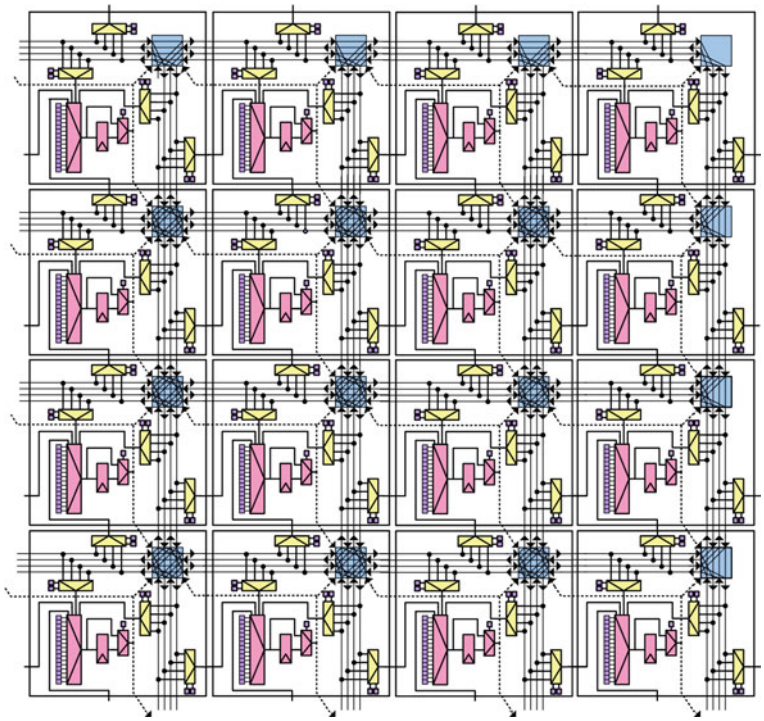

Fig. 3.19 Detailed interconnect of 16 tiles of mesh-based architecture

congested architecture. An overview of the two architectures is already presented in Sect. 3.1.

As discussed earlier, mesh-based architecture uses a unidirectional interconnect and the detailed interconnect of different tiles of mesh-based architecture is shown in Fig. 3.19. In this figure the detailed interconnect of 16 tiles is shown that are arranged in a 4x4 manner. As it can be seen from the figure that each tile contains switch box, a connection box, a CLB and routing wires on top and right side of CLB. These tiles can be further replicated to build larger architectures. In order to generate an optimized VHDL model of the architecture, these tiles are numbered in an appropriate manner and the logic and routing resources covered by each tile are also numbered. These numbers are later used while generating the VHDL model of the architecture.

For tree-based architecture, the approximate layout scheme needs an arrangement of logic and routing resources and it can be explained with the help of Fig. 3.20. In this figure, second level cluster of a tree-base architecture is shown where this cluster contains four level 1 clusters and each level 1 cluster in turn contains 4 CLBs which makes it a part of arity-4 tree-based architecture. Contrary to Fig. 3.6, in this figure switches and wires are depicted in different colors and for clarity only a small portion of the total interconnect is shown here. This is done to differentiate between the switches and wires of downward and upward interconnect of different levels. It

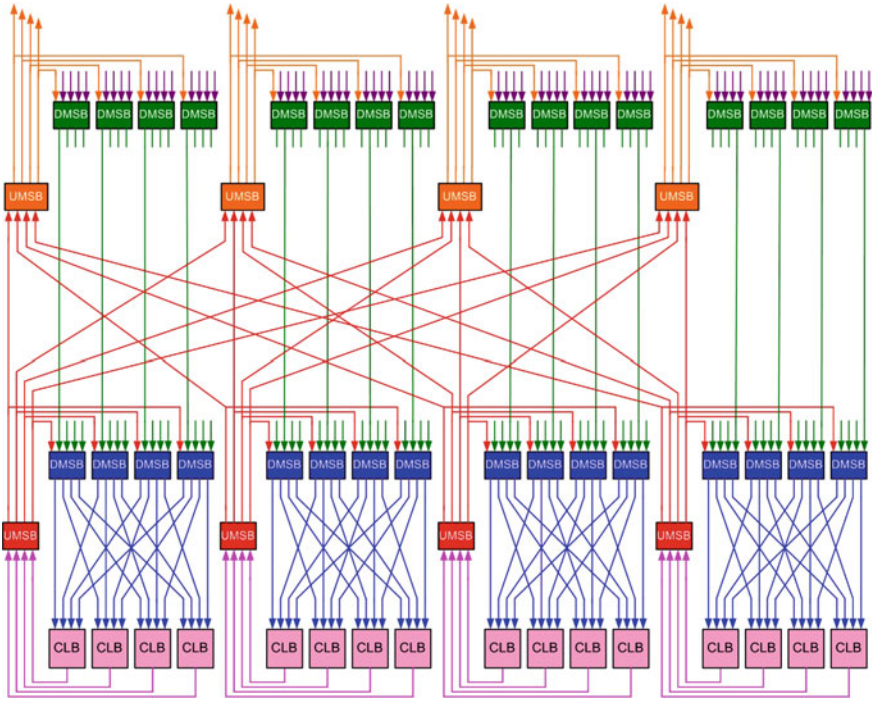

Fig. 3.20 Sparse interconnect of a level 2 cluster of a tree-based architecture

can be seen from this figure that every DMSB of level 1 cluster and its wires are shown in blue and every UMSB of level 1 and its wires are shown in red. Similarly for level 2 cluster the color is green for DMSBs and orange for UMSBs.

The approximate layout scheme of this cluster is shown in Fig. 3.21. This is a detailed but topologically equivalent scheme of the cluster shown in Fig. 3.20. In this figure switch blocks are broken down into small switches where each switch is placed horizontally or vertically in front of its successor. The division of a switch block into smaller switches is explained in Fig. 3.22 where a full cross bar switch has 5 inputs and four outputs and it is divided into programmable switches (multiplexors). Every programmable switch is composed of a group of switches and these groups are either placed in the same row or same column as their successor. CLBs of level 1 cluster of Fig. 3.20 are arranged in rows and the interconnect of two levels is interwoven uniformly to build a regular structure based on tiles. Each tile in Fig. 3.21 contains a CLB, a set of level 1 switches and a set of level 2 switches. In order to vary the arity of the architecture, we vary the number of CLBs in a row and in order to vary the number of inputs and outputs in a cluster level, we vary the number of switches in the tile. This corresponds to change in multiplexor size in level 1 or level 2. Thus scalability is taken care of in terms of arity and number of I/Os. It can be seen from the figure that all tiles are equivalent in terms of logic and switches distribution. Similarly all

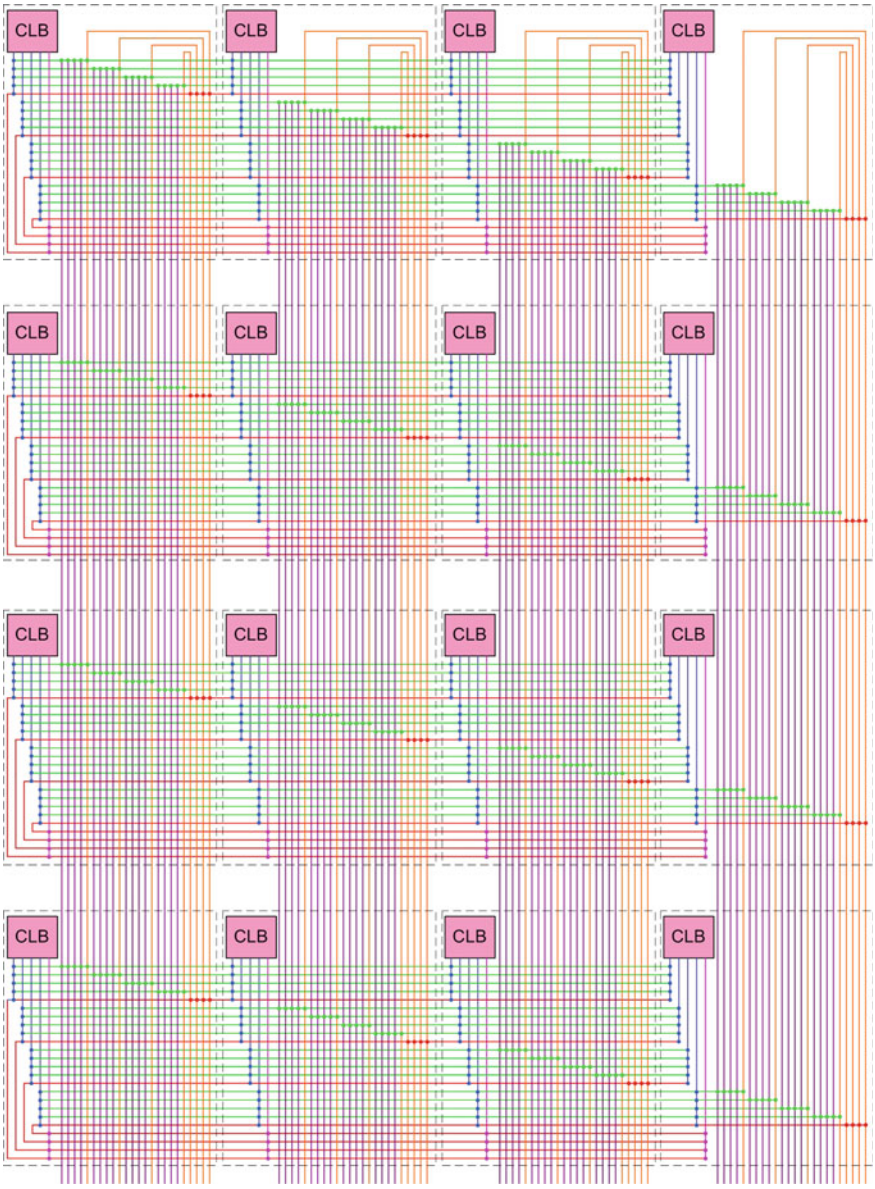

Fig. 3.21 Floor-plan of level 2 cluster of tree-based FPGA architecture

tiles of a column are equivalent. Although tiles of same row are different in routing topology, they are still equivalent in terms of number of switches. To generate the VHDL model for larger tree-based architectures, same technique of interweaving is applied as it gives flexibility in terms of arity and number of inputs and outputs of a

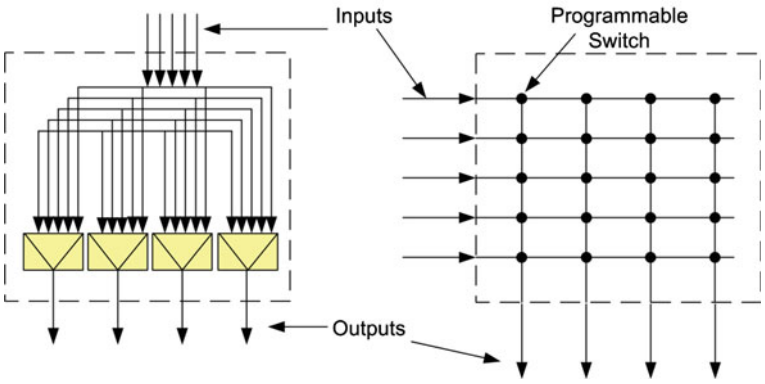

Fig. 3.22 Detailed topology of a switch block

cluster. However, it is important to mention here that it is just an approximate scheme and it does not represent the custom layout of tree-based architecture.

3.6.1 FPGA Generation Flow

Figure 3.23 shows the generalized flow that is used to generate the FPGA hardware in an automated manner. This flow is applicable to both mesh-based and tree-based architectures except that the two architectures use their respective architecture description mechanisms and exploration environments to generate the hardware. As it can be seen from the figure that the flow takes three parameters as its input:

1. Database of blocks contains the blocks definition of all the blocks that are supported by the FPGA architecture.
2. Architecture description file contains different architecture parameters using which the architecture is built. Details of the architecture description and block definition mechanism in mesh-based and tree-based architectures is given in Sect. 3.3.
3. Third parameter of the flow is the netlist that contains the blocks that are defined using block definition database and this netlist is then placed and routed on the architecture which is constructed using the parameters of architecture description file.

FPGA architecture exploration environment uses the architecture description file to construct the architecture and later the netlist is efficiently placed and routed on this architecture. Once the netlist is placed and routed on the architecture, exploration environment generates results like the floor-planning and routing graph of the architecture. An FPGA is basically represented by the floor-planning of different blocks and the routing graph connecting these blocks. The FPGA VHDL model generator

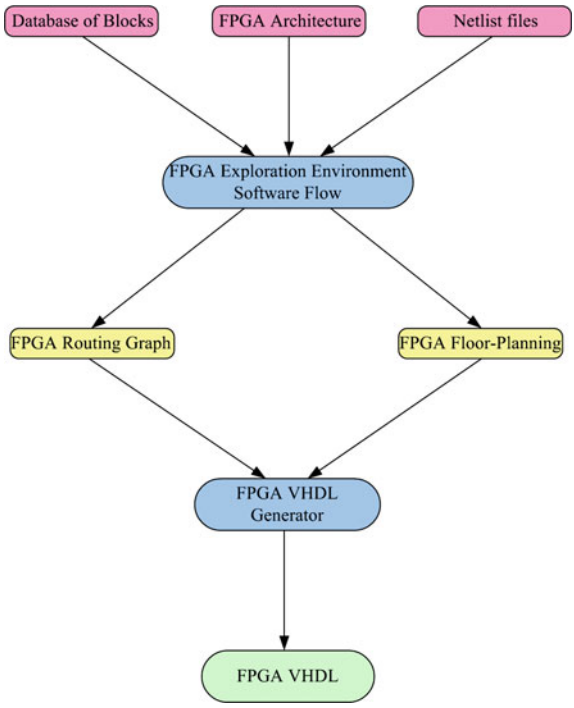

Fig. 3.23 FPGA VHDL model generation flow

uses FPGA floor-planning and FPGA routing graph to generate the FPGA VHDL model.

3.6.2 FPGA VHDL Model Generation

As shown in Fig. 3.23, the VHDL model of an FPGA is generated by using FPGA floor-planning and FPGA routing graph. An FPGA architecture may contain different kinds of blocks including I/Os, CLBs. The FPGA floor-planning gives the position of different blocks on the FPGA. These blocks are interconnected through a routing network. The routing network of the FPGA is represented by a routing graph. An FPGA routing graph contains nodes that are connected through edges; nodes represent a wire, and an edge represent the connections between different wires. A wire in the routing graph can be an input or output pin of a block, or a routing wire of the routing network.

The VHDL model generation using routing graph is explained with the help of a small example as shown in Fig. 3.24. Figure 3.24a shows a generalized full cross bar switch block which has four inputs and four outputs. The routing graph for this switch block is shown in Fig. 3.24b. This routing graph can be parsed to generate its

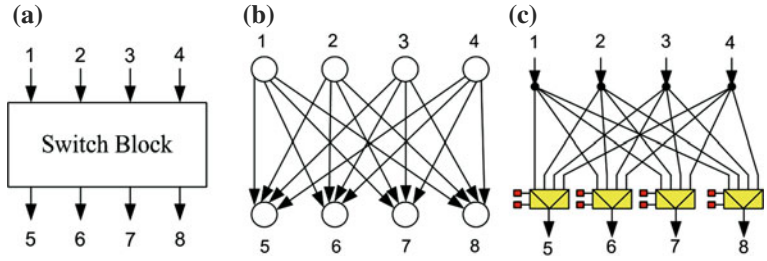

Fig. 3.24 FPGA VHDL model generation from routing graph. **a** Full cross-bar switch block, **b** routing graph for FPGA switch block, **c** physical representation for FPGA switch box

physical representation. The physical representation of FPGA switch block is shown in Fig. 3.24c. If a node is driven by more than one nodes, a multiplexor along with the required SRAMs is used to drive multiple nodes to the receiver node. If a node is driven by only a single node, a buffer is used to drive the receiver node. The physical representation of the routing graph is later translated to a VHDL model. This VHDL model is generated using a symbolic standard cell library, SXLIB [9].

The FPGA routing graph is parsed to generate its physical representation. If a receiver wire is driven by two or more wires, a multiplexor of appropriate size is connected to the receiver wire. If the receiver wire is being driven by only one wire, a buffer is used to connect the driver wire to the receiver wire. When a multiplexor is inserted, the SRAM bits required by the multiplexor are also declared along. The multiplexors and buffers belong to the same tile to which the receiver wire belongs. The SRAM bits connected to the multiplexor also belong to the same tile to which the multiplexor belongs.

The IO and logic block instances are also declared. The logic block can be a soft-block such as a CLB, or a hard-block such as multiplier or adder etc. The input and output pins of these blocks are already represented in the routing graph. Thus, these blocks are automatically linked to physical representation of an FPGA. These blocks are declared in their respective tiles. The VHDL model of these blocks is provided along with the architecture description. The SRAMs used by any logic blocks are also placed in the same way as the SRAMs of routing network are placed.

3.6.3 FPGA Layout Generation

The FPGA VHDL model is generated using a symbolic standard cell library, SXLIB [9]. The generated VHDL model is synthesized to 130 nm standard cell library of STMicroelectronics, and then passed to Cadence encounter for layout generation. The fanout loads are fixed through automatic buffer insertion and gate resizing. The default parameters of encounter enforce at least 5% space reserved for empty space

(fillers). FPGA layout generation uses these default parameters. If there is too much congestion in the chip, more fillers can be inserted to facilitate routing of a chip.

3.7 Summary and Conclusion

In this chapter, a brief overview of homogeneous mesh-based and tree-based architectures is given. Separate environments are designed for the exploration of the two architectures. Exploration of the architectures starts with a detailed architecture description mechanism and once the architecture is defined, netlists are placed and routed on the architecture using a specifically designed software flow. The new feature of this software flow is that now it can use both uni-directional or bi-directional routing networks for mesh-based architecture. Experiments are performed to evaluate the two architectures and results show that for a set of 16 MCNC benchmarks, on average, tree-based architecture is 29% better than uni-directional mesh-based architecture. A generalized hardware generation method for mesh-based and tree-based FPGA architectures is also presented. The method is automated as it is directly integrated with the exploration environments of both architectures and it takes into account the characteristics that are possessed by different exploration techniques of both architectures. Further the method presented here is generalized in the sense that it can be applied to either homogeneous or heterogeneous architectures.

Results presented in this chapter are for homogeneous architectures (i.e. architectures containing only CLBs and I/Os). Environments presented in this chapter are valid only for homogeneous architecture. However, they act as a stepping stone for a number of aspects that are designed and explored in the remaining chapters of this work. In the next chapter the two architectures and their respective environments are modified to support a heterogeneous mixture of blocks. Heterogeneity in FPGAs has become increasingly important as it gives them advantages in terms of area, speed and power consumption over their homogeneous counterparts. In the next chapter a number of techniques are explored for both mesh-based and tree-based architectures and their effect is evaluated by placing and routing a number of heterogeneous benchmarks on the two architectures. The work and some of the results presented in this chapter are also published in [132].