Chapter 2
FPGA Architectures: An Overview

Field Programmable Gate Arrays (FPGAs) were first introduced almost two and a half decades ago. Since then they have seen a rapid growth and have become a popular implementation media for digital circuits. The advancement in process technology has greatly enhanced the logic capacity of FPGAs and has in turn made them a viable implementation alternative for larger and complex designs. Further, programmable nature of their logic and routing resources has a dramatic effect on the quality of final device's area, speed, and power consumption.

This chapter covers different aspects related to FPGAs. First of all an overview of the basic FPGA architecture is presented. An FPGA comprises of an array of programmable logic blocks that are connected to each other through programmable interconnect network. Programmability in FPGAs is achieved through an underlying programming technology. This chapter first briefly discusses different programming technologies. Details of basic FPGA logic blocks and different routing architectures are then described. After that, an overview of the different steps involved in FPGA design flow is given. Design flow of FPGA starts with the hardware description of the circuit which is later synthesized, technology mapped and packed using different tools. After that, the circuit is placed and routed on the architecture to complete the design flow.

The programmable logic and routing interconnect of FPGAs makes them flexible and general purpose but at the same time it makes them larger, slower and more power consuming than standard cell ASICs. However, the advancement in process technology has enabled and necessitated a number of developments in the basic FPGA architecture. These developments are aimed at further improvement in the overall efficiency of FPGAs so that the gap between FPGAs and ASICs might be reduced. These developments and some future trends are presented in the last section of this chapter.

2.1 Introduction to FPGAs

Field programmable Gate Arrays (FPGAs) are pre-fabricated silicon devices that can be electrically programmed in the field to become almost any kind of digital circuit or system. For low to medium volume productions, FPGAs provide cheaper solution and faster time to market as compared to Application Specific Integrated Circuits (ASIC) which normally require a lot of resources in terms of time and money to obtain first device. FPGAs on the other hand take less than a minute to configure and they cost anywhere around a few hundred dollars to a few thousand dollars. Also for varying requirements, a portion of FPGA can be partially reconfigured while the rest of an FPGA is still running. Any future updates in the final product can be easily upgraded by simply downloading a new application bitstream. However, the main advantage of FPGAs i.e. flexibility is also the major cause of its draw back. Flexible nature of FPGAs makes them significantly larger, slower, and more power consuming than their ASIC counterparts. These disadvantages arise largely because of the programmable routing interconnect of FPGAs which comprises of almost 90% of total area of FPGAs. But despite these disadvantages, FPGAs present a compelling alternative for digital system implementation due to their less time to market and low volume cost.

Normally FPGAs comprise of:

- Programmable logic blocks which implement logic functions.
- Programmable routing that connects these logic functions.
- I/O blocks that are connected to logic blocks through routing interconnect and that make off-chip connections.

A generalized example of an FPGA is shown in Fig. 2.1 where configurable logic blocks (CLBs) are arranged in a two dimensional grid and are interconnected by programmable routing resources. I/O blocks are arranged at the periphery of the grid and they are also connected to the programmable routing interconnect. The "programmable/reconfigurable" term in FPGAs indicates their ability to implement a new function on the chip after its fabrication is complete. The reconfigurability/programmability of an FPGA is based on an underlying programming technology, which can cause a change in behavior of a pre-fabricated chip after its fabrication.

2.2 Programming Technologies

There are a number of programming technologies that have been used for reconfigurable architectures. Each of these technologies have different characteristics which in turn have significant effect on the programmable architecture. Some of the well known technologies include static memory [122], flash [54], and anti-fuse [61].

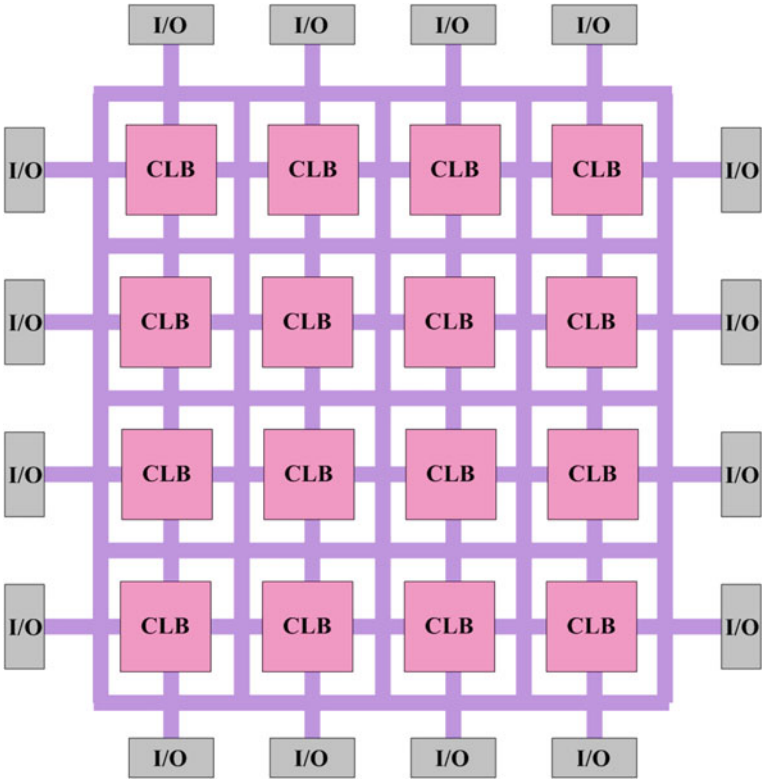


Fig. 2.1 Overview of FPGA architecture [22]

2.2.1 SRAM-Based Programming Technology

Static memory cells are the basic cells used for SRAM-based FPGAs. Most commercial vendors [76, 126] use static memory (SRAM) based programming technology in their devices. These devices use static memory cells which are divided throughout the FPGA to provide configurability. An example of such memory cell is shown in Fig. 2.2. In an SRAM-based FPGA, SRAM cells are mainly used for following purposes:

1. To program the routing interconnect of FPGAs which are generally steered by small multiplexors.
2. To program Configurable Logic Blocks (CLBs) that are used to implement logic functions.

SRAM-based programming technology has become the dominant approach for FPGAs because of its re-programmability and the use of standard CMOS process technology and therefore leading to increased integration, higher speed and lower

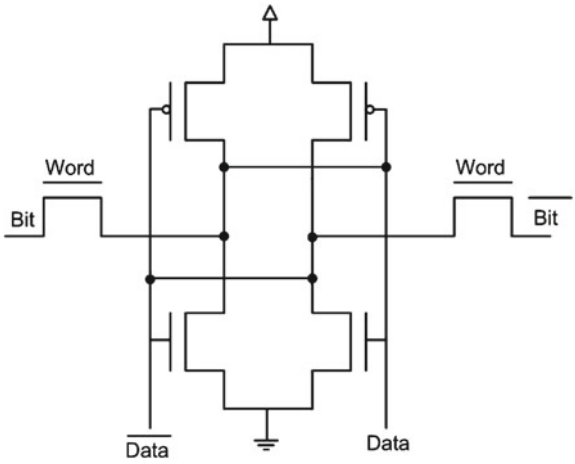


Fig. 2.2 Static memory cell

dynamic power consumption of new process with smaller geometry. There are however a number of drawbacks associated with SRAM-based programming technology. For example an SRAM cell requires 6 transistors which makes the use of this technology costly in terms of area compared to other programming technologies. Further SRAM cells are volatile in nature and external devices are required to permanently store the configuration data. These external devices add to the cost and area overhead of SRAM-based FPGAs.

2.2.2 Flash Programming Technology

One alternative to the SRAM-based programming technology is the use of flash or EEPROM based programming technology. Flash-based programming technology offers several advantages. For example, this programming technology is non-volatile in nature. Flash-based programming technology is also more area efficient than SRAM-based programming technology. Flash-based programming technology has its own disadvantages also. Unlike SRAM-based programming technology, flash-based devices can not be reconfigured/reprogrammed an infinite number of times. Also, flash-based technology uses non-standard CMOS process.

2.2.3 Anti-fuse Programming Technology

An alternative to SRAM and flash-based technologies is anti-fuse programming technology. The primary advantage of anti-fuse programming technology is its low area. Also this technology has lower on resistance and parasitic capacitance than other two

programming technologies. Further, this technology is non-volatile in nature. There are however significant disadvantages associated with this programming technology. For example, this technology does not make use of standard CMOS process. Also, anti-fuse programming technology based devices can not be reprogrammed.

In this section, an overview of three commonly used programming technologies is given where all of them have their advantages and disadvantages. Ideally, one would like to have a programming technology which is reprogrammable, non-volatile, and that uses a standard CMOS process. Apparently, none of the above presented technologies satisfy these conditions. However, SRAM-based programming technology is the most widely used programming technology. The main reason is its use of standard CMOS process and for this very reason, it is expected that this technology will continue to dominate the other two programming technologies.

2.3 Configurable Logic Block

A configurable logic block (CLB) is a basic component of an FPGA that provides the basic logic and storage functionality for a target application design. In order to provide the basic logic and storage capability, the basic component can be either a transistor or an entire processor. However, these are the two extremes where at one end the basic component is very fine-grained (in case of transistors) and requires large amount of programmable interconnect which eventually results in an FPGA that suffers from area-inefficiency, low performance and high power consumption. On the other end (in case of processor), the basic logic block is very coarse-grained and can not be used to implement small functions as it will lead to wastage of resources. In between these two extremes, there exists a spectrum of basic logic blocks. Some of them include logic blocks that are made of NAND gates [101], an interconnection of multiplexors [44], lookup table (LUT) [121] and PAL style wide input gates [124]. Commercial vendors like Xilinx and Altera use LUT-based CLBs to provide basic logic and storage functionality. LUT-based CLBs provide a good trade-off between too fine-grained and too coarse-grained logic blocks. A CLB can comprise of a single basic logic element (BLE), or a cluster of locally interconnected BLEs (as shown in Fig. 2.4). A simple BLE consists of a LUT, and a Flip-Flop. A LUT with k inputs (LUT-k) contains 2^k configuration bits and it can implement any k-input boolean function. Figure 2.3 shows a simple BLE comprising of a 4 input LUT (LUT-4) and a D-type Flip-Flop. The LUT-4 uses 16 SRAM bits to implement any 4 inputs boolean function. The output of LUT-4 is connected to an optional Flip-Flop. A multiplexor selects the BLE output to be either the output of a Flip-Flop or the LUT-4.

A CLB can contain a cluster of BLEs connected through a local routing network. Figure 2.4 shows a cluster of 4 BLEs; each BLE contains a LUT-4 and a Flip-Flop. The BLE output is accessible to other BLEs of the same cluster through a local routing network. The number of output pins of a cluster are equal to the total number of BLEs in a cluster (with each BLE having a single output). However, the number of input pins of a cluster can be less than or equal to the sum of input pins required

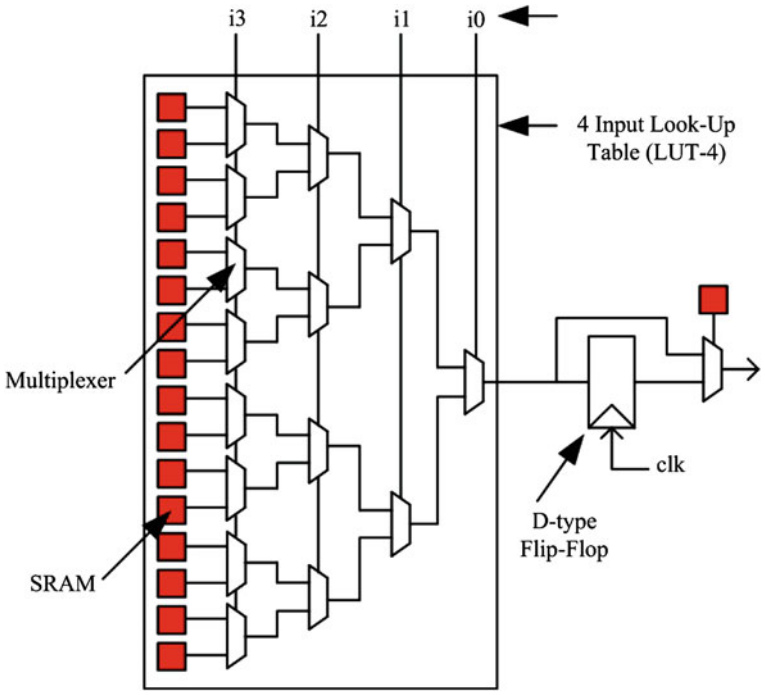


Fig. 2.3 Basic logic element (BLE) [22]

by all the BLEs in the cluster. Modern FPGAs contain typically 4 to 10 BLEs in a single cluster. Although here we have discussed only basic logic blocks, many modern FPGAs contain a heterogeneous mixture of blocks, some of which can only be used for specific purposes. Theses specific purpose blocks, also referred here as hard blocks, include memory, multipliers, adders and DSP blocks etc. Hard blocks are very efficient at implementing specific functions as they are designed optimally to perform these functions, yet they end up wasting huge amount of logic and routing resources if unused. A detailed discussion on the use of heterogeneous mixture of blocks for implementing digital circuits is presented in Chap. 4 where both advantages and disadvantages of heterogeneous FPGA architectures and a remedy to counter the resource loss problem are discussed in detail.

2.4 FPGA Routing Architectures

As discussed earlier, in an FPGA, the computing functionality is provided by its programmable logic blocks and these blocks connect to each other through programmable routing network. This programmable routing network provides routing

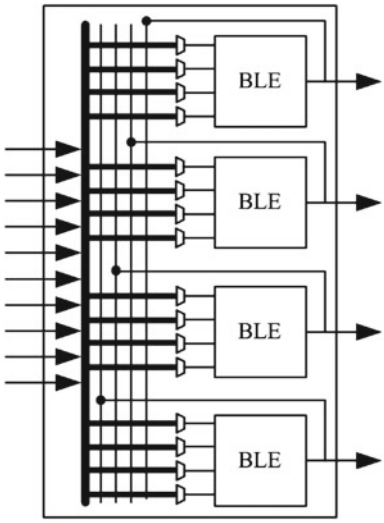


Fig. 2.4 A configurable logic block (CLB) having four BLEs [22]

connections among logic blocks and I/O blocks to implement any user-defined circuit. The routing interconnect of an FPGA consists of wires and programmable switches that form the required connection. These programmable switches are configured using the programmable technology.

Since FPGA architectures claim to be potential candidate for the implementation of any digital circuit, their routing interconnect must be very flexible so that they can accommodate a wide variety of circuits with widely varying routing demands. Although the routing requirements vary from circuit to circuit, certain common characteristics of these circuits can be used to optimally design the routing interconnect of FPGA architecture. For example most of the designs exhibit locality, hence requiring abundant short wires. But at the same time there are some distant connections, which leads to the need for sparse long wires. So, care needs to be taken into account while designing routing interconnect for FPGA architectures where we have to address both flexibility and efficiency. The arrangement of routing resources, relative to the arrangement of logic blocks of the architecture, plays a very important role in the overall efficiency of the architecture. This arrangement is termed here as global routing architecture whereas the microscopic details regarding the switching topology of different switch blocks is termed as detailed routing architecture. On the basis of the global arrangement of routing resources of the architecture, FPGA architectures can be categorized as either hierarchical [4] or island-style [22]. In this section, we present a detailed overview of both routing architectures.

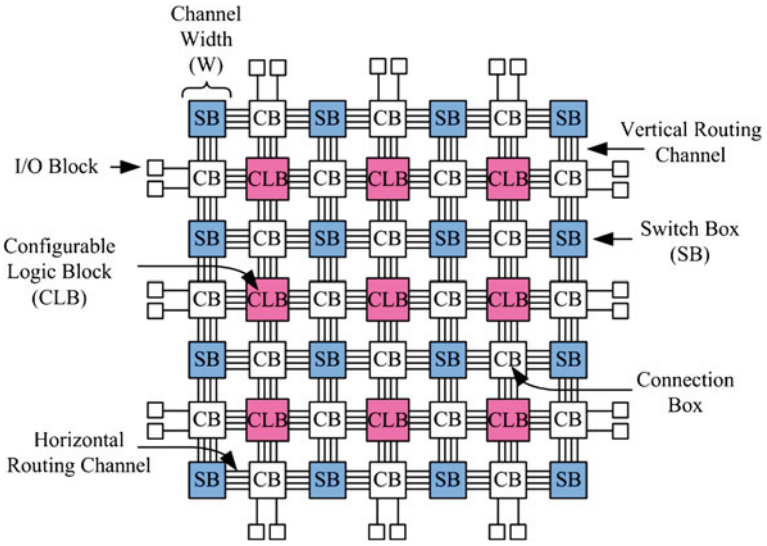


Fig. 2.5 Overview of mesh-based FPGA architecture [22]

2.4.1 Island-Style Routing Architecture

Figure 2.5 shows a traditional island-style FPGA architecture (also termed as mesh-based FPGA architecture). This is the most commonly used architecture among academic and commercial FPGAs. It is called island-style architecture because in this architecture configurable logic blocks look like islands in a sea of routing interconnect. In this architecture, configurable logic blocks (CLBs) are arranged on a 2D grid and are interconnected by a programmable routing network. The Input/Output (I/O) blocks on the periphery of FPGA chip are also connected to the programmable routing network. The routing network comprises of pre-fabricated wiring segments and programmable switches that are organized in horizontal and vertical routing channels.

The routing network of an FPGA occupies 80–90% of total area, whereas the logic area occupies only 10–20% area [22]. The flexibility of an FPGA is mainly dependent on its programmable routing network. A mesh-based FPGA routing network consists of horizontal and vertical routing tracks which are interconnected through switch boxes (SB). Logic blocks are connected to the routing network through connection boxes (CB). The flexibility of a connection box (Fc) is the number of routing tracks of adjacent channel which are connected to the pin of a block. The connectivity of input pins of logic blocks with the adjacent routing channel is called as Fc(in); the connectivity of output pins of the logic blocks with the adjacent routing channel is called as Fc(out). An Fc(in) equal to 1.0 means that all the tracks of adjacent routing channel are connected to the input pin of the logic block. The flexibility of switch box (Fs) is the total number of tracks with which every track entering in the switch

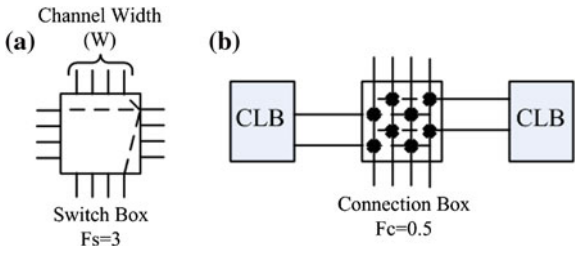


Fig. 2.6 Example of switch and connection box

box connects to. The number of tracks in routing channel is called the channel width of the architecture. Same channel width is used for all horizontal and vertical routing channels of the architecture. An example explaining the switch box, connection box flexibilities, and routing channel width is shown in Fig. 2.6. In this figure switch box has $Fs = 3$ as each track incident on it is connected to 3 tracks of adjacent routing channels. Similarly, connection box has $Fc(in) = 0.5$ as each input of the logic block is connected to 50% of the tracks of adjacent routing channel.

The routing tracks connected through a switch box can be bidirectional or unidirectional (also called as directional) tracks. Figure 2.7 shows a bidirectional and a unidirectional switch box having Fs equal to 3. The input tracks (or wires) in both these switch boxes connect to 3 other tracks of the same switch box. The only limitation of unidirectional switch box is that their routing channel width must be in multiples of 2.

Generally, the output pins of a block can connect to any routing track through pass transistors. Each pass transistor forms a tristate output that can be independently turned on or off. However, single-driver wiring technique can also be used to connect output pins of a block to the adjacent routing tracks. For single-driver wiring, tristate elements cannot be used; the output of block needs to be connected to the neighboring routing network through multiplexors in the switch box. Modern commercial FPGA architectures have moved towards using single-driver, directional routing tracks. Authors in [51] show that if single-driver directional wiring is used instead of bidirectional wiring, 25% improvement in area, 9% in delay and 32% in area-delay can be achieved. All these advantages are achieved without making any major changes in the FPGA CAD flow.

In mesh-based FPGAs, multi-length wires are created to reduce delay. Figure 2.8 shows an example of different length wires. Longer wire segments span multiple blocks and require fewer switches, thereby reducing routing area and delay. However, they also decrease routing flexibility, which reduces the probability to route a hardware circuit successfully. Modern commercial FPGAs commonly use a combination of long and short wires to balance flexibility, area and delay of the routing network.

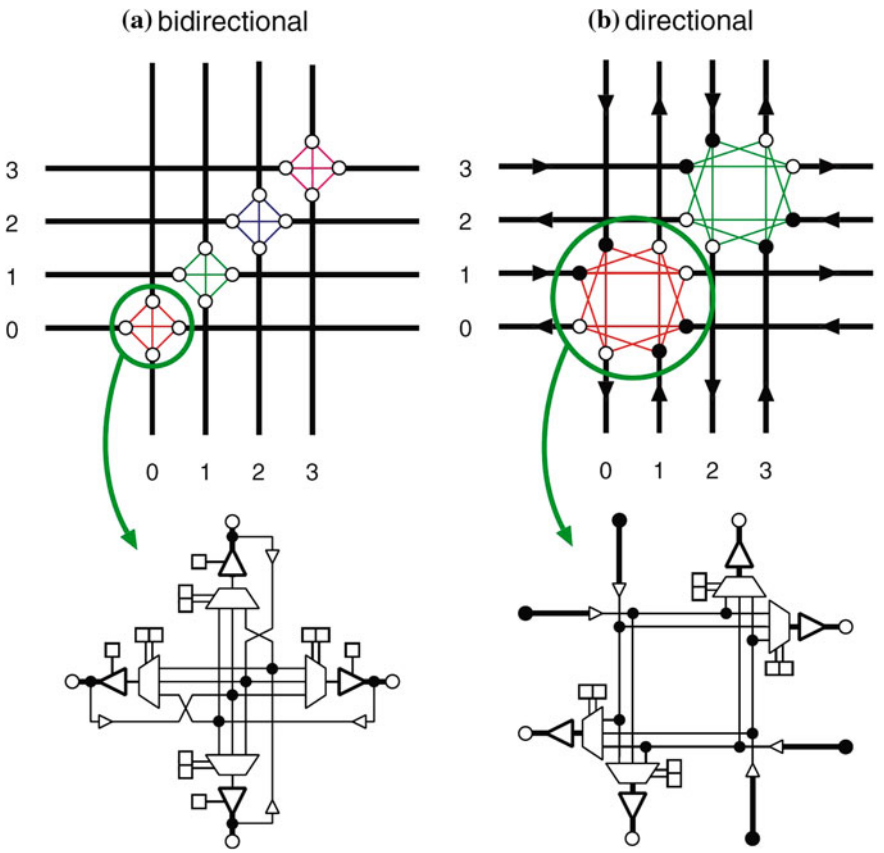


Fig. 2.7 Switch block, length 1 wires [51]

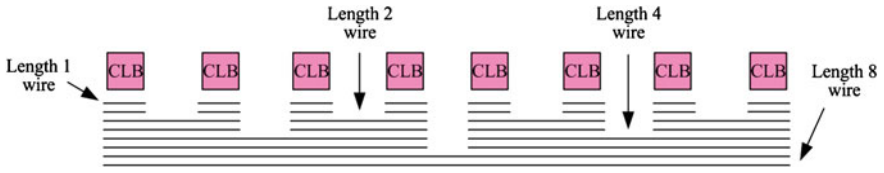


Fig. 2.8 Channel segment distribution

2.4.1.1 Altera's Stratix II Architecture

Until now, we have presented a general overview about island-style routing architecture. Now we present a commercial example of this kind of architectures. Altera's Stratix II [106] architecture is an industrial example of an island-style FPGA (Fig. 2.9). The logic structure consists of LABs (Logic Array Blocks), memory blocks, and digital signal processing (DSP) blocks. LABs are used to

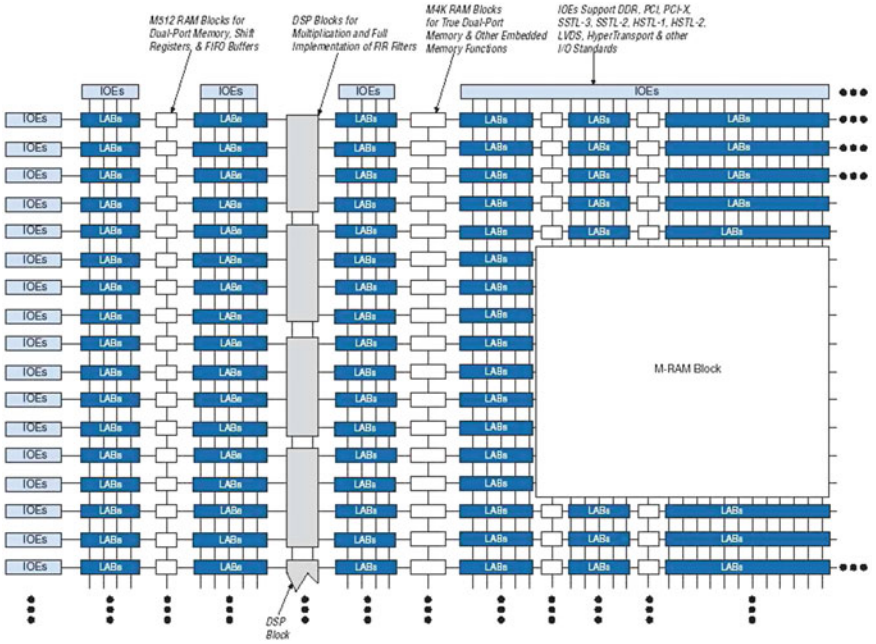


Fig. 2.9 Altera's stratix-II block diagram

implement general-purpose logic, and are symmetrically distributed in rows and columns throughout the device fabric. The DSP blocks are custom designed to implement full-precision multipliers of different granularities, and are grouped into columns. Input- and output-only elements (IOEs) represent the external interface of the device. IOEs are located along the periphery of the device.

Each Stratix II LAB consists of eight Adaptive Logic Modules (ALMs). An ALM consists of 2 adaptive LUTs (ALUTs) with eight inputs altogether. Construction of an ALM allows implementation of 2 separate 4-input Boolean functions. Further, an ALM can also be used to implement any six-input Boolean function, and some seven-input functions. In addition to lookup tables, an ALM provides 2 programmable registers, 2 dedicated full-adders, a carry chain, and a register-chain. Full-adders and carry chain can be used to implement arithmetic operations, and the register-chain is used to build shift registers. Outputs of an ALM drive all types of interconnect provided by the Stratix II device. Figure 2.10 illustrates a LAB interconnect interface.

Interconnections between LABs, RAM blocks, DSP blocks and the IOEs are established using the Multi-track interconnect structure. This interconnect structure consists of wire segments of different lengths and speeds. The interconnect wire-segments span fixed distances, and run in the horizontal (row interconnects) and vertical (column interconnects) directions. The row interconnects (Fig. 2.11) can be used to route signals between LABs, DSP blocks, and memory blocks in the same row. Row interconnect resources are of the following types:

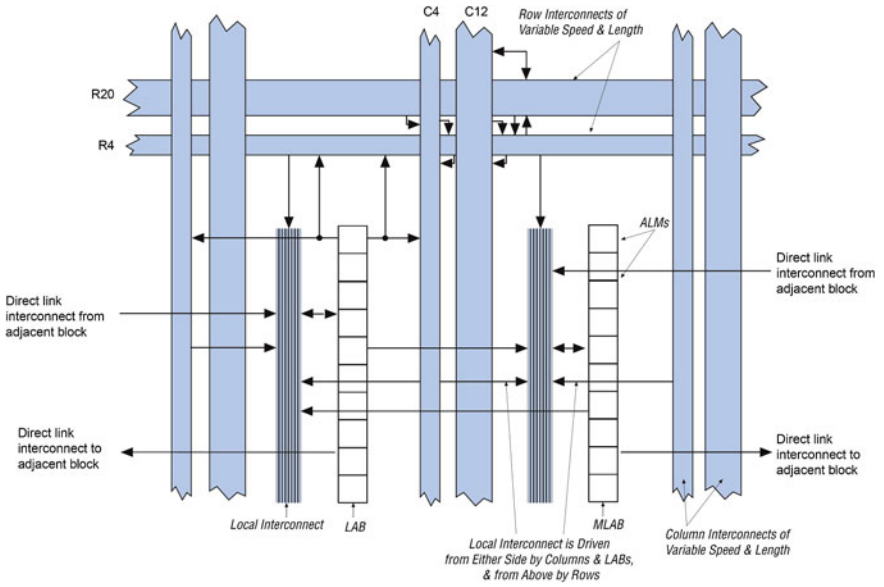


Fig. 2.10 Stratix-II logic array block (LAB) structure

- Direct connections between LABs and adjacent blocks.
- R4 resources that span 4 blocks to the left or right.
- R24 resources that provide high-speed access across 24 columns.

Each LAB owns its set of R4 interconnects. A LAB has approximately equal numbers of driven-left and driven-right R4 interconnects. An R4 interconnect that is driven to the left can be driven by either the primary LAB (Fig. 2.11) or the adjacent LAB to the left.

Similarly, a driven-right R4 interconnect may be driven by the primary LAB or the LAB immediately to its right. Multiple R4 resources can be connected to each other to establish longer connections within the same row. R4 interconnects can also drive C4 and C16 column interconnects, and R24 high speed row resources.

Column interconnect structure is similar to row interconnect structure. Column interconnects include:

- Carry chain interconnects within a LAB, and from LAB to LAB in the same column.
- Register chain interconnects.
- C4 resources that span 4 blocks in the up and down directions.
- C16 resources for high-speed vertical routing across 16 rows.

Carry chain and register chain interconnects are separated from local interconnect (Fig. 2.10) in a LAB. Each LAB has its own set of driven-up and driven-down C4 interconnects. C4 interconnects can also be driven by the LABs that are immediately

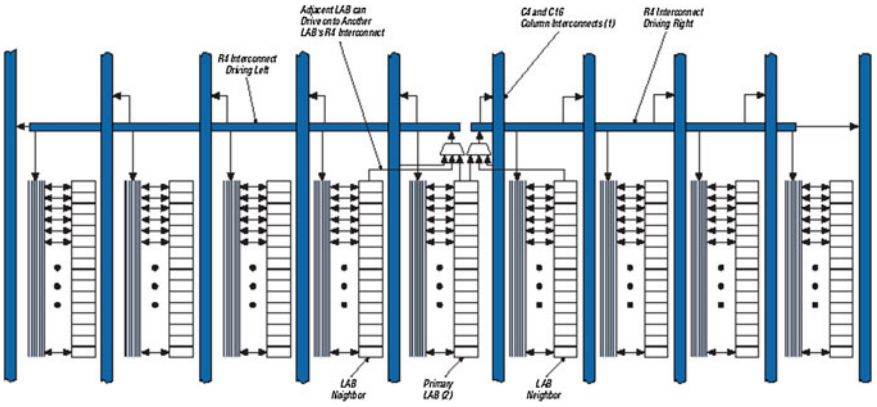


Fig. 2.11 R4 interconnect connections

adjacent to the primary LAB. Multiple C4 resources can be connected to each other to form longer connections within a column, and C4 interconnects can also drive row interconnects to establish column-to-column interconnections. C16 interconnects are high-speed vertical resources that span 16 LABs. A C16 interconnect can drive row and column interconnects at every fourth LAB. A LAB local interconnect structure cannot be directly driven by a C16 interconnect; only C4 and R4 interconnects can drive a LAB local interconnect structure. Figure 2.12 shows the C4 interconnect structure in the Stratix II device.

2.4.2 Hierarchical Routing Architecture

Most logic designs exhibit locality of connections; hence implying a hierarchy in placement and routing of connections between different logic blocks. Hierarchical routing architectures exploit this locality by dividing FPGA logic blocks into separate groups/clusters. These clusters are recursively connected to form a hierarchical structure. In a hierarchical architecture (also termed as tree-based architecture), connections between logic blocks within same cluster are made by wire segments at the lowest level of hierarchy. However, the connection between blocks residing in different groups require the traversal of one or more levels of hierarchy. In a hierarchical architecture, the signal bandwidth varies as we move away from the bottom level and generally it is widest at the top level of hierarchy. The hierarchical routing architecture has been used in a number of commercial FPGA families including Altera Flex10K [10], Apex [15] and ApexII [16] architectures. We assume that Multilevel hierarchical interconnect regroups architectures with more than 2 levels of hierarchy and Tree-based ones.

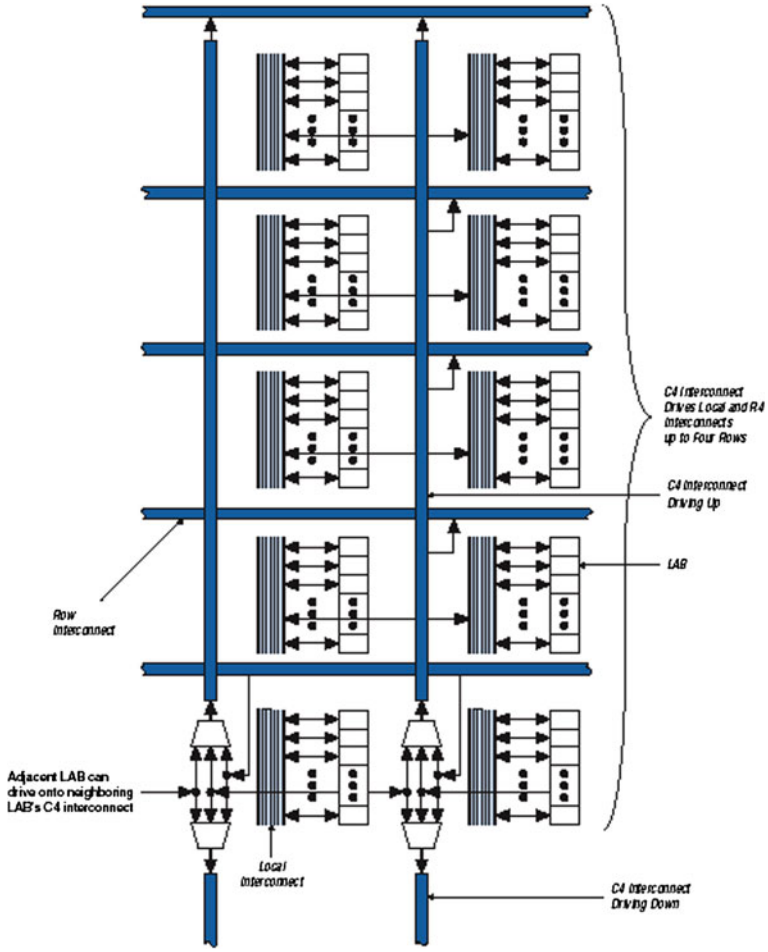


Fig. 2.12 C4 interconnect connections

2.4.2.1 HFPGA: Hierarchical FPGA

In the hierarchical FPGA called HFPGA, LBs are grouped into clusters. Clusters are then grouped recursively together (see Fig. 2.13). The clustered VPR mesh architecture [22] has a Hierarchical topology with only two levels. Here we consider multilevel hierarchical architectures with more than 2 levels. In [1] and [129] various hierarchical structures were discussed. The HFPGA routability depends on switch boxes topologies. HFPGAs comprising fully populated switch boxes ensure 100% routability but are very penalizing in terms of area. In [129] authors explored the HFPGA architecture, investigating how the switch pattern can be partly depopulated while maintaining a good routability.

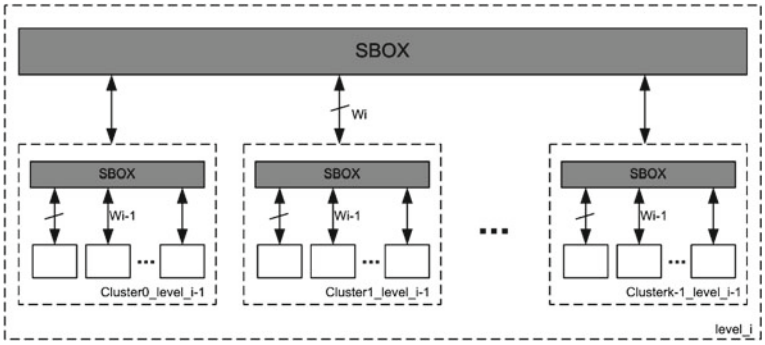


Fig. 2.13 Hierarchical FPGA topology

2.4.2.2 HSRA: Hierarchical Synchronous Reconfigurable Array

An example of an academic hierarchical routing architecture is shown in Fig. 2.14. It has a strictly hierarchical, tree-based interconnect structure. In this architecture, the only wire segments that directly connect to the logic units are located at the leaves of the interconnect tree. All other wire segments are decoupled from the logic structure. A logic block of this architecture consists of a pair of 2-input Look Up Table (2-LUT) and a D-type Flip Flop (D-FF). The input-pin connectivity is based on a choose-k strategy [4], and the output pins are fully connected. The richness of this interconnect structure is defined by its base channel width c and interconnect growth rate p. The base channel width c is defined as the number of tracks at the leaves of the interconnect Tree (in Fig. 2.14, $c = 3$). Growth rate p is defined as the rate at which the interconnect bandwidth grows towards the upper levels. The interconnect growth rate can be realized either using non-compressing or compressing switch blocks. The details regarding these switch blocks is as follows:

- Non-compressing (2:1) switch blocks—The number of tracks at the upper level are equal to the sum of the number of tracks of the children at lower level. For example, in Fig. 2.14, non-compressing switch blocks are used between levels 1, 2 and levels 3, 4.
- Compressing (1:1) switch blocks—The number of tracks at the upper level are equal to the number of tracks of either child at the lower level. For example, in Fig. 2.14, compressing switch blocks are used between levels 2 and 3.

A repeating combination of non-compressing and compressing switch blocks can be used to realize any value of p less than one. For example, a repeating pattern of (2:1, 1:1) switch blocks realizes $p = 0.5$, while the pattern (2:1, 2:1, 1:1) realizes $p = 0.67$. An architecture that has only 2:1 switch blocks provides a growth rate of $p = 1$.

Another hierarchical routing architecture is presented in [132] where the global routing architecture (i.e. the position of routing resources relative to logic resources

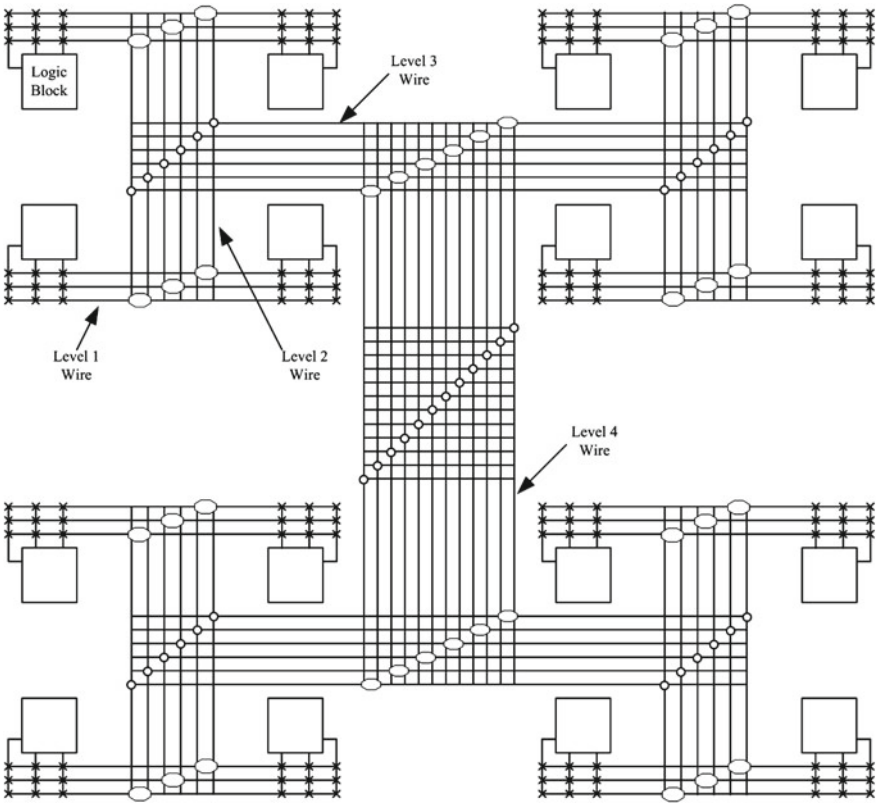


Fig. 2.14 Example of hierarchical routing architecture [4]

of the architecture) remains the same as in [4]. However, there are several key differences at the level of detailed routing architecture (i.e. the way the routing resources are connected to each other, flexibility of switch blocks etc.) that separate the two architectures. For example the architecture shown in Fig. 2.14 has one bidirectional interconnect that uses bidirectional switches and it supports only arity-2 (i.e. each cluster can contain only two sub-clusters). On contrary, the architecture presented in [132] supports two separate unidirectional interconnect networks: one is downward interconnect whereas other is upward interconnect network. Further this architecture is more flexible as it can support logic blocks with different sizes and also the clusters/groups of the routing architecture can have different arity sizes. Further details of this architecture, from now on alternatively termed as tree-based architecture, are presented in next chapter.

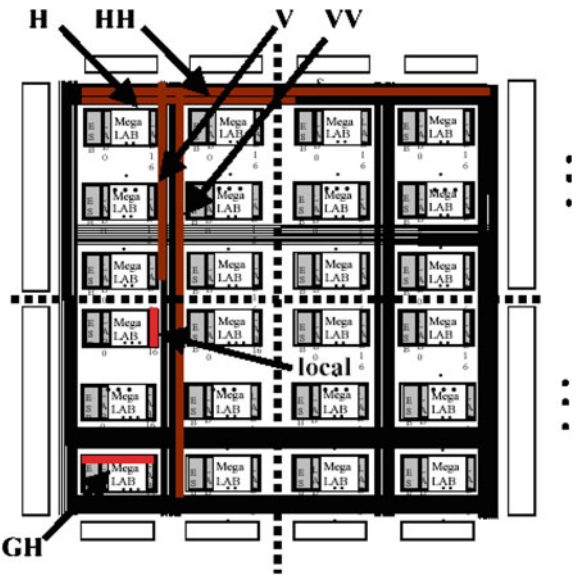


Fig. 2.15 The APEX programmable logic Devices [87]

2.4.2.3 APEX: Altera

APEX architecture is a commercial product from Altera Corporation which includes 3 levels of interconnect hierarchy. Figure 2.15 shows a diagram of the APEX 20K400 programmable logic device. The basic logic-element (LE) is a 4-input LUT and DFF pair. Groups of 10 LEs are grouped into a logic-array-block or LAB. Interconnect within a LAB is complete, meaning that a connection from the output of any LE to the input of another LE in its LAB always exists, and any signal entering the input region can reach every LE.

Groups of 16 LABs form a MegaLab. Interconnect within a MegaLab requires an LE to drive a GH (MegaLab global H) line, a horizontal line, which switches into the input region of any other LAB in the same MegaLab. Adjacent LABs have the ability to interleave their input regions, so an LE in LAB_i can usually drive LAB_{i+1} without using a GH line. A 20K400 MegaLab contains 279 GH lines.

The top-level architecture is a 4 by 26 array of MegaLabs. Communication between MegaLabs is accomplished by global H (horizontal) and V (vertical) wires, that switch at their intersection points. The H and V lines are segmented by a bidirectional segmentation buffer at the horizontal and vertical centers of the chip. In Fig. 2.15, We denote the use of a single (half-chip) line as H or V and a double or full-chip line through the segmentation buffer as HH or VV. The 20K400 contains 100 H lines per MegaLab row, and 80 V lines per LAB-column.

In this section, so far we have given an overview of the two routing architectures that are commonly employed in FPGAs. Both architectures have their positive and negative points. For example, hierarchical routing architectures exploit the

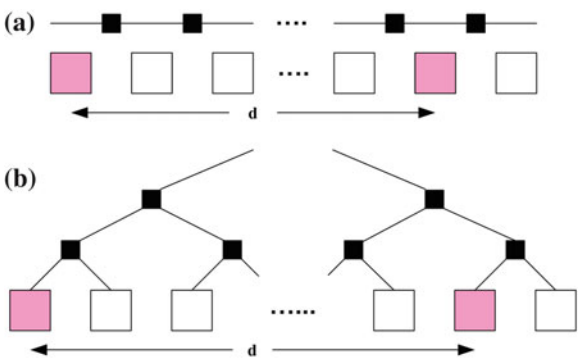


Fig. 2.16 **a** Number of series switches in a mesh structure **b** Number of series switches in a tree structure

locality exhibited by the most of the designs and in turn offer smaller delays and more predictable routing compared to island-style architectures. The speed of a net is determined by the number of routing switches it has to pass and the length of wires. In a mesh-based architecture, the number of segments increase linearly with manhattan distance d between the logic blocks to be connected. However, for tree-based architecture the distance d between the blocks to be connected increases in a logarithmic manner [82]. This fact is illustrated in Fig. 2.16. On the other hand, scalability is an issue in hierarchical routing architectures and there might be some design mapping issues. But in the case of mesh-based architecture, there are no such issues as it offers a tile-based layout where a tile once formed can be replicated horizontally and vertically to make as large architecture as we wish.

2.5 Software Flow

FPGA architectures have been intensely investigated over the past two decades. A major aspect of FPGA architecture research is the development of Computer Aided Design (CAD) tools for mapping applications to FPGAs. It is well established that the quality of an FPGA-based implementation is largely determined by the effectiveness of accompanying suite of CAD tools. Benefits of an otherwise well designed, feature rich FPGA architecture might be impaired if the CAD tools cannot take advantage of the features that the FPGA provides. Thus, CAD algorithm research is essential to the necessary architectural advancement to narrow the performance gaps between FPGAs and other computational devices like ASICs.

The software flow (CAD flow) takes an application design description in a Hardware Description Language (HDL) and converts it to a stream of bits that is eventually programmed on the FPGA. The process of converting a circuit description into a format that can be loaded into an FPGA can be roughly divided into five distinct steps, namely: synthesis, technology mapping, mapping, placement and routing. The final output of FPGA CAD tools is a bitstream that configures the state of the memory

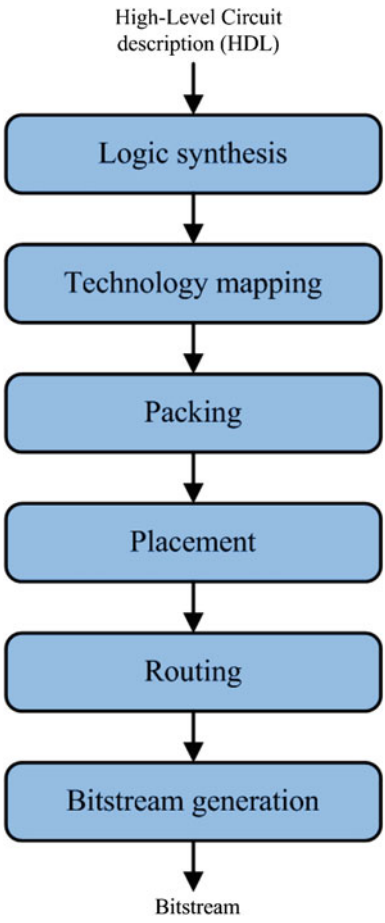


Fig. 2.17 FPGA software flow

bits in an FPGA. The state of these bits determines the logical function that the FPGA implements. Figure 2.17 shows a generalized software flow for programming an application circuit on an FPGA architecture. A description of various modules of software flow is given in the following part of this section. The details of these modules are generally indifferent to the kind of routing architecture being used and they are applicable to both architectures described earlier unless otherwise specified.

2.5.1 Logic Synthesis

The flow of FPGA starts with the logic synthesis of the netlist being mapped on it. Logic synthesis [26, 27] transforms an HDL description (VHDL or Verilog) into a set of boolean gates and Flip-Flops. The synthesis tools transform the

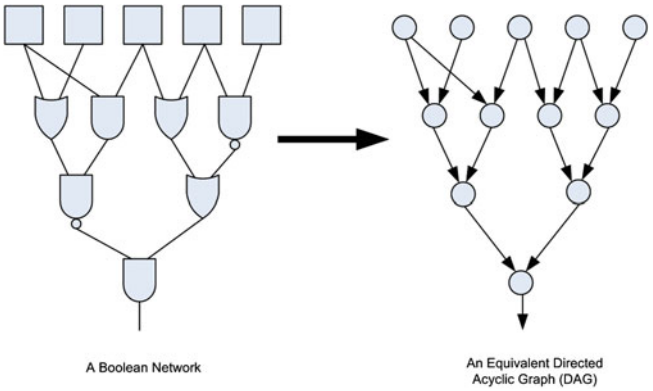


Fig. 2.18 Directed acyclic graph representation of a circuit

register-transfer-level (RTL) description of a design into a hierarchical boolean network. Various technology-independent techniques are applied to optimize the boolean network. The typical cost function of technology-independent optimizations is the total literal count of the factored representation of the logic function. The literal count correlates very well with the circuit area. Further details of logic synthesis are beyond the scope of this book.

2.5.2 Technology Mapping

The output from synthesis tools is a circuit description of Boolean logic gates, flip-flops and wiring connections between these elements. The circuit can also be represented by a Directed Acyclic Graph (DAG). Each node in the graph represents a gate, flip-flop, primary input or primary output. Each edge in the graph represents a connection between two circuit elements. Figure 2.18 shows an example of a DAG representation of a circuit. Given a library of cells, the technology mapping problem can be expressed as finding a network of cells that implements the Boolean network. In the FPGA technology mapping problem, the library of cells is composed of k-input LUTs and flip-flops. Therefore, FPGA technology mapping involves transforming the Boolean network into k-bounded cells. Each cell can then be implemented as an independent k-LUT. Figure 2.19 shows an example of transforming a Boolean network into k-bounded cells. Technology mapping algorithms can optimize a design for a set of objectives including depth, area or power. The FlowMap algorithm [64] is the most widely used academic tool for FPGA technology mapping. FlowMap is a breakthrough in FPGA technology mapping because it is able to find a depth-optimal solution in polynomial time. FlowMap guarantees depth optimality at the expense of logic duplication. Since the introduction of FlowMap, numerous technology mappers have been designed that optimize for area and run-time while still maintaining

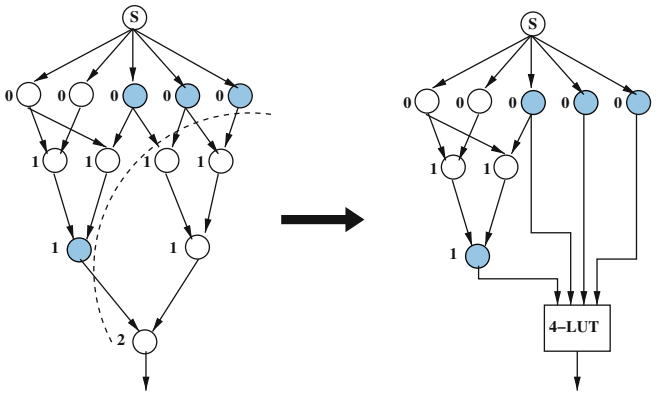


Fig. 2.19 Example of technology mapping

the depth-optimality of the circuit [65–67]. The result of the technology mapping step generates a network of k-bounded LUTs and flip-flops.

2.5.3 Clustering/Packing

The logic elements in a Mesh-based FPGA are typically arranged in two levels of hierarchy. The first level consists of logic blocks (LBs) which are k-input LUT and flip-flop pairs. The second level hierarchy groups k LBs together to form logic blocks clusters. The clustering phase of the FPGA CAD flow is the process of forming groups of k LBs. These clusters can then be mapped directly to a logic element on an FPGA. Figure 2.20 shows an example of the clustering process.

Clustering algorithms can be broadly categorized into three general approaches, namely top-down [39, 78], depth-optimal [84, 100] and bottom-up [14, 17, 43]. Top-down approaches partition the LBs into clusters by successively subdividing the network or by iteratively moving LBs between parts. Depth-optimal solutions attempt to minimize delay at the expense of logic duplication. Bottom-up approaches are generally preferred for FPGA CAD tools due to their fast run times and reasonable timing delays. They only consider local connectivity information and can easily satisfy clusters pin constraints. Top-down approaches offer the best solutions; however, their computational complexity can be prohibitive.

2.5.3.1 Bottom-up Approaches

Bottom-up approaches build clusters sequentially one at a time. The process starts by choosing an LB which acts as a cluster seed. LBs are then greedily selected and added to the cluster, applying various attraction functions. The VPack [14] attraction

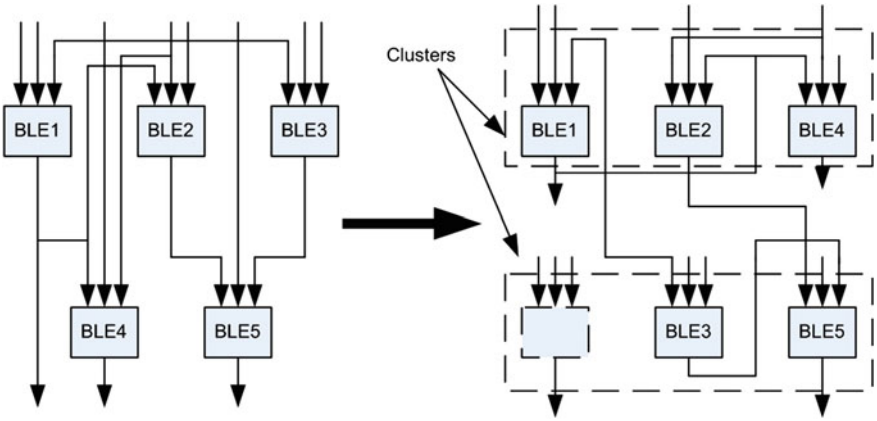


Fig. 2.20 Example of packing

function is based on the number of shared nets between a candidate LB and the LBs that are already in the cluster. For each cluster, the attraction function is used to select a seed LB from the set of all LBs that have not already been packed. After packing a seed LB into the new cluster, a second attraction function selects new LBs to pack into the cluster. LBs are packed into the cluster until the cluster reaches full capacity or all cluster inputs have been used. If all cluster inputs become occupied before this cluster reaches full capacity, a hill-climbing technique is applied, searching for LBs that do not increase the number of inputs used by the cluster. The VPack pseudo-code is outlined in algorithm 2.1.

T-VPack [22] is a timing-driven version of VPack which gives added weight to grouping LBs on the critical path together. The algorithm is identical to VPack, however, the attraction functions which select the LBs to be packed into the clusters are different. The VPack seed function chooses LBs with the most used inputs, whereas the T-VPack seed function chooses LBs that are on the most critical path. VPack's second attraction function chooses LBs with the largest number of connections with the LBs already packed into the cluster. T-VPack's second attraction function has two components for a LB B being considered for cluster C:

$$Attraction(B, C) = \alpha.Crit(B) + (1 - \alpha)\frac{\mid Nets(B) \cap Nets(C) \mid}{G} \qquad (2.1)$$

where $Crit(B)$ is a measure of how close LB B is to being on the critical path, $Nets(B)$ is the set of nets connected to LB B, $Nets(C)$ is the set of nets connected to the LBs already selected for cluster C, α is a user-defined constant which determines the relative importance of the attraction components, and G is a normalizing factor. The first component of T-VPack's second attraction function chooses critical-path LBs, and the second chooses LBs that share many connections with the LBs already packed into the cluster. By initializing and then packing clusters with


```
UnclusteredLBs = PatternMatchToLBs(LUTs,Registers);
LogicClusters = NULL;
while UnclusteredLBs != NULL do
    C = GetLBwithMostUsedInputs(UnclusteredLBs);
    while | C |< k do
        /*cluster is not full*/
        BestLB = MaxAttractionLegalLB(C,UnclusteredLBs);
        if BestLB == NULL then
            /*No LB can be added to this cluster*/
            break;
        endif
        UnclusteredLBs = UnclusteredLB − BestLB;
        C = C ∪ BestLB;
    endw
    if | C |< k then
        /*Cluster is not full - try hill climbing*/
        while | C |< k do
            BestLB = MinClusterInputIncreaseLB(C,UnclusteredLBs);
            C = C ∪ BestLB;
            UnclusteredLBs = UnclusteredLB − BestLB;
        endw
        if ClusterIsIllegal(C) then
            RestoreToLastLegalState(C,UnclusteredLBs);
        endif
    endif
    LogicClusters = LogicClusters ∪ C;
endw
```

Algorithm 2.1 Pseudo-code of the VPack Algorithm [22]

critical-path LBs, the algorithm is able to absorb long sequences of critical-path LBs into clusters. This minimizes circuit delay since the local interconnect within the cluster is significantly faster than the global interconnect of the FPGA. RPack [43] improves routability of a circuit by introducing a new set of routability metrics. RPack significantly reduced the channel widths required by circuits compared to VPack. T-RPack [43] is a timing driven version of RPack which is similar to T-VPack by giving added weight to grouping LBs on the critical path. iRAC [17] improves the routability of circuits even further by using an attraction function that attempts to encapsulate as many low fanout nets as possible within a cluster. If a net can be completely encapsulated within a cluster, there is no need to route that net in the external routing network. By encapsulating as many nets as possible within clusters, routability is improved because there are less external nets to route in total.

2.5.3.2 Top-down Approaches

The K-way partitioning problem seeks to minimize a given cost function of such an assignment. A standard cost function is net cut, which is the number of hyper-edges that span more than one partition, or more generally, the sum of weights of

such hyperedges. Constraints are typically imposed on the solution, and make the problem difficult. For example some vertices can be fixed in their parts or the total vertex weight in each part must be limited (balance constraint and FPGA clusters size). With balance constraints, the problem of partitioning optimally a hypergraph is known to be NP-hard [85]. However, since partitioning is critical in several practical applications, heuristic algorithms were developed with near-linear runtime. Such move-based heuristics for k-way hypergraph partitioning appear in [24, 34, 110].

Fiduccia-Mattheyses Algorithm

The Fiduccia-Mattheyses (FM) heuristics [34] work by prioritizing moves by gain. A move changes to which partition a particular vertex belongs, and the gain is the corresponding change of the cost function. After each vertex is moved, gains for connected modules are updated.

```
partitioning = initial_solution;
while solution quality improves do
    Initialize gain_container from partitioning;
    solution_cost = partitioning.get_cost();
    while not all vertices locked do
        move = choose_move();
        solution_cost += gain_container.get_gain(move);
        gain_container.lock_vertex(move.vertex());
        gain_update(move);
        partitioning.apply(move);
    endw
    roll back partitioning to best seen solution;
    gain_container.unlock_all();
endw
```

Algorithm 2.2 Pseudo-code for FM Heuristic [38]

The Fiduccia-Mattheyses (FM) heuristic for partitioning hypergraphs is an iterative improvement algorithm. FM starts with a possibly random solution and changes the solution by a sequence of moves which are organized as passes. At the beginning of a pass, all vertices are free to move (unlocked), and each possible move is labeled with the immediate change to the cost it would cause; this is called the gain of the move (positive gains reduce solution cost, while negative gains increase it). Iteratively, a move with highest gain is selected and executed, and the moving vertex is locked, i.e., is not allowed to move again during that pass. Since moving a vertex can change gains of adjacent vertices, after a move is executed all affected gains are updated. Selection and execution of a best-gain move, followed by gain update, are repeated until every vertex is locked. Then, the best solution seen during the pass is adopted as the starting solution of the next pass. The algorithm terminates when a

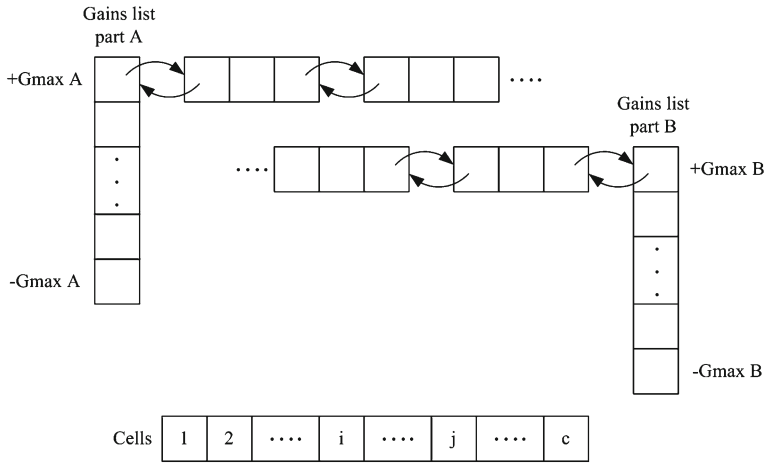


Fig. 2.21 The gain bucket structure as illustrated in [34]

pass fails to improve solution quality. Pseudo-code for the FM heuristic is given in algorithm 2.2.

The FM algorithm has 3 main components (1) computation of initial gain values at the beginning of a pass; (2) the retrieval of the best-gain (feasible) move; and (3) the update of all affected gain values after a move is made. One contribution of Fiduccia and Mattheyses lies in observing that circuit hypergraphs are sparse, and any move's gain is bounded between plus and minus the maximal vertex degree G_{max} in the hypergraph (times the maximal hyperedge weight, if weights are used). This allows prioritizing moves by their gains. All affected gains can be updated in amortized-constant time, giving overall linear complexity per pass [34]. All moves with the same gain are stored in a linked list representing a "gain bucket". Figure. 2.21 presents the gain bucket list structure. It is important to note that some gains G may be negative, and as such, FM performs hill-climbing and is not strictly greedy.

Multilevel Partitioning

The multilevel hypergraph partitioning framework was successfully verified by [31, 48, 49] and leads to the best known partitioning results ever since. The main advantage of multilevel partitioning over flat partitioners is its ability to search the solution space more effectively by spending comparatively more effort on smaller coarsened hypergraphs. Good coarsening algorithms allow for high correlation between good partitioning for coarsened hypergraphs and good partitioning for the initial hypergraph. Therefore, a thorough search at the top of the multilevel hierarchy is worthwhile because it is relatively inexpensive when compared to flat partitioning of the original hypergraph, but can still preserve most of the possible improvement.

The result is an algorithmic framework with both improved runtime and solution quality over a completely flat approach. Pseudo-code for an implementation of the multilevel partitioning framework is given in algorithm 2.3.

```
level = 0;
hierarchy[level] = hypergraph;
min_vertices = 200;
while hierarchy[level].vertex_count() > min_vertices do
    next_level = cluster(hierarchy[level]);
    level = level + 1;
    hierarchy[level] = next_level;
endw
partitioning[level] = a random initial solution for top-level hypergraph;
FM(hierarchy[level], partitioning[level]);
while level>0 do
    level = level - 1;
    partitioning[level] = project(partitioning[level+1], hierarchy[level]);
    FM(hierarchy[level], partitioning[level]);
endw
```

Algorithm 2.3 Pseudo-code for the Multilevel Partitioning Algorithm [38]

As illustrated in Fig. 2.22, multilevel partitioning consists of 3 main components: clustering, top-level partitioning and refinement or "uncoarsening". During clustering, hypergraph vertices are combined into clusters based on connectivity, leading to a smaller, clustered hypergraph. This step is repeated until obtaining only several hundred clusters and a hierarchy of clustered hypergraphs. We describe this hierarchy, as shown in Fig. 2.22, with the smaller hypergraphs being "higher" and the larger hypergraphs being "lower". The smallest (top-level) hypergraph is partitioned with a very fast initial solution generator and improved iteratively, for example, using the FM algorithm. The resulting partitioning is then interpreted as a solution for the next hypergraph in the hierarchy. During the refinement stage, solutions are projected from one level to the next and improved iteratively. Additionally, the hMETIS partitioning program [49] introduced several new heuristics that are incorporated into their multilevel partitioning implementation and are reportedly performance critical.

2.5.4 Placement

Placement algorithms determine which logic block within an FPGA should implement the corresponding logic block (instance) required by the circuit. The optimization goals consist in placing connected logic blocks close together to minimize the required wiring (wire length-driven placement), and sometimes to place blocks to balance the wiring density across the FPGA (routability-driven placement) or to maximize circuit speed (timing-driven placement). The 3 major classes of

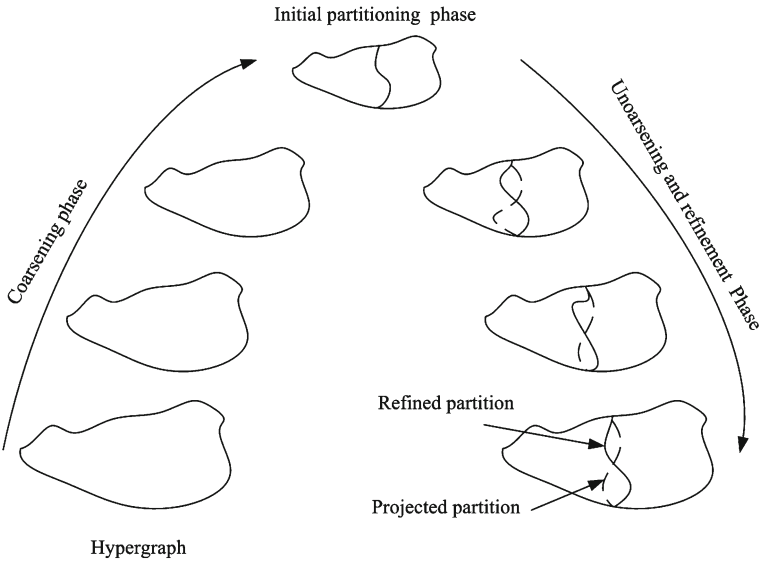


Fig. 2.22 Multilevel hypergraph bisection

placers in use today are min-cut (Partitioning-based) [6, 40], analytic [32, 53] which are often followed by local iterative improvement, and simulated annealing based placers [37, 105]. To investigate architectures fairly we must make sure that our CAD tools are attempting to use every FPGA's feature. This means that the optimization approach and goals of the placer may change from architecture to architecture. Partitioning and simulated annealing approaches are the most common and used in FPGA CAD tools. Thus we focus on both techniques in the sequel.

2.5.4.1 Simulated Annealing Based Approach

Simulated annealing mimics the annealing process used to cool gradually molten metal to produce high-quality metal objects [105]. Pseudo-code for a generic simulated annealing-based placer is shown in algorithm 2.4. A cost function is used to evaluate the quality of a given placement of logic blocks. For example, a common cost function in wirelength-driven placement is the sum over all nets of the half perimeter of their bounding boxes. An initial placement is created by assigning logic blocks randomly to the available locations in the FPGA. A large number of moves, or local improvements are then made to gradually improve the placement. A logic block is selected at random, and a new location for it is also selected randomly. The change in cost function that results from moving the selected logic block to the proposed new location is computed. If the cost decreases, the move is always accepted and the block is moved. If the cost increases, there is still a chance to accept the move, even though it makes the placement worse. This probability of acceptance is


```
S = RandomPlacement();
T = InitialTemperature();
R_limit = InitialR_limit;
while ExitCriterion() == false do
    while InnerLoopCriterion() == false do
        S_new = GenerateViaMove(S, R_limit);
        ΔC = Cost(S_new) − Cost(S);
        r = random(0,1);
        if r < e^(−ΔC/T) then
            S = S_new;
        endif
    endw
    T = UpdateTemp();
    R_limit = UpdateR_limit();
endw
```

Algorithm 2.4 Generic Simulated Annealing-based Placer [22]

given by $e^{-\frac{\Delta C}{T}}$, where ΔC is the change in cost function, and T is a parameter called temperature that controls probability of accepting moves that worsen the placement. Initially, T is high enough so almost all moves are accepted; it is gradually decreased as the placement improves, in such a way that eventually the probability of accepting a worsening move is very low. This ability to accept hill-climbing moves that make a placement worse allows simulated annealing to escape local minima of the cost function.

The R_{limit} parameter in algorithm 2.4 controls how close are together blocks must be to be considered for swapping. Initially, R_{limit} is fairly large, and swaps of blocks far apart on a chip are more likely. Throughout the annealing process, R_{limit} is adjusted to try to keep the fraction of accepted moves at any temperature close to 0.44. If the fraction of moves accepted, α, is less than 0.44, R_{limit} is reduced, while if α is greater than 0.44, R_{limit} is increased.

In [22], the objective cost function is a function of the total wirelength of the current placement. The wirelength is an estimate of the routing resources needed to completely route all nets in the netlist. Reductions in wirelength mean fewer routing wires and switches are required to route nets. This point is important because routing resources in an FPGA are limited. Fewer routing wires and switches typically are also translated into reductions of the delay incurred in routing nets between logic blocks. The total wirelength of a placement is estimated using a semi-perimeter metric, and is given by Eq. 2.2. N is the total number of nets in the netlist, $bbx(i)$ is the horizontal span of net i, $bby(i)$ is its vertical span, and $q(i)$ is a correction factor. Figure 2.23 illustrates the calculation of the horizontal and vertical spans of a hypothetical net that has 6 terminals.

$$WireCost = \sum_{i=1}^{N} q(i) \times (bb_x(i) + bb_y(i)) \tag{2.2}$$

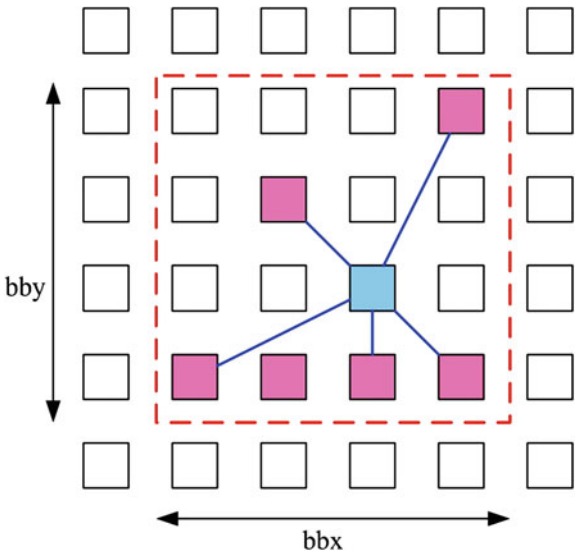


Fig. 2.23 Bounding box of a hypothetical 6-terminal net [22]

The temperature decrease rate, the exit criterion for terminating the anneal, the number of moves attempted at each temperature (InnerLoopCriterion), and the method by which potential moves are generated are defined by the annealing schedule. An efficient annealing schedule is crucial to obtain good results in a reasonable amount of CPU time. Many proposed annealing schedules are "fixed" schedules with no ability to adapt to different problems. Such schedules can work well within the narrow application range for which they are developed, but their lack of adaptability means they are not very general. In [86] authors propose an "adaptive" annealing schedule based on statistics computed during the anneal itself. Adaptive schedules are widely used to solve large scale optimization problems with many variables.

2.5.4.2 Partitioning Based Approach

Partitioning-based placement methods, are based on graph partitioning algorithms such as the Fiduccia-Mattheyses (FM) algorithm [34], and Kernighan Lin (KL) algorithm [6]. Partitioning-based placement are suitable to Tree-based FPGA architectures. The partitioner is applied recursively to each hierarchical level to distribute netlist cells between clusters. The aim is to reduce external communications and to collect highly connected cells into the same cluster.

The partitioning-based placement is also used in the case of Mesh-based FPGA. The device is divided into two parts, and a circuit partitioning algorithm is applied to determine the adequate part where a given logic block must be placed to minimize the number of cuts in the nets that connect the blocks between partitions, while leaving highly-connected blocks in one partition.

A divide-and-conquer strategy is used in these heuristics. By partitioning the problem into sub-parts, a drastic reduction in search space can be achieved. On the whole, these algorithms perform in the top-down manner, placing blocks in the general regions which they should belong to. In the Mesh FPGA case, partitioning-based placement algorithms are good from a "global" perspective, but they do not actually attempt to minimize wirelength. Therefore, the solutions obtained are sub-optimal in terms of wirelength. However, these classes of algorithms run very fast. They are normally used in conjunction with other search techniques for further quality improvement. Some algorithms [130] and [95] combine multi-level clustering and hierarchical simulated annealing to obtain ultra-fast placement with good quality. In the following chapters, the partitioning-based placement approach will be used only for Tree-based FPGA architectures.

2.5.5 Routing

The FPGA routing problem consists in assigning nets to routing resources such that no routing resource is shared by more than one net. $Pathfinder$ [80] is the current, state-of-the-art FPGA routing algorithm. $Pathfinder$ operates on a directed graph abstraction $G(V, E)$ of the routing resources in an FPGA. The set of vertices V in the graph represents the IO terminals of logic blocks and the routing wires in the interconnect structure. An edge between two vertices represents a potential connection between them. Figure 2.24 presents a part of a routing graph in a Mesh-based interconnect.

Given this graph abstraction, the routing problem for a given net is to find a directed tree embedded in G that connects the source terminal of the net to each of its sink terminals. Since the number of routing resources in an FPGA is limited, the goal of finding unique, non-intersecting trees for all the nets in a netlist is a difficult problem.

$Pathfinder$ uses an iterative, negotiation-based approach to successfully route all the nets in a netlist. During the first routing iteration, nets are freely routed without paying attention to resource sharing. Individual nets are routed using $Dijkstra$'s shortest path algorithm [111]. At the end of the first iteration, resources may be congested because multiple nets have used them. During subsequent iterations, the cost of using a resource is increased, based on the number of nets that share the resource, and the history of congestion on that resource. Thus, nets are made to negotiate for routing resources. If a resource is highly congested, nets which can use lower congestion alternatives are forced to do so. On the other hand, if the alternatives are more congested than the resource, then a net may still use that resource.

The cost of using a routing resource n during a routing iteration is given by Eq. 2.3.

$$c_n = (b_n + h_n) \times p_n \tag{2.3}$$

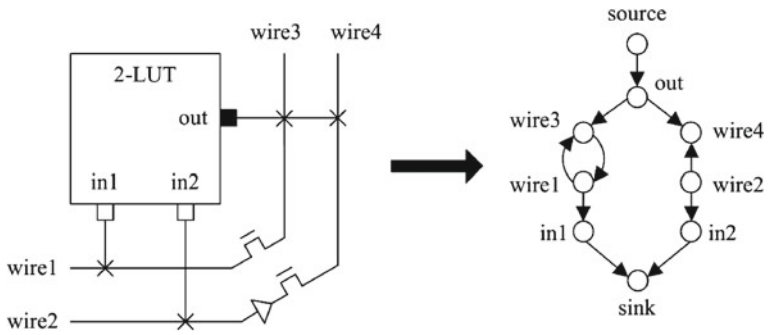


Fig. 2.24 Modeling FPGA architecture as a directed graph [22]

b_n is the base cost of using the resource n, h_n is related to the history of congestion during previous iterations, and p_n is proportional to the number of nets sharing the resource in the current iteration. The p_n term represents the cost of using a shared resource n, and the h_n term represents the cost of using a resource that has been shared during earlier routing iterations. The latter term is based on the intuition that a historically congested node should appear expensive, even if it is slightly shared currently. Cost functions and routing schedule were described in details in [22]. The Pseudo-code of the $Pathfinder$ routing algorithm is presented in algorithm 2.5.

```
Let: RT_i be the set of nodes in the current routing of net i
while shared resources exist do
    /*Illegal routing*/
    foreach net, i do
        rip-up routing tree RT_i;
        RT(i) = s_i;
        foreach sink t_ij do
            Initialize priority queue PQ to RT_i at cost 0;
            while sink t_ij not found do
                Remove lowest cost node m from PQ;
                foreach fanout node n of node m do
                    Add n to PQ at PathCost(n) = c_n + PathCost(m);
                endfch
            endw
            foreach node n in path t_ij to s_i do
                /*backtrace*/
                Update c_n;
                Add n to RT_i;
            endfch
        endfch
    endfch
    update h_n for all n;
endw
```

Algorithm 2.5 Pseudo-code of the $Pathfinder$ Routing Algorithm [80]

An important measure of routing quality produced by an FPGA routing algorithm is the critical path delay. The critical path delay of a routed netlist is the maximum delay of any combinational path in the netlist. The maximum frequency at which a netlist can be clocked has an inverse relationship with critical path delay. Thus, larger critical path delays slow down the operation of netlist. Delay information is incorporated into $Pathfinder$ by redefining the cost of using a resource n (Eq. 2.4).

$$c_n = A_{ij} \times d_n + (1 - A_{ij}) \times (b_n + h_n) \times p_n \tag{2.4}$$

The c_n term is from Eq. 2.3, d_n is the delay incurred in using the resource, and A_{ij} is the criticality given by Eq. 2.5.

$$A_{ij} = \frac{D_{ij}}{D_{max}} \tag{2.5}$$

D_{ij} is the maximum delay of any combinational path going through the source and sink terminals of the net being routed, and D_{max} is the critical path delay of the netlist. Equation 2.4 is formulated as a sum of two cost terms. The first term in the equation represents the delay cost of using resource n, while the second term represents the congestion cost. When a net is routed, the value of A_{ij} determines whether the delay or the congestion cost of a resource dominates. If a net is near critical (i.e. its A_{ij} is close to 1), then congestion is largely ignored and the cost of using a resource is primarily determined by the delay term. If the criticality of a net is low, the congestion term in Eq. 2.4 dominates, and the route found for the net avoids congestion while potentially incurring delay.

$Pathfinder$ has proved to be one of the most powerful FPGA routing algorithms to date. The negotiation-based framework that trades off delay for congestion is an extremely effective technique for routing signals on FPGAs. More importantly, $Pathfinder$ is a truly architecture-adaptive routing algorithm. The algorithm operates on a directed graph abstraction of an FPGA's routing structure, and can thus be used to route netlists on any FPGA that can be represented as a directed routing graph.

2.5.6 Timing Analysis

Timing analysis [99] is used for two basic purposes:

- To determine the speed of circuits which have been completely placed and routed,
- To estimate the slack [68] of each source-sink connection during routing (placement and other parts of the CAD flow) in order to decide which connections must be made via fast paths to avoid slowing down the circuit.

First the circuit under consideration is presented as a directed graph. Nodes in the graph represent input and output pins of circuit elements such as LUTs, registers,

and I/O pads. Connections between these nodes are modeled with edges in the graph. Edges are added between the inputs of combinational logic Blocks (LUTs) and their outputs. These edges are annotated with a delay corresponding to the physical delay between the nodes. Register input pins are not joined to register output pins. To determine the delay of the circuit, a breadth first traversal is performed on the graph starting at sources (input pads, and register outputs). Then the arrival time, $T_{arrival}$, at all nodes in the circuit is computed with the following equation:

$$T_{arrival}(i) = \max_{j \in fanin(i)} \{T_{arrival}(j) + delay(j, i)\}$$

where node i is the node currently being computed, and $delay(j, i)$ is the delay value of the edge joining node j to node i. The delay of the circuit is then the maximum arrival time, D_{max}, of all nodes in the circuit.
To guide a placement or routing algorithm, it is useful to know how much delay may be added to a connection before the path that the connection is on becomes critical. The amount of delay that may be added to a connection before it becomes critical is called the slack of that connection. To compute the slack of a connection, one must compute the required arrival time, $T_{required}$, at every node in the circuit. We first set the $T_{required}$ at all sinks (output pads and register inputs) to be D_{max}. Required arrival time is then propagated backwards starting from the sinks with the following equation:

$$T_{required}(i) = \min_{j \in fanout(i)} \{T_{required}(j) - delay(j, i)\}$$

Finally, the slack of a connection (i, j) driving node, j, is defined as:

$$Slack(i, j) = T_{required}(j) - T_{arrival}(i) - delay(i, j)$$

2.5.7 Bitstream Generation

Once a netlist is placed and routed on an FPGA, bitstream information is generated for the netlist. This bitstream is programmed on the FPGA using a bitstream loader. The bitstream of a netlist contains information as to which SRAM bit of an FPGA be programmed to 0 or to 1. The bitstream generator reads the technology mapping, packing and placement information to program the SRAM bits of Look-Up Tables. The routing information of a netlist is used to correctly program the SRAM bits of connection boxes and switch boxes.

2.6 Research Trends in Reconfigurable Architectures

Until now in this chapter a detailed overview of logic architecture, routing architecture and software flow of FPGAs is presented. In this section, we highlight some of the disadvantages associated with FPGAs and further we describe some of the trends that

are currently being followed to remedy these disadvantages. FPGA-based products are basically very effective for low to medium volume production as they are easy to program and debug, and have less NRE cost and faster time-to-market. All these major advantages of an FPGA come through their reconfigurability which makes them general purpose and field programmable. But, the very same reconfigurability is the major cause of its disadvantages; thus making it larger, slower and more power consuming than ASICs.

However, the continued scaling of CMOS and increased integration has resulted in a number of alternative architectures for FPGAs. These architectures are mainly aimed to improve area, performance and power consumption of FPGA architectures. Some of these propositions are discussed in this section.

2.6.1 Heterogeneous FPGA Architectures

Use of hard-blocks in FPGAs improves their logic density. Hard-Blocks, in FPGAs increase their density, performance and power consumption. There can be different types of hard-blocks like multipliers, adders, memories, floating point units and DSP blocks etc. In this regard, [19] have incorporated embedded floating-point units in FPGAs, [30] have developed virtual embedded block methodology to model arbitrary embedded blocks on existing commercial FPGAs. Here some of the academic and commercial architectures are presented that make use of hard-blocks to improve overall efficiency of FPGAs.

2.6.1.1 Versatile Packing, Placement and Routing VPR

Versatile Packing, Placement and Routing for FPGAs (commonly known as VPR) [14, 22, 120] is the most widely used academic mesh-based FPGA exploration environment. It allows to explore mesh-based FPGA architectures by employing an empirical approach. Benchmark circuits are mapped, placed and routed on a desired FPGA architecture. Later, area and delay of FPGAs are measured to decide best architectural parameters. Different CAD tools in VPR are highly optimized to ensure high quality results.

Earlier version of VPR supported only homogeneous achitectures [120]. However, the latest version of VPR known as VPR 5.0 [81] supports hard-blocks (such as multiplier and memory blocks) and single-driver routing wires. Hard-blocks are restricted to be in one grid width column, and that column can be composed of only similar type of blocks. The height of a hard-block is quantized and it must be an integral multiple of grid units. In case a block height is indivisible with the height of FPGA, some grid locations are left empty. Figure 2.25 illustrates a heterogeneous FPGA with 8 different kinds of blocks.

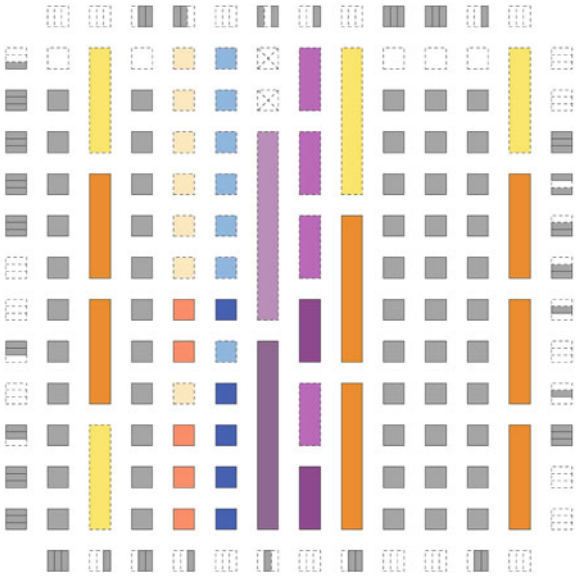


Fig. 2.25 A heterogeneous FPGA in VPR 5.0 [81]

2.6.1.2 Madeo, a Framework for Exploring Reconfigurable Architectures

Madeo [73] is another academic design suite for the exploration of reconfigurable architectures. It includes a modeling environment that supports multi-grained, heterogeneous architectures with irregular topologies. Madeo framework initially allows to model an FPGA architecture. The architecture characteristics are represented as a common abstract model. Once the architecture is defined, the CAD tools of Madeo are used to map a target netlist on the architecture. Madeo uses same placement and routing algorithms as used by VPR [120]. Along with placement and routing algorithms, it also embeds a bitstream generator, a netlist simulator, and a physical layout generator in its design suite. Madeo supports architectural prospection and very fast FPGA prototyping. Several FPGAs, including some commercial architectures (such as Xilinx Virtex family) and prospective ones (such as STMicro LPPGA) have been modeled using Madeo. The physical layout is produced as VHDL description.

2.6.1.3 Altera Architecture

Altera's Stratix IV [107] is an example of a commercial architecture that uses a heterogeneous mixture of blocks. Figure 2.26 shows the global architectural layout of Stratix IV. The logic structure of Stratix IV consists of LABs (Logic Array Blocks), memory blocks and digital signal processing (DSP) blocks. LABS are distributed symmetrically in rows and columns and are used to implement general purpose logic. The DSP blocks are used to implement full-precision multipliers of different

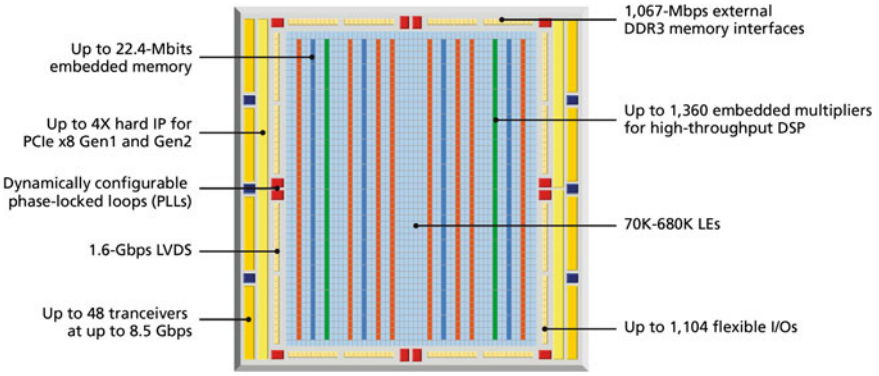


Fig. 2.26 Stratix IV architectural elements

granularities. The memory blocks and DSP blocks are placed in columns at equal distance with one another. Input and Output (I/Os) are located at the periphery of architecture.

Logic array blocks (LABs) and adaptive logic modules (ALMs) provide the basic logic capacity for Stratix IV device. They can be used to configure logic functions, arithmetic functions, and register functions. Each LAB consists of ten ALMs, carry chains, arithmetic chains, LAB control signals, local interconnect, and register chain connection lines. The local interconnect connects the ALMs that are inside same LAB. The direct link allows a LAB to drive into the local interconnect of its left or right neighboring LAB. The register chain connects the output of ALM register to the adjacent ALM register in the LAB. A memory LAB (MLAB) is a derivative of LAB which can be either used just like a simple LAB, or as a static random access memory (SRAM). Each ALM in an MLAB can be configured as a 64×1, or 32×2 blocks, resulting in a configuration of 64×10 or 32×20 simple dual-port SRAM block. MLAB and LAB blocks always coexist as pairs in Stratix IV families.

The DSP blocks in Stratix IV are optimized for signal processing applications such as Finite Impulse Response (FIR), Infinite Impulse Response (IIR), Fast Fourier Transform functions (FFT) and encoders etc. Stratix IV device has two to seven columns of DSP blocks that can implement different operations like multiplication, multiply-add, multiply-accumulate (MAC) and dynamic arithmetic or logical shift functions. The DSP block supports different multiplication operations such as 9×9, 12×12, 18×18 and 36×36 multiplication operations. The Stratix IV devices contain three different sizes of embedded SRAMs. The memory sizes include 640-bit memory logic array blocks (MLABs), 9-Kbit M9K blocks, and 144-Kbit M144K blocks. The MLABs have been optimized to implement filter delay lines, small FIFO buffers, and shift registers. M9K blocks can be used for general purpose memory applications, and M144K are generally meant to store code for a processor, packet buffering or video frame buffering.

2.6.2 FPGAs to Structured Architectures

The ease of designing and prototyping with FPGAs can be exploited to quickly design a hardware application on an FPGA. Later, improvements in area, speed, power and volume production can be achieved by migrating the application design from FPGA to other technologies such as Structured-ASICs. In this regard, Altera provides a facility to migrate its Stratix IV based application design to HardCopy IV [56]. Altera gives provision to migrate FPGA-based applications to Structured-ASIC. Their Structured-ASIC is called as HardCopy [56]. The main theme is to design, test and even initially ship a design using an FPGA. Later, the application circuit that is mapped on the FPGA can be seamlessly migrated to HardCopy for high volume production. Their latest HardCopy-IV devices offer pin-to-pin compatibility with the Stratix IV prototype, making them exact replacements for the FPGAs. Thus, the same system board and softwares developed for prototyping and field trials can be retained, enabling the lowest risk and fastest time-to-market for high-volume production. Moreover, when an application circuit is migrated from Stratix IV FPGA prototype to Hardcopy-VI, the core logic performance doubles and power consumption reduces by half.

The basic logic unit of HardCopy is termed as HCell. It is similar to Stratix IV logic cell (LAB) in the sense that the fabric consists of a regular pattern which is formed by tiling one or more basic cells in a two dimensional array. However, the difference is that HCell has no configuration memory. Different HCell candidates can be used, ranging from fine-grained NAND gates to multiplexors and coarse-grained LUTs. An array of such HCells, and a general purpose routing network which interconnects them is laid down on the lower layers of the chip. Specific layers are then reserved to form via connections or metal lines which are used to customize the generic array into specific functionality. Figure 2.27 illustrates the correspondence between an FPGA and a compatible structured ASIC. There is a one to one layout-level correspondence between MRAMs, phase-lock loops (PLLs), embedded memories, transceivers, and I/O blocks. The soft-logic DSP multipliers and logic cell fabric of the FPGA are re-synthesized to structured ASIC fabric. However, they remain functionally and electrically equivalent in FPGAs and HardCopy ASICs.

Apart from Altera, there are several other companies that provide a solution similar to that of Altera. For example, the eASIC Nextreme [41] uses an FPGA-like design flow to map an application design on SRAM programmable LUTs, which are later interconnected through mask programming of few upper routing layers. Tierlogic [113] is a recently launched FPGA vendor that offers 3D SRAM-based TierFPGA devices for prototyping and early production. The same design solution can be frozen to a TierASIC device with one low-NRE custom mask for error-free transition to an ASIC implementation. The SRAM layer is placed on an upper 3D layer of TierFPGA. Once the TierFPGA design is frozen, the bitstream information is used to create a single custom mask metal layer that will replace the SRAM programming layer.

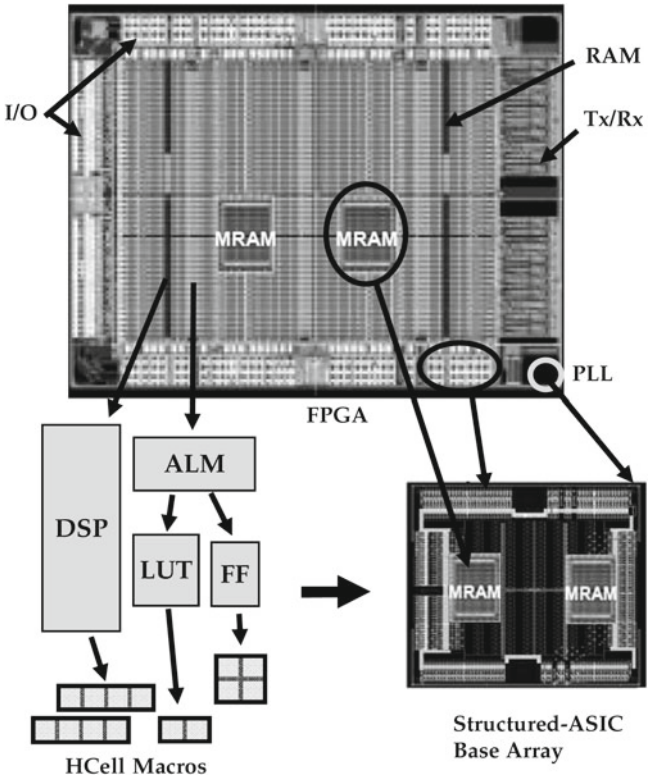


Fig. 2.27 FPGA/Structured-ASIC (HardCopy) Correspondence [59]

2.6.3 Configurable ASIC Cores

Configurable ASIC Core (cASIC) [35] is another example of reconfigurable devices that can implement a limited set of circuits which operate at mutually exclusive times. cASICs are intended as accelerator in domain-specific systems-on-a-chip, and are not designed to replace the entire ASIC-only chip. The host would execute software code, whereas compute-intensive sections can be executed on one or more cASICs. So, to execute the compute intensive sections, cASICs implement only data-path circuits and thus supports full-word blocks only (such as 16-bit wide multipliers, adders, RAMS, etc). Since the application domain of cASICs is more specific, they are significantly smaller than FPGAs. As hardware resources are shared between different netlists, cASICs are even smaller than the sum of the standard-cell based ASIC areas of individual circuits.

2.6.4 Processors Inside FPGAs

Considerable amount of FPGA area can be reduced by incorporating a microprocessor in an FPGA. A microprocessor can execute any less compute intensive task, whereas compute-intensive tasks can be executed on an FPGA. Similarly, a microprocessor based application can have huge speed-up gains if an FPGA is attached with it. An FPGA attached with a microprocessor can execute any compute intensive functionality as a customized hardware instruction. These advantages have compelled commercial FPGA vendors to provide microprocessor in their FPGAs so that complete system can be programmed on a single chip. Few vendors have integrated fixed hard processor on their FPGA (like AVR Processor integrated in Atmel FPSLIC [18] or PowerPC processors embedded in Xilinx Virtex-4 [126]). Others provide soft processor cores which are highly optimized to be mapped on the programmable resources of FPGA. Altera's Nios [90] and Xilinx's Microblaze [88] are soft processor meant for FPGA designs which allow custom hardware instructions. [96] have shown that considerable area gains can be achieved if these soft processors for FPGAs are optimized for particular applications. They have shown that unused instructions in a soft processor can be removed and different architectural tradeoffs can be selected to achieve on average 25% area gain for soft processors required for specific applications. Reconfigurable units can also be attached with microprocessors to achieve execution time speedup in software programs. [28, 70, 104] have incorporated a reconfigurable unit with microprocessors to achieve execution-time speedup.

2.6.5 Application Specific FPGAs

The type of logic blocks and the routing network in an FPGA can be optimized to gain area and performance advantages for a given application domain (controlpath-oriented applications, datapath-oriented applications, etc). These types of FPGAs may include different variety of desired hard-blocks, appropriate amount of flexibility required for the given application domain or bus-based interconnects rather than bit-based interconnects. Authors in [83] have presented a reconfigurable arithmetic array for multimedia applications which they call as CHESS. The principal goal of CHESS was to increase arithmetic computational density, to enhance the flexibility, and to increase the bandwidth and capacity of internal memories significantly beyond the capabilities of existing commercial FPGAs. These goals were achieved by proposing an array of ALUs with embedded RAMs where each ALU is 4-bit wide and supports 16 instructions. Similarly, authors in [42] present a coarse-grained, field programmable architecture for constructing deep computational pipelines. This architecture can efficiently implement applications related to media, signal processing, scientific computing and communications. Further, authors in [128] have used bus-based routing and logic blocks to improve density of FPGAs

for datapath circuits. This is a partial multi-bit FPGA architecture that is designed to exploit the regularity that most of the datapath circuits exhibit.

2.6.6 Time-Multiplexed FPGAs

Time-multiplexed FPGAs increase the capacity of FPGAs by executing different portions of a circuit in a time-multiplexed mode [89, 114]. An application design is divided into different sub-circuits, and each sub-circuit runs as an individual context of FPGA. The state information of each sub-circuit is saved in context registers before a new context runs on FPGA. Authors in [114] have proposed a time-multiplexed FPGA architecture where a large circuit is divided into sub-circuits and each sub-circuit is sequentially executed on a time-multiplexed FPGA. Such an FPGA stores a set of configuration bits for all contexts. A context is shifted simply by using the SRAM bits dedicated to a particular context. The combinatorial and sequential outputs of a sub-circuit that are required by other sub-circuits are saved in context registers which can be easily accessed by sub-circuits at different times.

Time-Multiplexed FPGAs increase their capacity by actually adding more SRAM bits rather than more CLBs. These FPGAs increase the logic capacity by dynamically reusing the hardware. The configuration bits of only the currently executing context are active, the configuration bits for the remaining supported contexts are inactive. Intermediate results are saved and then shared with the contexts still to be run. Each context takes a micro-cycle time to execute one context. The sum of the micro-cycles of all the contexts makes one user-cycle. The entire time-multiplexed FPGA or its smaller portion can be configured to (i) execute a single design, where each context runs a sub-design, (ii) execute multiple designs in time-multiplexed modes, or (iii) execute statically only one single design. Tabula [109] is a recently launched FPGA vendor that provides time-multiplexed FPGAs. It dynamically reconfigures logic, memory, and interconnect at multi-GHz rates with a Spacetime compiler.

2.6.7 Asynchronous FPGA Architecture

Another alternative approach that has been proposed to improve the overall performance of FPGA architecture is the use of asynchronous design elements. Conventionally, digital circuits are designed for synchronous operation and in turn FPGA architectures have focused primarily on implementing synchronous circuits. Asynchronous designs are proposed to improve the energy efficiency of asynchronous FPGAs since asynchronous designs offer potentially lower energy as energy is consumed only when necessary. Also the asynchronous architectures can simplify the design process as complex clock distribution networks become unnecessary.

The first asynchronous FPGA was developed by [57]. It consisted the modified version of previously developed synchronous FPGA architecture. Its logic block was

similar to the conventional logic block with added features of fast feedback and a latch that could be used to initialize an asynchronous circuit. Another asynchronous architecture was proposed in [112]. This architecture is designed specifically for dataflow applications. Its logic block is similar to that of synchronous architecture, along with it consists of units such as split unit which enables conditional forwarding of data and a merge unit that allows for conditional selection of data from different sources. An alternative to fully asynchronous design is a globally asynchronous, locally synchronous approach (GALS). This approach is used by [69] where authors have introduced a level of hierarchy into the FPGA architecture. Standard hard or soft synchronous logic blocks are grouped together to form large synchronous blocks and communication between these blocks is done asynchronously. More recently, authors in [131] have applied the GALS approach on Network on Chip architectures to improve the performance, energy consumption and the yield of future architectures in a synergistic manner.

It is clear that, despite each architecture offering its own benefits, a number of architectural questions remain unresolved for asynchronous FPGAs. Many architectures rely on logic blocks similar to those used for synchronous designs [57, 69] and, therefore, the same architectural issues such as LUT size, cluster size, and routing topology must be investigated. In addition to those questions, asynchronous FPGAs also add the challenge of determining the appropriate synchronization methodology.

2.7 Summary and Conclusion

In this chapter initially a brief introduction of traditional logic and routing architectures of FPGAs is presented. Later, different steps involved in the FPGA design flow are detailed. Finally various approaches that have been employed to reduce few disadvantages of FPGAs and ASICs, with or without compromising their major benefits are described. Figure 2.28 presents a rough comparison of different solutions used to reduce the drawbacks of FPGAs and ASICs. The remaining chapters of this book will focus on the exploration of tree-based FPGA architectures using hard-blocks, tree-based application specific Inflexible FPGAs (ASIF), and their automatic layout generation methods.

This book presents new environment for the exploration of tree-based heterogeneous FPGAs. This environment is used to explore different architecture techniques for tree-based heterogeneous FPGA architecture. This book also presents an optimized environment for mesh-based heterogeneous FPGA. Further, the environments of two architectures are evaluated through the experimental results that are obtained by mapping a number of heterogeneous benchmarks on the two architectures.

Altera [11] has proposed a new idea to prototype, test, and even ship initial few designs on an FPGA, later the FPGA based design can be migrated to Structured-ASIC (known as HardCopy). However, migration of an FPGA-based product to Structured-ASIC supports only a single application design. An ASIF retains this

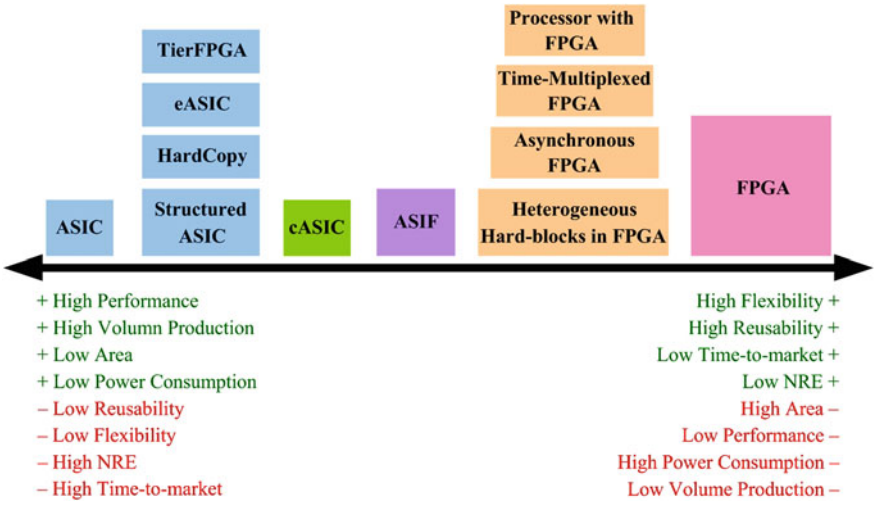


Fig. 2.28 Comparison of different solutions used to reduce ASIC and FPGA drawbacks

property, and can be a possible future extension for the migration of FPGA-based applications to Structured-ASIC. Thus when an FPGA-based product is in the final phase of its development cycle, and if the set of circuits to be mapped on the FPGA are known, the FPGA can be reduced to an ASIF for the given set of application designs. This book presents a new tree-based ASIF and a detailed comparison of tree-based ASIF is performed with mesh-based ASIF. This book also presents automatic layout generation techniques for domain-specific FPGA and ASIF architectures.

http://www.springer.com/978-1-4614-3593-8

Tree-based Heterogeneous FPGA Architectures
Application Specific Exploration and Optimization
Farooq, U.; Marrakchi, Z.; Mehrez, H.
2012, XVI, 188 p., Hardcover
ISBN: 978-1-4614-3593-8

Altera FLEX 8000 Block Diagram

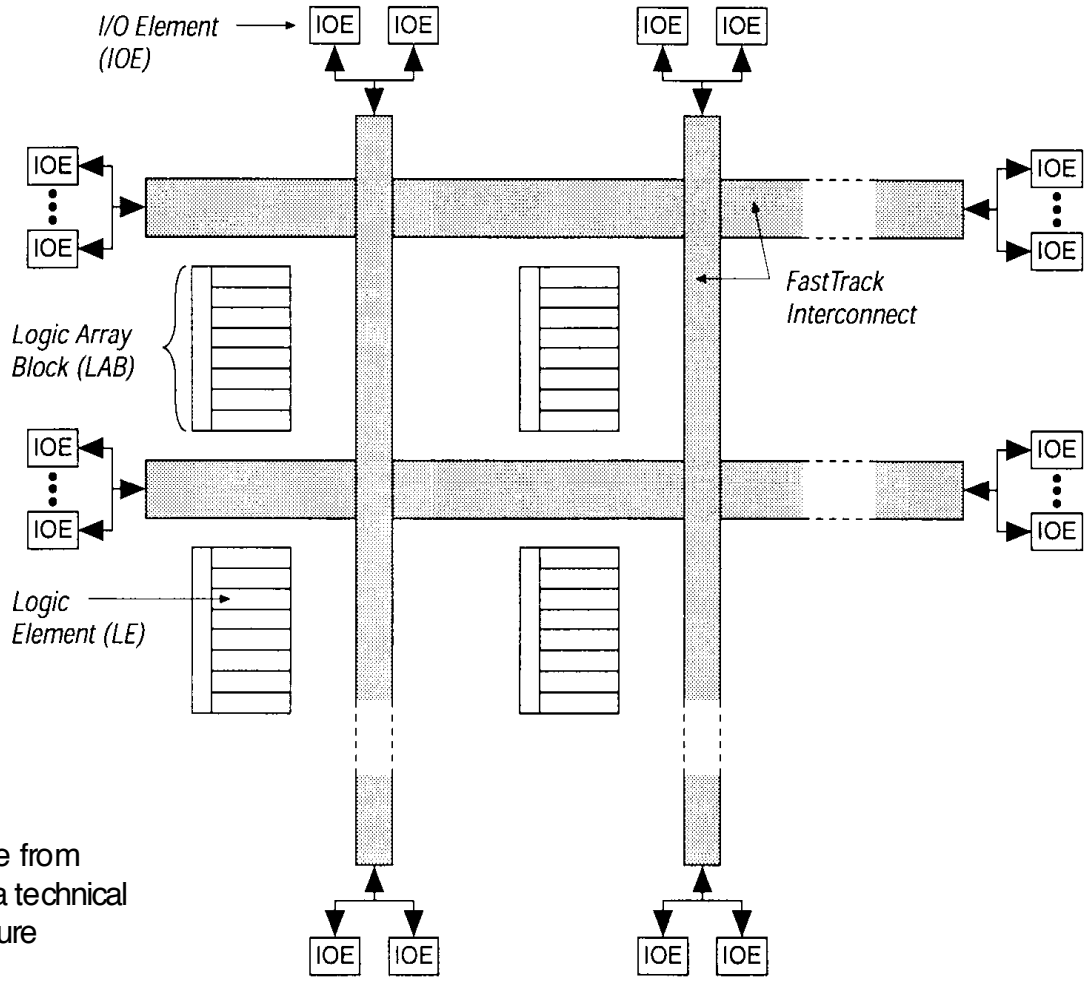


Figure from Altera technical literature

- FLEX 8000 chip contains 26–162 LABs
  - Each LAB contains 8 Logic Elements (LEs), so a chip contains 208–1296 LEs, totaling 2,500–16,000 usable gates
  - LABs arranged in rows and columns, connected by FastTrack Interconnect, with I/O elements (IOEs) at the edges

Altera FLEX 8000 Logic Array Block

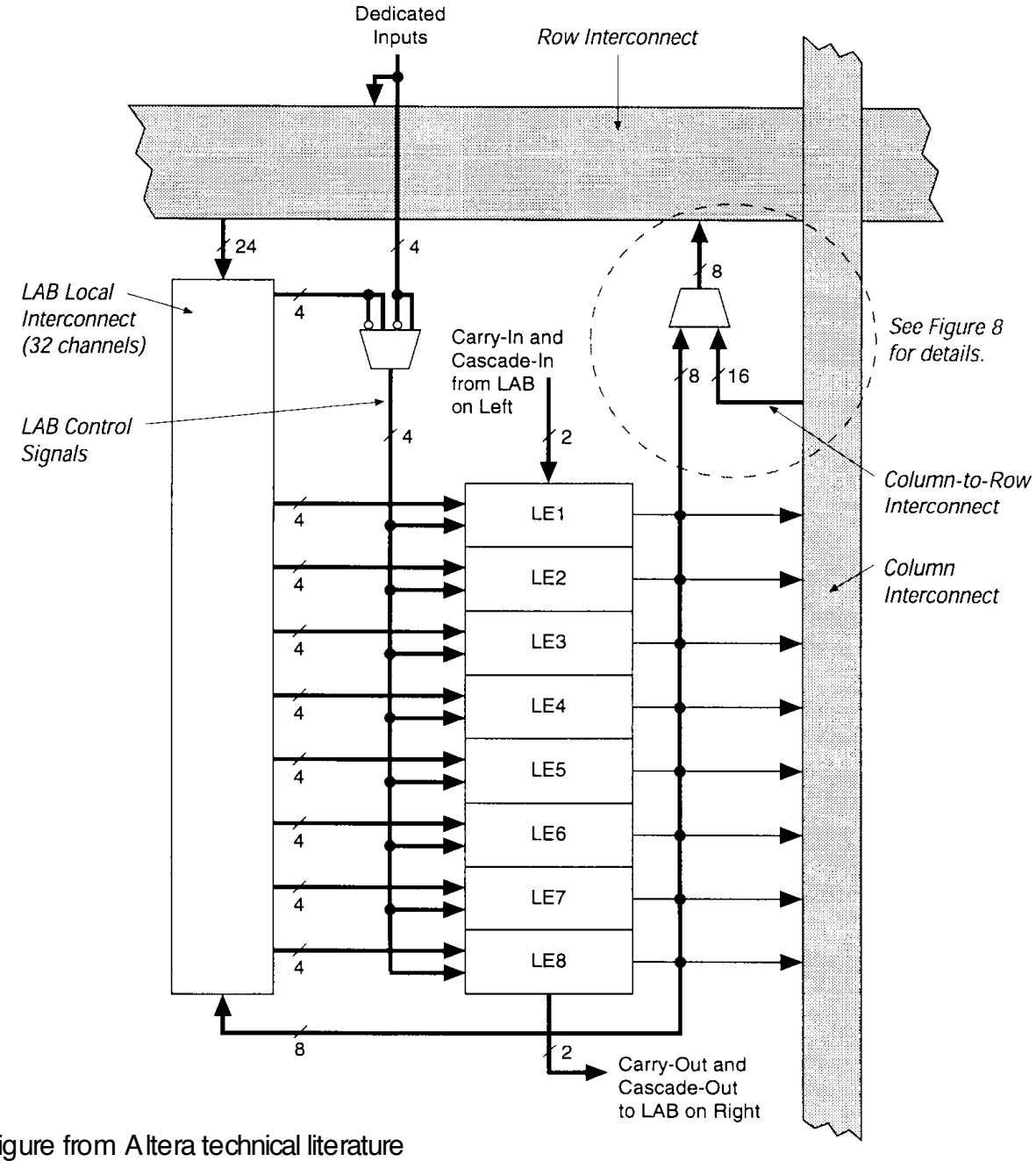


Figure from Altera technical literature

- LAB = 8 LEs, plus local interconnect, control signals, carry & cascade chains

Altera FLEX 8000 Logic Element

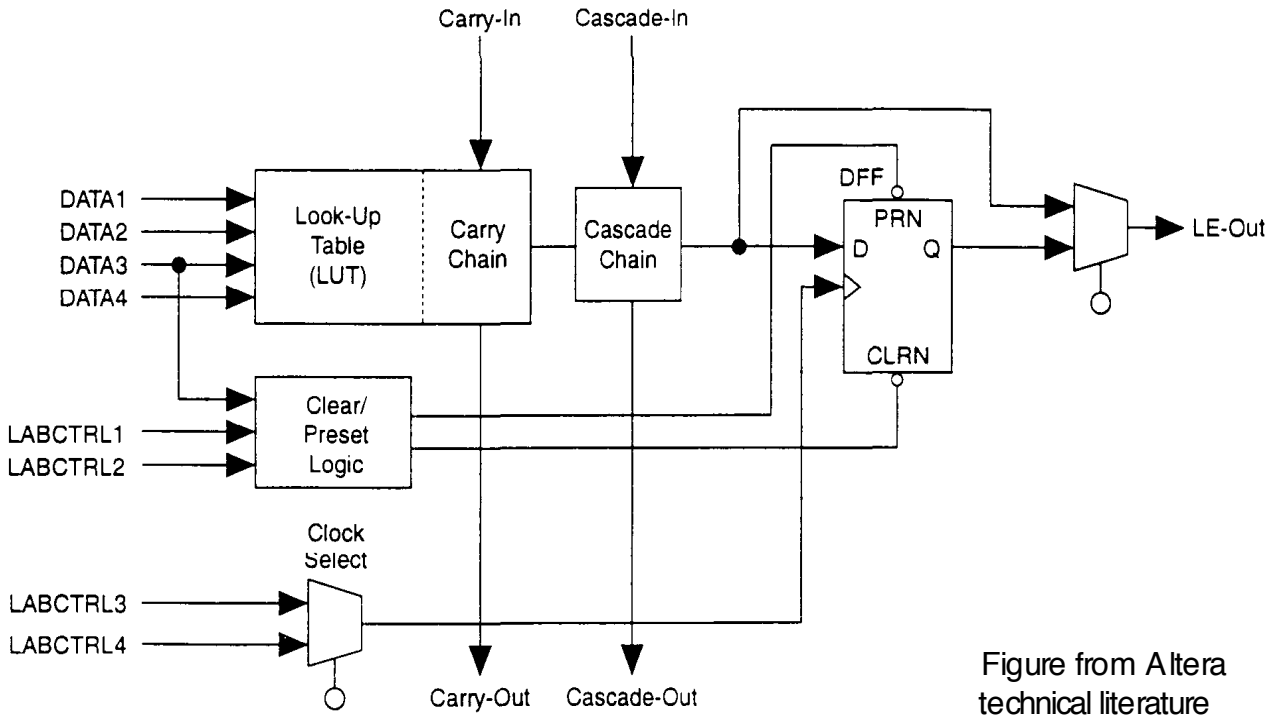


Figure from Altera technical literature

- Each Logic Element (LE) contains:
  - 4-input Look-Up Table (LUT)
    - Can produce any function of 4 variables
  - Programmable flip-flop
    - Can configure as D, T, JR, SR, or bypass
    - Has clock, clear, and preset signals that can come from dedicated inputs, I/O pins, or other LEs
  - Carry chain & cascade chain

Altera FLEX 8000 Carry Chain (Example: n-bit adder)

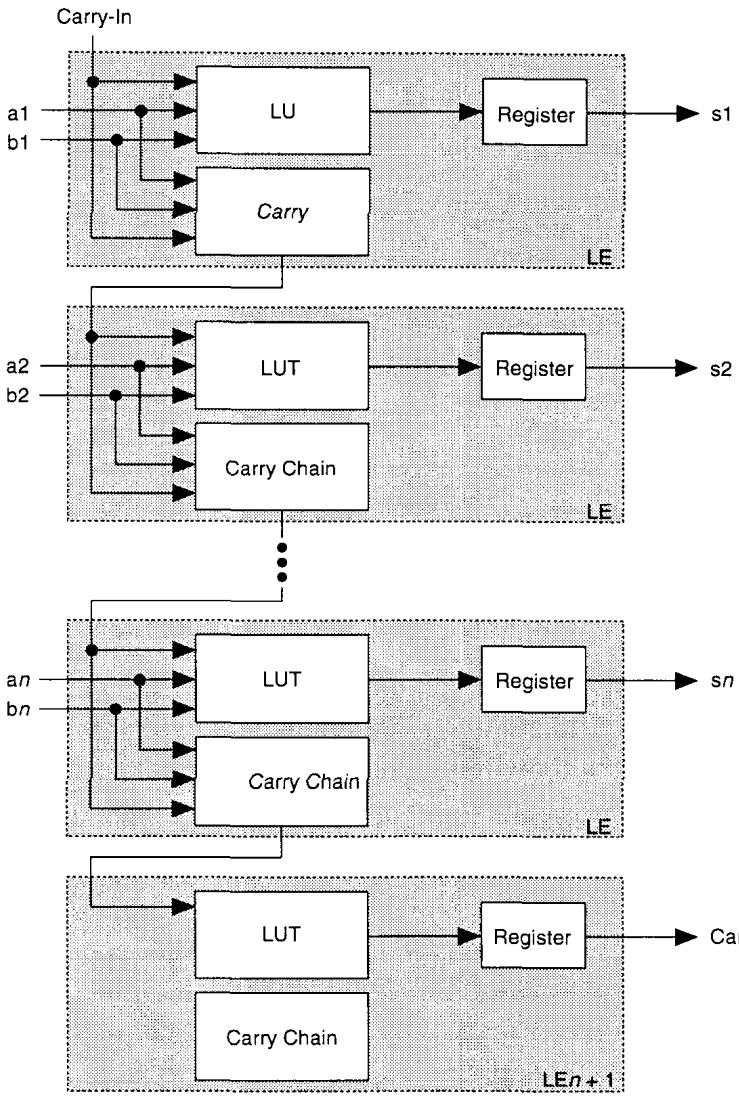


Figure from Altera technical literature

- *Carry chain* provides very fast (< 1ns) carry-forward between LEs
  - Feeds both LUT and next part of chain
  - Good for high-speed adders & counters

Altera FLEX 8000 Cascade Chain

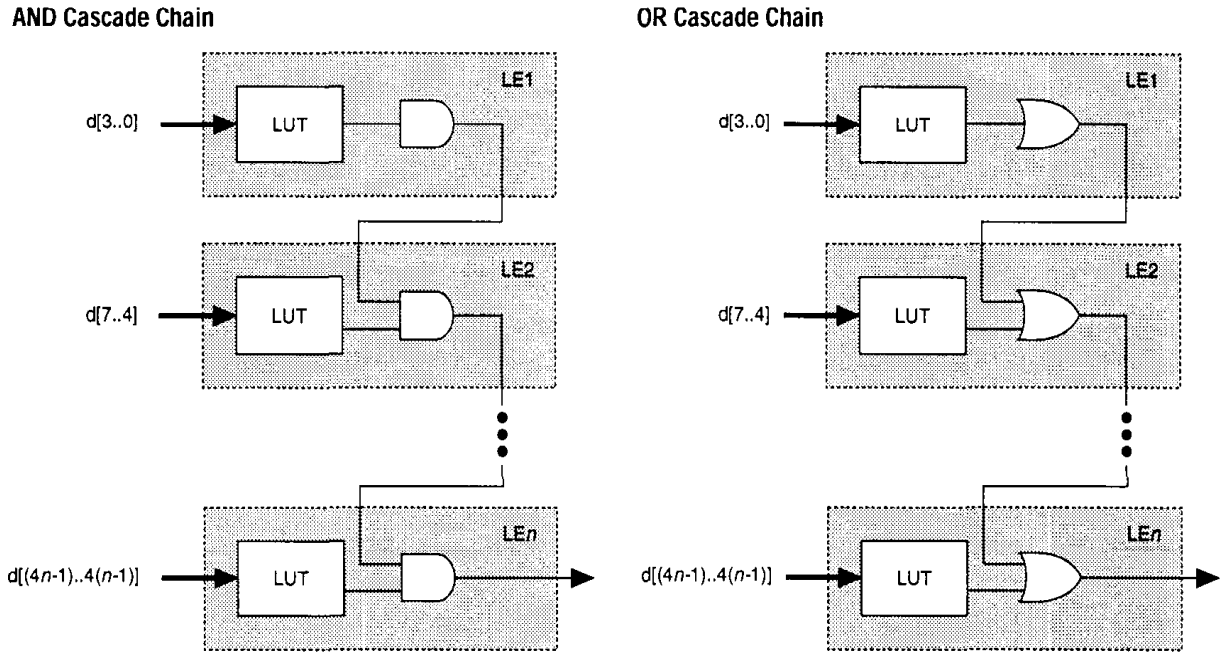
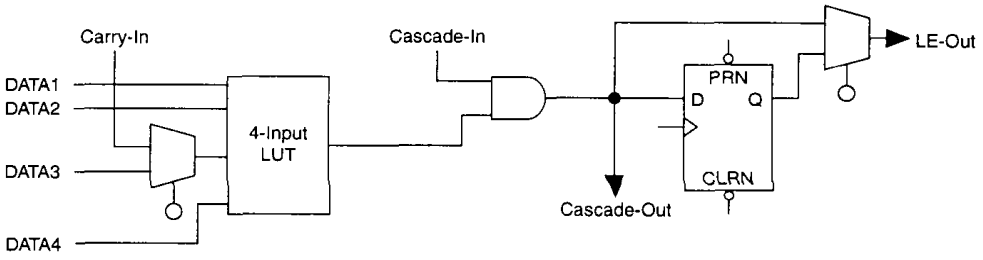


Figure from Altera technical literature

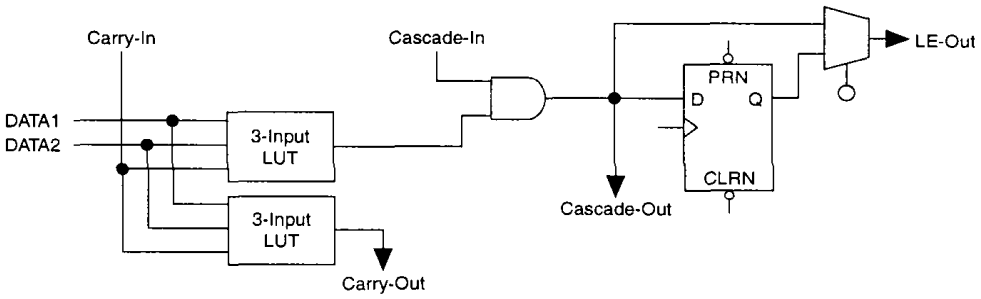
- *Cascade chain* provides wide fan-in
  - Adjacent LE's LUTs can compute parts of the function in parallel; cascade chain then serially connects intermediate values
  - Can use either a logical AND or a logical OR (using DeMorgan's theorem) to connect outputs of adjacent LEs
  - Each additional LE provides 4 more inputs to the width of the function

Altera FLEX 8000 LE Operating Modes

Normal



Arithmetic



Up/down Counter

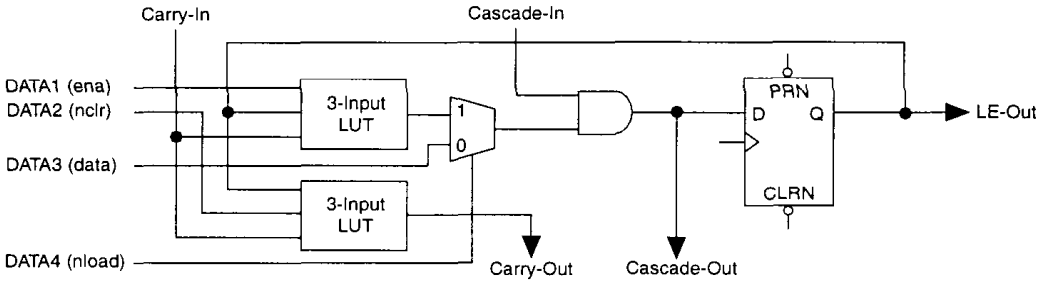


Figure from Altera technical literature

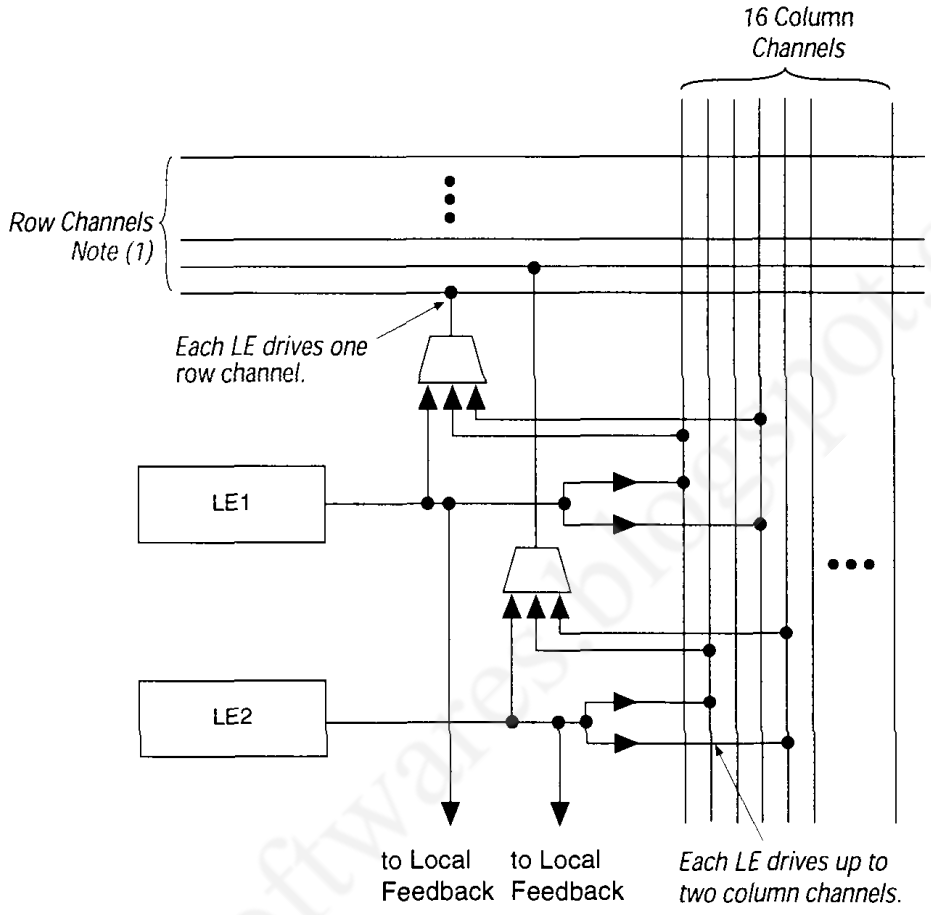
- Each mode uses LE resources differently
  - 7 out of 10 inputs (4 data from LAB local interconnect, feedback from register, and carry-in & cascade-in) go to specific destinations to implement the function
  - Remaining 3 provide clock, clear, and preset for register

Altera FLEX 8000 Operating Modes (cont.)

- Normal mode
  - Used for general logic applications, and wide decoding functions that can benefit from the cascade chain
- Arithmetic mode
  - Provides two 3-input LUTs to implement adders, accumulators, and comparators
    - One LUT provides a 3-bit function
    - Other LUT generates a carry bit
- Up/down counter mode
  - Provides counter enable, synchronous up / down control, and data loading options
  - Uses two 3-input LUTs
    - One LUT generates counter data
    - Other LUT generates fast carry bit
    - Use 2-to-1 multiplexer for synchronous data loading, clear and preset for asynchronous data loading

Altera FLEX 8000
FastTrack Interconnect

Figure from Altera technical literature



Note:
(1) See Table 4 for the number of row channels.

- Device-wide rows and columns
  - Each LE in LAB drives 2 column (total 16) channels, which connects… that column
  - Each LE in LAB drives 1 row channel, which connects to other LABs in that row
    - 3-to-1 muxs connect either LE outputs or column channels to row channels

Altera FLEX 8000 I/O Elements

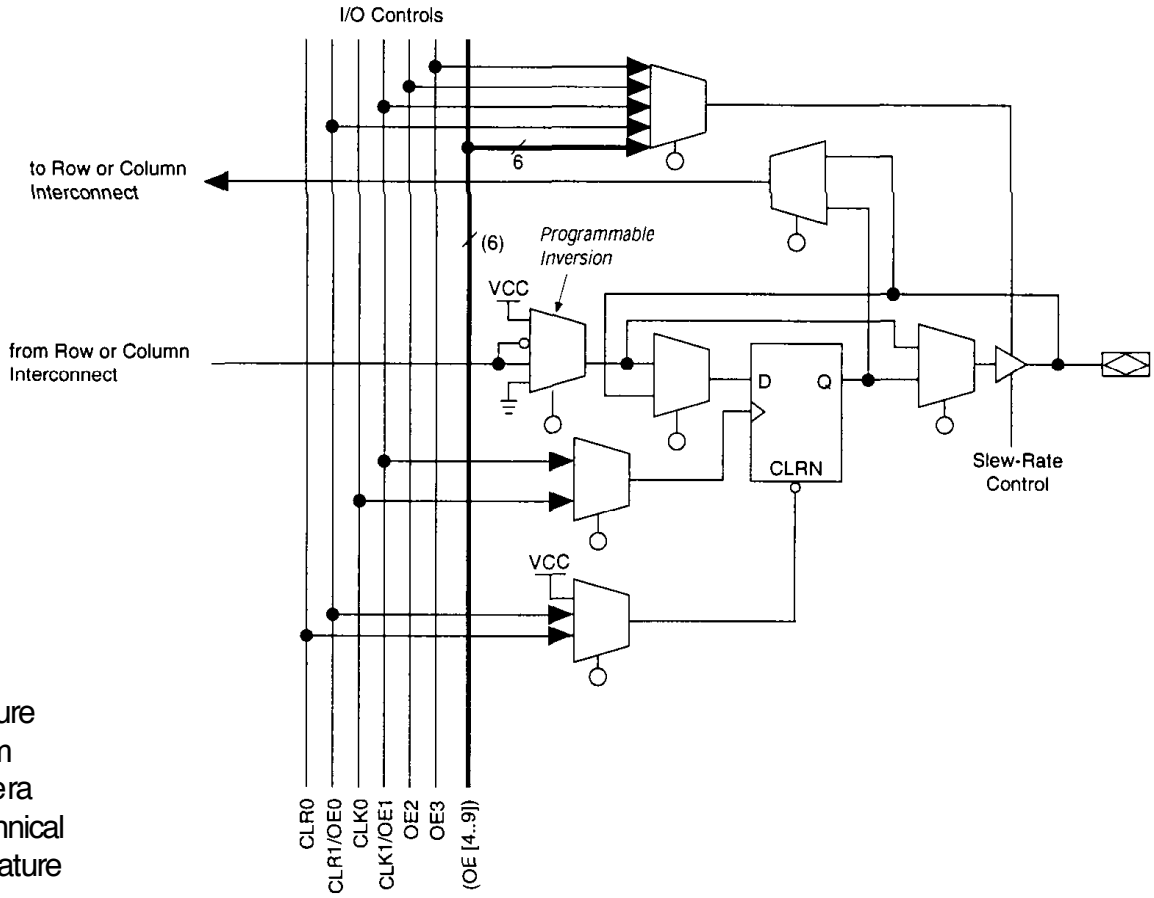


Figure from Altera technical literature

- Eight I/O Elements (IOEs) are at the end of each row and column
  - Some restrictions on how many row / column channels each IOE connects to
  - Contains a register that can be used for either input or output
    - Associated I/O pins can be used as either input, output, or bidirectional pins

Altera FLEX 8000 Configuration

- Loading the FLEX 8000's SRAM with programming information is called *configuration*, and takes about 100ms
  - After configuration, the device initializes itself (resets its registers, enables its I/O pins, and begins normal operation)
  - Configuration & initialization = command mode, normal operation = user mode
- Six configuration schemes are available:
  - Active serial — FLEX gives configuration EPROM clock signals (not addresses), keeps getting new values in sequence
  - Active parallel up, active parallel down — FLEX 8000 gives configuration EPROM sequence of addresses to read data from
  - Passive parallel synchronous, passive parallel asynchronous, passive serial — passively receives data from some host

Altera FLEX 8000 Block Diagram (Review)

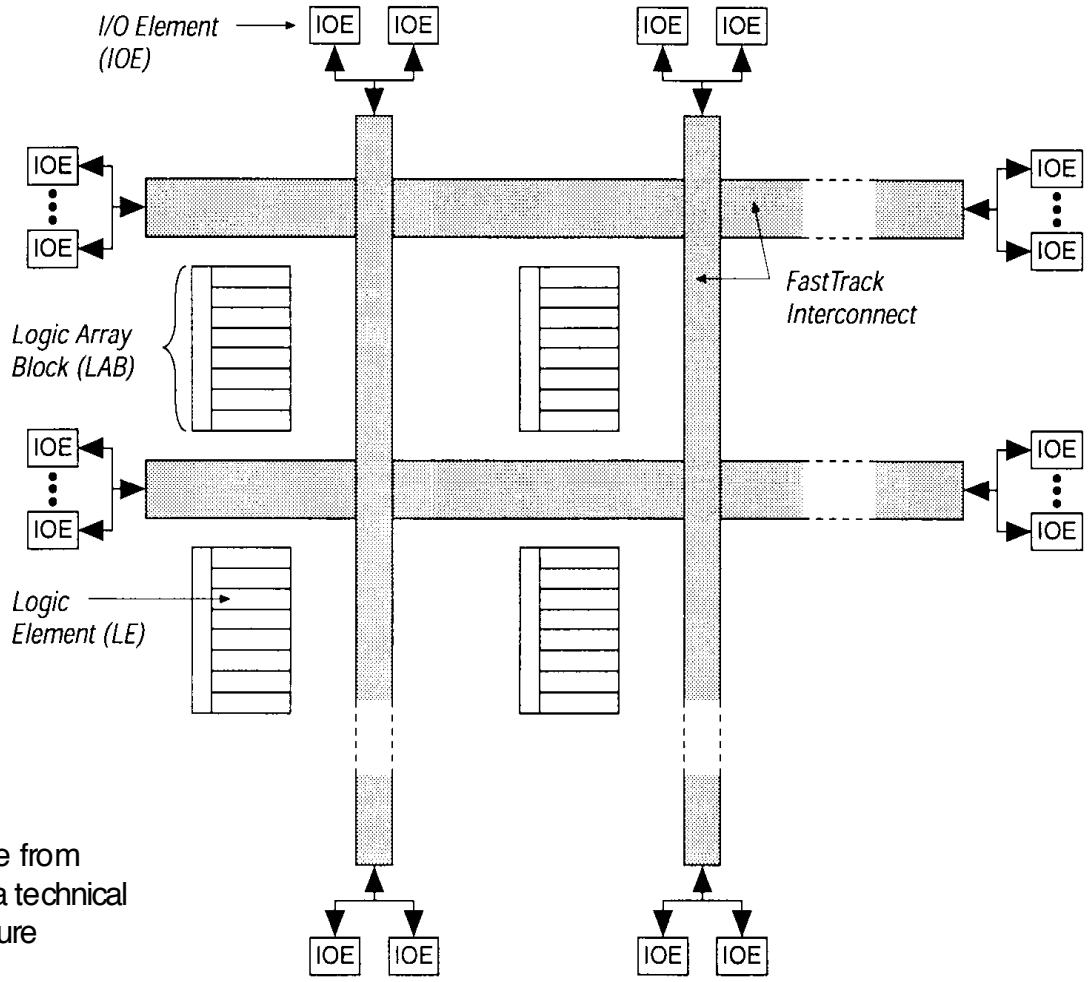


Figure from Altera technical literature

- FLEX 8000 chip contains 26–162 LABs
  - Each LAB contains 8 Logic Elements (LEs), so a chip contains 208–1296 LEs, totaling 2,500–16,000 usable gates
  - LABs arranged in rows and columns, connected by FastTrack Interconnect, with I/O elements (IOEs) at the edges

Altera FLEX 10K Block Diagram

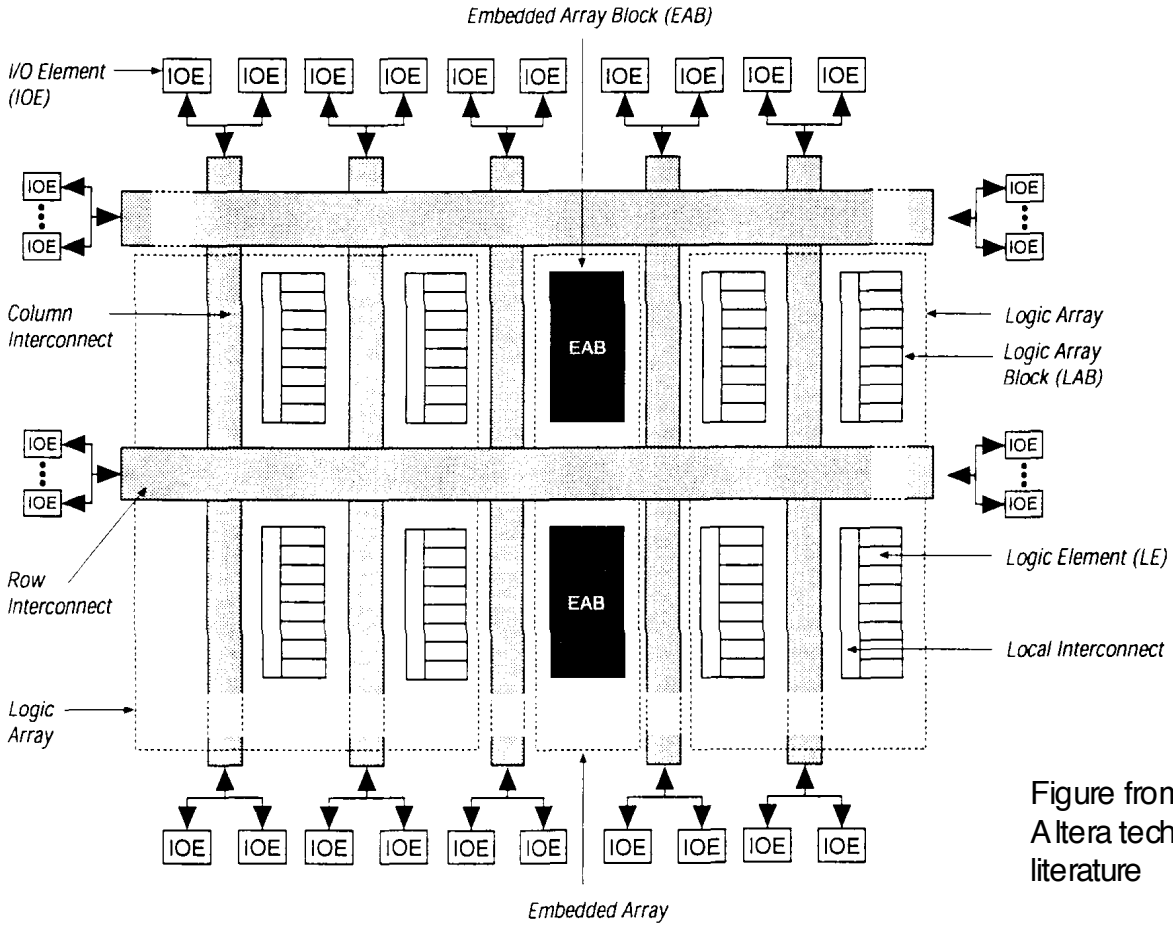


Figure from Altera technical literature

- FLEX 10K chip contains 72–1520 LABs
  - Each LAB contains 8 Logic Elements (LEs), so a chip contains 576–12,160 LEs, totaling 10,000–250,000 usable gates
- Each chip also contains 3–20 Embedded Array Blocks (EABs), which can provide 6,164–40,960 bits of RAM

Altera FLEX 10K
Embedded Array Blocks (EABs)

- Each chip contains 3–20 EABs, each of which can be used to implement either logic or memory

- When used to implement logic, an EAB can provide 100 to 600 gate equivalents (in contrast, a LAB provides 96 g.e.'s)
  - Provides a very large LUT
    - Very fast — faster than general logic, since it's only a single level of logic
    - Delay is predictable — each RAM block is not scattered throughout the chip as in some FPGAs
  - Can be used to create complex logic functions such as multipliers (e.g., a 4x4 multiplier with 8 inputs and 8 outputs), microcontrollers, large state machines, and DSPs
  - Each EAB can be used independently, or combined to implement larger functions

Altera FLEX 10K
Embedded Array Blocks (cont.)

- Using EABs to implement memory, a chip can have 6K–40K bits of RAM
  - Each EAB provides 2,048 bits of RAM, plus input and output registers
  - Can be used to implement synchronous RAM, ROM, dual-port RAM, or FIFO
  - Each EAB can be configured in the following sizes:
    - 256x8, 512x4, 1024x2, or 2048x1
  - To get larger blocks, combine multiple EABs:
    - Example: combine two 256x8 RAM blocks to form a 256x16 RAM block
    - Example: combine two 512x4 RAM blocks to form a 512x8 RAM block
    - Can even combine all EABs on the chip into one big RAM block
    - Can combine so as to form blocks up to 2048 words without impacting timing

Altera FLEX 10K
Embedded Array Blocks (cont.)

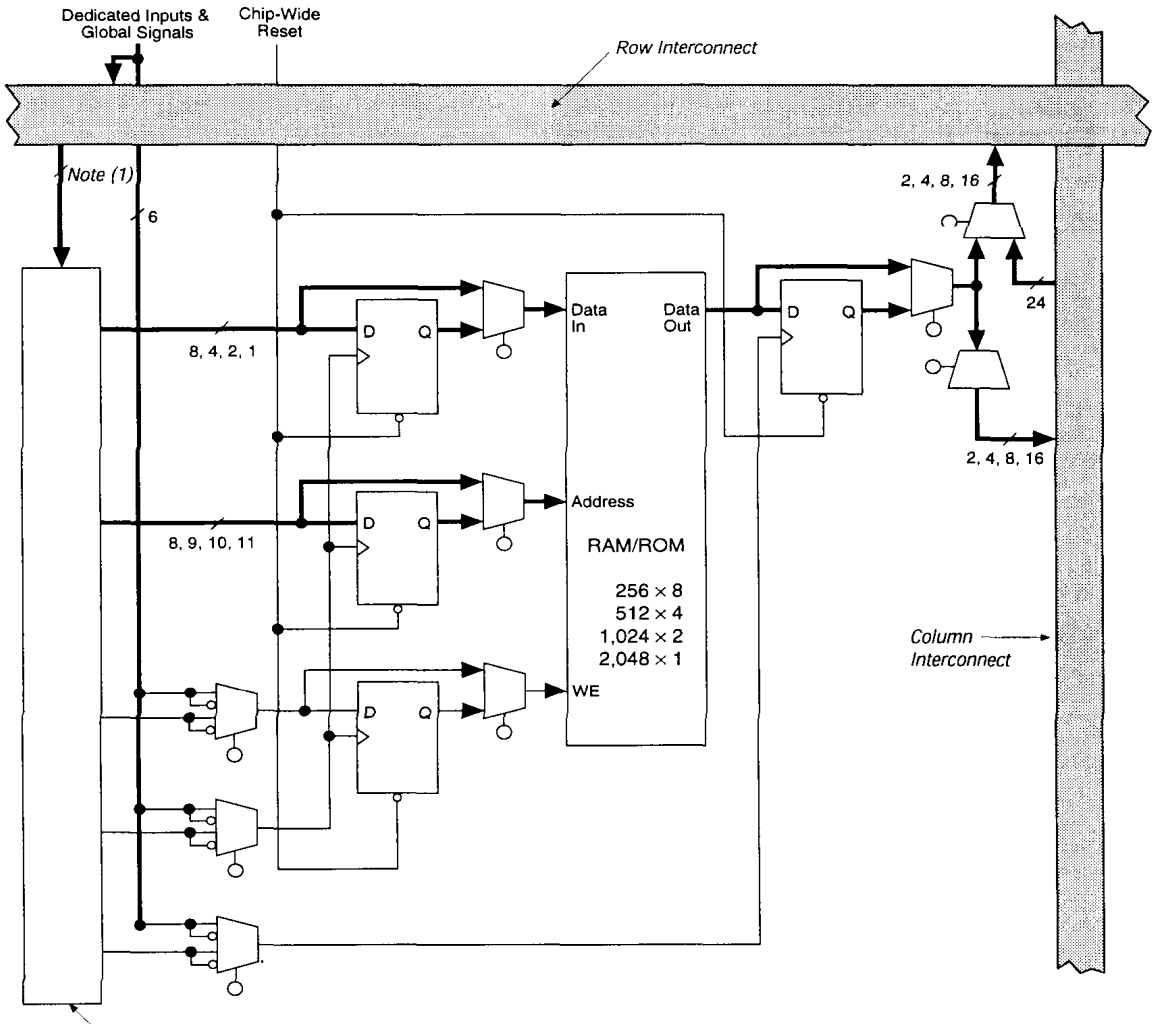


Figure from Altera technical literature

- EAB gets input from a row channel, and can output to up to 2 row channels and 2 column channels
- Input and output buffers are available

Altera APEX 20K Overview

- APEX 20K chip contains:
  - 256–3,456 LABs, each of which contains 10 Logic Elements (LEs), so a chip contains 2,560–51,840 Les, 162,000–2,391,552 usable gates
  - 16–216 Embedded System Blocks (EABs), each of which can provide 32,768–442,368 bits of memory
    - Can implement CAM, RAM, dual-port RAM, ROM, and FIFO
- Organization:
  - MultiCore architecture, combining LUT, product-terms, & memory in one structure
    - Designed for “system on a chip”
  - MegaLAB structures,each of which contains 16 LABs, one ESB, and a MegaLAB interconnect (for routing within the MegaLAB)
    - ESB provides product terms ***or*** memory

APEX LABs and Interconnect

- Logic Array Block (LAB)
  - 10 LEs
  - Interleaved local interconnect (each LE connects to 2 local interconnect, each local interconnect connects to 10 LEs)
    - Each LE can connect to 29 other Les through local interconnect
- Logic Element (LE)
  - 4-input LUT, carry chain, cascade chain, same as FLEX devices
  - Synchronous and asynchronous load and clear logic
- Interconnect
  - MegaLAB interconnect between 16 LABs, etc. inside each MegaLAB
  - FastTrack row and column interconnect between MegaLABs

APEX Embedded System Blocks (ESBs)

- Each ESB can act as a macrocell and provide product terms
  - Each ESB gets 32 inputs from local interconnect, from adjacent LAB or MegaLAB interconnect
  - In this mode, each ESB contains 16 macrocells, and each macrocell contains 2 product terms and a programmable register (parallel expanders also provided)
- Each ESB can also act as a memory block (dual-port RAM, ROM, FIFO, or CAM memory) configured in various sizes
  - Inputs from adjacent local interconnect, which can be driven from MegaLAB or FastTrack interconnect
  - Outputs to MegaLAB and FastTrack, some outputs to local interconnect