Foundations and Trends® in
Electronic Design Automation
Vol. 1, No 3 (November 2006) 195–330

DOI: 10.1561/1000000003

now
the essence of knowledge

FPGA Design Automation: A Survey

Deming Chen[1], Jason Cong[2] and Peichen Pan[3]

[1] *Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, dchen@uiuc.edu*
[2] *Department of Computer Science, University of California at Los Angeles, cong@cs.ucla.edu*
[3] *Magma Design Automation, Inc., Los Angeles, CA, peichen@magma-da.com*

Abstract

Design automation or computer-aided design (CAD) for field programmable gate arrays (FPGAs) has played a critical role in the rapid advancement and adoption of FPGA technology over the past two decades. The purpose of this paper is to meet the demand for an up-to-date comprehensive survey/tutorial for FPGA design automation, with an emphasis on the recent developments within the past 5–10 years. The paper focuses on the theory and techniques that have been, or most likely will be, reduced to practice. It covers all major steps in FPGA design flow which includes: routing and placement, circuit clustering, technology mapping and architecture-specific optimization, physical synthesis, RT-level and behavior-level synthesis, and power optimization. We hope that this paper can be used both as a guide for beginners who are embarking on research in this relatively young yet exciting area, and a useful reference for established researchers in this field.

Keywords: computer-aided design; FPGA design

1

Introduction

The semiconductor industry has showcased the spectacular exponential growth of device complexity and performance for four decades, predicted by Moore's Law. Programmable logic devices (PLDs), especially field programmable gate arrays (FPGAs), have also experienced an exponential growth in the past 20 years, in fact, at an even faster pace compared to the rest of the semiconductor industry. For example, when FPGAs were first debuted in the mid- to late-80s, the Xilinx XC2064 FPGA had only 64 lookup tables (LUTs) and it was used as simple glue logic. Now, both Altera's Stratix II [10] and Xilinx's Virtex-4 chips [207] offer up to over 200,000 programmable logic cells (i.e., LUTs), plus a large number of hard-wired macro blocks such as embedded memories, DSP blocks, embedded processors, high-speed IOs, and clock synchronization circuitry, representing an over 3,000 times increase in logic capacity. These FPGA devices are being used to implement highly complex system-on-a-chip (SoC) designs. To support the design of such complex programmable devices, computer-aided design (CAD) plays a critical role in delivering high-performance, high-density, and low-power design solutions using these high-end FPGAs. We witnessed the

establishment of FPGA design automation as a research area and a dramatic increase in research activities in this area in the past 17–18 years. However, there is lack of comprehensive references for the latest FPGA CAD research. Most existing books (e.g., [23, 27, 93, 151, 188]) and survey/tutorial papers (e.g., [28, 52]) in this area are 10–15 years old, and do not reflect vast amount of recent research on FPGA CAD. The purpose of this paper is to meet the demand for a comprehensive survey/tutorial on the state of FPGA CAD—with an emphasis on the recent developments that have taken place within the past 5–10 years and a focus on the theory and techniques that have been, or most likely will be, reduced to practice. We hope that this paper can be useful for both beginners and established researchers in this exciting and dynamic field.

In the remainder of this section we shall first briefly introduce some typical FPGA architectures and define the basic terminologies that will be used in the rest of this paper. Then, we shall provide an overview of the FPGA design flow.

1.1 Introduction to FPGA Architectures

An FPGA chip includes input/output (I/O) blocks and the core programmable fabric. The I/O blocks are located around the periphery of the chip, providing programmable I/O connections and support for various I/O standards. The core programmable fabric consists of programmable logic blocks and programmable routing architectures. Figure 1.1 shows a high-level view of an island-style FPGA [23], which represents a popular architecture framework that many commercial FPGAs are based on, and is also a widely accepted architecture model used in the FPGA research community. *Logic blocks* represented by gray squares consist of circuitry for implementing logic. Logic blocks are also called configurable logic blocks (CLBs). Each logic block is surrounded by routing channels connected through switch blocks and connection blocks. The wires in the channels are typically segmented and the length of each wire segment can vary. A *switch block* connects wires in adjacent channels through programmable switches such as pass-transistors or bi-directional buffers. A *connection block* connects

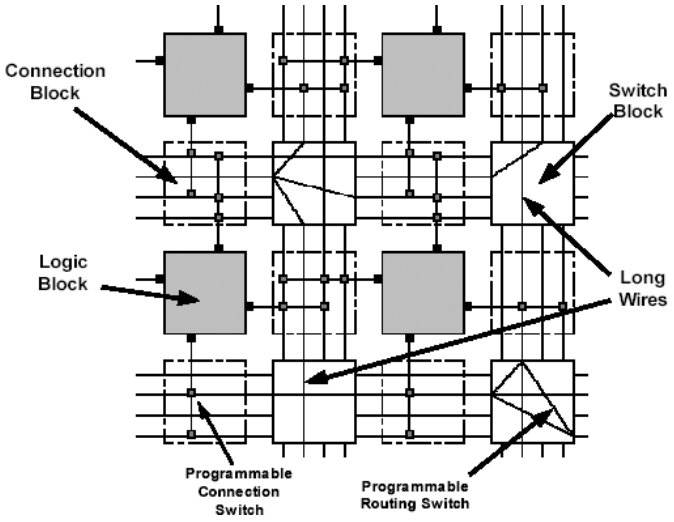


Fig. 1.1 An island-style FPGA [23].

the wire segments around a logic block to its inputs and outputs, also through programmable switches. Notice that the structures of the switch blocks are all identical. The figure illustrates the different switching and connecting situations in the switch blocks (the structures of all the connection blocks are identical as well). In [23] routing architectures are defined by the parameters of channel width (W), switch block flexibility (F_s – the number of wires to which each incoming wire can connect in a switch block), connection block flexibility (F_c – the number of wires in each channel to which a logic block input or output pin can connect), and segmented wire lengths (the number of logic blocks a wire segment spans). Modern FPGAs also provide embedded IP cores, such as memories, DSP blocks, and processors, to facilitate the implementation of SoC designs.

Commercial FPGA chips contain a large amount of dedicated interconnects with different fixed lengths. These interconnects are usually point-to-point and uni-directional connections for performance improvement. For example, Altera's Stratix II chip [10] has vertical or horizontal interconnects across 4, 16 or 24 logic blocks. There are dedicated carry chain and register chain interconnects within and between

logic blocks as well. Xilinx's Spartan-3E chip [206] has long lines, hex lines, double lines, and direct connections between the logic blocks. These lines cross different numbers of logic blocks. Specifically, the direct connect lines can route signals to neighboring tiles vertically, horizontally, and diagonally. For example, Figure 1.2 shows the direct connect lines (a) and hex lines (b) between a CLB and its neighbors in the Spartan-3E chip. The use of segmented routing makes the FPGA interconnect delays highly non-linear, discrete, and in some cases, even non-monotone (with respect to the distance). This presents unique challenges for FPGA placement and routing tools because a simple interconnect delay model using Manhattan distance between the source and the sink may not work well any more. Accurate interconnect delay modeling is a mandate for meaningful performance-driven physical design tools for FPGAs.

Further down the logic hierarchy, each logic block contains a group of basic logic elements (BLEs), where each BLE contains a

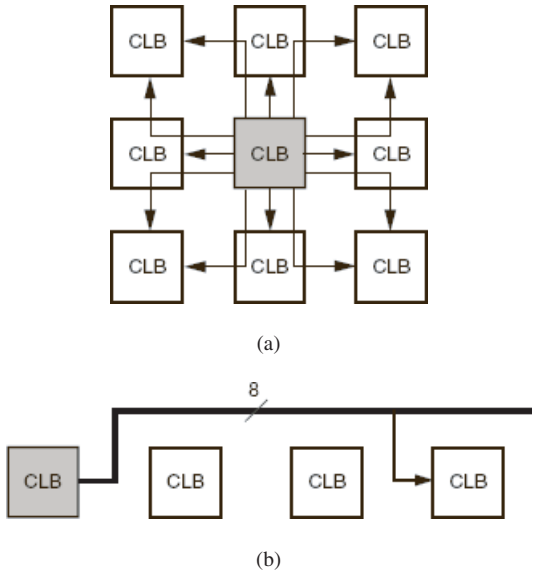


Fig. 1.2 Direct connect lines (a) and hex lines (b) in Xilinx Spartan-3E architecture [206].

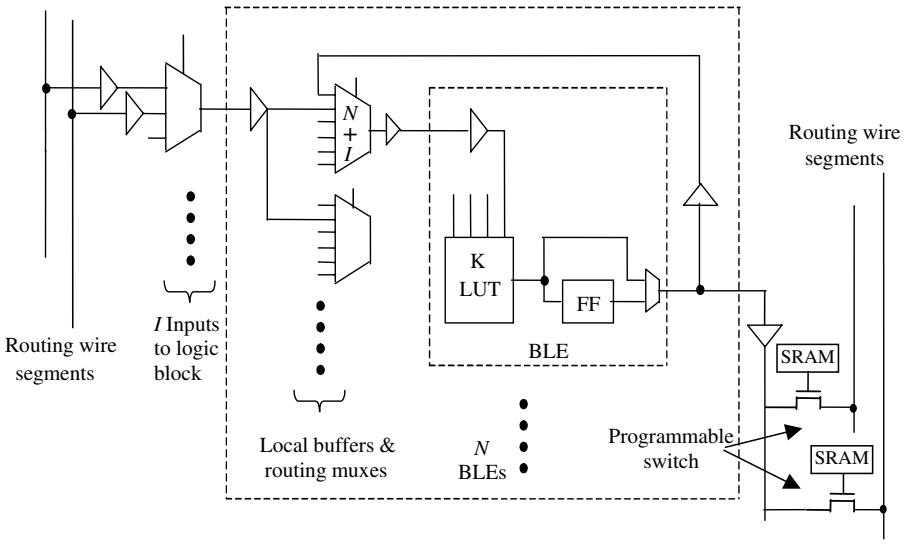


Fig. 1.3 A logic block and its peripheries.

LUT[1] and a register. Figure 1.3 shows part of a logic block with a block size N (the logic block contains N BLEs). The logic block has I inputs and N outputs. These inputs and outputs are fully connected to the inputs of each LUT through multiplexers. The figure also shows some details of the peripheral circuitry in the routing channels.

In addition to logic and routing architectures, clock distribution networks is another important aspect of FPGA chips. An H-tree based FPGA clock network is shown in Fig. 1.4 [131]. A tile is a logic block. Each clock tree buffer in the H-tree has two branches. There is a local clock buffer for each flip-flop in a tile. Both clock tree buffers in the H-tree and local clock buffers in the tiles are considered to be clock network resources. Chip area, tile size, and channel width determine the depth of the clock tree and the lengths of the tree branches.

[1] We focus on the LUT-based FPGA architecture in which the BLE consists of one k-input lookup table (k-LUT) and one flip-flop. The output of the k-LUT can be either registered or un-registered. We want to point out that commercial FPGAs may use slightly different logic architectures. For example, Altera's Stratix II FPGA [10] uses an adaptive logic module which contains a group of LUTs and a pair of flip-flops.

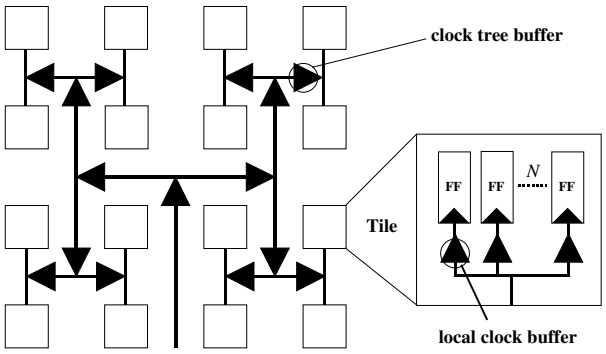


Fig. 1.4 A clock tree [131].

There is another type of programmable logic device called complex programmable logic device (CPLD). The general architecture topology of a CPLD chip is similar to that of island-based FPGAs, where routing resources surround logic blocks. One attribute of CPLD is that its interconnected structures are simpler than those of FPGAs. Therefore, the interconnect delay of CPLD is more predictable compared to that of FPGAs. The basic logic elements in the CPLD logic blocks are not LUTs. Instead, they are logic cells based on two-level AND-OR structures, where a fixed number of AND gates (also called p-terms) drive an OR gate. The output from the OR gate can be registered as well. For example, Fig. 1.5 shows such a structure (called *macrocell*) used in Altera's MAX7000B CPLD [6]. Each macrocell has five p-terms by default. It can borrow some p-terms from its neighbors. The interconnect structure PIA (programmable interconnect array) connects different logic blocks together.

1.2 Overview of FPGA Design Flow

As the FPGA architecture evolves and its complexity increases, CAD software has become more mature as well. Today, most FPGA vendors provide a fairly complete set of design tools that allows automatic synthesis and compilation from design specifications in hardware specification languages, such as Verilog or VHDL, all the way down

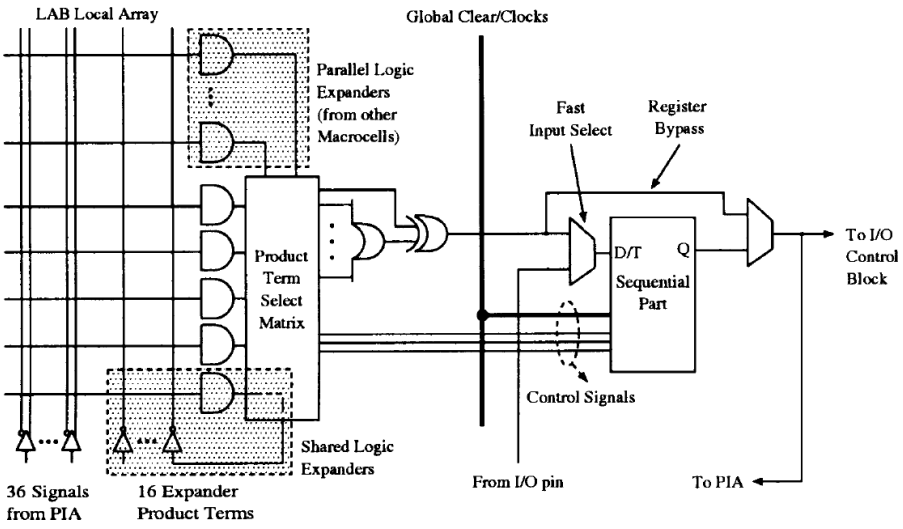


Fig. 1.5 An example of CPLD logic element, MAX 7000B macrocell [6].

to a bit-stream to program FPGA chips. A typical FPGA design flow includes the steps and components shown in Fig. 1.6.

Inputs to the design flow typically include the HDL specification of the design, design constraints, and specification of target FPGA devices. We further elaborate on these components of the design input in the following:

- The most widely used design specification languages are Verilog and VHDL at the register transfer level (RTL) which specify the operations at each clock cycle. There is a general (although rather slow) trend toward moving to specification at a higher level of abstraction, using general-purpose behavior description languages like C or SystemC [182], or domain-specific languages, such as MatLab [185] or Simulink [185]. Using these languages, one can specify the behavior of the design without going through a cycle-accurate detailed description of the design. A behavior synthesis tool is used to generate the RTL specification in Verilog or VHDL, which is then fed into the design flow as shown in Fig. 1.6. We shall discuss the behavior synthesis techniques in Section 5.

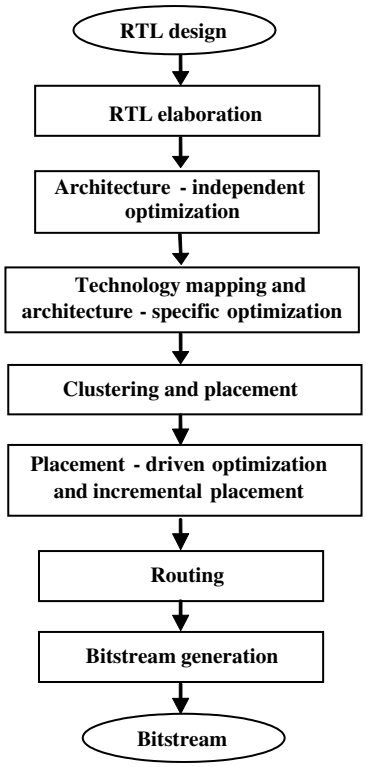


Fig. 1.6 A typical FPGA design flow starting from RTL specifications.

- Design constraints typically include the expected operating frequencies of different clocks, the delay bounds of the signal path delays from input pads to output pads (I/O delay), from the input pads to registers (setup time), and from registers to output pads (clock-to-output delay). In some cases, delays between some specific pairs of registers may be constrained. Design constraints may also include specifications of so-called *false paths* and *multi-cycle paths*. False paths will not be activated during normal circuit operation, and therefore can be ignored; multi-cycle paths refer to signal paths that carry a valid signal every few clock cycles, and therefore have a relaxed timing requirement. Typically, the designer

specifies the false paths and multi-cycle paths based on his knowledge of the design, although recently attempts have been made to automatically extract these paths from the RTL designs [87]. Finally, the design constraints may include physical location constraints, which specify that certain logic elements or blocks be placed at certain locations or a range of locations. These location constraints may be specified by the designer, or inherited from the previous design iteration (for making incremental changes), or generated automatically by the physical synthesis tools in the previous design iterations. We shall discuss the physical synthesis concept and techniques in Section 4.

- The third design input component is the choice of FPGA device. Each FPGA vendor typically provides a wide range of FPGA devices, with different performance, cost, and power tradeoffs. The selection of target device may be an iterative process. The designer may start with a small (low capacity) device with a nominal speed-grade. But, if synthesis effort fails to map the design into the target device, the designer has to upgrade to a high-capacity device. Similarly, if the synthesis result fails to meet the operating frequency, he has to upgrade to a device with higher speed-grade. In both the cases, the cost of the FPGA device will increase—in some cases by 50% or even by 100%. This clearly underscores the need to have better synthesis tools, since their quality directly impacts the performance and cost of FPGA designs.

We now briefly describe each step in the design flow shown in Fig. 1.6 and, following that, we present an outline of the remainder of this paper. Given an RTL design, a set of design constraints, and the target FPGA device, the overall FPGA synthesis process goes through the following steps:

- RTL elaboration. This identifies and/or infers *datapath* operations, such as additions, multiplications, register files, and/or memory blocks, and *control logic*, which is elaborated into a set of finite-state machines and/or generic Boolean

networks. It is important to recognize the datapath elements as most of them have special architectural support in modern FPGAs, such as adders with dedicated fast-carry chains and embedded multipliers.

- Architecture-independent optimization. This includes both *datapath optimization*, using techniques such as constant propagation, strength reduction, operation sharing, and expression optimization; and *control logic optimization*, which includes both sequential optimization, such as finite-state machine encoding/minimization and retiming, and combinational logic optimization, such as constant propagation, redundancy removal, logic network restructuring and optimization, and don't-care based optimization.
- Technology mapping and architecture-specific optimization. This maps: (i) the optimized datapath to on-chip dedicated circuit structures, such as on-chip multipliers, adders with dedicated carry-chains, and embedded memory blocks for datapath implementation, and (ii) the optimized control logic to BLEs. Note that datapath operations can be mapped to BLEs as well if the dedicated circuit structures are not available or not convenient to use.
- Clustering and placement. Placement determines the location of each element in the mapped netlist. Since most modern FPGAs are hierarchical, a separate clustering step may be performed prior to placement to group BLEs into logic blocks. Alternatively, such clustering or grouping may be carried out during the placement process.
- Placement-driven optimization and incremental placement. Once placement is available, interconnects are defined and may become a performance bottleneck (since the delay of a long interconnect can be multiples of a BLE's delay). Further optimization may be carried out in the presence of interconnect delays, including logic restructuring, duplication, rewiring, etc. After such operations, an incremental placement step is needed to legalize the placement again.

The step of placement-driven optimization is optional, but may improve design performance considerably.

- Routing. Global routing and detail routing will be performed to connect all signal paths using the available programmable interconnects on-chip.
- Bit-stream generation. This is the final step of the design flow. It takes the mapped, placed, and routed design as input and generates the necessary bit-stream to program the logic and interconnects to implement the intended logic design and layout on the target FPGA device.

Following sections present the algorithms and techniques used in these steps in reverse order of the design flow. We start with routing and placement (Section 2), then present techniques used in technology mapping and architecture-specific optimization (Section 3). The architecture-dependent optimization phase of FPGA design typically shares techniques widely used for ASIC synthesis and optimization, and we refer the reader to the available textbooks [79, 99] for details. Section 4 presents the techniques used in physical synthesis of FPGA designs, which cover the algorithms used in clustering and placement-driven optimization. Section 5 presents the techniques used in RT-level and behavior-level synthesis for FPGA designs. Section 6 discusses synthesis techniques used for FPGA power optimization, which is a design objective that has received a lot of interest in recent years. This design objective cuts cross all design steps in the flow and interacts with performance and area optimization. Finally, we conclude this paper and touch on future trends of FPGA design automation in Section 7.

2

Routing and Placement for FPGAs

2.1 Routing

Routing is one of the most basic, tedious, yet important steps in FPGA designs. It is the last step in the design flow prior to generating the bit-stream to program the FPGA. FPGA routing is similar to the general ASIC problem in terms of the objective—we need to successfully connect all signal nets subject to timing constraints. However, FPGA routing is more restricted in the sense that it can use only the prefabricated routing resources, including available wire segments, programmable switches, and multiplexers. Therefore, achieving 100% routability is more challenging.

FPGA routing typically goes through routing-resource graph generation, (optional) global routing, and detailed routing. The remainder of this subsection describes these steps in detail.

2.1.1 Routing-resource graph generation

In order to model all the available routing resources in an FPGA, a routing-resource graph is created as an abstract data representation to be used by the global and detailed routers [147, 22]. Given an FPGA

architecture, the vertices in the routing-resource graph represent the input and output pins of the logic blocks as well as the wire segments in the routing channels. The edges represent the programmable switches that connect the two vertices. A unidirectional switch, such as a buffer, is represented by a directed edge, while a bi-directional switch, such as a pass transistor, is represented by a pair of directed edges. To model the equivalent pins, we introduce a source vertex that connects to all the logically equivalent output pins of a logic block, and a sink vertex to connect from all the logically equivalent input pins of a logic block. Figure 2.1 shows an example of a routing-resource graph for a portion of an FPGA whose logic block contains a single two-input, one-output LUT. In general, a node may have a capacity that indicates the maximum number of nets that can use this vertex in a legal routing. In our example, the source vertex has capacity one, while the sink node has capacity two.

Since modern FPGAs may have millions of logic blocks, the routing-resource graph can be very large. Its generation is typically done automatically by a software program, which models the given FPGA, builds the routing-resource graph for a basic tile of the architecture, and then replicates the graph many times and stitches them all together to form the routing-resource graph for the entire FPGA.

In many cases, we need to build the placement and routing tools for an FPGA under development in order to provide quantitative

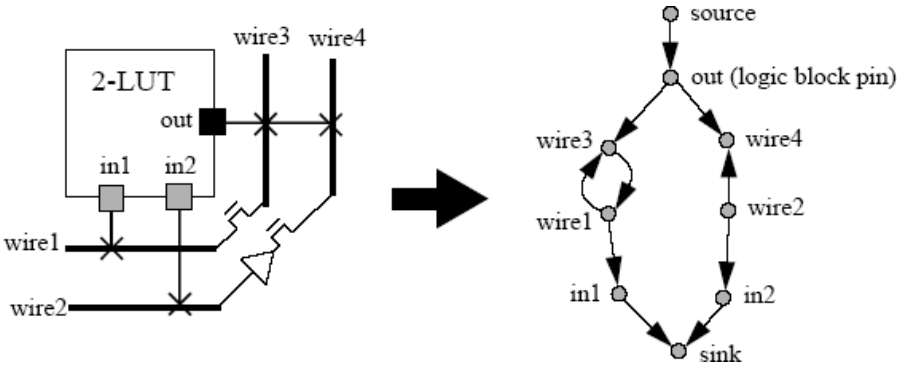


Fig. 2.1 Modeling FPGA routing architecture using a routing resource graph [23].

evaluation of the choice of various architecture parameters before we finalize the FPGA architecture. In this case, we need to generate a routing-resource graph from a set of architecture parameters, as the real FPGA model is not yet available. The typical set of parameters needed for routing include [23]:

a) Number of logic block input and output pins.
b) Side(s) of the logic block from which each input and output is accessible.
c) Logic equivalence between the various input and output pins.
d) Number of I/O pads that fit into one row or column of the FPGA.
e) Relative widths of the horizontal and vertical channels.
f) Relative widths of the channels in different regions of the FPGA.
g) Switch block topology used to connect the routing tracks.
h) F_c values for logic block inputs and outputs, as well as I/O; pads (F_c represents the number of routing tracks in the channel that each input or output pin connects. The F_c value may vary for an input pin, an output pin, or an I/O pad).
i) Wire segment types and distributions: for each segment type, we need to specify segment length, fraction of tracks in the channel with such type, type of switches, and population of the switches on the segment, etc.

Parameters (a) to (f) are needed for global routing, and additional parameters (g) to (i) are needed for detailed routing. A good routing-resource generation tool should be able to: (i) detect any inconsistency in architecture parameter specification, and (ii) provide reasonably good assumptions of the missing parameter in case of partial architecture specification (which is quite common in the early stage of architecture exploration). One important contribution of the VPR placement and routing tool [23] is that it provides a simple language for the user to specify a reasonable set of architecture parameters for an FPGA under investigation and generates the corresponding routing-resource graph automatically.

2.1.2 Global routing

Most IC routing tools go through global routing and detailed routing steps. Global routing divides the available routing area into channels or routing regions. It determines the coarse routing topology of each net in terms of channels or routing regions that the net passes through, typically with the objectives of minimizing overall congestion and satisfying timing constraints of critical nets. Detailed routing generates the design-rule-correct detailed routing geometry to implement every net or subnet in each routing channel or region. The advantage of such a two-step approach is obviously the reduction of problem complexity, as the general routing problem is NP-hard, and it is highly complex to determine the exact routing details of tens of thousands or even millions of nets directly in one step. The problem with such a two-step approach, however, is the possible miscorrelation between global and detailed routing, as the global router has to use a rough model for the available routing resources in each channel or routing region, and does not see the details of routing obstacles, pre-routed nets, etc. Such a problem is more serious in FPGA routing since the detailed distribution of different types of wire segments and programmable switches may greatly affect the success of detailed routing, but is hard to model during the global routing step. Therefore, while a number of FPGA routers still follow the two-step global and detailed routing approach, several other FPGA routers perform global and detailed routing in a single step and demonstrate good results. For completeness, we shall discuss the global and detailed routing techniques used in FPGA designs in this and the following section, and present the approaches for combined global and detailed routing in Section 2.1.4.

For global routing, the routing-resource graph defined in the preceding section can be simplified, resulting in a *coarse routing-resource graph* (or simply *routing graph* when there is no ambiguity), where we represent each routing channel (as opposed to each wire segment) by a vertex, with the capacity being the number of tracks in the channel. We still represent each pin in a logic block by a vertex, and use the source and sink vertices to represent the logically equivalent pins [23]. The edges represent the available connections from the logic block input

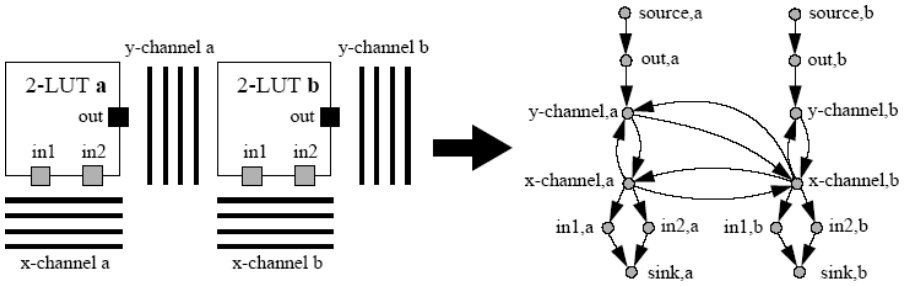


Fig. 2.2 An example of the coarse routing resource graph [23].

and output pins to the channels and available connections between adjacent channels. Figure 2.2 shows an example of the coarse routing resource graph for the portion of the FPGA which is being modeled.

Given the coarse routing-resource graph G, the FPGA global routing problem is to determine the routing of each net on the graph G such that: (i) all the channel capacity constraints are satisfied, and (ii) the signal timing constraints on all the nets are satisfied. We shall defer the discussion on signal timing consideration in FPGA routing to Section 2.1.5 and focus on routability issues in this section and next two sections.

In fact, given the abstract graph-based formulation, the FPGA global routing problem is very similar to that of traditional metal-programmable gate-array (MPGA) or standard cell designs. Therefore, many ASIC global routing techniques may be used for FPGA global routing. The early FPGA routers CGE [29] and SEGA [130] adopted the global router LocusRoute [165] for standard cell designs. But by far, the most successful FPGA global routing approach, like the one used in PathFinder [147] and VPR [22, 23], is based on the negotiation-based global router [155] for standard cell designs.

The basic framework used in the PathFinder and VPR routers is based on the iterative routing. At each iteration, all nets are routed, each using the minimum cost based on the current costs associated with the vertices in the routing graph, even though the solution may lead to over-congestion in some routing channels. Then, we readjust each vertex cost based on whether the corresponding channel has overflowed

in the current iteration and previous iterations. Then, all nets are re-routed based on this new cost function so that congestion, hopefully, can be reduced or eliminated. This process is repeated until all congestion is removed or some pre-defined stopping criteria (such as the maximum number of iterations) is met. Specifically, the cost function in VPR for using a routing resource n when it is reached from routing resource m is the following [23]:

$$\text{Cost}(n) = \text{b}(n) * \text{h}(n) * \text{p}(n) + \text{BendCost}(n, m)$$

where the terms $\text{b}(n)$, $\text{h}(n)$, and $\text{p}(n)$ relate to the base cost, historical congestion, and present congestion. The term $\text{BendCost}(n, m)$ discourages the bends in the routing solution. The base cost remains unchanged throughout the routing process. The present congestion penalty term $\text{p}(n)$ depends on the amount of overflow at resource n, while the historical penalty term $\text{h}(n)$ accumulates the congestion penalty in the previous iterations. The exact forms of these functions are available in [23]. During each iteration, routing of each net is based on maze expansion on the routing graph. For multi-pin nets, maze expansion is first carried out to connect a pair of closest pins in the net. Then, partial routes that connect a subset of terminals in the same net are used for further expansion to connect the next nearest pin. It was shown in [23] that this scheme works remarkably well, and the same approach can be easily extended to combined global and detailed routing (see Section 2.1.4) to produce highly competitive results.

Given the similarity between FPGA and standard cell global routing, we expect that some recent advances in standard cell global routing, such as multi-commodity flow based global routing [3] and multilevel global routing [56], can be successfully extended to FPGA global routing as well, although we have not seen such attempts reported in relevant literature.

2.1.3 Detailed routing

In this section we present the detailed routing algorithms used in a two-step approach for FPGA routing. The following section presents the combined global and detailed routing approach. Given a global

routing solution, the detailed routing step implements each route in the coarse routing-resource graph in the detailed routing-resource graph so that there is no resource conflict. Since there are different types of wire segments and programmable switches in a channel, the number of possible ways to implement each route in the coarse routing graph is still quite large.

The detailed routers used in CGE [26] and SEGA [130] go through two phases. In the first phase, for each global route, the router enumerates all the possible detailed routes in the routing-resource graph that go through the same set of channels, and adds them into *an expansion graph*. For example, given the global route from (0,4) to (4,0) (as shown in Fig. 2.3), three possible detailed routes are available and added to the expansion graph. In the second phase, the router's algorithm repeatedly: (i) selects the detailed route of the lowest cost (defined later), (ii) removes the other alternative detailed routes of the selected route for the same net, and (iii) removes the detailed routes of other nets that conflict with the selected route until all the global routes are implemented by the detailed routes. Note that operation (iii) may result in some nets being unroutable. To avoid this, when a detailed route becomes the only alternative for a global route, it is called an *essential route*, and essential routes are routed with high priority. In

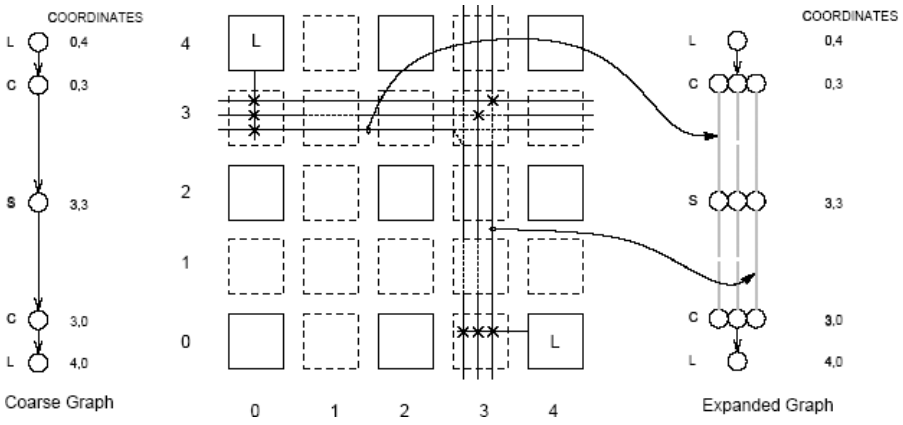
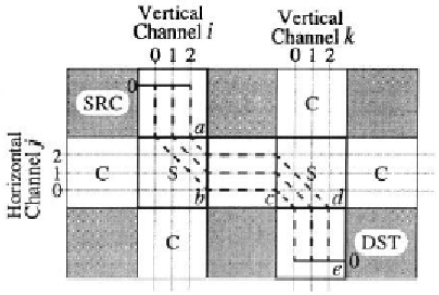


Fig. 2.3 Expansion of a global (coarse) route into three detailed routes [130].

general, a cost is associated with each detailed route p in the expansion graph, which reflects the number of segments used, the waste of long segments by short connections, the number of alternative paths to p, and the impact that the selection of p has on other paths in the expansion graph. The exact formulation of the cost function is available in [130]. Since all the detailed routes are enumerated in Phase 1, it is also possible to use the iterative deletion approach as introduced in [67] in Phase 2 to select the proper detailed route for each global route, although this approach was not attempted in [26] and [130].

Another approach to FPGA detailed routing uses the Boolean satisfiability (SAT) formulation [156, 157]. Given a global routing solution, the SAT-based detailed router divides each net into several horizontal and vertical net segments. Then it generates the connectivity constraints and exclusivity constraints using Boolean expressions in the conjunctive normal form, so that its truth assignment gives a legal detailed routing solution. The *connectivity constraints* ensure the existence of a connecting path for each two-pin net through a sequence of connection and switch boxes, and model the flexibility in using the different wire segment and programmable switches on the path defined by the global route. For example, given the global route of net N shown in Fig. 2.4(a) from the source (SRC) logic block (also called CLB in [156]) to the destination (DST) logic block, the Boolean expression C_{a} in Fig. 2.4(b) specifies that net N can be assigned to any of the three tracks in the vertical channel i. Similarly, the Boolean expression S_{b} encodes the constraint that if a route enters from track j at the top of switchbox b, it must exit from the right, also on track j. On the other hand, the *exclusivity constraints* ensure that different nets will not share the same routing resource. For example, given the three nets A, B, and C in the horizontal channel in Fig. 2.4(c), the Boolean expression E_{m} encodes the constraints that net A cannot share a track with net B or net C. After connectivity and exclusivity constraints are generated for all nets, they are given to a SAT-solver, such as GRASP [173], as used in [156]. If a satisfiable solution is found, we have a detailed routing solution, otherwise, we are certain that the given global routing solution cannot be implemented in the current architecture. In theory, this approach provides an exact formulation to the FPGA detailed routing



(a) Global route and possible detailed routes for net N (shaded boxes are CLBs).

$C_a = [(V_i \quad 0) \quad (V_i \quad 1) \quad (V_i \quad 2)]$

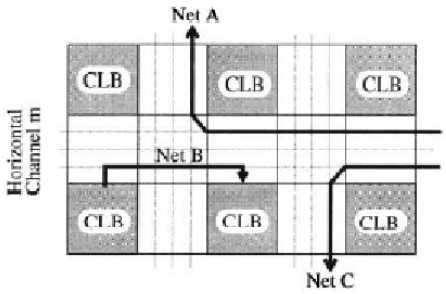
$S_b = [(V_i \quad 0) \quad (H_j \quad 0)]$
$[(V_i \quad 1) \quad (H_j \quad 1)]$
$[(V_i \quad 2) \quad (H_j \quad 2)]$

$C_c = [(H_j \quad 0) \quad (H_j \quad 1) \quad (H_j \quad 2)]$

$S_d = [(H_j \quad 0) \quad (V_k \quad 0)]$
$[(H_j \quad 1) \quad (V_k \quad 1)]$
$[(H_j \quad 2) \quad (V_k \quad 2)]$

$Ce = [(V_k \quad 0) \quad (V_k \quad 1) \quad (V_k \quad 2)]$

(b) Connectivity constraints for net X
C formulas model C-blocks;
S formulas model S-blocks.



$E_m = [(A \quad B) \quad (A \quad C)]$

(c) Exclusivity constraint in horizontal channel m

Fig. 2.4 Examples of connectivity constraints and exclusivity constraints [157].

problem. The experimental results in [156] indeed reported considerably smaller routing track usage when compared to the SEGA detailed router [130]. In general, the runtime complexity is a concern for the SAT-based approach, especially for large designs, as the SAT problem is NP-complete. However, the recent progress in efficient SAT-solvers (e.g., [219]) will make this approach more scalable.

2.1.4 Combined global and detailed routing

In order to avoid the possible mismatch between global and detailed routing due to the difficulty of approximating all available routing

resources in FPGA designs, several FPGA routers combine global and detailed routing in one step and produce very good results.

The early attempt was the greedy bin-packing (GBP)-based FPGA router reported in [203]. It decomposes each multi-pin net into a set of two-pin subnets. It also divides all wire segments in the underlying FPGA into a set of *track domains*, where each track domain includes a set of wire segments that can be connected using programmable switches. Then, the GBP router greedily packs the two-pin nets into the track domain based on the best-fit-decreasing (BFD) bin-packing heuristic and a few other (heuristic) considerations in selecting the nets for packing. This simple approach worked surprisingly well and reported better routing results than those of CGE [26] and SEGA [130]. The GBP heuristic was further enhanced with another "orthogonal" greedy growth heuristic for packing the nets into track domains. The resulting router, named orthogonal greedy coupling (OGC) [204], reported an improved routing result over that of GBP [203].

Another approach to combined global and detailed routing was the simulated evolution-based router named Tracer-fpga, reported in [125]. It first routes every net on the (detailed) routing-resource graph using the classical maze routing expansion algorithm. When routing a net, it will consider the existence of already-routed nets and try to avoid routing violations. If not possible, it will select a route with the minimum number of routing violations. After this stage of initial routing of all the nets, Tracer-fpga goes through the rip-up and re-routing stage based on simulated evolution. Each routed net is assigned a cost based on its routing tree length and the number of routing violations involved. The simulated evolution scheme selects a subset of nets for rip-up and re-route based on their costs. A net with a higher cost will have a higher probability of being selected for re-routing in the presence of already-routed nets; the objective here is to minimize the routing cost and violation. This process is repeated a number of times until a solution free of routing violation is obtained, or some predefined stopping criteria is met. Note that the probabilistic scheme used by simulated evolution may occasionally choose a "good" net (free of routing violation) for re-routing, which makes this

approach less greedy and more robust. This approach reported better routing results than those of CGE [26] and SEGA [130], as well as GBP [203].

Up to this point, however, the most successful router that combines global and detailed routing is the VPR router [22, 23] which uses the negotiation-based approach presented in Section 2.1.2. It applies exactly by the same global routing engine for combined global and detailed routing on the detailed routing-resource graph. Through careful evaluation and selection of various parameters in the negotiation-based routing algorithm, such as assignment of base routing cost, penalty factors for historical and present congestions, etc., the authors of [22] developed a highly optimized and robust FPGA routing tool that has been widely used in the community. The experimental results reported in [23] show that VPR consistently requires fewer numbers of routing tracks compared to all other FPGA routers in the literature at that point of time, including CGE [26], SEGA [130], GBP [203], OGC [204], IKMB [5], and Tracer-fpga [125].

Other than using maze expansion for routing tree construction of a multi-pin net in the routing-resource graph, one may use graph-based Steiner heuristics to construct a near-optimal Steiner tree in the graph. In [5], two graph-based Steiner heuristics, IKMB and IZEL were developed. Both use the idea of iterated Steiner tree construction. IKMB is based on the heuristic of Kou, Markowsky, and Berman [117], and IZEL is based on a more recent heuristic of Zelikovsky [218] which has a performance bound of 11/6 from the optimal Steiner tree. The implementation of IKBM in a FPGA router [5] leads to a smaller routing track usage compared to those of CGE [26], SEGA [130], and GBP [203], but falls behind those of Tracer-fpga [125] and VPR [22], even though both Tracer-fpga and VPR use maze expansion for the routing tree construction. This is not necessarily a negative reflection of the effectiveness of the graph-based Steiner heuristics, since many other implementation details, such as the choice of cost functions and cost updating schemes (which are well done in VPR), will affect the routing solution quality considerably. It would be interesting to try replacing the maze expansion engine in VPR with a graph-Steiner-based algorithm to better measure the impact.

2.1.5 Timing optimization in routing

So far we have focused only on the routability issue in routing. It is important to perform timing optimization in routing since routing delays in FPGA designs are significant, largely due to the extensive use of programmable switches. We can group timing optimization techniques roughly into four categories: routing order optimization, routing tree topology optimization, slack distribution, and net weighting.

Timing constraints are typically specified as the maximum path delay constraints from the primary inputs and/or FF outputs to primary outputs and/or FF inputs. Given a mapped and placed circuit, one can perform static timing analysis to compute the signal arrival times and required times at every pin in the design, and then compute the slack at every pin and every source–sink pair in each net. The nets with smaller slacks are more critical. The simplest form of timing optimization is to order the nets by their timing criticality: timing-critical nets are routed first so that they can avoid long detours. This simple approach is used in [125] and almost every timing-driven router.

The next level of timing optimization is to optimize the routing tree topologies of the timing-critical nets. For example, the work in [5] extended the A-tree algorithm [62] used for timing-driven IC routing and proposed that two graph Steiner arborescence (GSA) heuristics be performed on the FPGA routing-resource graph. This routes a timing-critical net by an *arborescence*, which is a routing tree with the shortest path from the source to every sink in the routing graph (for delay minimization), and also tries to minimize the total routing cost of the arborescence. The timing-driven FPGA router in [36] uses a bounded-delay minimum-cost spanning tree formulation, where the delay of each route is estimated by the number of programmable switches it goes through. However, it shows that the problem is NP-hard, and presents a heuristic based on the methods in [18] and [60]. Both PathFinder [147] and VPR [22, 23] use a delay penalty term in the routing cost function in their iterative negotiation-based routing framework to balance the delay and congestion optimization. In particular, VPR uses the Elmore delay model, which is more accurate. When routing net i to

its sink j, the modified routing cost function at each vertex n in the routing graph is

$$\begin{aligned}\text{Cost}(n) = &\text{Crit}(i,j) * \text{delay}(n, \text{topology}) \\ &+ [1 - \text{Crit}(i,j)] * \text{b}(n) * \text{h}(n) * \text{p}(n)\end{aligned}$$

where $\text{Crit}(i,j)$ is a balance factor in the interval [0,1]. Depending on the criticality of sink j of net i in the design, delay(n, topology) is the Elmore delay at the vertex n given the partial routing topology constructed so far, and the terms b(n), h(n), and p(n) are the same as defined in Section 2.1.2. If $\text{Crit}(i,j)$ is 0 (for non-critical nets), we ignore the delay term completely; however, if $\text{Crit}(i,j)$ is 1, we completely ignore the congestion term, which may not be good. Therefore, VPR chooses the balance factor in such a way that for the most timing-critical nets, the corresponding $\text{Crit}(i,j)$ is slightly below 1 (in fact, 0.99 in its implementation).

Another consideration of timing optimization in routing is slack distribution. If we do not distribute slacks in advance, the nets that are routed earlier may use more slacks than those that are routed later. Early work used the zero-slack algorithm [100] introduced for custom IC designs for slack distribution. An improved algorithm, called the limited bumping algorithm, was presented in [90] and applied to FPGA routing. It presents a general heuristic and allows the slacks to be distributed based on the net's estimated load capacitance, fanout numbers, etc. In fact, it was recently shown that the slack distribution problem can be solved optimally under some reasonable objective functions [94], although we have not seen such a method being applied to FPGA routing or placement in the literature.

In general, it is difficult to determine the best way to allocate slacks to different nets in the design. An alternative approach is to assign weights to the nets based on their timing criticality so that they can compete for slacks and routing resources based on those weights. There are conceivably many ad hoc heuristics for weight assignment, but the method presented in [124] gives a systematic way for net weight assignment in FPGA routing. It formulates the timing-driven routing problem as a constrained optimization problem, and solves it by

Lagrangian relaxation. The Lagrangian relaxation approach transforms the timing-constrained routing problem into a sequence of Lagrangian subproblems. The Lagrangian multipliers, which are updated during each iteration, can be viewed as the weights of the source–sink pairs of all the nets. This guides the router to properly allocate timing slacks and routing resources. Experimental results show an improved critical path delay for 13 out of 17 benchmarks with comparable runtimes when compared to the VPR router in the timing-driven mode.

2.2 Placement and Floorplanning

Placement has a significant impact on the performance and routability of circuit design in nanometer designs, because a placement solution, to a large extent, defines the amount of interconnect in the design, which now becomes the bottleneck of circuit performance. The interconnect performance bottleneck is even worse in FPGA designs since the programmable switches incur more delays. A comprehensive survey on modern placement techniques was recently compiled in [61], and we do not want to duplicate the effort. In this section, we focus on FPGA placement, which was not thoroughly covered in [61]. We divide the existing approaches to FPGA placement in four categories: simulated annealing-based placement, partitioning-based placement, analytical method-based placement, and fast placement and floorplanning.

2.2.1 Simulated annealing-based approach

The well-known VPR package for FPGA placement and routing [22, 23] uses the simulated annealing method as its optimization engine for placement. The basic operation (move) is the exchange of two logic blocks, with one of them possibly being an empty logic block. VPR follows the basic template of simulated annealing, but with several placement-specific enhancements like:

a) A new temperature updating scheme, which decreases the temperature faster when the move acceptance rate is very high or very low, so that the annealing process spends more

time at the most productive temperature region—when a significant portion of moves, but not all, are being accepted.

b) A limitation on the range of cell exchanges so that the move acceptance rate is as close to 0.44 as possible and for as long as possible.

c) A linear congestion model, based on the cost function, to handle cases where the channel capacity is non-uniform in the given FPGA architecture. The model shows that this cost function can be computed as efficiently as the traditional half-perimeter bounding box based model, but with good improvement for routability optimization in the case of non-uniform channel capacity.

d) A faster method for incremental net bounding box updating, with some small memory overhead for additional bookkeeping.

The timing-driven mode [145] of the VPR placement tool carries out timing optimization by including an additional timing cost term to the objective function. The timing cost is the summation of the delay times and the timing criticality over all connections in the design, where the criticality of a connection depends on its timing slack. Since every accepted move may change delays in a number of connections, which may in turn change the slack distribution, static timing analysis is required to recompute all the slacks after each accepted move. But this is too costly in terms of runtime. The VPR placement tool chooses to update slacks after a number of moves (typically at the end of each temperature iteration), which leads to shorter runtime and more robust optimization (by avoiding frequent changes of the coefficients in the cost function).

Adding two terms for two very different objectives (in this case, timing and wirelength) usually requires careful scaling. The VPR placement tool uses the idea of self-normalization, where the changes of timing cost and wirelength of a move are scaled by the total timing cost and total wirelength at the end of the previous temperature iteration, respectively. This ensures that the relative importance of timing and wirelength is captured in the cost function, independent of their actual

values and measuring units. Overall, the VPR placement tool provides very good results and is widely used in the FPGA research community.

Given the timing cost term in the objective function, it is clear that the VPR placement tool minimizes the weighted delays of all connections (as part of the objective function), where the weight of a connection depends on its slack. But it ignores another important consideration in timing optimization—path sharing. Intuitively, a connection appearing in many critical paths should be given a higher weight for optimization. This is difficult to capture and compute in general, as there might be an exponential number of paths going through a connection, each with a different timing criticality. A proper solution has recently been proposed [115]. The algorithm, named PATH, can properly scale the impact of all paths by their relative timing criticalities (measured by their slacks), respectively. It was shown in [115] that for certain *discount* functions, this method is equivalent to enumerating all the paths in the circuit, counting their weights, and then distributing the weights to all edges in the circuit. Such computation can be carried out very efficiently in linear time, and experimental results have confirmed its effectiveness. Compared with VPR [145] under the same placement framework, PATH reduces the longest path delay by 15.6% on average with no runtime overhead and only a 4.1% increase in total wirelength. Note that this weight scheme is not limited to the use of an annealing-based placer. It can be used by any placer that optimizes the weighted delays of all connections as part of its objective function.

2.2.2 Partitioning-based approach

Min-cut or partitioning-based placement is one of the earliest approaches to circuit placement, and it has also been applied to FPGA placement [4, 142]. In this section, we briefly highlight the key features of a recent partitioning-based FPGA placement tool, named PPFF, and reported in [142].

a) PPFF uses the state-of-the-art multilevel partitioner hMetis [110] as its partitioning engine.
b) PPFF performs recursive bi-partitioning in a breadth-first manner. At each level of the partitioning hierarchy, it

considers terminal alignment (i.e., trying to align terminals of a net in the same horizontal or vertical channel) in addition to the traditional cut-size minimization objective. It argues that the terminal alignment is helpful for FPGA placement in general, and shows that incorporating terminal alignment in the timing-driven VPR placement tool can also lead to a slight improvement in delay reduction.

c) In order to facilitate terminal alignment, PPFF optimizes the partitioning order of the regions in the same level of partitioning hierarchy, and shows that the ordering problem is equivalent to that of linear placement with the minimum sum of incoming edges, and that the problem can be solved optimally by a simple greedy algorithm.

d) PPFF estimates the delay of each net by computing this on its minimum span of the net at the current level of the partitioning hierarchy and performing table lookup of delay values stored in a pre-computed table (obtained by analyzing some routed placement solutions).

e) At the end of partitioning-based placement, PPFF goes through a legalization step and a post-optimization step of low-temperature annealing to further improve the solution quality.

The experimental results in [142] report a slight degradation in the solution quality of PPFF, with 3–4X improvement on runtime when compared to the VPR placement tool.

2.2.3 Embedding-based approach

In this section, we briefly introduce one recent FPGA placement algorithm based on graph embedding and metric geometry, named convex assigned placement for regular ICs (CAPRI) [96]. CAPRI considers the dependence of routing delays on the FPGA routing architecture. Figure 2.5 illustrates the motivation for the approach [96]. The contour lines in the figure join points on the chip surface that are equidistant from the origin using geometric metrics such as (a) Euclidean

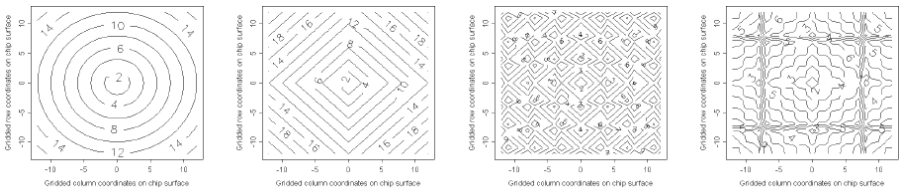


Fig. 2.5 Contour plots showing points of equal distance (i.e., delay) from the origin using the following metrics: (a) Euclidean distance, (b) Manhattan distance, (c) Delays along an FPGA routing grid with two different kinds of route segments, and (d) Delays along routing grid that mimics the Xilinx Virtex FPGA [96].

and (b) Manhattan distances, as well as delays measured in terms of the number of routing switches on two different FPGA routing architectures (c) and (d). These plots demonstrate that to model delays accurately in FPGA placement, there is a need for a metric that captures the delay contours of the FPGA routing architecture, rather than the Euclidean or Manhattan metrics used in ASIC placement.

CAPRI views the placement task as an embedding of a graph representing the netlist into a chosen metric space. It first defines an analytic metric of "distance" in terms of the total delay through switches on the FPGA routing architecture, and then uses it to construct a metric space that captures FPGA performance. CAPRI then embeds the netlist graph into this metric space based on the binary quadratic assignment formulation (which is NP-hard), and solved the problem with a heuristic technique based on matrix projections followed by online bipartite graph matching. The resulting solution is a legal initial placement, which tries to minimize delays on driver–sink connections and is thus "good" from a global timing perspective. Subsequently, CAPRI applies local optimization using an existing low-temperature simulated-annealing in VPR for local optimization to improve specific critical paths and routability.

When compared with running VPR alone, CAPRI shows an improvement of 10.1% (median) and 11.1% (mean) in the post-routing delay of top critical paths. Total placement runtime is improved by 2X, and CAPRI itself is reported to take just 4.8% of this total runtime.

2.2.4 Fast placement and floorplanning

As modern FPGAs reach close to a million logic blocks, more efficient and scalable FPGA placement algorithms are needed. Multilevel placement and floorplanning techniques are introduced to improve the runtime of existing FPGA placement algorithms.

The Ultra-Fast Placement (UFP) algorithm in [166] aims at significant runtime improvement of the VPR placement tool, which produces good placement results but are not very scalable due to the use of simulated annealing. UFP uses multilevel optimization, and starts with multilevel clustering. It requires cluster sizes at each level to be the same (in fact, to be the powers of 2, such as $4, 8, 16, 32, \ldots$) to facilitate pair-wise exchange at each level later on by simulated annealing. It shows experimentally, that the cluster sizes in the first three levels should be (64, 4, 4), (64, 16, 4), or (256, 4, 4). The clustering algorithm starts with a cluster with a random seed occupying an arbitrary slot in the cluster. Then, the algorithm grows the cluster based on a connectivity-based scoring function. The score of adding a logic block to the cluster is determined by two components: (i) the strength of connections between the block and the cluster, measured by the summation of the shared nets, with the smaller nets favored over larger nets; and (ii) the number of nets that are absorbed if the block is merged into the cluster. (We say a net is absorbed into a cluster if all the blocks on that net belong to the cluster.) The block with the highest score is added to the next available slot in the cluster, and if the cluster is full, a new cluster is started with a random seed. This process is repeated until all blocks are clustered. The result is a clustered netlist with the absorbed nets removed. Then, we may proceed to create the next level of clustering hierarchy. After the clustering hierarchy is created, low-temperature simulated annealing is performed at each level of the clustering hierarchy. When we de-cluster from a coarser level to a finer level, the position of each cluster (or logic block) in the finer level is determined by the mean of the positions of the I/O pads and the parent clusters that are connected to it. The experimental result reported in [166] showed smooth runtime and quality trade-off. At one extreme, UFP achieved

over 50X speed-up over the VPR placement tool, with 33% wirelength overhead.

To the best of our knowledge, UFP is the first reported multilevel placement algorithm in the literature. Recently, multilevel placement became a very active research topic, with several high-quality multilevel placers developed for standard cell-based designs (e.g., [31, 69, 109]). It is likely that the multilevel placement techniques developed in these works can be used to further enhance the quality of UFP.

Another way of reducing placement runtime is through floorplanning or hierarchical placement based on the design hierarchy specified in the RTL designs (note that UFP applies to the flattened netlist only). Such an approach was taken in [82] and [184]. Here, we briefly outline the fast floorplan and placement system, named Frontier and reported in [184]. It starts with a macro-based netlist of soft and hard macros targeted to an FPGA device. Initially, the FPGA device is decomposed into an array of placement *bins*, each having the same physical dimensions. Then it groups the macros into clusters, with each cluster being placed into a bin. Each cluster will accommodate the volume of macro logic blocks and the physical dimensions of hard macros inside a bin. If, there is an insufficient number of available bins to place all clusters, following clustering bin sizes are increased and clustering is restarted. After clustering, each cluster is assigned to a physical bin location on the target device, and entire bin clusters are subsequently swapped between physical bins using simulated annealing to minimize inter-bin placement cost, including connectivity to the I/O pads. Since the number of bins allocated to a device is frequently much smaller than the number of device logic blocks, this process proceeds rapidly. Following bin placement, hard and soft macro blocks are placed within each bin in a space-filling fashion. All intra-bin placement is based on inter- and intra-bin connectivity. Soft macros are resized at this point to meet bin-shape constraints. After this step, detailed estimates of the placement wirelength and post-route design performance are carried out, taking into account the special features of the FPGA device. These wirelength and performance estimates are used to evaluate whether subsequent device routing will complete quickly, require a long period of time, or ultimately fail, based on the estimation method

in [180]. For floorplans that are impossible or difficult to route, another low-temperature simulated annealing is performed on soft macros to smooth wirelength inefficiencies. This approach reported a 2.6X speed-up in the total placement and routing time compared to the place-and-route system which was available at that time from Xilinx used on multiple designs with 2,000 to 3,000 CLBs in the Xilinx 4000 series FPGAs [184].

2.3 Combined Placement and Routing

Given the fact that it is hard to achieve 100% routability, especially for the earlier generation of FPGAs (1990s), several attempts were made to combine placement and routing, so that the placement solution is assured to be routable. In this section, we briefly summarize the work in this area.

The earliest attempt at combining placement and routing was reported in [154], which embedded a fast router inside the inner loop of a simulated annealing-based placement engine. After each placement move, incremental routing was performed on the nets affected by the move. Although an 8–15% performance improvement was reported over the commercial FPGA place-and-route tool which was available at that time from Xilinx, the runtime overhead was very high, ranging from 6X for the smallest design to 11X for the largest design.

Another approach to integrating placement and routing is to embed global routing in a partitioning-based placement algorithm, so that global routing is performed at every level of the placement hierarchy during the recursive partitioning process. Such an approach is more scalable, and was used in [4] and [187]. However, this approach has not shown results that demonstrate the superiority of the combined approach.

A more recent work in [35] combines a simple cluster growth placer with a maze router. It places and routes nets one by one. For each net being placed, it chooses a position to optimize a cost function with three components: (i) the number of segments used (for delay minimization), (ii) the type and length of the segments used (for both delay and routability optimization), and (iii) the density of the channels (for

routability optimization). Although it was shown that using this cost function is helpful compared to the traditional metric of wirelength and channel density minimization, there is no direct comparison of this approach with the commonly used flow with separate placement and routing.

In general, the research in the area of combined placement and routing has had only very limited success so far. With modern FPGAs that have a much higher logic capacity and much richer routing resources, one may question whether it is feasible to compute or even still necessary to carry out simultaneous placement and routing.

3

Technology Mapping

FPGA technology mapping transforms a network of technology-independent logic gates into one comprised of logic cells in the target FPGA architectures. In a typical FPGA design flow, mapping is the last step in which the design is transformed. As a result, technology mapping has a significant impact on the quality (area, performance, power, etc.) of the final implementation.

Technology mapping for FPGAs is a subject of extensive study. Many algorithms have been proposed and various techniques have been developed. Technology mapping algorithms can be classified in several different ways. One classification is based on the optimization objectives: area [72, 89, 152], timing [38, 46, 50, 58, 88, 152], power [13, 42, 86, 121, 136, 195], and routability [167]. Another classification is based on the type of transformation techniques employed during mapping. Algorithms can be structural or functional. The structural approach does not modify the input netlist other than logic duplication [46, 50]. It reduces technology mapping to a problem of covering the input netlist with logic cells (e.g., K-LUTs) of the target FPGAs. Due to their combinatorial nature, structural mapping algorithms are efficient for large designs. Functional approach, on the other hand,

treats technology mapping as Boolean transformation/decomposition of the input design into a set of interconnected logic cells [126, 152]. Functional mapping mixes Boolean optimization with covering. It can potentially explore a larger solution space than structural mapping. However, functional mapping algorithms tend to be time-consuming, which limits their use to small designs or small portions of a large designs. Recently, algorithms have been proposed to explore functional mapping in the context of structural mapping to take advantages of both structural and functional approaches [43, 49, 148].

Technology mapping algorithms can also be classified according to the types of input networks. Algorithms for combinational networks assume fixed positions for sequential elements and only consider the combinational logic between sequential elements [50, 85]. Algorithms for sequential networks, on the other hand, may relocate the sequential elements using retiming during mapping [71, 159]. Such algorithms can explore a much larger solution space to derive mapping solutions with better quality. Algorithms for combinational networks can further be divided into those for general DAGs [50] and those for special networks such as *tree* and *leaf-DAGs* [85, 89]. Algorithms for special networks can be applied to general networks through partitioning, which obviously can compromise the solution quality.

Recent advances in technology mapping try to combine mapping with up-stream and/or down-stream optimization steps in the design flow. Such integrated algorithms have the potential for exploring large solution spaces to arrive at mapping solutions with better overall quality. Algorithms have been proposed to combine mapping with retiming [71, 160, 70, 146], with synthesis and decomposition [43, 59, 148], and with clustering and placement [139, 140].

We will be focusing on recent advances in FPGA technology mapping. For early technology mapping works, the reader is referred to the excellent and comprehensive survey in [52].

3.1 Preliminaries

The input design to technology mapping is a network consisting of logic gates and sequential elements. The network can be represented

by a directed graph where the nodes denote logic gates. There is a directed edge (i, j) if the output of gate i is an input to gate j. An edge may have a weight that represents the number of sequential elements on the connection. A weight of zero means the connection is combinational without any sequential element. Most mapping algorithms operate only on the combinational portion of the network which is a DAG obtained by removing sequential elements. Each removed sequential element introduces a (pseudo) PI—the output of the sequential element, and a (pseudo) PO—the input of the sequential elements. When describing technology mapping for combinational circuits, we make no distinction between a network and its combinational portion. We also make no distinction between pseudo PIs/POs and the real ones. The following concepts are defined mainly for combinational technology mapping although many of them are applicable to sequential networks. We use input(ν) to denote the set of nodes that are fanins of gate ν and use output(ν) to denote the set of nodes that are fanouts of gate ν. We use O_ν to denote a fanin cone rooted at node ν. O_ν is a sub-network consisting of ν and some of its predecessors, such that for any node $u \in O_\nu$, there is a path from u to ν that lies entirely in O_ν. The maximum cone of ν, consisting of all the predecessors of ν, is called a *fanin cone* of ν, denoted as F_ν. A *fanout-free cone* is a cone in which the fanouts of every node other than the root (node) are inside the cone, i.e., all its paths converge to the root. For each node ν, there is a unique maximum fanout-free cone [52], denoted MFFC_v, which contains every fanout-free cone rooted at ν. We use input(O_ν) to denote the set of distinct nodes outside of O_ν and supplying inputs to one or more gates in O_ν. O_ν is K-feasible if $|\text{input}(O_\nu)| \leq K$. A *cut* is a partition (X, X') of the fanin cone F_ν of ν such that X' is a cone of ν. We call input(X') the *cutset* of the cut. A cut is K-feasible (or a K-cut) if X' is a K-feasible cone, or equivalently, X' is a K-LUT that implements ν with the inputs in the cutset. For convenience, we will use cuts, cutsets, cones, and LUTs interchangeably when the meaning is clear. Finally, a Boolean network is *t-bounded* if $|\text{input}(\nu)| \leq t$ for each node ν.

Most mapping algorithms are structural and view mapping as a covering problem by covering a network of logic gates using K-feasible cones which can then be implemented by K-LUTs. A mapping solution

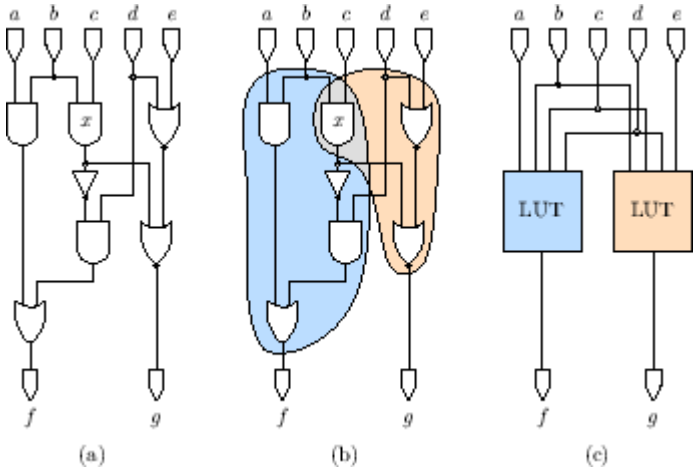


Fig. 3.1 Structural technology mapping: (a) Original network; (b) Covering using K-feasible cones, (c) Mapping solution derived from the covering [141].

is simply a network of K-LUTs where there is an edge (O_u, O_ν) if u is in input(O_ν) or equivalently, u is an input to the LUT selected for ν. Figure 3.1 is an example of structural mapping. The original network in (a) can be covered by two 4-feasible cones as indicated in (b). Note that node x is included in both cones and will be duplicated. Some of the nodes are completely covered and no LUTs are needed for them in the final mapping solution. The corresponding mapping solution is shown in (c).

3.2 Structural Mapping Framework

Structural mapping is done as part of a logic synthesis flow which typically consists of three steps. First, the initial network is optimized using technology-independent optimization techniques such as node extraction/substitution, don't-care optimization, and timing-driven logic restructuring. The optimized network is then decomposed into a two-bounded network to give maximum flexibility for the ensuing technology mapping step. Several decomposition techniques have been proposed. They include the Huffman-tree-like AND/OR decomposition algorithm *dmig* [46], and bin packing-based algorithms [89]. The final

step, called structural mapping, is to cover the two-bounded network with K-LUTs to optimize one or more objectives, such as timing and area among others.

Most structural mapping algorithms are based on dynamic programming and consist of the following steps:

- Cut generation/enumeration
- Cut ranking
- Cut selection
- Final mapping solution generation

Cut generation produces one or more cuts for cut selection and mapping solution generation. *Cut ranking* evaluates generated cuts to see how good they are for timing and/or area. Cuts are normally evaluated following a topological order of the nodes from PIs to POs. *Cut selection* picks a "best" cut for each node based on the ranking info. It is typically done in a reverse topological order from POs to PIs. Cut ranking and selection may be done multiple times to refine the mapping solution successively.

We will first discuss cut generation and general ideas in cut ranking. Cut selection and enhancements to cut ranking will be discussed when we present details of some of the mapping algorithms.

3.2.1 Cut generation

Early mapping algorithms mix cut generation and ranking to generate one or a few "good" cuts for each node. The most successful example is the FlowMap algorithm, which finds a single cut with optimal mapping depth at each node based on max-flow computation [50, 48]. It computes the optimal mapping depth of each node in the topological order from the PIs to POs based on dynamic programming. At each node, it uses max-flow computation to test if the current node can have the optimal mapping depth as its predecessors or have to be incremented by one, which were shown to be the only two possible mapping depths at the node. FlowMap algorithm was the first polynomial-time algorithm that computes a depth-optimal mapping solution for K-LUT based FPGAs.

Because K is a small constant in practice, most recent mapping algorithms compute all K-cuts for each node before selecting cuts to cover the nodes. With all cuts available during covering, we have the added flexibility in selecting cuts to optimize multiple and/or complex objectives.

For combinational mapping, cuts can be enumerated by a topological traversal of the nodes from PIs to POs [72, 167]. The result of cut enumeration is a set of K-feasible cuts for each node. For a PI, the set of cuts contains only the trivial cut consisting of the node itself. For an internal node ν with two fanins, u_1 and u_2, the set of cuts $\Phi(\nu)$ is computed by merging the sets of cuts of fanin nodes u_1 and u_2 as follows:

$$\Phi(\nu) = \{\{\nu\} \cup \{c_1 \cup c_2 | c_1 \in \Phi(u_1), c_2 \in \Phi(u_2), |c_1 \cup c_2| \leq K\}.$$

In other words, the set of cuts of a node can be obtained by the pair-wise union of the cuts of its fanins and drop those that are not K-feasible. For propagation purpose, we also add the trivial cut of each node to its set of cuts. In practice, the set of cuts, $\Phi(\nu)$, may contain dominated ones which are supersets of other cuts. Dominated cuts can be removed without impacting the quality of mapping solutions.

Once we have the set of all cuts for each node, a mapping algorithm will select a cut for each node to cover the network. To help choose cuts to cover the network, mapping algorithms evaluate and rank the cuts based on the mapping objectives. Criteria for ranking cuts are discussed in the following section.

3.2.2 Cut ranking — area

For LUT mapping, the area of a mapping solution can be measured by the total number of LUTs. Area minimization for LUT mapping has been shown to be NP-hard [85]. Therefore, it is unlikely that there is an efficient and yet accurate way to rank cuts for area. The difficulty of area estimation during mapping is mainly due to the existence of multiple fanout nodes and their reconvergence [51].

In [72], the authors proposed the concept of *effective area* as a heuristic to measure the area cost of a cut. (A similar concept called

area flow was later proposed in [143].) The intuition of effective area is to distribute the area cost of each node to its fanout edges to consider logic sharing and reconvergence during area cost propagation. The effective areas are computed in topological order from PIs to POs. For each PI ν, its effective area $a(c)$ is set to zero. The following formula is used to compute the effective area of a cut:

$$a(c) = \sum_{u \in c} [a(u)/|\text{output}(u)|] + A_c,$$

where A_c is the area of the LUT corresponding to the cut c. The area cost of a non-PI/PO node is then the minimum area of its cuts:

$$a(\nu) = \min\{a(c) \,|\, \forall u \in \Phi(\nu)\}.$$

It can be shown [72] that for duplication-free mapping based on MFFCs, effective area is accurate in that there is a mapping solution whose area is equal to the sum of the effective areas of the POs. Since the effective area is computed by distributing the area of a node evenly among its fanouts, it does not account for the situation where the node may be duplicated in a mapping solution. When there is duplication, effective area may be inaccurate. In the example shown in Fig. 3.2, with $K = 3$, the LUT for u covers w. In this case, the LUT for w is introduced solely for the LUT for ν. However, in effective area computation, only one half is counted for ν. As a result, the LUT for w is under-counted. In this particular case, the effective area of the overall mapping solution (sum of the effective areas of the POs) is 2.5 while the mapping solution has three LUTs. In general, effective area is a lower bound of the actual area [72].

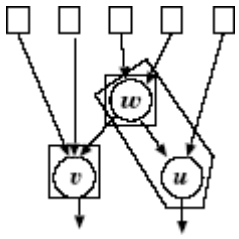


Fig. 3.2 Inaccuracy in effective area when duplication is allowed [72].

3.2.3 Cut ranking — timing

Because the exact layout information is not available during the technology mapping stage, most FPGA technology mapping algorithms only consider the cell delays. The delay of a mapping solution is defined as the total cell delays on a longest combinational path from PIs to POs.

Most delay optimal combinational mapping algorithms use the dynamic programming-based labeling process introduced in the FlowMap algorithm [50]. The label at each node is the minimum arrival time that can be achieved for the node in any mapping solution. The label of a PI is set to zero assuming that the signal arrives at the PI at the beginning of the clock edge. After the labels for all the nodes in F_ν except ν itself are found, the label of gate ν can be computed by first calculating the label of each cut c of ν using the following formula:

$$l(c) = \max\{l(u) + D_c \,|\, \forall u \in c\},$$

where D_c is the delay of the LUT corresponding to the cut c. Intuitively, $l(c)$ is the arrival time at ν if ν is covered using the cut c. The best arrival time at ν is the smallest label among all its cuts, i.e.,

$$l(\nu) = \min\{l(c) \,|\, \forall c \in \Phi(\nu)\}.$$

To obtain a delay optimal mapping solution, one can follow the reverse topological order starting from POs going backward to PIs. At each node, select a cut with the label, then trace back to the nodes in the cut. This process is continued until we reach the PIs. At that point, a complete mapping solution with best delay is obtained.

3.3 Structural Mapping Algorithms

In this section, we present recent advances in structural technology mapping algorithms based on the framework presented in the preceding section. For early mapping algorithms, the readers are referred to [52]. We will discuss mapping algorithms for area optimization first, then for timing optimization.

3.3.1 Area optimization

Area optimal mapping is an NP-hard problem [85]. The problem can be formulated as a binate covering problem [152]. Exact solution of the binate covering problem takes exponential time.[1] For practical designs, we have to resort to heuristics.

The PRAETOR algorithm [72] is an area-oriented mapping algorithm based on cut enumeration and ranking. In addition to the area-based cut ranking discussed earlier, the PRAETOR algorithm presents a number of techniques for both additional area reduction and possible runtime improvement. One technique is to encourage the use of common sub-cuts. A cut for a fanout of a node ν induces a cut for ν (maybe the trivial cut consisting of ν itself). If two fanouts of a node induce different cuts for the node, this will most likely result in area increase due to the need to duplicate ν and possibly some of its predecessors. To alleviate this problem, the PRAETOR algorithm sorts and selects cuts with the same effective area in a predetermined order to avoid arbitrary selection. It assigns an integer ID to each node. Then, all cuts with the same effective area are sorted according to the lexicographic order based on the IDs of the nodes in the cuts. If we choose the first cut with minimum effective area for each node, different fanouts of the same node tend to use the same cut for the node. Therefore, the final mapping solution will have a smaller area.

Another area reduction technique introduced in the PRAETOR algorithm is to carry out cut selection twice. The purpose of the first pass is to generate candidate LUT roots that will be declared non-duplicable. Non-duplicable nodes will become cut boundaries. Any cut that contains non-duplicable nodes will be dropped in the second pass of cut selection. By doing so, we not only exclude those cuts with possible duplication, but also encourage cuts with less duplication. For example in Fig. 3.3, in the first cut selection, we may generate a mapping solution as shown in (a) with four LUTs. In the second pass, the cut including ν for u_1 will be excluded from the set of cuts for ν_1. As a

[1] In the case of tree networks, area-optimal mapping can be solved efficiently using dynamic programming [85]. An exponential algorithm can be used to solve the general problem optimally for small designs, e.g., the ILP-based solution in [47].

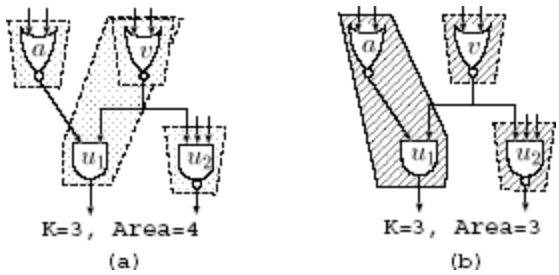


Fig. 3.3 Effect of excluding cuts with non-duplicable nodes [72].

result, we remove the duplication of ν and at the same time encourage the cut including a for u_1 to finally arrive at the mapping solution in (b) with three LUTs. Experimental results show that the PRAETOR algorithm can significantly improve area over previous algorithms. Moreover, it can achieve results that are only 14% larger than the effective areas, which, as mentioned earlier, is the lower bound on the optimal areas.

The IMap algorithm proposed in [143] is another mapping algorithm that targets for area reduction. The two enhancements to the basic framework are: 1) it iterates between cut ranking and cut selection multiple times; and 2) it adjusts the area ranking between successive iterations using history information. In the effective area formula, we use $|\text{output}(\nu)|$, i.e., the number of fanouts of ν in the initial network, to divide up the effective area for the node ν. Ideally, it should be the fanout count of the node, if it exists, in the final mapping solution, which is not available until cut selection is done. In IMap, between iterations, the fanout count estimation is also updated by using a weighted combination of the estimated fanout count and the real fanout count in previous iterations. The formula is as follows:

$$\text{estimated\_fc}(\nu) = (\text{estimated\_fc}_{\text{prev}}(\nu) + \alpha\,|\text{output}(\text{LUT}_\nu)|)/(1+\alpha),$$

where $\text{estimated\_fc}(\nu)$ is the estimated fanout count for current iteration, $\text{estimated\_fc}_{\text{prev}}(\nu)$, the estimated fanout count for the previous iteration; $\text{output}(\text{LUT}_\nu)$ is the actual fanout count of the previous mapping solution; and α is a weighting factor. Note that for nodes that are fully covered in the previous mapping solution, i.e., no LUT is gen-

erated for them, is empty. At the beginning, estimated_fc(ν) is set to |output(ν)|. The following formula is used for the area cost of a cut c of ν:

$$a(c) = \sum_{u \in c} [a(u)/\text{eatimated\_fc}(u)] + A_c.$$

The authors suggest setting α to be between 1.5 and 2.5 when the number of iterations is limited to 8. When applying the enhanced area cost on delay-oriented mapping, the authors show good improvement in area compared to previous delay-oriented mapping algorithms.

In [141], the authors present a mixed structural and functional area mapping algorithm based on solving a sequence of SAT problems. The algorithm starts with an existing mapping solution (e.g., obtained from a structural mapper described earlier). The key idea is a SAT formulation for the problem of mapping a small circuit into the smallest possible number of LUTs. The algorithm iteratively selects a small logic cone to remap to fewer LUTs using a SAT-solver. It is shown that for some highly structured (albeit small) designs, area can be improved significantly.

Most area optimization techniques are heuristics. A natural question is how close or far away existing mapping algorithms are from optimal. In [66], the authors construct a set of designs with known optimal area mapping solutions (called LEKO examples). They tested existing academic algorithms and commercial tools on the LEKO examples. The average gap from optimal varies from 5 to 23%, with an average of 15%. From the LEKO examples, they further derived the logic synthesis examples with known upper bounds (LEKU). These examples require both logic optimization and mapping. Existing FPGA synthesis algorithms and tools perform very poorly on LEKU examples, with an average optimality gap of over 70X. This indicates that further studies are needed for area-oriented mapping and optimization.

3.3.2 Delay minimization

Timing optimization is important for FPGAs due to the performance disadvantage introduced by programmability. FlowMap and its deriva-

tives can find a mapping solution with optimal delay. Recent advances in delay minimization focus on optimizing area while maintaining performance.

DAOmap [38] is a mapping algorithm that guarantees optimal delay, while at the same time reducing the area significantly compared to previous delay-oriented mapping algorithms. The DAOmap algorithm introduces three techniques to optimize area without degrading timing. First, it enhances effective area computation to control potential node duplications more effectively. Second, it exploits the extra timing slacks on non-critical paths for area reduction. It uses an iterative cut selection procedure to further explore and perturbs the solution space to improve solution quality.

In DAOmap, the effective area for a cut c of node ν is enhanced using the following formula:

$$a(c) = \sum_{u \in c} [a(u)/|\text{output}(u)|] + U_c + P_{u_1} + P_{u_2}$$

where U_c is the area contributed by the cut itself and P_{u_1} and P_{u_2} are correction terms to account for potential duplication from the fanins u_1 and u_2 of ν. Specifically, the following formula is used,

$$U_c = \alpha|c|/(N_c + \beta(|\text{output}(\nu)| + R_c)$$

where N_c is the number of nodes in the cone of the cut and R_c is the number of reconvergent paths completely covered by c. Parameters α and β are two weighting factors determined empirically. The intuition of the formula is to encourage the use of a small cut (in terms of the number of cut nodes) that covers a large cone. Obviously, such cuts have fewer chances to introduce unnecessary logic duplication. The correction terms $P_{u_i} (i = 1 \text{ or } 2)$ are introduced to gauge the potential of node duplication at the fanin u_i of ν. If ν has only one fanout, they are set to zero; otherwise, $P_{u_i} = N_{c_i}/|c|$, where c_i is the cut induced on u_i by c. The intuition is that the larger N_{c_i} is, the more likely the nodes in c_i will be duplicated. The size of the cut $|c|$ is added as a normalizing factor.

DAOmap first picks cuts with minimum timing cost for each node. Among all cuts with minimum timing cost, it then picks a cut with

minimum area cost. However, when there is extra slack (the difference between required time and arrival time from the timing labels), it will pick a cut with minimum area cost as long as the timing increase doesn't exceed the slack. Cut selection starts from POs and works backward toward PIs.

Recognizing the heuristic nature of the area cost, DAOmap also employs the technique of multiple passes for cut selection (i.e., mapping generation). Moreover, DAOmap adjusts area costs based on input sharing to encourage using nodes that have already been contained in other cuts. This reduces the chance that a newly picked cut cuts into the interior of existing LUTs. Between successive iterations of cut selection, it also adjusts area cost to encourage using LUT roots with a large number of fanouts in previous iterations. There are also a few other secondary techniques used in DAOmap. The interested reader is referred to [38] for details.

Based on the results reported, DAOmap can improve the area by about 13% compared to previous mapping algorithms for optimal depth. It is also many times faster than previous flow computation based mapping algorithms, mainly due to efficient implementation of cut enumeration.

A more recent work [148] introduces several techniques that further improve the area while preserving delay optimality. As DAOmap, this algorithm also goes through several passes of cut selection. Each pass selects cuts with better areas among the ones that do not violate optimal timing. The basic framework is also based on the concept of effective area (or area flow). However, it processes nodes from PIs to POs, instead of from POs to PIs during cut selection. With this processing order, the algorithm tries to use extra slack on nodes close to PIs to reduce area cost. This is based on the observation that logic is typically denser close to PIs. Delay relaxation is more effective following the topological order from PIs to POs.

The algorithm [148] also uses a local heuristic for area recovery during cut selection. It introduces the concept of *exact area of a cut* which is defined as the number of LUTs to be added to the mapping solution if the cut is selected for the node. The algorithm tries to improve the current mapping solution by selecting a cut with minimum exact

area at each node without violating timing. An iterative procedure is proposed to calculate the exact area for each cut. Experimental data show 7% better area over DAOmap with the same timing.

The cut enumeration framework can also be used for technology mapping for power minimization, which will be discussed in Sections 6.4 and 6.5.

3.4 Integrated Technology Mapping

Technology mapping is a step in the middle of a typical FPGA CAD flow with optimization steps before and after it. We carry out technology-independent optimization and logic decomposition before technology mapping. We may do sequential optimization such as retiming after (or before) mapping. A separate approach can miss the best overall solutions even if we can solve each individual step optimally. To obtain better overall solutions, it is desirable to combine some of the optimization steps with mapping. In this section we discuss mapping algorithms that integrate with decomposition, logic synthesis, and retiming. Mapping has also been integrated with clustering and placement; these algorithms will be presented when we discuss layout-driven synthesis.

3.4.1 Simultaneous logic decomposition and mapping

It is advantageous to decompose an input netlist into a two-bounded one before technology mapping. This is because a mapping solution for the original network can always be found in the decomposed one.

A network of complex gates (in SOP form) can be turned into a network of simple gates (e.g., AND, OR, XOR, and INV gates) by expressing each SOP as a set of AND-OR gates (e.g., using the *tech_decomp* algorithm in SIS [169]). Structural decomposition then decomposes the network of simple gates into two-bounded simple gates using associativity. The *dmig* algorithm is one such method; it tries to minimize the depth of the decomposed two-input network using Huffman-tree construction [52]. However, the resulting decomposition may not be the best as far as the final mapping solution is concerned. This is because

the depth of the two-bounded network is not an exact predictor for the depth of the final mapping solution.

In general, different decompositions can significantly impact the final mapping solution. To find the best mapping solution for the initial network, it is ideal that we combine decomposition with mapping by finding a structural decomposition such that the final mapping solution of the decomposed network has minimum depth among all possible decompositions of the input network. This problem was shown to be NP-hard [59].

An algorithm for integrated structural decomposition and mapping is proposed in [43]. The algorithm is based on the concept of *choice nodes* that were originally introduced for combining decomposition and technology mapping for standard cells [127, 128]. A choice node is not a physical gate. It is introduced to group all functionally equivalent, but structurally different decompositions of a node [127]. Each choice node has a complement copy, both of which form a so-called *ugate*. Fig. 3.4 shows an ugate formed by two choice nodes c_1 and c_2 that are the complement of each other.

The algorithm essentially encodes all possible structural decompositions of the input network in a concise *mapping graph*. The mapping graph is formed by adding the choice nodes to represent the

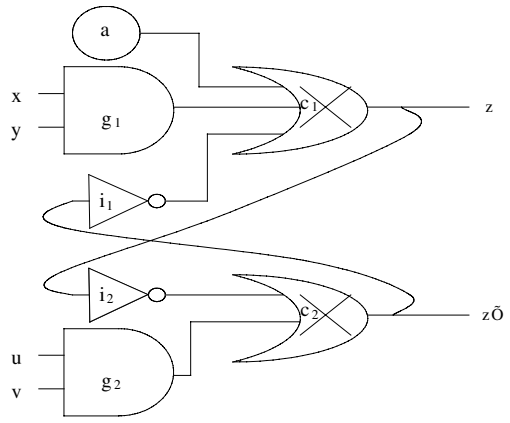


Fig. 3.4 An ugate example [43].

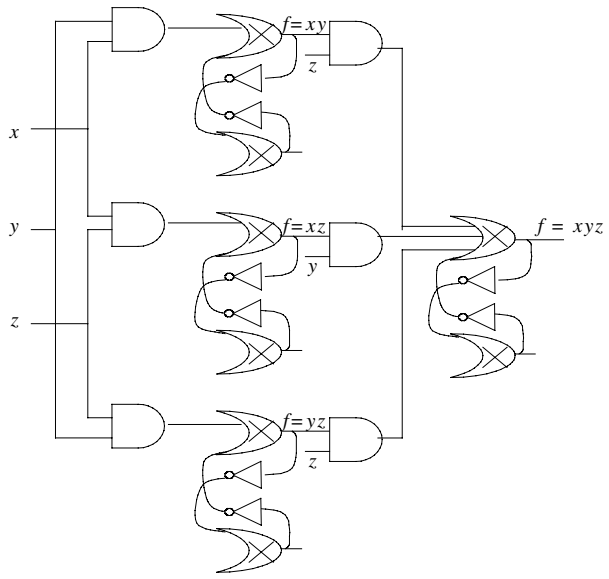


Fig. 3.5 A mapping graph for all decompositions of function xyz [43].

decomposition choices. Figure 3.5 is a mapping graph that encodes all possible decompositions of Boolean function $f = xyz$. By picking one branch at each choice node in the mapping graph, we can retrieve one decomposition of the function.

Cut generation and cut ranking can be extended to choice nodes. For example, the set of cuts of a choice node is simply the union of the sets of cuts of all its fanins. Similarly, the label of a choice node is the smallest one among the labels of its fanins. The rest of the approach is similar to a conventional mapping algorithm.

The algorithm does not actually generate a mapping solution directly from the mapping graph. Instead, it determines a decomposition from the mapping graph by selecting a decomposition with best timing label at each node. After that, it applies a state-of-the-art mapping algorithm for a fixed decomposition to obtain the final mapping solution. Significant timing and area improvements were observed compared to separate decomposition and mapping [43].

3.4.2 Simultaneous logic synthesis and mapping

Technology-independent Boolean optimizations carried out before technology mapping in a conventional flow can significantly change the network passed to mapping; so does the final mapping solution. During technology-independent optimization, we have freedom to change the network structures, but accurate estimation of the impact to downstream mapping is not available. During technology mapping, we can achieve optimal or close to optimal solutions using one of the algorithms discussed earlier. However, we are stuck with a fixed network. It is desirable to capture the interactions between logic optimization and mapping to arrive at a solution with better quality.

Lossless synthesis has been proposed as way to consider technology-independent optimization during mapping [148]. Lossless synthesis is based on the concept of choice networks; this is similar to the mapping graphs in [127, 128]. As mapping graphs, a choice network contains choice nodes which encode functionally equivalent, but structurally different alternatives. The algorithm operates on a simple yet powerful data structure called AIG which is a network of AND2 and INV gates. A combination of SAT and simulation techniques are used to detect functionally equivalent points in different networks and compress them

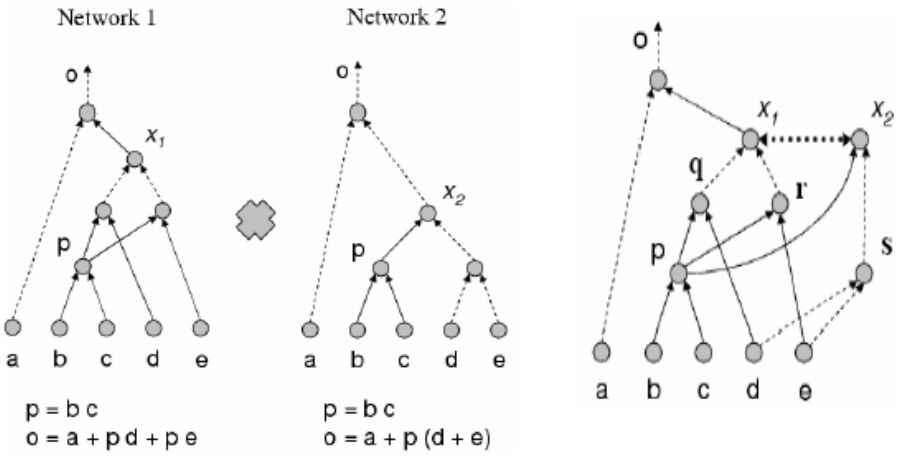


Fig. 3.6 Combining networks to create a choice network [148].

to form one choice network. Figure 3.6 illustrates the construction of a network with choices from two equivalent networks with different structures. The nodes x_1 and x_2 in the two networks are functionally equivalent (up to complementation). They are combined in an equivalence class in the choice network, and an arbitrary member (x_1 in this case) is set as the class representative. Node p does not lead to a choice because its implementation is structurally the same in both networks. Note also that there is no choice corresponding to the output node o since the procedure detects the maximal commonality between the two networks.

Using structural choices leads to a new way of thinking about logic synthesis: rather than trying to come up with a "good" final netlist to use as an input to mapping, the algorithm in [148] accumulates choices by combining intermediate networks seen during logic synthesis to generate a network with many choices. In a sense, it does not make judgments on the goodness of the intermediate networks or portions of the networks and defers the decision to the mapping phase. The best combination among these choices is selected during mapping. In the final mapping solution, different sections may come from different intermediate networks. For example, the timing-critical sections of the final mapping solution may come from networks which are optimized for timing, while the timing non-critical sections of the final mapping solution may come from networks which are optimized for area.

Cut generation and ranking techniques are extended to network with choices as in the case of integrated decomposition and mapping. Results reported in [148] show that the proposed algorithm can improve both area and timing by 7% on a large set of benchmark designs over mapping solutions produced using just one "optimized" network as the input.

3.4.3 Simultaneous retiming and mapping

Retiming is an optimization technique that relocates flip-flops (FF) to improve the performance or area of a design while preserving its functionality [129]. Retiming can shift FF boundaries and change the

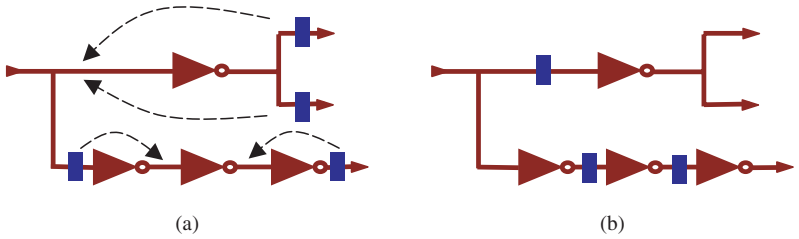


Fig. 3.7 A retiming example: (a) The initial design; (b) The retimed design.

timing of a design. For the design (a) in Fig. 3.7, if we relocate the FFs as indicated, we arrive at the design in (b). If we assume each gate has one unit of delay, the original design has a clock period of three, while the retimed design has a clock period of one. The number of FFs is also reduced from four to three.

If we apply retiming after mapping, mapping may optimize the wrong paths, because the critical paths seen during mapping may not be critical after retiming. If we do retiming before mapping, retiming will be carried out using less accurate timing information since the design is not mapped. In either case, we cannot account for the impact of retiming on the cut or LUT generation as logic can be shifted from one side of FFs to the other. All of these point to the importance of combining retiming and technology mapping.

In [159], the authors propose the first polynomial time mapping algorithm that can find the best clock period in the combined solution space of retiming and mapping. In other words, the mapping solution obtained at the end is the best among all possible ways of retiming and mapping a network. The algorithm is based on two important concepts: *sequential arrival times* and *expanded cones* (circuits). Sequential arrival times anticipate the impact of retiming on performance without actually doing retiming. Expanded cones at a node make it possible to form cuts across time frames. So cut generation is oblivious of register boundaries.

Although the algorithm in [159] has polynomial time complexity, the runtime can be high. An improved algorithm is proposed in [71] that significantly reduces the runtime while still preserving the optimality of the final mapping solution. Both algorithms are based on flow com-

putation, as in FlowMap. To further improve runtime in practice, an algorithm based on cut enumeration is proposed in [158]; this will be discussed next.

FF boundaries are not fixed anymore with retiming. Cut generation is extended to go across FF boundaries to generate sequential cuts [158]. In a sequential design, a gate may go through zero or more FFs in addition to logic gates before reaching gate ν. To capture this information, an element in a cut for a node ν is represented as a pair consisting of the driving gate u and the number of FFs d on the paths from u to ν. The element will be denoted by u^d. Note that a node may reach a root node through paths with different FF counts. In that case, the node will appear in the cut multiple times with different d values. Here is the formula relating the set of cuts of a node ν to its two fanins u_1 and u_2:

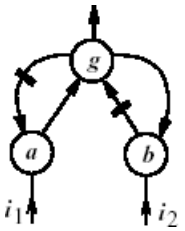
$$\Phi(\nu) = \{\{\nu^0\} \cup \{c_1^{d_1} \cup c_2^{d_2} | c_1 \in \Phi(u_1), c_2 \in \Phi(u_2), |c_1^{d_1} \cup c_2^{d_2}| \leq K\},$$

where d_i is number of FFs from u_i to ν and $c_i^{d_i} = \{u^{d+d_i} | u^d \in c_i\}$ for $i = 1, 2$.

Unlike the combinational case, the above formula does not give us a direct way to compute all cuts because a general sequential design may contains loops, so the sets of cuts are inter-dependent. A procedure is proposed in [158] to determine the sets of cuts for all nodes through successive approximation. The procedure starts with $\Phi(\nu)$ containing the trivial cut $\{\nu^0\}$ for each node ν, and then updating cuts using the above formula by going through all nodes in passes until no new cut is discovered. Figure 3.8 shows an example. For the design on the left, the table on the right shows three iterations in cut generation. In the first iteration, every node has the trivial cut. Row 1 shows the new cuts discussed in the first iteration. In iteration 2, two more cuts are discovered, one for a and one for b. After that, further cut combination does not yield any new cut, and the procedure stops. In practice, cut generation stops very quickly. For example, cut generation stops in, at most five iterations for all ISCAS89 benchmarks with $K = 4$.

To consider retiming effect, the concept of (timing) labels is extended to that based on sequential arrival times [160, 159]. The label of a cut c is now defined as follows:

$$l(c) = \max\{l(u) - d \cdot \phi + 1 | u^d \in c\}$$



itr	i_1	i_2	a	b	g
0	$\{i_1^0\}$	$\{i_2^0\}$	$\{a^0\}$	$\{b^0\}$	$\{g^0\}$
1			$\{i_1^0, g^1\}$	$\{i_2^0, g^0\}$	$\{a^0, b^1\}$ $\{a^0, i_2^1, g^1\}$ $\{i_1^0, g^1, b^1\}$ $\{i_1^0, g^1, i_2^1\}$
2			$\{i_1^0, a^1, b^2\}$	$\{i_2^0, a^0, b^1\}$	

Fig. 3.8 Example of iterative cut generation for sequential circuits [158].

where ϕ is the target clock period. The label of a gate ν is then defined as,

$$l(\nu) = \min\{l(c) | \forall c \in \Phi(\nu)\}.$$

The label for each PI is zero, and the label for each PO is that of its driven gate.

An algorithm is proposed in [158] to find the labels for cuts and nodes. Due to the cyclic nature of general sequential designs, the labels cannot be determined in one pass, as in the case of combinational networks. They are computed through successive approximation by going through the nodes in several passes. At the beginning, the labels for all nodes are set to $-\infty$ except PIs whose labels are always zero. The successive improvement stops if either one of the POs has a label larger than ϕ or no more change in the labels is observed. It is shown in [161] that *the initial design has a mapping solution with a clock period ϕ or less among all possible retiming and mapping if the label of each PO is less than or equal to ϕ.*

After the labels for all nodes are computed and the target clock period is determined to be achievable, we can generate a mapping solution. As in the combinational case, the algorithm generates a mapped network starting from POs and going backward. At each node ν, it selects one of the cuts that realizes the label of the node, and then moves on to select a cut for u if u^d is in the cut selected for ν. On the edge from u to ν, d FFs are added. To obtain the final mapping solution with the clock period ϕ, it retimes the LUT for each non-PI/PO node ν by $\lceil l(\nu)/\phi \rceil - 1$.

Heuristics are used to reduce the LUT counts in the mapping solution [158]. Experimental results show that the algorithm is very efficient and consistently produces mapping solutions with better performance than optimal combinational mapping followed by optimal retiming.

3.5 Specialized Mapping Algorithms

In previous sections, we discussed LUT mapping with a single value K. In reality, FPGA devices typically contain heterogeneous resources e.g., embedded memory blocks and LUTs of difference input sizes. There are also commercial FPGA architectures with logic cells that can only implement a subset of functions of their inputs. In this section, we briefly summarize mapping algorithms for these special architecture features.

3.5.1 Mapping for FPGAs with heterogeneous resources

In this subsection, we summarize FPGA technology mapping algorithms for heterogeneous resources. We first discuss mapping for architectures with different sizes of LUTs. Then, we examine the problem of mapping logic to on-chip memory.

Mapping with different LUT sizes: There are a number of commercial FPGA architectures that can support LUTs with several different input sizes. The adaptive logic modules (ALMs) in Altera's Stratix II devices can be configured to two 4-LUTs, one 5-LUT and one 3-LUT, and certain 6/7-LUTs. Other architectures such as Xilinx Virtex II, Virtex 4 devices can also implement LUTs with different input sizes.

Without loss of generality, we assume there are two types of LUTs with sizes K_1 and K_2 and delays d_1 and d_2, respectively. We further assume $K_1 < K_2$ and $d_1 < d_2$. A number of mapping algorithms have been proposed for mapping to such architectures: for area [72, 101, 107, 116], for delay [73, 76].

For area minimization, the PRAETOR algorithm discussed earlier has also been applied to such architectures using different area costs for different LUTs. Good improvements are observed over a previous

algorithm. In the special case of tree networks, a polynomial optimal algorithm has also been proposed based on dynamic programming [116].

For timing optimization, the algorithm proposed in [73] is an extension of FlowMap. As FlowMap, the algorithm is also based on flow computation. The basic ideas in the algorithm can also be cast in the cut generation framework by enumerating all K_2 cuts. We can just extend the timing cost by replacing D_c in the formula with d_1 or d_2 depending on the cut size. We set D_c to d_1 if $|c| \leq K_1$; otherwise, we set it to d_2. With this simple modification, an algorithm for homogeneous LUT architectures can be used for architectures with different LUT sizes.

When there are resource bounds on the available LUTs of different sizes, the mapping problem becomes harder since mapping with area bound is NP-hard in general. Two heuristic algorithms are proposed for the case in which there can be at most r K_2 LUTs [74]. The algorithm BinaryHM [74] employs a mapping algorithm that does not consider resource limitation. It calls on the algorithm repeatedly to bring resource utilization under control. The basic idea is as follows. Let D_{FM} be the delay obtained using only K_1 LUTs, and D_{HM} be the delay obtained using both K_1 and K_2 LUTs with no resource limitation. Obviously, D_{FM} and D_{HM} are upper and lower bounds on the delay with the resource limitation, respectively. If the solution with delay D_{HM} meets the resource bound, the best solution is found. If not, one can increase d_2, the delay of K_2 LUT, and solve the unconstrained version again, which should result in a mapping solution with fewer K_2 LUTs. Binary search is used, and the change in d_2 is done through adjusting the ratio of d_1 and d_2. The reader is referred to [74] for more details.

Mapping with embedded memory blocks: On-chip memory has become an essential component of high-performance FPGAs. Dedicated embedded memory blocks (EMBs) can be used to improve clock frequencies and lower costs for large systems that require memory. If a design does not need all the available EMBs, unused EMBs can be used to implement logic, since they typically can be configured as ROMs on

most commercial FPGA devices, which essentially turns EMBs into large multi-input multi-output LUTs.

EMBs usually have configurable widths and depths. They can be used to implement functions with different numbers of inputs/outputs. For example, a 2K-bit memory with configurations, $2048 \times 1, 1024 \times 2$, and 512×4 can be used to implement an 11-input/1-output, 10-input/2-output, or 9-input/4-output logic function. Using EMBs for logic can reduce interconnect delays since they can potentially replace many small LUTs. On the other hand, EMBs usually have big internal delays (memory access time). Care must be taken to map logic to EMBs. Several mapping algorithms have been proposed to take advantage of unused EMBs [75, 198, 200, 199]. Mapping logic to EMBs is typically done as a post-processing step after LUT mapping. It starts with an optimized mapping solution for LUTs and then packs groups of LUTs for EMB implementation.

The algorithm SMAP in [198] maps one EMB at a time. It begins by selecting a seed node. A fanin cone of the seed node is generated by finding a d-feasible cut that covers as many nodes as possible, where d is the bit width of the address line of the target EMB. Since d is considerably larger than the typical LUT input size, flow-based cut generation is used. After the cone is generated, the output selection process selects signals to be the outputs of the EMB. Output selection tries to select a set of signals so that the resulting EMB can eliminate as many LUTs as possible. Each node is assigned a score that is equal to the number of nodes in its MFFC within the cone. The w highest-scoring nodes are selected as the EMB outputs, where w is the number of outputs of the target EMB. The selection of the seed node is critical for this method. The algorithm tests each candidate node and selects the one that leads to the maximum number of eliminated LUTs. Heuristics are introduced to consider EMBs with different configurations and to preserve timing.

Another algorithm, EMB_Pack, presented in [75] takes a slightly different approach. It finds the logic to map to EMBs altogether instead of one at a time, as in SMAP. The algorithm first selects a set of maximum fanout-free subgraphs (MFFSs)—a generalization of MFFCs with one or more outputs for possible EMB implementation. It goes through each node to find a MFFS. For each node, it searches in the

fanout cone of the node to find output nodes for the MFFS. MFFS selection is transformed into a hyper-graph clustering problem. The MFFSs selected may exceed the input bound of the target EMBs. If this happens, an iterative procedure is used to remove nodes from the MFFSs to reduce the input size. Finally, it selects a set of MFFSs among all candidate MFFSs to maximize area improvement without increasing the overall circuit delay. The selected MFFSs will be implemented as EMBs.

3.5.2 Mapping for CPLDs

Complex programmable logic devices (CPLDs) are a class of programmable logic devices that are more coarse-grained than typical FPGAs. Each CPLD logic cell (p-term block) is essentially a PLA that consists of a set of product terms (p-terms) with multiple outputs. A p-term block can be characterized by a 3-tuple (k, m, p) where k is the number of inputs, p is the number of outputs, and m is the number of p-terms for the block. The input size k is normally much larger than that of FPGA logic cells. In this section we discuss mapping algorithms for CPLDs.

Relatively speaking, there has been a lot less mapping algorithms proposed for CPLDs. A fast heuristic partition method for PLA-based structures is presented in [98]. The algorithm DDMap [118] adapts a LUT mapper for CPLD mapping. It uses wide cuts to form big LUTs and decomposes the big LUTs into p-terms allowed in target CPLDs. Packing is used to form multi-output p-term cells. An area-oriented mapping algorithm is proposed for CPLDs in [15]. The algorithm is based on tree mapping. It uses heuristic partial collapsing and bin packing to form p-term cells. In [57], the authors investigate an FPGA architecture consisting of the k/m macrocell which is a p-term block with one output. A mapping algorithm similar to the FlowMap algorithm is proposed for this architecture.

A more recent mapping algorithm for CPLDs, PLAmap, is proposed in [40]. Like most of mapping algorithms, it has two phases: the labeling phase and the mapping phase. In the labeling phase, it finds a minimal mapping depth for each node using logic cell $(k, m, 1)$, i.e., a

singled out p-term block, assuming each logic cell has one unit delay. The labeling procedure is based on Lawler's clustering algorithm [123]. The labeling process follows topological order from PIs whose labels are set to zero. At each node, let l be the max label of the nodes in its fanin cone. The algorithm forms a logic cluster by grouping the node with all nodes in its fanin cone that have the label l. If the cluster can be implemented by a $(k,m,1)$-cell, the node is assigned the label l; otherwise, the node gets the label $l+1$ with a cluster consisting only of the node itself. Note that this is a heuristic in that the label may not be the best. This is because even if the cluster formed by the node and its (transitive) fanins with the label l cannot be implemented by a $(k,m,1)$-cell, a super-cluster (one containing the cluster) may be – the so-called *non-monotone property*. The mapping phase is done in reverse topological order from the POs. The algorithm tries to merge the clusters generated in the labeling phase to form (k,m,p)-cells whenever possible. Cluster merging is done in such a way that duplication is minimized and the labels of the POs do not exceed the performance target. There is also a post-processing packing to further reduce p-term cell count. In commercial CPLDs, there is extra logic in each p-term block for borrowing p-terms across blocks. PLAmap is also extended to take advantage of such extra logic. Experimental results show PLAmap out-performs commercial tools and other algorithms in performance with none or very small area penalty.

P-term blocks or macrocells are suitable for implementing wide-fanin, low-density logic, such as finite-state machines. They can potentially complement fine-grain LUTs to improve both performance and utilization. Both academic and commercial architectures have been proposed that contain a mix of LUTs and P-team blocks or macrocells to take the advantages of different types of logic cells. Technology mapping algorithms were proposed for such hybrid architectures [111, 119, 138]. These algorithms are similar to the hybrid mapping algorithms described earlier in that they try to identify sub-circuits to be implemented using coarse-grain structures.

4

Physical Synthesis

In a conventional FPGA design flow, synthesis is separated from physical design. In the synthesis stage, the design is transformed from one representation to another. Along the way, the design morphs from a high-level generic representation to a netlist in terms of the logic cells of the target FPGA device (a mapped netlist). Physical design places the logic cells on the selected FPGA device and finally connects the cells.

With continued shrinking of the feature size, the locations of the cell and the wirings among them become the dominating factors on the quality of the final implementation. It is common for interconnect delays to take up to 70–80% of the total delay on the critical paths of final designs these days. The conventional design flow cannot adequately address this challenge. This is because the physical effects are not considered during synthesis. On the other hand, we are limited by the netlist from synthesis during physical design. As a result, we cannot correct the "bad" synthesis decisions made earlier in the flow.

This is where physical synthesis comes into play. Physical synthesis can mean different things for different people. However, at higher level, physical synthesis can be viewed as techniques/methods that try to

link physical design with synthesis. So synthesis can consider physical impacts and physical design can introduce design transformations.

Although some physical synthesis techniques for standard cells can also be applied to FPGAs, physical synthesis for FPGAs has its unique constraints. For example, in FPGAs, we cannot size cells up and down to trade off timing, area, and power. In the following, we review physical synthesis techniques developed for FPGAs.

4.1 Logic Clustering

Most modern FPGA architectures contain a physical hierarchy *logic blocks* for improved area-efficiency and speed. To implement a design on such architectures, a logic clustering step is typically needed between technology mapping and placement. Logic clustering transforms a netlist of logic cells into a netlist of logic clusters each of which can be implemented using a logic block.

A typical logic block contains N logic cells with I inputs and N outputs. Here, N and I are fixed for the given architecture. There can be other architecture constraints, such as control signals for sequential elements, which have been omitted here for ease of discussion. The logic clustering problem for FPGAs takes as input a mapped netlist and produces a clustered netlist satisfying the cluster parameters.

One of the early FPGA clustering algorithms is VPack [22, 21]. The VPack algorithm forms one cluster at a time. At the beginning of each cluster formation, VPack selects as a seed an unclustered logic cell with the most used inputs and places this seed into the cluster. It then calculates the *attraction* of each unclustered cell to the new cluster. The attraction of a logic cell to a cluster is the number of inputs and outputs that are shared by the cell and the cluster. The cell whose addition doesn't violate cluster constraints and has the largest attraction value will be added to the cluster. The packing process is repeated until we cannot add in new cells to the current cluster. At that time, packing begins with a new cluster. The process terminates when all logic cells have been assigned a cluster.

The objective of the VPack algorithm is to minimize the number of clusters needed. Later, an enhanced algorithm called T-VPack was

proposed to optimize timing [144]. The delay model used in T-VPack can be described by three delay values: logic cell delay, intra-cluster delay, and the inter-cluster delay. Usually, the inter-cluster delay is much larger than the intra-cluster delay. The basic idea in T-VPack is to reduce the inter-cluster connects on the critical paths.

T-VPack follows the general steps of VPack. It also forms one cluster at a time and packs as many logic cells into each cluster as possible. The main difference is in the selection of seeds and selection of logic cells to absorb into partial clusters. Both selections are based on timing criticality values. The criticality of the connection driving an input i is defined as Connection_Criticality(i) = 1− slack(i)/MaxSlack, where *MaxSlack* is the largest slack of all interconnects and slack(i) is the slack at input i. T-VPack selects as the seed an unclustered cell that is driven by the most critical connection.

The concept of attraction is enhanced to include timing criticalities. T-VPack defines the base criticality of an unclustered logic cell B with respect to the cluster C currently being packed as the maximum connection criticality of all the connections between B and cells C, denoted by Base_BLE_Criticality(B), when the cluster is understood. The criticality of B is defined using the following formula:

$$\begin{aligned}\text{Critically}(B) = &\text{Base\_BLE\_Critically}(B)\\ &+\varepsilon \cdot \text{total\_path\_affected}(B),\end{aligned}$$

where total_path_affected(B) is an estimate of the number of critical paths B is involved in and ε is a very small value (so total_path_affected(B) acts a tie-breaker for the base criticality). The new attraction formula incorporating timing is defined as follows:

$$\text{Attraction}(B) = \alpha \cdot \text{Critically}(B) + (1 - \alpha) \cdot \text{Attraction}_{\text{area}}(B)/G,$$

where $\text{Attraction}_{\text{area}}(B)$ is the attraction for area packing used in VPack, G is a normalizing factor, and α is a trade-off factor that determines the amount of attraction coming from timing vs. area. It is recommended to set α to 0.75 for timing.

T-VPack performs much better for timing than VPack. Comparison data also show T-VPack results in better final chip area after placement and routing. This is because T-VPack has the tendency to completely

absorb many low fanout nets into clusters. This reduces the number of inter-cluster nets, which in turn, improves the routing area.

One problem with T-VPack is the delay model. It uses a simplistic model since there is no accurate interconnect delay. To overcome this problem, a simultaneous clustering and placement algorithm is proposed in [45]. This algorithm can move cells among clusters inside an annealing-based placement engine. It starts with an initial clustered netlist and a random placement. During the annealing process, it tries to remove suboptimal clustering structures by introducing *fragment-level moves* in addition to block-level moves which move clusters around. Fragment-level moves relocate cells among clusters without changing the locations of the clusters. The integrated approach is particularly effective when utilization is high and a large amount of unrelated packing occurs. Unrelated packing means logic that is not directly connected is packed into the same clusters to reduce number of clusters. Experimental results show significant improvement in timing and area compared to a separate approach using T-VPack followed by VPR.

Clustering has a significant impact on the routability of a design. A routability-driven clustering algorithm is presented in [25]. The algorithm prioritizes a set of factors that can impact routability, then incorporates those factors to form an routabilty-oriented attraction formula to guide the cell selection process. A typical clustering algorithm packs as much logic into each cluster as possible. In practice, routability can be improved if the clustered netlist matches the device structure. A connectivity-based clustering algorithm is proposed that tries to achieve "spatial uniformity" to reduce stress on routing [174]. The algorithm is based on *Rent's* rule and may purposely leave some clusters unsaturated for better routability.

Some FPGA architectures have more than one level of hierarchy, e.g., Altera APEX families have two levels of physical hierarchy. In [68], the authors study the problem of performance-driven multi-level clustering. They show that the problem is NP-hard and propose an efficient heuristic for two-level clustering. The heuristic algorithm is based on label computation, like many of the area-constrained performance-driven clustering algorithms [123, 150, 164]. Experimental results show

an average improvement of 15% delay reduction from the two-level clustering algorithm after final place and route.

4.2 Placement-Driven Mapping

In this section, we examine physical synthesis techniques that combine placement and technology mapping. One problem with technology mapping in the traditional flow is the lack of accurate information about interconnects. Many mapping algorithms use simple models that are inaccurate when the design is placed and routed. For timing optimization, most mapping algorithms use the unit delay model in which the interconnect delays are totally ignored. "Good" mapping solutions produced based on such inaccurate models may not be good after placement. To overcome this problem, algorithms have been proposed to combine placement and mapping.

A number of algorithms try to carry out placement and mapping simultaneously. The MIS-pga algorithm [152] performs iterative logic optimization and placement. The algorithm in [37] tightly couples technology mapping and placement by mapping each cell and assigning it to a preferred location on a 2-D grid using a maximum weighted matching formulation. Another approach [186] combines mapping, placement, and routing by integrating mapping into a bi-partitioning-based placement framework. The algorithm in [24] refines mapping solution during placement using simulated annealing to move logic among LUTs to improve routability. Ideally, such integrated approaches would generate the best solutions. In practice, they have a serious limitation. Due to the complexity of the combined problem, often simple mapping and placement techniques are employed for ease of integration. Because of this, the benefit of the combined approach is reduced.

Another approach to combining mapping and placement is by iterating between mapping and placement (or placement refinement). The design is first mapped and placed. Then, the netlist is back-annotated and mapped again under the given placement. This process can be repeated until a satisfactory solution is found. Figure 4.1 outlines the major steps in an iterative mapping and placement algorithm for timing optimization presented in [140]. The key step is the placement-driven

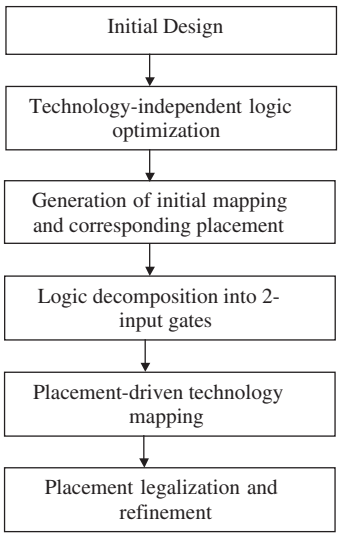


Fig. 4.1 One pass of iterative mapping and placement [140].

mapping problem that involves decomposing a mapped/placed netlist, then mapping it again to improve timing. The mapping step may make the placement illegal (e.g., two or more cells are placed at the same location). The placement of the new mapped netlist is then legalized and refined to produce a mapped/placed solution with potentially better timing.

The algorithm uses table-lookup to estimate edge delays based on placement locations. Given two locations, it looks up the estimated delay for the wiring between the two locations in a pre-stored table. This is more accurate and realistic than the "fixed" interconnect delays used in earlier layout-based mapping algorithms [146, 216].

The decomposition method is a straightforward extension of the *dmig* algorithm [46]. The only difference is that interconnect delays are added to the arrival time propagation during the decomposition process. New nodes inherit the locations of the LUTs from which they are generated.

The mapping algorithm works in a fashion similar to typical cut enumeration based technology mappers. It has two phases namely labeling and mapping generation. The labeling phase is the same as in

conventional mapping, except it adds interconnect delays into the label propagation.

One difficulty in placement-driven mapping is that the new solution may not be legal since it is possible to assign two or more cells to the same location. Another difficulty is that timing predicted in the labeling phase may be invalid due to congestion in the new mapping solution. Congestion means many overlapping locations in a small region. It requires many cell relocations to legalize the placement, which in turn, perturbs the timing. To overcome this problem, the algorithm employs an iterative process with multiple passes in the mapping phase. Each pass uses the cell congestion information gathered during previous iterations to guide the mapping decisions. Several techniques are proposed to relieve congestion while trying to meet the labels at the POs. One of the techniques is a hierarchical area control scheme to evaluate the local congestion cost. In this scheme, the chip is divided into bins with different granularities. Area increase is tallied in each bin. Penalty costs will be given to bins with area overflows.

After the mapping phase completes, a mapping solution with a possibly illegal placement is generated. This is followed by a timing-driven legalization step that moves overlapping cells to empty locations in their neighborhood based on the timing slack available for the cell. Finally, a simulated annealing-based placement refinement phase is carried out to improve the circuit performance. Experimental results show the method can improve timing by over 12% with little area penalty incurred by remapping.

4.3 Placement-Driven Duplication

Logic replication is a simple, yet effective technique for improving timing. It can be used to distribute fanout load and isolate critical paths. Logic replication can improve design quality without any real area cost if it can be done using only the unused cells on the target device.

Recently, logic replication has been used to reduce interconnect delays after placement. The idea is to use replication to straighten paths that are otherwise circuitous (and therefore with big delays). Figure 4.2 shows an example. Suppose A, B, D, and E cannot be moved (e.g., pads

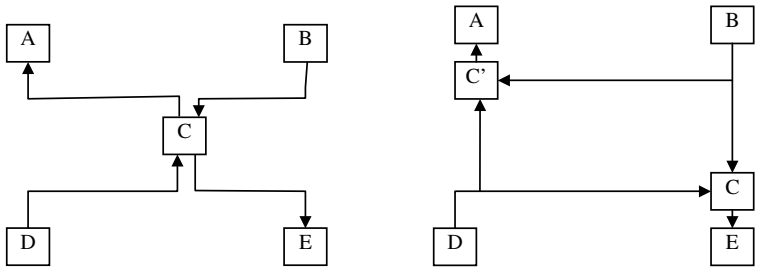


Fig. 4.2 Straightening paths using replication [20].

with fixed positions). We further assume that delay is proportional to the wirelength. To minimize the maximum path delay, we have to place C at the middle of the region enclosed by the four fixed pads as shown in the left of Fig. 4.2. Obviously, all paths between pads have detours going through the middle cell C. This is bad for timing. On the other hand, if we duplicate cell C and move C and its duplicate close to the two sink pads A and E, we arrive at the placement shown in the right of Fig. 4.2. The new placement is much better in delay. What happens is that once we replicated C, we have the freedom to place the cell and its duplicate differently to straighten paths and improve timing.

An iterative algorithm is presented in [20] that selectively replicates cells and perturbs the placement to straighten circuitous paths for timing optimization. One of the key components of the algorithm is the notion of *local monotonicity*. A cell, together with a fanin and a fanout, is on a non-monotone local sub-path if the distance between the fanin cell and the fanout cell is smaller than the sum of the distance from the fanin to the cell and the distance from the cell to the fanout. Let C be the cell and P and N be a fanin and a fanout of C, respectively. The *deviation* of C on this sub-path can be defined as follows:

$$\text{deviation}(C) = \text{distance}(P, C) + \text{distance}(C, N) - \text{distance}(P, N),$$

If $\text{deviation}(C) > 0$, then P, C, and N form a non-monotone sub-path.

The algorithm, starting from a placed design, iteratively selects one cell on a non-monotone sub-path of a critical path to replicate. Cell selection focuses on cells that are on critical paths and have non-zero deviation. The cells with large deviations will have a high probability

of being selected. After a cell is selected and replicated, it finds a new location for the replicated cell to reduce the deviation. The fanouts of the selected cell will then be distributed between the cell and its replicate with the goal of improving timing. It could also happen that all the fanouts go to the replicate. In this case, the cell is essentially relocated.

The new location may not be legal since cells can overlap. Cell overlaps are eliminated in the ensuing legalization procedure. The objective of legalization is to produce a legal placement with minimal impact to the performance of other paths. A procedure based on "ripple move," similar to the one in [105], is used. The algorithm terminates when it fails to generate improvement for a certain number of iterations in a row. Experiments show good performance improvement over the placement results produced by the VPR algorithm.

One limitation of the preceding algorithm is that it only targets local monotone sub-paths. While effective, it may not be able to remove more global non-monotone sub-paths. Figure 4.3 demonstrates this limitation. In this example, the sub-path s- >a-> b and a- >b- >t are both non-monotone (based on rectilinear distance). However, the path from s to t is not so.

An enhanced algorithm is proposed in [103] to overcome the limitation of local monotonicity. The algorithm is also iterative. However, it may replicate a set of cells in each iteration. The algorithm determines a timing-critical section of the design, then replicates the cells in the critical section to generate a *Replication Tree* in which every node has one fanout except the leaves which may have multiple fanouts (a leaf-DAG). The algorithm then embeds the replication tree by adapting the dynamic programming procedure for S-tree embedding [102] to

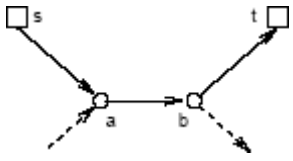


Fig. 4.3 Limitation of local monotonicity [103].

consider both timing and wirelength. Placement legalization is invoked after embedding. The legalization procedure is the same as the previous algorithm except for the fact that both timing and wirelength are considered. A delay reduction of over 14% was observed on a set of benchmark designs, about twice the improvement from the approach based on local monotonicity removal.

The previous algorithms carry out logic replication as a post-optimization step after placement. An algorithm is presented in [44] that integrates replication into a simulated annealing placement engine. At the end of each annealing iteration, it performs a placement-driven logic replication based on the current placement. The replication algorithm has several unique features. It introduces the notion of *feasible region* and *super-feasible region* to improve the critical path monotonicity globally. An enhanced placement legalization procedure is proposed that can take into consideration the complex architecture constraints in real commercial FPGA architectures. Replication can be carried out multiple times with this approach, which may result in redundancies—multiple duplicates of the same nodes. An effective technique is presented to remove redundancies globally while preserving timing. Experimental results show over 18% delay reduction over VPR on average. With the path-counting-based net weighting scheme in [115], the algorithm achieves over 25% delay reduction.

4.4 Other Techniques

There are many other physical synthesis techniques, from post-layout pin permutation [80] to a general incremental physical synthesis framework [178]. In this section, we briefly review a few other physical-aware synthesis and optimization techniques.

Integrated retiming and placement: Traditionally, retiming is applied during the synthesis stage where accurate estimate of interconnect delays is not available. In [175] an integrated retiming and placement algorithm is presented. The algorithm has three components: retiming-aware placement, retiming with minimal placement

disruption, and incremental clustering and placement to legalize the disrupted placement after retiming.

The algorithm first enhances the simulated annealing-based FPGA placement algorithm VPR (see Section 2.2.1) to make it retiming-aware. VPR is a net-weighting based timing-driven placement algorithm that uses connection criticalities to weigh interconnection delays with the objective of encouraging cells on critical connections to stay together during placement. In the original VPR algorithm, the criticality of an interconnection c is defined as follows: $\text{Critically}(c) = 1.0 - \beta \cdot \text{slack}(c)$, where $\text{slack}(c)$ is the slack of the interconnection and β is a scaling factor. To make VPR retiming-aware, the criticality formula is enhanced by replacing $\text{slack}(c)$ with $\text{CycleSlack}(c)$. The cycle slack of an interconnection is the maximum amount of delay that can be added to the interconnection without violating a given performance target under retiming. The notion of cycle slacks is similar to sequential slacks computed using sequential arrival and required times [63, 159].

Once the retiming-aware placement step is completed, the next step of the algorithm is to find and apply a retiming to improve performance based on more accurate interconnect delays determined by the placement. The algorithm finds a retiming using the standard formulation of minimizing a weighted sum of the register counts on interconnections subject to the timing requirements. The weights or costs of the interconnections are assigned in such a way that they discourage disruption to the placements. The final step is incremental clustering and placement to place the new registers introduced during retiming. This step is based on a greedy iterative improvement method that moves logic cells in an attempt to minimize a cost function. The reader is referred to [175] for details on the cost function and the moves used. An average of 19% performance improvement is reported compared to a sequential approach in which retiming is done before placement.

SPFD-based rewiring: Rewiring is a technique that changes interconnections in a design by removing and adding wires without touching logic cells, while preserving design functionality [83, 33, 34, 215, 64]. This technique is very attractive in physical synthesis since it does not perturb placement. It can be used for timing

improvement (replacing timing-critical connections with non-critical ones) or routability improvement (replacing interconnections in congested regions with those in less congested ones).

Most rewiring algorithms use ATPG-based redundancy addition and removal and do not modify the functionality of any node in a design during rewiring. However, they cannot take advantage of the flexibility of K-LUT (which can implement any function with up to K inputs). Set of pairs of functions to be distinguished (SPFD) [215] is a rewiring technique that may change node internal functions during rewiring. SPFD approaches can potentially find more rewiring opportunities than ATPG-based approaches while possessing the same advantage of no placement disruption.

A function f is said to distinguish a function pair (π_1, π_0) where $\pi_1 \neq 0, \pi_0 \neq 0, \pi_1 \cdot \pi_0 = 0$, if either one of the following two conditions holds: $\pi_1 \leq f \leq \overline{\pi}_0$ or $\pi_1 \leq f \leq \overline{\pi}_1$. Intuitively, $\pi_1 \leq f \leq \overline{\pi}_0$ means f contains π_1 and is outside of π_0, namely, it separates or distinguishes the two functions π_1 and π_0. A function f is said to *satisfy* an SPFD $P = \{(\pi_{11}, \pi_{10}), (\pi_{21}, \pi_{20}), \ldots, (\pi_{m1}, \pi_{m0})\}$ if f distinguishes all the function pairs in P. The notion of SPFDs is a way to express don't-care conditions by providing flexibility for implementing functions [26].

A SPFD-based rewiring algorithm starts by computing the *global function* of each pin in a design (a global function is defined in terms of the PIs). It then determines the SPFD for each pin in topological order from the POs. After the SPFDs for all pins are available, rewiring is carried out next. As an example, for the partial design shown in the left of Fig. 4.4, assume there is another node p' with global function $g' = \overline{x}_1 + x_2$. It can be shown that g' can also satisfy the SPFD of p_2. As

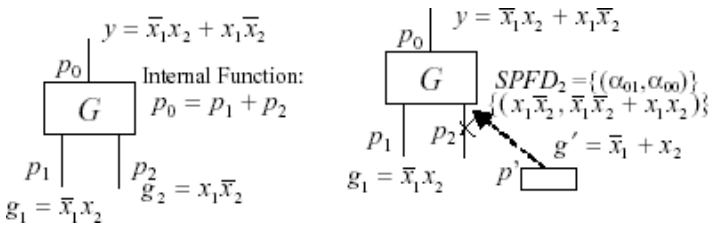


Fig. 4.4 SPFD-based rewiring [65].

a result, we can connect p' to p_2 and remove the original wire connected to p_2. The internal function of G should be changed to $p_0 = p_1 + \overline{p}_2$ to maintain the functionality of the design.

The original SPFD-based rewiring algorithm proposed in [215] finds alternative wires locally. It requires the destination node of the alternative wire to be the same as the target wire, as indicated in Fig. 4.5. This obviously limits the effectiveness of the rewiring algorithm. The global SPFD-based rewiring algorithm presented in [64] is capable of finding global alternative wires that may not share the same destination nodes as target wires. Let G_1 be the destination node of a target wire as shown in Fig. 4.5. The global rewiring algorithm may add an alternative wire to a *dominator*, G_D, of G_I. A dominator of G_1 is a node that is on all paths from G_1 to POs. Comparitive results show that ATPG-based rewiring can find alternative wires for around 10% of the wires, while local and global SPFD-based rewiring can find alternative wires for 25% and 36% of the wires, respectively. This clearly shows the potential of SPFD-based rewiring for LUT-based FPGAs.

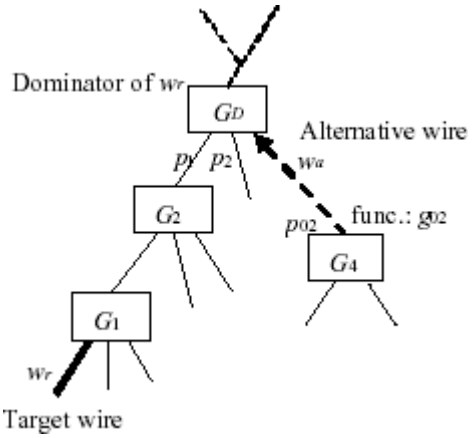


Fig. 4.5 Global SPFD rewiring [65].

Integrated mapping and clustering: The quality of clustering depends significantly on the mapping solution. To address this dependency, mapping and clustering needs to be performed together. In [139] an integrated mapping and clustering algorithm was proposed. The

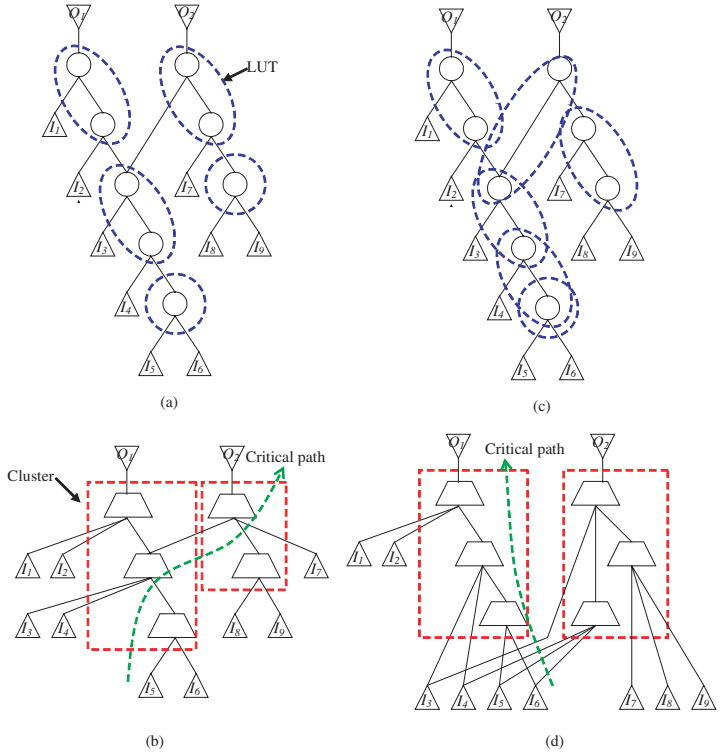


Fig. 4.6 Mapping and clustering.

algorithm can find a clustering solution with optimal delay under the commonly-used clustering delay model described in Section 4.1.

Figure 4.6 is an example that illustrates the sub-optimality of separate mapping and clustering. We assume 3-input LUTs and a cluster capacity of 3 (i.e. at most 3 LUTs in each cluster). For the example netlist in (a), delay-optimal mapping generates a netlist with 5 LUTs, and delay-optimal clustering afterwards produces a clustered netlist in (b). The critical path contains three inter-cluster connections (counting the edges between I/Os and clusters) and one intra-cluster connection. On the other hand, with node duplication, we can obtain another mapping solution with 6 LUTs as shown in (c) which results in the clustered netlist in (d). The critical path of the second clustering solution contains two inter-cluster connections and two intra-cluster

connections. The latter solution is better than the one in (b) generated by separate delay-optimal mapping and clustering, assuming inter-clustering is larger than intra-cluster delay as is the case in practice.

The proposed algorithm carries out mapping and clustering simultaneously. It first uses dynamic programming to determine the optimal delay at each node while considering both mapping and clustering. After the optimal delay at each node is determined, the algorithm generates mapping and clustering solutions simultaneously to realize the optimal delays under the cluster capacity constraint. The paper also presents a number of heuristics to reduce area overhead introduced by duplication. Compared to a sequential approach using state-of-the-art mapping and clustering algorithms, the proposed algorithm achieves 25% performance gain with 22% area overhead under the inter-/intra-clustering delay model. After placement and routing using VPR, the performance is still 12% better on a set of benchmark designs.

5

Design and Synthesis with Higher Level of Abstraction

Modern SoC FPGA (or field-programmable SoC) contains embedded processors (hard or soft), busses, memory, and hardware accelerators on a single device. On one hand, these types of FPGAs provide opportunities and flexibilities for system designers to develop high-performance systems targeting various applications. On the other hand, they also immediately increase the design complexity considerably. To realize the promise of this vision, a complete tool chain from concept to implementation is required [210]. System-level, behavior-level, and RT-level synthesis techniques are the building blocks for this automated system design flow. System-level synthesis compiles a complex application in a system-level description (such as SystemC [182]) into a set of tasks to be executed on various processors, or a set of functions to be implemented in customized logic, as well as the communication protocols and the interface logic connecting different modules. Such capabilities are part of the so-called electronic system-level (ESL) design automation. ESL design automation has caught much attention from the industry recently. Many design challenges still remain in this level, such as the standardization of IP integration, system modeling, performance estimation, overall design flow, and verification methodology, etc. A key

component of ESL design automation is behavior-level synthesis which is a process that takes a given behavioral description of a circuit and produces an RTL design to satisfy area, delay, or power constraints for the hardware. It primarily consists of three subtasks namely: scheduling, allocation, and binding. The next step after high-level synthesis is RTL synthesis.[1] Usually, input to an RTL synthesis tool includes the number of data path components, the binding of operations to data path components, and a controller (finite state machine) that contains the detailed schedule (related to clock edge) of computational, I/O, and memory operations. The output of the RTL synthesis provides logic level implementations of the design that can be evaluated through the optimization of the data path, memory, and controller components, individually or in a unified manner. Section 1 introduced RTL design and RTL elaboration steps.

System-level design is a vast area. It can include topics on software/hardware partitioning and codesign, reconfigurable computing, and synthesis for dynamically reconfigurable FPGAs, which are all beyond the scope of this paper. Therefore, we will primarily focus on behavior-level and RTL synthesis in this section. Continuing with the bottom-up approach guided by design levels, as laid out in previous sections, we will introduce RTL synthesis first and behavior-level synthesis second.

5.1 RTL Synthesis

RTL synthesis is a key step in the FPGA design flow, as shown in Section 1. However, there are relatively few publications on this subject in relevant literature. Part of the reason is that RTL synthesis for FPGAs can take advantage of existing RTL synthesis techniques used in ASIC designs, which are already pretty mature. Meanwhile, RTL synthesis for FPGAs does need to consider the specific FPGA architecture features. For example, the regularity of FPGA logic fabric offers opportunities for directly mapping datapath components to FPGA logic blocks, producing regular layout, and reducing chip delay

[1] Designers can skip high-level synthesis and directly write RTL codes for circuit design. This design style is facing challenges due to the growing complexity of FPGA designs.

and synthesis runtime. The synthesis tools also need to pay attention to circuitry features enclosed in the logic blocks, such as the fast carry chains, to achieve better performance for the design. In addition, most of the modern FPGAs offer both hard structures, such as blocks of memory and multipliers, and flexible soft programmable logic to provide domain-specific programmable solutions. These FPGAs are called platform FPGAs or heterogenous FPGAs. Examples include Altera Stratix II device families [10] and Xilinx Virtex-5 device families [209]. This brings new challenges for RTL synthesis to simultaneously target both hard structures and soft logic. In this subsection, we will present several research works and provide readers with the flavor of how these issues have been addressed. We will first talk about datapath synthesis, which deals with mapping datapath components to FPGA directly. We then cover RTL synthesis for heterogeneous FPGAs. RTL synthesis for FPGA power reduction will be discussed in Section 6.4.3. We believe there are still many interesting research topics for further study in RTL synthesis, such as retiming for glitch power reduction, resource sharing for multiplexer optimization, and layout-driven RTL synthesis, just to name a few.

5.1.1 Datapath synthesis

Large circuits typically contain a large portion of highly regular datapath logic. The traditional gate/CLB-level CAD flow first implements each datapath node with a datapath component, then flattens the datapath components to gates (discarding information about regularity), and feeds the resulting netlist to the gate-level design flow. It is unlikely that an efficient bit-slice layout will be rediscovered during placement, and the generated irregular layout leads to a difficult routing problem. Moreover, once the circuit is flattened to gates, it is usually not possible to rediscover uses of specialized features of the CLBs in the FPGA, such as the fast carry chain circuitry. Flattening to gates also leads to a much larger problem size — there are many multiples of gates than there are nodes in the DFG. To address this problem, *datapath synthesis* algorithms preserve these datapath structures rather than flattening them to gates. They also explore the specialized datapath features in

an FPGA and try to map datapath operations directly to the available modules in the FPGA. Datapath synthesis intrinsically needs to deal with bit-slices for maintaining the regularity of the datapath and the layout. Therefore, it is natural to be combined with packing, placement, and routing tools to improve speed or area. The input of datapath synthesis is a scheduled DFG or CDFG. Thus, the data storage elements are already determined and need to be considered during synthesis. There are several related works in this area.

In [114] a design strategy named SDI was presented. It is a strategy for the efficient implementation of regular datapaths with fixed topology on FPGAs. It employs parametric module generators, a floorplanner based on a genetic algorithm, and a circuit compaction phase through local technology mapping and placement. Figure 5.1 shows the design flow of SDI. The chip topology targeted by SDI is characterized by a fixed tri-partite layout (Fig. 5.2). The large middle section holds the regular part of the datapath. This part consists of a horizontal arrangement of modules, each composed of vertically stacked bit-slices. The area below the datapath is intended to hold the controller. A small area above the regular section can hold irregularities in the modules as *cap cells*, e.g., the processing of overflow and carry bits in a signed adder.

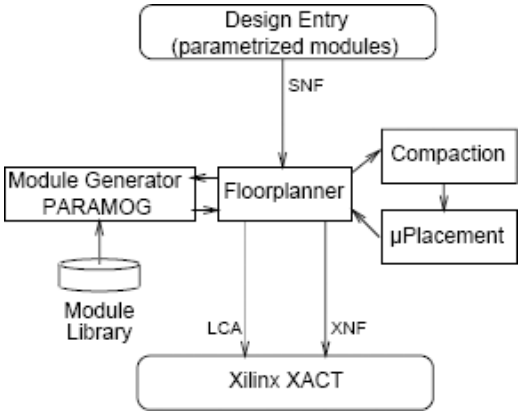


Fig. 5.1 SDI design flow in [114].

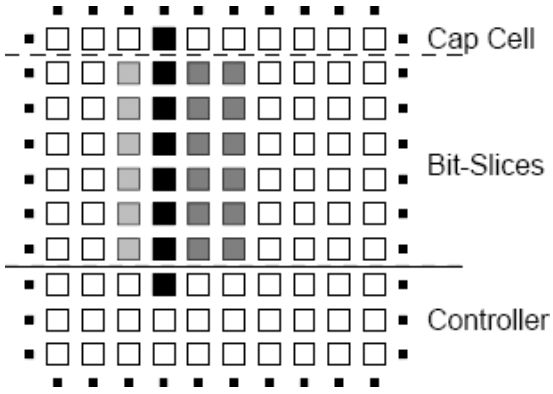


Fig. 5.2 On-chip topology shown in [114].

As shown in Fig. 5.1, the designs are expressed in the SDI netlist format *SNF*, which is a textual netlist of module declarations, module instantiations, and interconnections. Modules include arithmetic modules, logic, shifters, comparators, counters, storage elements, etc. SNF associates values with module parameters such as bus widths, data types, and optimization requests (speed vs. area).

The module generation procedure (PARAMOG) takes user-specified parameters for each module instance and prepares a list of possible layouts with different topologies. The *FloorPlanner* will read the available layout topologies for all module instances of the datapath and begin to linearly place instances in the regular region of the FPGA. During this process, different concrete layouts are selected and evaluated in context. The FloorPlanner is based on a genetic algorithm, and thus considers various different layout choices and placements simultaneously. When FloorPlanner has finished its work and created a suitable linear placement of module instances in the datapath area, a compaction phase follows. Compaction is performed by merging all logic (across module boundaries) within a logic equivalence class and processing the resulting functions with classical logic synthesis and optimization tools. Afterwards, since all placement information within an equivalence class is lost during compaction, the CLBs in the classes have to be replaced. One of the primary criteria for this placement is

the restoration of a regular structure consistent with the one created by PARAMOG. This placement (μPlacement) is based on an integer linear programming (ILP) formulation of the problem. Afterward, a routing tool available through Xilinx physical design tool *PPR* is used to finish the routing for the design. The final result of SDI is a bit-stream ready for downloading.

The paper showed two layouts of the same circuit (a 16-bit datapath consisting of two instances of a sample combinational module with a structure common to many bit-slices), one conventionally generated by the Xilinx tool PPR, the other one processed by SDI. Based on the layout, they showed that the SDI-generated solution is more regular. The SDI layout is also less congested than the PPR layout, and the routing delay in the critical path of the SDI solution is 13% shorter than the PPR layout. SDI also runs about two times faster. Although FPGAs with fixed bit-slice topology enable efficient datapath synthesis and faster optimization flow, it does imply restrictions on the overall layout of the design. In general, it should work well for data-intensive designs with simple control logic. However, for control-intensive designs, layout restriction can cause larger wire delays due to the difficulty of distributing control logic flexibly on the chip.

In [217] the authors proposed an enhancement to the module compaction algorithm proposed in [114]. They observed that typical datapath synthesis algorithms sacrificed area to gain regularity. They proposed two word-level optimizations—multiplexer tree collapsing and operation reordering. They reduced the area inflation to 3% to 8% as compared to flat synthesis. Their synthesis results retained a significant amount of regularity from the original designs.

In another follow-up work [30], the authors observed that the limitation of [114] was that the module compaction step could not handle specialized CLB features such as a fast carry chain, and thus did not attempt to merge modules that used such features. Another limitation of [114] was that only physically adjacent modules in the previously determined floorplan were considered for compaction. To address these issues, [30] presented a datapath mapping tool *GAMA*. GAMA consists of the following optimization steps.

Tree splitting. GAMA uses a tree-covering algorithm. Therefore, the input design (control dataflow graph) must be split into a forest of trees. Cycles are broken at appropriate places, usually at storage elements demarking iteration boundaries. This produces a directed acyclic graph (DAG), which will be further split into trees.

Tree covering. GAMA tries to take advantage of the *compound module* present in typical CLBs. It is often possible to implement multiple nodes from the DFG together in a compound module that is much smaller and/or faster than if they were implemented separately. When such compound modules exist, there may be many different ways in which the DFG can be covered with module patterns from the library of possible modules. The authors designed a dynamic programming algorithm (similar to what was used in DAGON [112]) to find the best cover in linear time for a tree. Each tree is passed to the tree-covering algorithm separately. The trees are covered in the topological order defined by the original design before tree splitting. Each tree covering is optimal in terms of delay or area, but the overall solution is not necessarily optimal. Relative module placement in the linear datapath occurs simultaneously with tree covering.

Post-covering optimizations. This phase may consider rearranging the modules after they have been placed by the tree-covering algorithm. This allows layout possibilities that are not considered by the tree-covering algorithm, such as intermingling the modules from different trees.

Module generation. Finally, each specified module is generated. A rich variety of functions can be implemented using a column of 4-input LUTs augmented with fast carry chain circuitry. The generator, given a pattern of DFG nodes, values of constant inputs, datapath width (in bits), etc., creates the module. All modules are generated with the same pitch and width (in bits).

Experimental results showed that for 32-bit datapath designs mapped to the Xilinx 4000 architecture, GAMA gave compilation speeds 3.24 times faster than compiling flattened netlists. Designs generated by GAMA were roughly of the same quality or better than

their flattened equivalents in terms of both CLB usage and critical path delay.

5.1.2 Heterogenous FPGAs

In [108] the authors presented an RTL synthesis tool for heterogenous FPGAs. Modern heterogenous FPGAs contain "hard" specific-purpose structures such as blocks of memory and multipliers in addition to the completely flexible "soft" programmable logic and routing. These hard structures provide major benefits, yet raise interesting questions in FPGA CAD and architecture. The authors presented a synthesis tool, called *Odin*, and an algorithm that permits flexible targeting of hard structures in FPGAs. Odin maps Verilog designs to two different FPGA CAD flows: Altera's Quartus, and the academic VPR CAD flow. Figure 5.3 shows the overall design flow of Odin.

First, a front-end parser parses the Verilog design and generates a hierarchical representation of the design. Second, Odin has an elaboration stage that traverses the intermediate representation of a design to create a flat netlist that consists of structures including logic blocks, memory blocks, *if* and *case* blocks, arithmetic operations, and registers. Each of these structures within the netlist is a node in the netlist. Third, some simple synthesis and mapping is performed on this netlist. This includes examining adders and multipliers for constants, collapsing mul-

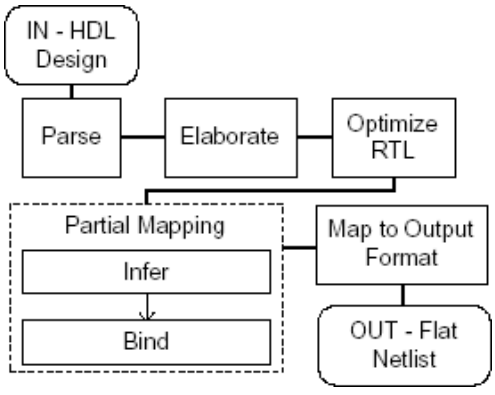


Fig. 5.3 RTL synthesis flow for heterogenous FPGAs in [108].

tiplexers, and detecting and re-encoding finite state machines to one-hot encoding. Fourth, an inferencing stage searches for structures in the design that could be mapped to hard circuits on the target FPGA. These structures are connected sub-graphs of nodes that exist in the design netlist. A matching algorithm is used to carry out this search. This matching problem is a form of sub-graph isomorphism, which has been used in instruction generation for configurable processor systems [55] and other applications.

Finally, a binding stage guides how each node in the netlist will be implemented. This is done by mapping nodes in the netlist to either hard circuits, soft programmable logic, or a mixture of both. One way to do this is to map structures to library parameterized modules (LPMs), which, in later stages of the industrial CAD flow, will bind to an implementation on the FPGA whether it is a hard or soft implementation. The output from Odin is a flat netlist consisting of connected complex logic structures and primitive gates. The authors showed that the quality of their tool is comparable to Altera's front-end synthesis tool. They also showed that their binding/mapping results compared favorably to those from Altera's Quartus tool.

5.2 Behavior-Level Synthesis

The basic problem of high-level synthesis (or behavior-level synthesis) is the mapping of a behavioral description of a digital system into a cycle-accurate RTL design consisting of a datapath and a control unit. A datapath is composed of three types of components: functional units (e.g., ALUs, multipliers, and shifters), storage units (e.g., registers and memory), and interconnection units (e.g., buses and multiplexers). The control unit is specified as a finite state machine which controls the set of operations for the datapath to perform during every control step. The high-level synthesis process mainly consists of three tasks: scheduling, allocation, and binding. Scheduling determines when a computational operation will be executed; allocation determines how many instances of resources (functional units, registers, or interconnection units) are needed; binding binds operations, variables, or data transfers to these resources. In general, it has been shown that the code density and

simulation time can be improved by 10X and 100X, respectively, when moved to the behavior-level synthesis from RTL synthesis [193, 194]. Such an improvement in efficiency is much needed for design in the deep submicron era. However, most of the behavioral synthesis problems are difficult to solve optimally due to various constraints, including latency and resource constraints. Meanwhile, the subtasks of behavioral synthesis are highly interrelated with one another. For example, the scheduling of operations to control steps is directly constrained by resource allocation. Meanwhile, a performance/cost tradeoff exists in the design space exploration. An area-efficient design with a smaller number of resources will increase the total number of control steps to execute the desired function. On the other hand, allocating more resources to exploit parallel executions of operations can achieve a higher performance, but at the expense of a larger area.

Traditionally, people are more concerned with the area and power of functional units and registers. As technology advances, the area and power of multiplexers and interconnects have by far outweighed the area and power of functional units and registers. Multiplexers are particularly expensive for FPGA architectures. It is shown that the area, delay and power data of a 32-to-1 multiplexer are almost equivalent to an 18-bit multiplier in 0.1 μm technology in FPGA designs [41]. In general, having a smaller number of functional units or registers allocated, with a larger number of wide multiplexers and larger amount of interconnects, may lead to a completely unfavorable solution for both performance and the area/power cost. Tackling this increasingly alarming problem will require an efficient search engine to explore a sufficiently large solution space while considering multiple constraining factors.

On top of these difficulties, behavioral synthesis also faces challenges on how to connect better to the physical reality. Without physical layout information, the interconnect delay cannot be accurately estimated. Since interconnect delay is the dominate element to determining the performance of designs in submicron technology, ignoring it will make it even more difficult for behavioral synthesis to deliver satisfactory solutions. These unique challenges are driving the need for developing new behavioral synthesis techniques and design flows that

are able to overcome or mitigate negative impacts from the separation of traditional high-level synthesis and low-level design. In addition, there is a need of powerful data-dependence analysis tools to analyze the operational parallelism available in the design before we can allocate proper amount of resources to carry out the computation in parallel. Meanwhile, memory operations usually represent the bottleneck for performance optimization. How to carry out memory partitioning, bitwidth optimization, and memory access pattern optimization, together with behavioral synthesis for different application domains prompts unique challenges for delivering satisfiable quality of design results. Given all these challenges, much research is still needed in this area.

We will first present work in behavioral synthesis for multi-FPGA systems. We then present some initial work on layout-driven behavioral synthesis, which presented some promising results for improving design quality. Finally, we briefly introduce some other techniques involved in behavioral synthesis, including loop transformation, branch optimization, and memory allocation. Behavioral synthesis for power minimization will be presented in Section 6.4.4.

5.2.1 Behavioral synthesis for multi-FPGA systems

Due to the capacity limitation of FPGA devices, there has been work on mapping large designs onto multiple-FPGA systems. The traditional flow usually consists of two phases [84]. In the first phase, a synthesizer is used to transform a design specification into a CLB-based netlist by performing high-level compilation, RTL/logic synthesis, and CLB-based technology mapping tasks. In the second phase, a circuit-level partitioner is used to partition the CLB netlist into FPGA chips. This method is mainly constrained by pin limitations on the FPGAs. In [168] the authors experimented with multiple FPGA partitioning methods at behavioral and structural levels. They observed that during structural partitioning, the IO limitation can be reduced if the partitioner is able to decompose and place portions of structural components, such as multiplexers and controllers, into different FPGA chips.

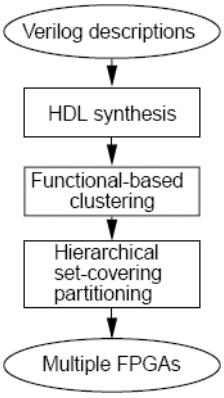


Fig. 5.4 The synthesis flow in [84].

In [84] the authors presented a new integrated synthesis and partitioning method for multiple-FPGA applications. Their approach tried to bridge the gap between high-level synthesis and physical partitioning by exploiting the design hierarchy. The input to their system is a design specification described in Verilog. Figure 5.4 shows their synthesis flow.

In the first step, a synthesizer performs RTL and FPGA synthesis tasks, including hardware description language (HDL) compilation, unit selection, unit/storage/interconnect binding, logic minimization, and CLB-based technology mapping. The synthesizer uses a fine-grained synthesis method to generate a structural tree, which can represent the structural hierarchy of the HDL description of a design. In a structural tree, the root node represents the top-level design, each intermediate node represents a higher-level design such as *modules*, *processes*, and *tasks*, and each leaf node represents a circuit cluster. A HDL description of the design is usually expressed as a set of hierarchical interconnected modules. Each module may contain a set of concurrent processes. From the hardware point of view, each process can be implemented as an independent hardware block. Furthermore, a process usually consists of a set of statements with input and output signals. The outcome of the outputs is dependent on the executions of the statements embedded in the process. To further decompose these statements, each output can be represented as a function of a set of

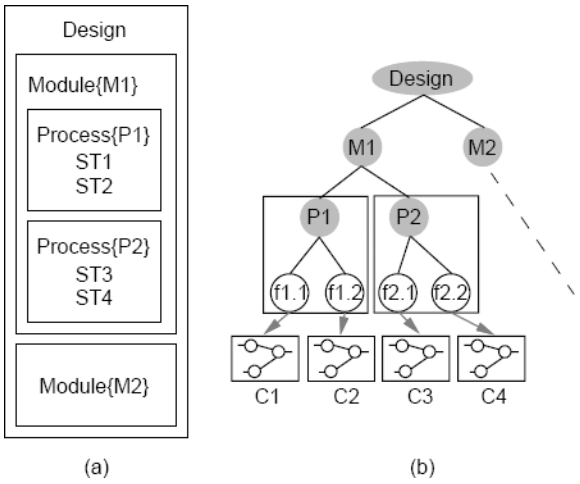


Fig. 5.5 (a) the design hierarchy; (b) the structural tree in [84].

inputs and internal signals in the process. Figure 5.5 shows a design with (a) its functional hierarchy, and (b) the structural tree for the design. The authors then map the nodes in the structural tree to a hierarchical connected graph. Each module node in the structural tree is mapped to a top-level node in the graph, while each process and functional node is mapped to a second-level and a third-level node, respectively. Figure 5.6 shows an example of their hierarchical connected graph.

The authors then formulate their problem as follows: given a hierarchical connected graph $G$and the CLB/IO-pin constraint of the FPGA chips, find a minimum number of FPGAs to cover G. They used a heuristic called *hierarchical set-covering partitioning*. The basic idea of the covering method is to start the set-covering procedure from the top-level nodes (i.e., *module* nodes). If no more feasible covers can be found in the top level, then the set-covering process continues on the nodes at the lower level. By exploiting the design structural hierarchy during the multiple-FPGA partitioning, they showed that their method produced fewer FPGA partitions with higher CLB and lower I/O-pin utilizations.

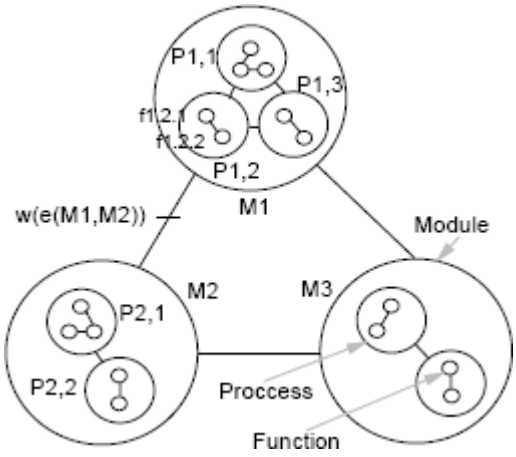


Fig. 5.6 A hierarchical connected graph mapped from a structural tree in [84].

Authors in [81] presented an overview of their COBRA-ABS high-level synthesis tool. COBRA-ABS has been designed to synthesize custom architectures for arithmetic-intensive algorithms, specified in C, for implementation on multi-FPGA platforms. It performs global optimization of high-level synthesis using simulated annealing, and integrating all of the following operations: datapath partitioning over multiple FPGAs, functional unit (FU) operation scheduling, FU selection and operator binding, FU allocation, register allocation, inter-FPGA communication scheduling, and inter-FPGA communication binding. COBRA-ABS synthesizes a custom very long instruction word (VLIW) architecture for the given algorithm for implementation on the specified FCCM. To illustrate the operation of this tool, a number of results for synthesis of a Fast Fourier Transform algorithm were presented.

5.2.2 Layout-driven behavioral synthesis

In [214] a layout-driven behavioural synthesis approach was presented to reduce the gap between predicted metrics during the synthesis and the actual data after implementation of the FPGA. This allows more efficient exploration of the design space and thus avoids unnecessary

iterations through the design process. By producing not only an RTL netlist but also an approximate physical topology of implementation at the chip level, the solution would perform at the predicted metric once it is implemented. The problem is formulated as follows: given (1) a data flow graph (DFG), (2) maximum allowable clock period and execution time, which are usually part of the system specification, and (3) component power/performance tradeoff functions due to different implementations, identify whether there is a feasible RTL datapath solution or not. If there is a solution, perform scheduling and binding, and generate an RTL netlist and its corresponding floorplan; otherwise, report it to the user and output the best solution that can be achieved. Figure 5.7 shows their overall design flow.

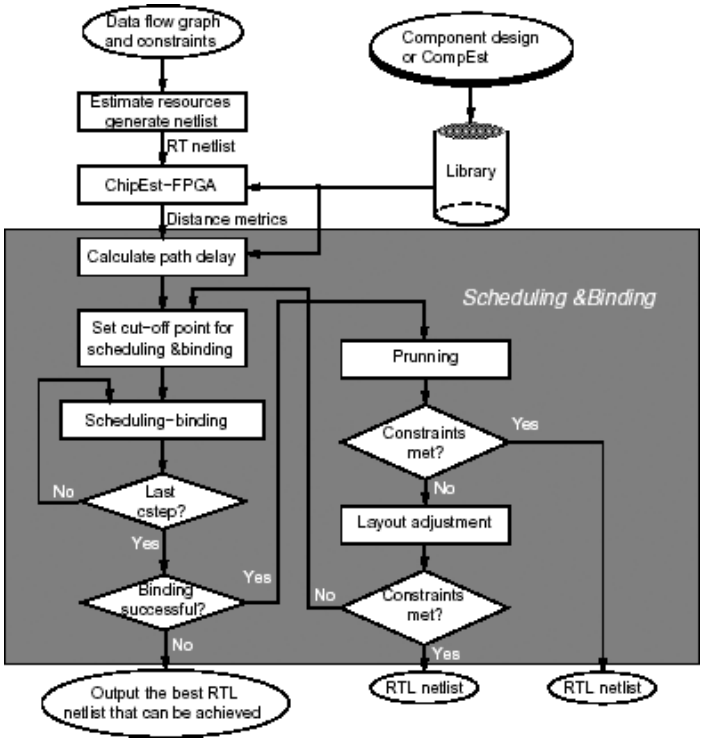


Fig. 5.7 Layout-driven high-level synthesis design flow in [214].

The authors first estimate the resources required for a design and use their physical-level estimation tool ChipEst-FPGA [213] to obtain an approximate topology of the layout. They then obtain distance metrics between the different units and use this step to provide feedback to the scheduling and binding task. Their scheduling and binding is carried out as one control step at a time. Given timing and resource constraints, they build a search tree encoding different binding and scheduling solutions. They then apply pruning techniques to certain branches when those branches lead to larger area or latency. Once a proper search solution is found, the algorithm will record its scheduling-binding information and proceed to do scheduling and binding for the next cycle. This process will be repeated until all the cycles are processed. If the scheduling-binding succeeds, the algorithm will then update the area and timing information based on the component information in the library and generate an optimized RTL netlist. At this point, if the limit exceeds the maximum limit layout adjustment will be invoked to re-run the ChipEst-FPGA on the updated RTL netlist. After layout adjustment, if the cycle time still cannot satisfy the maximum clock period constraint, the authors then relax the latency constraint and redo the scheduling and binding until the latency hits a threshold or a feasible solution is found. Experimental results showed that when using this approach, the authors could find a result that satisfied the constraints, while the timing constraints were violated using a traditional method (without layout information). They also found that interconnection delay could contribute up to 55% on the circuit performance for their benchmarks.

In [54] the authors presented layout-driven architectural synthesis algorithms, including scheduling-driven placement, placement-driven simultaneous scheduling with rebinding, distributed control generation, etc. The synthesis engine can target both microarchitecture and FPGAs. To target designs in nanometer technologies, their technique supports multicycle on-chip communication (data transfers on global interconnects can take multiple clock cycles). Their architecture model—regular distributed register (RDR) divides the entire chip into an array of islands. The island size is chosen so that all local computation and communication within an island can be performed in a

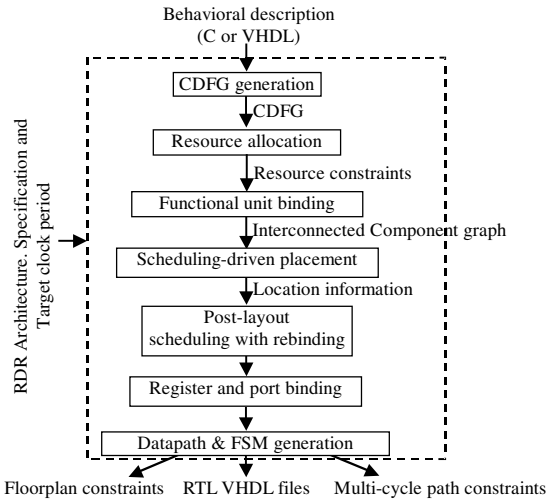


Fig. 5.8 Overall synthesis flow MCAS in [54].

single clock cycle. Signals traveling between two different islands will take 1 to k clock cycles depending on the distance between these two islands, where k is the maximum number of cycles needed to communicate across the chip. Figure 5.8 shows their overall synthesis flow, which is named architectural synthesis system for multi-cycle communication (MCAS).

At the front-end, MCAS first generates the control data flow graph (CDFG) from the behavioral descriptions. Based on the CDFG, it performs resource allocation, followed by an initial functional unit binding. The objective of resource allocation is to minimize the resource usage (e.g., functional units, registers, etc.) without violating the timing constraint. It uses the time-constrained force-directed scheduling algorithm [155] to obtain the resource allocation. After resource allocation, it employs an algorithm proposed in [113] to bind operational nodes to functional units for minimizing the potential global data transfers. An interconnected component graph (ICG) is derived from the bound CDFG. An ICG consists of a set of components (i.e., functional units) to which operation nodes are bound. They are interconnected by a set of connections that denote data transfers between components.

At the core, this flow performs the scheduling-driven placement, which takes the ICG as input, places the components in the island structure of the architecture using a simulated-annealing-based placer, and returns the island index of each component. After the scheduling-driven placement, both the CDFG schedule and the layout information are produced. To further minimize the schedule latency, MCAS performs placement-driven scheduling with rebinding. The algorithm is based on the force-directed list-scheduling framework, and is integrated with simultaneous rebinding.

At the back-end, MCAS performs register and port binding, followed by datapath and distributed controller generation. The final output of MCAS includes a datapath in structural VHDL format and a set of distributed controllers in behavioral FSM style (these RT-level VHDL files will be fed into logic synthesis tools), and floorplan and multi-cycle path constraints for the downstream place-and-route tools.

To obtain the final performance results, Altera's Quartus II version 2.2 is used to implement the datapath portion into a real FPGA device, the Stratix^{TM} EP1S40F1508C5 [12]. All the pipelined multipliers are implemented into the dedicated DSP blocks of the Stratix^{TM} device. For data-flow-intensive examples, the authors obtained a 44% improvement on average in terms of the clock period and a 37% improvement on average in terms of the final latency compared to a traditional non-layout-driven flow. For designs with control flow, their approach achieved a 28% clock-period reduction and a 23% latency reduction on average.

5.2.3 Other techniques

There are other types of studies on specific behavioral synthesis tasks for FPGA designs, such as loop transformation, branch optimization, memory allocation, module selection and resource sharing, and communication optimization. We will briefly introduce these works.

Loop transformation. In [172] the authors tried to develop fast and accurate performance and area models to quickly understand the impact and interaction of program transformations. They presented a combined analytical performance and area modeling approach for

complete FPGA designs in the presence of loop transformations. Their approach took into account the impact of I/O memory bandwidth and memory interface resources (often the limiting factor in the effective implementation of computations). The preliminary results revealed that their modeling was accurate, and was therefore amenable to being used as a compiler tool to quickly explore very large design spaces.

Branch optimization. In [177] the authors explored using the information about program branch probabilities to optimize reconfigurable designs. The basic premise is to promote utilization by dedicating more resources to branches that execute more frequently. A hardware compilation system was developed for producing designs that were optimized for different branch probabilities. The authors proposed an analytical queuing network performance model to determine the best design from observed branch probability information. The branch optimization space was characterized in an experimental study of two complex applications for Xilinx Virtex FPGAs: video feature extraction and progressive refinement radiosity. For designs of equal performance, branch-optimized designs require 24% and 27.5% less area. For designs of equal area, branch optimized designs run up to three times faster.

Memory allocation. In [95] the authors observed that FPGA-based processors, like many conventional DSP systems, often associate small high-performance memories with each processing chip. These memories may be onboard embedded SRAMs or discrete parts. In the process of mapping a computation onto an FPGA processor, it is necessary to map the application's data to memories. The authors presented an algorithm that had been implemented in their NAPA C compiler to assign data automatically to memories to produce a minimum overall execution time of the loops in the program. The algorithm used a search technique known as implicit enumeration to reduce the otherwise exponential search space. They showed the correctness of their implementation.

Module selection and resource sharing. In [179] the authors developed a synthesis methodology that generated pipelined data-path circuits from a high-level data-flow specification. This methodology carried out module selection (selecting a module implementation from

a variety of circuit implementation options for each operation) and resource sharing (i.e., resource binding) together. They used a module library created for Xilinx Virtex-4 architecture. They presented two types of algorithms. One was based on a recursive branch and bound algorithm. However, the runtime complexity was high. Then they presented another algorithm based on iterative modulo scheduling with a backtracking feature. Their objective was to minimize the area cost of the resulting circuit while meeting a user-specified minimum throughput constraint. They showed that even for small benchmark circuits, combining module selection and resource sharing could offer significant area savings relative to applying them alone.

Communication optimization. In [53] the authors proposed a communication synthesis approach targeting systems with sequential communication media (SCM). Since SCMs require that the reading sequence and writing sequence must have the same order, different transmission orders may have a dramatic impact on the final performance. The goal of their work was to consider behaviors in communication synthesis for SCM, detect appropriate transmission order to optimize latency, automatically transform the behavior descriptions, and automatically generate driver routines and glue logics to access physical channels. They showed that solving the order detection problem was equivalent to solving the resource-constrained scheduling problem. Since the scheduling problem with resource constraints is NP-complete in general, they used a list-scheduling-based heuristic algorithm to tackle this problem. To deal with loops in CDFG, they completely expanded the loop iteration space and used iteration reordering techniques to generate reconstructed loops and reduced associated storage overhead. To get real simulation numbers, they developed a FIFO module in VHDL which resembled the behaviors of the Xilinx FSL (Fast Simplex Link) [205]. The algorithm, named *SCOOP*, achieved an average 20% improvement in total latency on a set of real-life benchmarks compared to the results without optimization.

6

Power Optimization

With the exponential growth in the performance and capacity of integrated circuits, power consumption has become one of the most critical design factors in the IC design process. FPGAs are not power-efficient. The post-fabrication flexibility provided by these devices is implemented using a large number of prefabricated routing tracks and programmable switches. Also, the generic logic structures in FPGAs consume more power than the dedicated circuitry that is found in an ASIC. It has been shown that a typical FPGA chip consumes about 50 to 100X more power than a functionally equivalent ASIC chip [120, 221]. It is projected that a high-end FPGA chip with 7 million logic cells using 35 nm technology can consume close to 200 W power [92]. This power dissipation level is almost equivalent to that of a high-performance microprocessor using the same technology that has 3.5 billion transistors and runs 20X faster [1]. The large power consumption of FPGA chips limits its use in mainstream low-power applications. Meanwhile, large power consumption and heat dissipation typically lead to higher costs for thermal packaging, fans, and electricity, and also have a negative impact on signal integrity. The growing demand of power reduction for FPGA designs has caught the

attention of both industry and academia. A large amount of research has been published in this area, especially in the past five years. We will touch on power estimation, power breakdown, synthesis for power minimization, and synthesis and design for novel power-efficient FPGA architectures in this section. Low-power RTL and high-level synthesis techniques will be discussed separately in the next section.

6.1 Sources of Power Consumption

There are three sources of power consumption in FPGAs: (1) switching power, (2) short-circuit power, and (3) static power. The first two types of power can only occur when a signal transition happens at the gate output; together they are called *dynamic power*. There are two types of signal transitions: one is the signal transition necessary to perform the required logic functions between two consecutive clock ticks (called *functional transition*); the other is the unnecessary signal transition due to the unbalanced path delays to the inputs of a gate (called *spurious transition* or *glitch)*. Glitch power can be a significant portion of the dynamic power. Static power is the power consumption when there is no signal transition for a gate or a circuit module. As technology advances to feature sizes of 90 nm and below, static power starts to become a dominating factor in the total chip power dissipation.

Switching power can be modeled by the following formula:

$$P_{\text{sw}} = 0.5f \cdot V_{dd}^2 \cdot \sum_{i=1}^{n} C_i S_i \tag{6.1}$$

where n is the total number of nodes, f is the clock frequency, Vdd is the supply voltage, C_i is the load capacitance for node i, and S_i is the transition density (switching activity) for node$i$. Switching activity is the average number of transitions ($0 \rightarrow 1$ or $1 \rightarrow 0$) a signal switches per unit time.

Short-circuit power is another type of dynamic power. When a signal transition occurs at a gate output, both the pull-up and pull-down transistors can be conducting simultaneously for a short period of time. Short-circuit power represents the power dissipated via the

direct current path from Vdd to GND during this period of time. It is a function of the input signal transition time and load capacitance.

Static power is also called leakage power. There are primarily three types of leakage power: sub-threshold leakage power, reverse-biased junction leakage power, and gate leakage power. Sub-threshold current is the weak inversion conduction current that flows between the source and drain of a MOSFET when the gate voltage is below the threshold voltage. Sub-threshold leakage is the dominant factor in leakage power. It is exponentially related to threshold voltage and temperature, as modeled by the following formula:

$$I_{\text{sub}} \propto \frac{W}{L} \exp\left(\frac{V_{\text{GS}} - V_{\text{TH}}}{n \cdot V_t}\right) \tag{6.2}$$

where W and L are the effective width and length of the device, V_{gs} is gate voltage, V_{th} is threshold voltage, $V_T = kT/q$ is thermal voltage (k and q are constants, and T is temperature), and n is a technology-dependent parameter.

MOS transistors have reverse biased pn junctions from the drain/source to the well. The reverse biased pn junctions give rise to static current passing across the junctions. This leakage is a function of junction area and doping concentration.

As gate oxide thickness scales down, there occurs an increased probability of direct tunneling current through the gate oxide. There are three components of gate leakage namely: gate leakage between the gate and the drain, between the gate and the substrate, and between the gate and the source. Although gate leakage is becoming increasingly important, it will have to be controlled with other techniques such as high-k dielectrics. Figure 6.1 schematically illustrates the various types of leakage currents.

6.2 Power Estimation

Power estimation is an important task for FPGAs. FPGA designers rely on power estimation tools in order to predict the power consumption of circuits and discover possible power violations during the design process. Power estimation also serves as the foundation for power optimization. It is well known that higher the design level the larger the

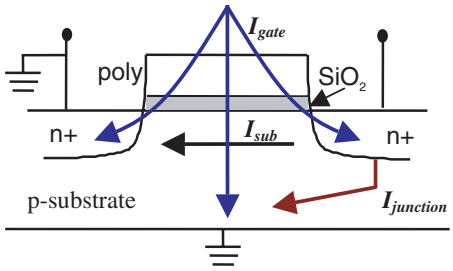


Fig. 6.1 Various leakage currents.

impact of power reduction techniques. Therefore, power estimation is required across the various levels in the FPGA design hierarchy in order to effectively control and reduce the power consumption of the end product.

As technology node scales down, power consumption in interconnects becomes the dominant source in sub-micron FPGAs. They can contribute 75–85% of the total power [120, 131] for most of the FPGA designs. Consequently, power estimation for FPGAs must consider routing interconnect capacitance. Interconnect estimation can be done in different design levels. The estimation becomes increasingly accurate as the design enters lower design levels. After placement and routing, wire-capacitance can be more accurately captured and back-annotated to the original netlist for better power estimation. However, there is a tradeoff between accuracy and runtime complexity. Although high-level interconnect estimation is not as accurate as low-level estimation, its runtime is much faster, which will be beneficial when using high-level synthesis to explore low-power design possibilities.

Most FPGA companies provide online spreadsheets for their customers to estimate power dissipation for particular devices in early design stages [7, 122, 208]. Some vendors, such as Xilinx and Altera, have incorporated the power estimation feature in their CAD tools, such as XPower [211] and PowerPlay [8], which can be launched after placement and routing for a more accurate power analysis of the design. We will introduce dynamic and static power estimation in Sections 6.2.1 and 6.2.2, and then we will briefly summarize the power estimation works published in the literature.

6.2.1 Dynamic power estimation

Dynamic power estimation is mainly concerned with switching activity estimation and load capacitance estimation (Formula 6.1). Both gates (including buffers) and wires contribute capacitance in the circuit. Gate-related capacitance is usually easy to obtain because FPGA chips are already fabricated and their gate sizes are already known. Wire capacitance estimation is more demanding. Gate-level power estimators can first perform placement and routing for more accurate capacitance extraction. The extracted capacitance can then be back-annotated to the power estimation flow for better estimation. For high-level power estimation, detailed placement and routing are usually not available. These estimators will have to rely on wire-length estimation methods, such as *Rent's* rule-base methods, for the estimation of the total amount of wires and wire capacitance involved in the design.

There are primarily three approaches reported in literature for FPGA switching activity estimation namely: characterization through board measurement, statistical model, and simulation model. The work in used an emulation board embedded with a Virtex FPGA for power measurement. The authors then calculated the average switching activity for logic elements using a power estimation formula published by Xilinx. The formula is as follows [212]:

$$P_{\mathrm{INT}} = V_{\mathrm{Core}} \cdot K_p \cdot f_{\mathrm{Max}} \cdot N_{\mathrm{LC}} \cdot Tog_{\mathrm{LC}} \tag{6.3}$$

P_{INT} is the internal power consumption caused by the charging and discharging of the capacitance on each logic element that is switched. V_{Core} is the core voltage; K_p is a technology-dependent constant; f_{Max} is the maximum clock speed; N_{LC} is number of logic elements used; and Tog_{LC} is the average switching activity of all the logic elements.

Works in [136, 162, 171] used statistical models to estimate switching activity. The *static probability* of a signal x, denoted by $P(x)$, is defined as the probability that signal $x$has the logic value 1. For each LUT in the circuit, the function implemented in that LUT can be expressed as a function $y = f(x_1,x_2,\ldots,x_n)$. For each input, x_i, two new Boolean functions fx_i and fx_i' can be generated by setting input x_i to 1 and 0, respectively in $f(x_1,x_2,\ldots,x_n)$ (these functions are called

cofactors of y with respect to x_i). The *Boolean difference* of the output with respect to an input, x_i, can be calculated by:

$$\frac{\partial y}{\partial x_i} = fx_i \oplus fx_i' \tag{6.4}$$

where $\oplus$ denotes the excusive-OR operation. The probability of this Boolean difference, $P(\partial y/\partial x_i)$, is the static probability that a transition inx_i causes a transition at the output. The switching activity at the output $S(y)$ can be calculated as follows:

$$S(y) = \sum_{i=1}^{n} P\left(\frac{\partial y}{\partial x_i}\right) S(x_i) \tag{6.5}$$

In [162] the authors assumed that all primary inputs have a static probability of 0.5 and a switching activity of 0.5. Notice that this model also assumes that the switching activities of input signals are not correlated, which is usually not the case. It is also hard for this model to capture glitch power.

The third type of switching activity estimation is based on simulation. A sequence of random input vectors can be applied on the primary inputs, and cycle-accurate gate-level simulation can be carried out for the whole circuit. Combined with back-annotated delay information available after placement and routing, this estimation model is most accurate for switching activity calculation because it can also capture activities due to glitches. Works in [131, 134] used this model. The down side of this approach is its larger runtime.

After switching power is estimated, short-circuit power can be estimated proportionally to the switching power. Some work used a fixed ratio. For example, [162] assumed that short-circuit power is always 10% of the total dynamic power. Some work developed detailed models to evaluate the short-circuit power. For example, [134] used a linear curve fitting method to derive the ratio between the short-circuit power and the switching power. This ratio is a linear function of the input transition time in the model. They reported that the short-circuit power is a significant power component due to the large signal transition time in FPGA designs. It can reach 70% of the global interconnect dynamic power for certain designs [134].

6.2.2 Static power estimation

We will introduce two approaches reported in the literature for FPGA static power estimation: the analytical method and the macro-modeling method. The analytical method estimates the values of various parameters involved in the calculation of the leakage. For example, the work in [162] used a detailed formula (with similar parameters as presented in Formula 6.2) to estimate the sub-threshold leakage. Given a specific process technology, they estimated the values of various parameters, such as the device width, channel length, and temperature. They also made some assumptions in the calculation. For instance, they assumed that the V_{gs} value was half of the threshold voltage V_{th}. They reported that the average error between the estimated values and the simulated results was 13.4%.

Micro-modeling mainly relies on SPICE simulation to achieve estimation results. For example, the work in [134] used SPICE simulation with randomly generated input vectors to obtain the average leakage power in the LUT. Since the number of all possible input vectors increases exponentially with the number of inputs for LUTs, it is infeasible to try all the input vectors for large-input LUTs. Therefore, different input vectors were mapped into a few typical vectors with representative Hamming distances and SPICE simulation was performed only for these typical vectors to build macromodels in [134]. With this model, [134] performed simulation for LUT sizes ranging from three to seven and buffers of various sizes in global/local interconnects, and then built the static power macromodels.

6.2.3 Power estimation works

We will briefly introduce several gate-level and high-level power estimation publications in this subsection. In [120] people used a Xilinx XC4003A FPGA test board to carry out power dissipation measurement, characterize capacitance of various FPGA components, and report the power breakdown of these components. In [171], the authors analyzed the dynamic power consumption and distribution for the Xilinx Virtex-II FPGA family. The work in [197] presented the power consumption estimation for the Xilinx Virtex architecture using their emu-

lation environment. Based on board measurement, the authors calculated the technology-dependent power factor K_p (Eq. (3)) and derived their own power estimation formulas. They reported a 5% estimation error for certain designs. However, they did not produce good estimation results for designs that were dominated by large combinatorial logic blocks like multipliers. The work in [162] presented a flexible FPGA power model associated with architectural parameters. This model estimates dynamic and leakage power for a wide variety of FPGA architectures. The authors in [14] developed an empirical estimation model and showed that estimation accuracy was improved by considering aspects of the FPGA interconnect architecture in addition to generic parameters, such as net fanout and bounding box perimeter length. The work in [190] made a detailed analysis of leakage power in Xilinx CLBs. It concluded that a significant reduction of FPGA leakage was needed to enable the use of FPGAs in mobile applications. Authors in [131, 134] developed a mixed-level FPGA power model that combines switch-level models for interconnects and macromodels for LUTs and flip-flops. It carried out gate-level simulation under real delay models and was able to capture glitch power. Work in [134] reported high fidelity compared to SPICE simulation, and the absolute estimation error was 8% on average.

There is limited high-level power estimation work for FPGAs in academia. Authors in [170] presented a high-level power modeling technique to estimate the power consumption of FPGAs. They captured the relationship between FPGA power dissipation and I/O signal statistics. Then they used an adaptive regression method to model the FPGA power consumption. Experimental results indicated that the average relative error was 3.1% compared to a low-level FPGA power simulation method for FPGA components such as ALUs, adders, DSP cores, etc. There was no report for power estimation for larger designs. Authors in [41] developed a high-level power estimator. It used a fast switching activity calculation algorithm, a *Rent*'s rule-based wire-length estimation [78], and a resource characterization flow using *DesignWare* library from Synopsys [181]. It takes into account various FPGA components, such as LUTs, local and global buffers, MUXes, etc. Later on, the same authors extended this power model to work on a real FPGA archi-

tecture, Altera's Stratix architecture [12], during high-level synthesis. Some device-specific functional blocks are considered in the model, such as memory blocks, DSP clocks, and I/O blocks. Experimental results showed an 18.5% average estimation error [77] compared to Altera's gate-level PowerPlay power analyzer using real designs.

6.3 Power Breakdown

In the FPGA architecture evaluation work for power [134], the authors concluded that logic block size = 6 and LUT input size = 7 represent the min-delay architecture, and logic block size = 8 and LUT input size = 4 represents the min-energy architecture under the 100 nm technology. The energy consumption difference between these two architectures is 48%, and the critical path delay difference is 12%. Figure 6.2 shows the breakdown of various power sources for the min-energy architecture [134].

Figure 6.3 presents the power breakdown for average designs using Altera's Stratix II FPGA architecture (90 nm technology) [11]. Total device power is the sum of three components: core dynamic power, core

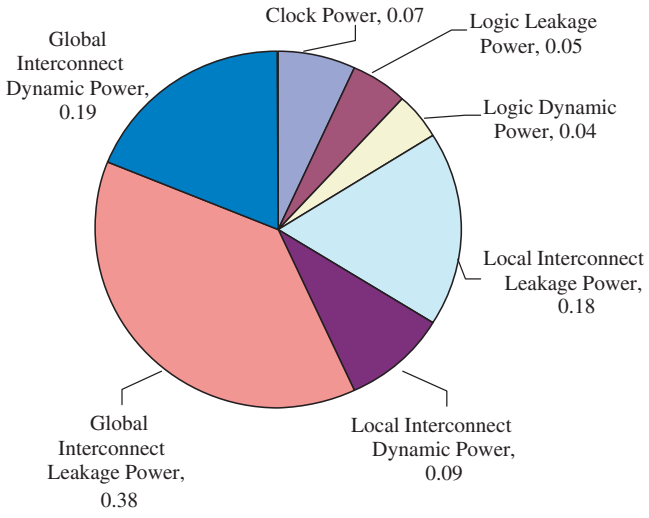


Fig. 6.2 Power breakdown of the min-energy architecture in [134].

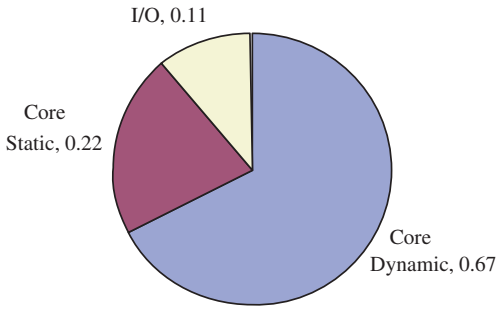


Fig. 6.3 Power breakdown of average designs using Altera's Stratix II FPGA in [11].

static power, and I/O power. Core dynamic power is the power dissipated by the operation of the FPGA core fabric. Core static power is the leakage power dissipated, which can be determined by stopping all clocks (an operating frequency of 0 MHz). I/O power is the power dissipated in the FPGA I/O cells when communicating with other chips. This data was obtained by estimating the power consumption of 99 complete designs with the Quartus II software version 5.0 SP1 PowerPlay power analyzer [11]. We can observe that leakage power in this figure represents a smaller portion of the total power than what has been shown in Fig. 6.2. We believe that the main reason for this is the architecture in [134] does not consider circuit-level optimization for leakage power reduction. However, there are some circuit optimization techniques for leakage power reduction in Stratix II FPGAs, such as higher V_{th} and longer transistor length for non-speed-critical paths [9]. We shall present leakage power minimization techniques in Section 6.5.1.

Figure 6.4 shows the power breakdown for Xilinx's Spartan-3 devices [189]. Dynamic power is estimated at 150 MHz clock frequency, 12.5% average switching activity, and typical configuration and resource utilization as determined by user benchmarks. Static power measures both subthreshold leakage and gate leakage [189]. The static power is about 10% of the total power consumption. We can observe that routing switches make up the largest part of the total dynamic power, and both routing switches and configuration SRAM represent significant parts of the total static power.

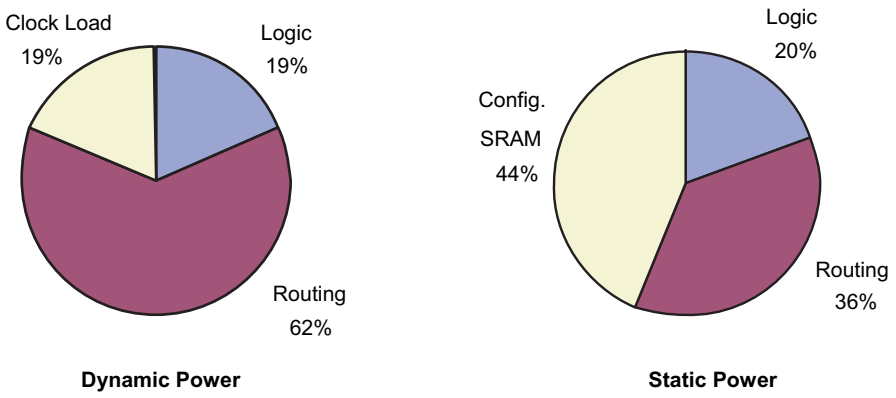


Fig. 6.4 Power breakdown of average designs using Xilinx's Spartan-3 in [189].

6.4 Synthesis for Power Optimization

In this section, we present low-power synthesis techniques for existing FPGA architectures, where performance is the main objective of the architecture design. However, power efficiency can be a synthesis objective with performance constraints. We will touch on technology mapping, circuit clustering, RTL synthesis, behavioral synthesis, and some other synthesis techniques for power minimization.

6.4.1 Technology mapping for low power

FPGA technology mapping for low power is a NP-hard problem [86]. Some heuristics have been proposed. These algorithms mainly worked on reducing the overall switching activity of the design. The smaller the total switching activity, the lesser the power consumption (Eq. (1)). In [86], besides the NP-hard proof, the authors also presented a heuristic for low-power mapping. They pointed out that the power consumed by a LUT depended on the switching activity and the fanout number of the LUT, and gave a formula to estimate the total power consumption of a technology mapping solution. The main idea of their algorithm was to hide nodes with higher switching activities inside LUTs (and hence, LUTs would have smaller switching activity at their outputs). The goals of most other algorithms were similar in terms of switching activity

reduction. In [13] a cut-enumeration-based algorithm was designed to keep nets with high switching activities out of the FPGA routing tracks. It also considered switching activity when logic replication was needed for optimizing mapping depth of the design. The authors reported that the result of FlowMap-r[51]+MP-Pack[46] required 14.2% more power than their algorithm when both were depth-optimal. When the mapping depth was relaxed by one level, they reported further power reduction by about 8% over the depth-optimal case for 4-LUTs and 10% for 5-LUTs. The authors in [193] and [121] used cut enumeration as well. In [195] both run time and memory space were considered and only a fixed number of cuts were performed. It reported up to a 14.18% power savings compared to [86]. The mapping algorithm in [121] designed a cost function for each cut, including switching activity, fanout number, node duplication consideration, etc. Authors reported an 8.4% energy reduction over CutMap [58]. The authors in [136] used a network flow formulation and carried out mapping while looking ahead at the impact of the mapping selection on the power consumption of the remaining network. An extension was also presented that computed depth-optimal mapping. They reported a 14% power savings without any depth penalty compared to CutMap.

6.4.2 Circuit clustering for low power

Clustering has traditionally been used in the VLSI industry to extract underlying circuit structures and construct a natural hierarchy in the circuits. In [164] the authors derived the first delay optimal clustering algorithm under the general delay model. In [192] the authors presented a low-power clustering algorithm with the optimal delay. Their algorithm is power optimal for trees. They enumerated all clustering solutions for a graph and selected a low-power clustering solution from all delay optimal clustering solutions. It has been shown that logic cluster-based FPGA logic blocks can improve FPGA performance, area, and power [2, 22, 174]. Section 4.1 presented some clustering algorithms optimizing FPGA area and performance. There are a few prior research efforts on clustering for low-power FPGA designs as well. An FPGA circuit-clustering work was reported in [121] as one of the optimization

steps in power-aware CAD flow, which tried to attract nets with high switching activity inside the logic blocks. It was 12.6% better than T-VPACK [22] in terms of energy consumption. Researchers presented a routability-driven clustering technique for area and power reduction in FPGAs [174]. It used *Rent's* rule-based estimation method to reduce the potential routing complexity due to clustering. In [77] a delay-optimal clustering algorithm was presented to improve FPGA performance and reduce FPGA power. The algorithm was delay and power optimal for trees. It also presented some heuristic to control duplications. The authors reported a 9% improvement on delay reduction with a 3% power overhead compared to [121].

6.4.3 RTL synthesis for low power

In [201] the authors worked on RTL synthesis for FPGA power minimization. They first characterized the power and performance data for the functional units on their targeted FPGAs using board measurement. For example, they characterized the power and delay of different implementations of adders, such as ripple adder, carry lookahead adder, conditional sum adder, etc. Then, their design flow took an RTL specification and began to tradeoff power with circuit speed by selecting different implementations of components iteratively. Figure 6.5 shows the optimization flow. They showed that their methodology was useful for designing a low-power digital filter.

6.4.4 Behavioral synthesis for low power

As we have mentioned before, multiplexers are particularly expensive for FPGA architectures. In general, when there is a smaller number of functional units or registers allocated but there is a larger number of wide multiplexers and larger amount of interconnects, it may lead to a completely unfavorable solution for both the performance and the area/power cost. Tackling this increasingly alarming problem will require an efficient search engine to explore a sufficiently large solution space considering multiple constraining factors—such as resource allocation and binding, MUX generation, and interconnection

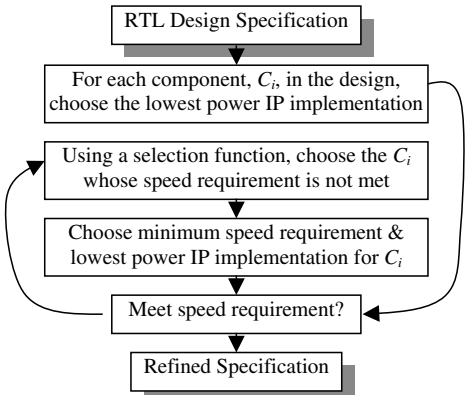


Fig. 6.5 Power optimization flow presented in [201].

generation—for optimizing performance or cost, or studying the trade-off between them.

With such a motivation, a behavioral synthesis engine for FPGA power optimization was presented in [41]. Here, the power optimization goal is to search a combined solution space for the subtasks in behavioral synthesis so that the power of FPGA designs can be optimized, and at the same time the performance/latency target can still be met. To achieve this goal, the authors adopted a simulated annealing-based algorithm. For each move during the annealing procedure, the optimization engine generates the full data path to capture the overall cost, considering all the contributing factors in the design. The cost function is the estimated power dissipation guided by the behavioral level power estimator. The algorithm carried out resource selection, scheduling, function unit binding, register binding, and steering logic and interconnection estimation simultaneously. Figure 6.6 shows a block diagram of the power optimization engine.

There are five different types of moves performed during simulated annealing. They are different functional unit binding operations. The moves are listed below:

Reselect: Select another FU of the same functionality but with a different implementation. For example, select a carry look-ahead adder to

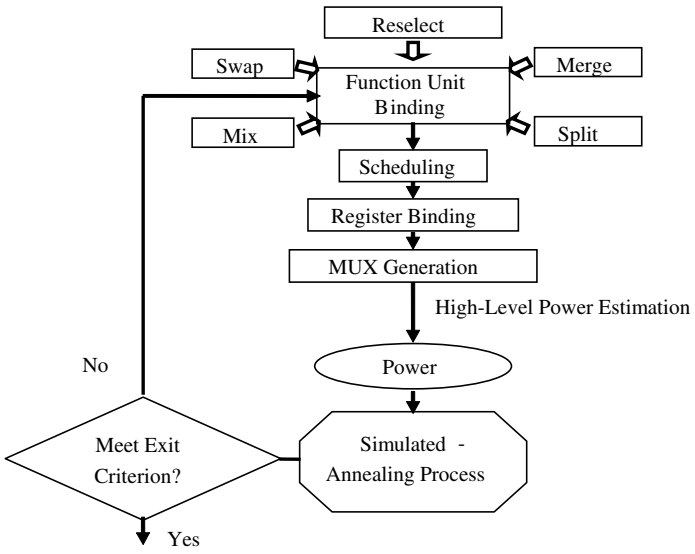


Fig. 6.6 The power optimization engine in [41].

replace a Brent-Kung adder. The operations bound to the adder stay unchanged.

Swap: Swap two bindings of the same functionality but with different implementations. This is equivalent to two *reselects* between two FUs in each direction.

Merge: Merge two bindings into one, i.e., the operations bound to the two FUs are combined into one FU. As a result, the total number of FUs decreases by 1. The two FUs have to be the same type.

Split: Split one binding into two. It is the reverse action of *merge*. As a result, the total number of FUs increases by 1. The operations of the original binding are distributed into the new bindings randomly.

Mix: Select two bindings, *merge* them, sort the merged operations according to their slacks, and then *split* the operations. For example, if there are N operations after sorting, operations 1 to $N/2$ will form one binding and the rest of the operations will form another binding.

An interconnection optimization step was also designed to reduce the interconnections between functional units and registers through

multiplexer optimization. Experimental results showed that a power reduction of 35.8% was achieved compared to the results of the Synopsys Behavioral Compiler [181].

6.4.5 Other techniques

There are also other works that focus on FPGA power minimization. We introduce two of them here. In [16] the authors considered active leakage power dissipation in FPGAs and presented a "no cost" approach for active leakage reduction. Leakage power consumed by a digital CMOS circuit depends strongly on the state of its inputs. This leakage reduction technique leveraged a fundamental property of LUTs which says that a logic signal in an FPGA design can be interchanged with its complemented form without any area or delay penalty. The authors applied this property to select polarities for logic signals so that FPGA hardware structures spent the majority of time in low leakage states. They optimized leakage power in circuits mapped into a 90 nm commercial FPGA. Results showed that the proposed approach reduced active leakage by 25%, on average. In [106] the authors presented a method to re-synthesize LUT-based FPGAs for low power design after technology mapping, placement and routing were performed. They used the set of pairs of functions to be distinguished (SPFD) method to express functional permissibility of each signal. Using different propagations of SPFD to fan-in signals, they changed the functionality of a CLB which drives a large load into one with low transition density. Experimental results showed that their method on an average could achieve, a 12% power reduction compared to the original circuits, without affecting placement and routing.

6.5 Synthesis for Power-Efficient Programmable Architectures

In this section, we present synthesis techniques for power-efficient FPGA architectures, where low power is the main objective of the architecture design with performance being the secondary objective. For example, this type of FPGA chip may support multiple supply voltages,

multiple threshold voltages, and power gating features to reduce both dynamic and leakage power.

As silicon technologies advance, smaller geometries become possible with lower Vdd. At the same time, threshold voltage V_{th} must be reduced in order to maintain or improve circuit performance. This decrease of V_{th} then drives significant increases in leakage power. As silicon technologies move into 90 nm and below, leakage currents become as important as active power in many applications. To reduce both dynamic power and leakage power, deploying multiple Vdd and V_{th} is a popular design technique. These techniques have been extensively used for ASIC designs [17, 153, 176, 183, 191, 196]. Low-Vdd reduces dynamic power, and high-V_{th} reduces leakage power, but each incurs a longer signal delay. If low-Vdd and/or high-V_{th} are only applied to non-critical paths carefully, the multi-Vdd/V_{th} technique has the advantage of reducing power dissipation without sacrificing system performance. Specific to FPGAs, multi-Vdd/V_{th} fabric and layout pattern must be pre-defined, because FPGAs do not have the freedom of using mask patterns to arrange different Vdd/V_{th} components in a flexible way (as in ASICs). This brings unique challenges for FPGA designers. We will introduce several recent research efforts in this area.

6.5.1 Leakage power reduction

Circuit design techniques can be applied to reduce leakage power for FPGAs. As mentioned before, SRAM-based FPGAs use a large amount of SRAM cells to provide programmability for both logic cells and interconnects. The work in [135] and [163] introduced high-V_{th} for SRAM cell design. Increasing V_{th} for SRAM cells in FPGAs has no delay penalty during normal operation of the FPGA. However, it will increase the SRAM write access time and slow down the FPGA configuration speed. It was shown that the V_{th} of SRAM cells could be increased to achieve a 15X SRAM-leakage reduction with only a 13% configuration time increase [135]. Figure 6.7 shows the schematic of a 4-LUT with dual-V_{th} (denoted as V_t in the figure) regions, where Region I presents the high-V_{th} region, and Region II for low-V_{th} region.

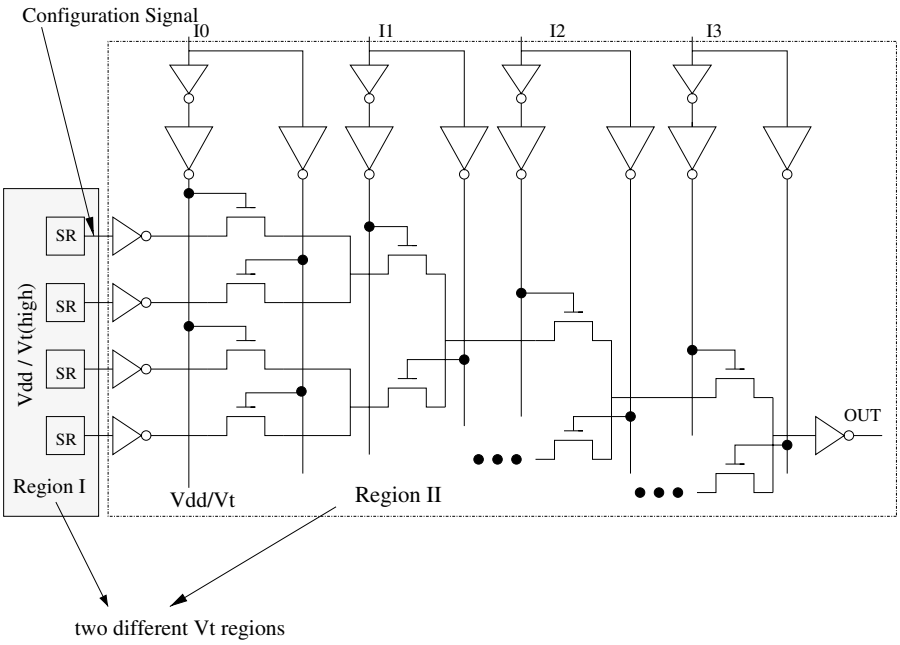


Fig. 6.7 Two different V_{th} applied to a 4-LUT cell in [135].

The authors in [163] proposed several circuit enhancement techniques for leakage power reduction, including redundant memory cells, dual-V_{th} devices, and body biasing. Specifically, they targeted low-leakage multiplexer designs because multiplexers are widely used in SRAM-based FPGAs. Figure 6.8(a) shows a two-stage implementation of a pass transistor-based multiplexer. It is composed of several smaller multiplexers, and the same SRAM cell configures one pass transistor from each multiplexer in stage 1. As a result, whenever there is an enabled input-to-output path, the intermediate nodes, such as nodes 1, 2, 3, and 4, are driven to Vdd. Therefore, the drain-to-source voltage, V_{DS} of all disabled pass transistors is Vdd, and these pass transistors still contribute leakage power. Figure 6.8(b) shows the schematic after adding some redundant memory cells in the multiplexer design. These SRAM cells can turn the inactive input-to-output paths off (e.g., the left upper portion in Fig. 6.8(b)) and reduce leakage power. These SRAM cells can be implemented using high-V_{th} devices. In the authors'

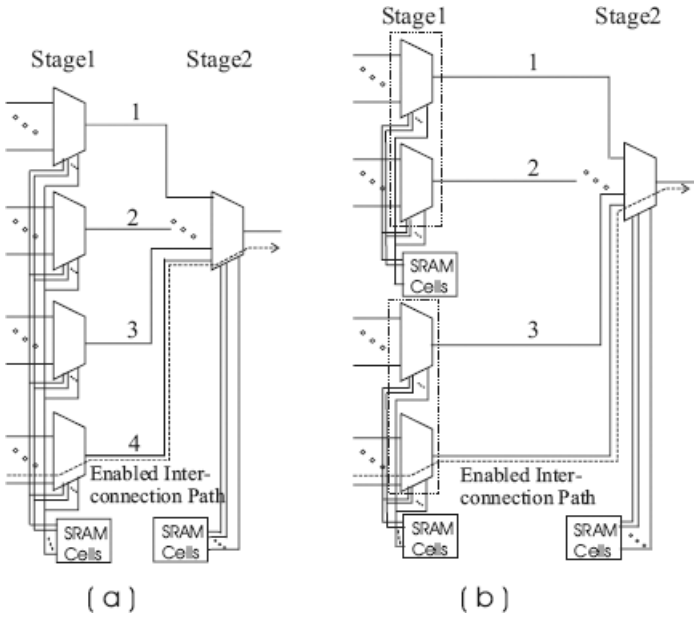


Fig. 6.8 (a) A two-stage traditional implementation of a pass transistor-based multiplexer; (b) A two-stage multiplexer with redundant memory cells for turning off inactive circuit portion in [163].

analysis, the V_{th} of transistors in SRAM cells is 25% higher than that of typical devices. They reported a 2X leakage power reduction with a 15–30% total chip area increase through use of this technique. They also studied the performance/leakage-power tradeoff scenarios when V_{th} is increased for the devices in routing switches.

The work in [91] divided the FPGA fabric into small regions and switched on/off the power supply to each region using a sleep transistor in order to reduce leakage energy. The regions not used by the placed design were power gated. The authors presented a placement strategy to increase the number of regions that could be power gated.

The authors in [189] described the design and implementation of *Pika*, a low-power FPGA core targeting battery-powered applications such as those in consumer and automotive markets. They used several key leakage power reduction techniques. First, they applied low-leakage configuration SRAMs by adopting mid-oxide, high-V_{th} transistors. This

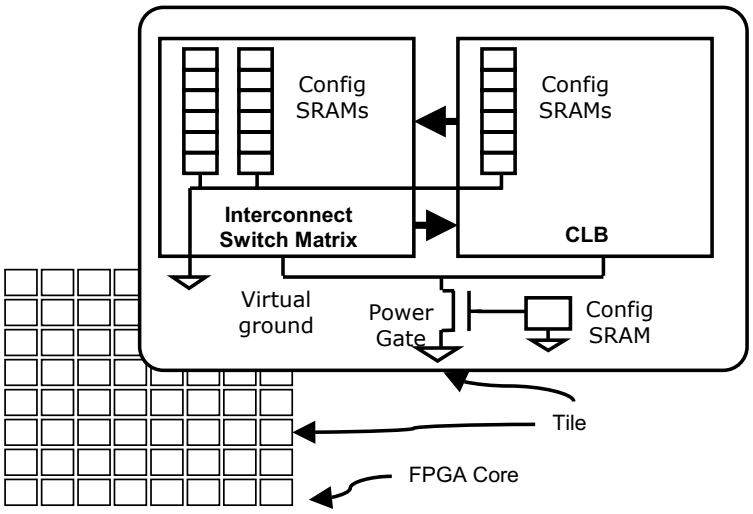


Fig. 6.9 Power gating architecture in [189].

corresponded to a reduction of 43% to the total standby power of the FPGA core. Second, they applied power gating at the level of individual tiles (each tile consists of a CLB and a programmable interconnect switch matrix). They used mid-oxide power gates and sized them at the point of 10% performance degradation. They did not power gate the configuration SRAM cells to enable a state-retaining standby mode when all logic and routing were power gated. Figure 6.9 shows their power gating architecture. Third, they made circuit modifications to prevent high-current paths. A high-current path may be a short-circuit path from supply to ground, or a high-leakage path that does not go through a power gate (refer to [189] for details). They reported that standby power was reduced by 99% when the chip was idle. The power optimizations incurred a 27% performance penalty and 40% area increase. They also reported that the core woke up from standby mode in approximately 100 ns.

6.5.2 Dynamic power reduction

A dual-Vdd FPGA with pre-defined voltage patterns was studied in [135]. Figure 6.10 shows the voltage patterns explored in this work.

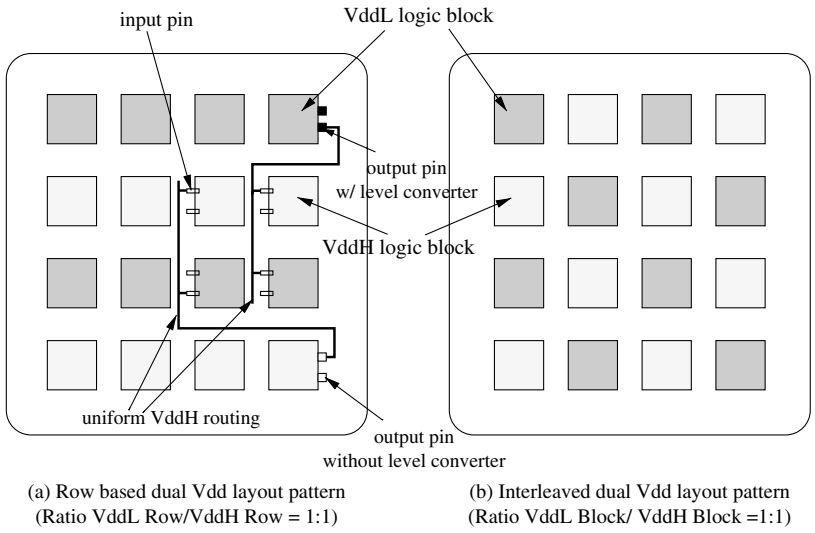


Fig. 6.10 Fixed dual-Vdd layout patterns in [135].

The architecture had the advantage of small area overhead and simple voltage regulation. The authors designed FPGA architectures with the support of dual supply voltages and dual threshold voltages and developed placement tools on top of the architecture. The placement was based on a simulated-annealing method modified from the one used in VPR [22]. The cell swapping during the simulated-annealing process considered different voltage assignments on the physical locations of the CLBs. It tried to reduce power while avoiding deterioration of the critical path delay. However, the authors found that it did not provide good opportunities for power reduction while still maintaining competitive circuit performance. This was due to the fixed voltage pattern that imposed strong constraints on the placement engine. Later on, researchers proposed a new architecture that provided voltage configurability for each CLB [132]. They inserted two PMOS transistors between the high-Vdd and low-Vdd power rails and the CLB (Fig. 6.11). Therefore, each CLB could be configured as driven by either the low-Vdd or high-Vdd. The area overhead of sleep transistors was 24% over the original CLB area with a 5% delay overhead. The flexibility offered through this architecture helped the placement

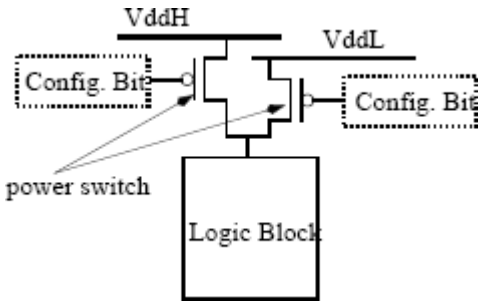


Fig. 6.11 Configurable Vdd for a logic block in [132].

engine to place cells on critical paths into CLBs configured to high-Vdd, and cells on non-critical paths into CLBs configured to low-Vdd. A total of 14% power reduction was reported [132]. In [133] the same authors designed a dual-Vdd architecture to reduce the interconnect power of FPGAs. They designed three Vdd states for interconnect switches namely: high Vdd, low Vdd and power-gating. This is similar to the technique used in [132], where each CLB can be Vdd-configured or power-gated. The authors developed a design flow to apply high Vdd to critical paths and low Vdd to non-critical paths and to power gate unused interconnect switches. They reported a significant amount of power savings. In [149], a partitioning algorithm for FPGAs with pre-defined voltage patterns (voltage island configurations) was presented. This power-driven partitioner created partitions of critical and non-critical CLBs and assigned these CLBs to different voltage islands according to their timing criticalities, followed by placement and routing. It showed that a dynamic power gain as high as 47% was possible with a 17% area/delay product penalty and a 30% power gain, with an area/delay product penalty as low as 6% for different voltage island configurations. In [104], the authors developed a technique to estimate power reduction using dual-Vdd for mixed length interconnects, and applied linear programming (LP) to solve slack budgeting to minimize power for mixed length interconnects. Experiments showed 53% power reduction on average compared to single-Vdd interconnects. Furthermore, this paper presented a simultaneous retiming and slack budgeting algorithm to reduce power in dual-Vdd FPGAs considering placement

and flip-flop binding constraints. The algorithm was based on mixed integer and linear programming (MILP) and achieved up to 20% power reduction compared to retiming followed by slack budgeting.

Low-power technology mapping and circuit clustering for FPGAs with dual Vdds were presented in [42] and [39], respectively. In [42] the authors used a cut-enumeration-based technique. They developed a detailed delay and power model for LUTs and level converters using different voltages. The algorithm built all the cases of LUT connections under dual-Vdd scenarios and generated one set of power and delay results for each case to enlarge the low-power solution search space. Their algorithm improved power savings by 11.6% on average over the single-Vdd case when both algorithms produced optimal mapping depth. In [39] the authors used a solution curve propagation technique to examine the quality of different clustering solutions. They built all the non-inferior *delay-power-Vdd* solution points for a node with considerations of dual Vdds and level converter delay. The algorithm is delay and power optimal for trees and delay optimal for DAGs under the general delay model. A limitation of the work was that each generated cluster (to be implemented by a logic block) was a single output cluster, which could cause a large area overhead. Later on, the authors extended the work to generate clustering solutions with multiple cluster outputs [77] while maintaining the optimal results. They showed a 13.5% power reduction compared to the single-Vdd case on average.

7

Conclusions and Future Trends

Tremendous advances have been made in FPGA design automation in the past decade. In this survey paper, we try to cover important algorithms and methodologies developed for various design tasks in modern FPGA design flow, including routing and placement, clustering, technology mapping, physical synthesis, RT-level and behavior-level synthesis, and power optimization. It is our hope that this paper can be a useful reference for both beginning and established researchers and tool developers in this field.

As the feature size continues to shrink and device capacity continues to increase in modern FPGAs, we are facing new challenges and opportunities in FPGA design automation. We would like to conclude the paper by listing some challenges and open problems in FPGA CAD that we are facing now or will face in the near future.

Physical Design

1. The time spent for placement and routing is still the dominating part of the entire FPGA compilation process. There is a need for more scalable and efficient placement and routing algorithms. We think that the interesting yet very challenging

goals are full-chip physical design in one hour and full-chip incremental physical design in five minutes. If these goals can be achieved with little or no compromise on design quality, it will be significant improvement on overall design productivity.

2. In connection to the challenges stated above, we believe an important means to achieve highly scalable placement and routing algorithms is to make the best use of multi-CPU computer systems which will be widely available very soon as standard engineering workstations. Although there were some early studies on parallel CAD algorithms (e.g., [19, 97]), we believe that more work is needed to come up commercial-strength parallel or distributed placement and routing algorithms with good scalability and quality of results.
3. Process variation is an increasing concern in nanometer designs (esp. 65 nm or below) (e.g., [32, 202, 220]), and its impact to FPGA architectures and designs is not fully understood. We hope that the regularity of the nanometer FPGA architecture can hide a large portion of variability effect from the designer. But in case it is not possible or economically feasible to shield all variability effect by the architecture optimization alone, it will be important to investigate novel physical design algorithms with consideration of process variability and be able to perform statistical optimization.
4. We expect that the future FPGAs to be designed in the "ultimate CMOS technologies" (32 nm and below) may have defects. On one hand, it is important to develop defect-tolerant FPGA architectures. On the other hand, we believe it is important to develop defect-aware physical design algorithms which can work hand-in-hand with the defect-tolerant architectures to continue to deliver high yield even in the era of the ultimate CMOS technologies. The design of defect-tolerant memory systems is well known (e.g., [137]) and widely used, but it remains challenging to come up equally

efficient defect-tolerant architecture and CAD algorithms for logic design.

Logic Synthesis and Physical Synthesis

1. Although FPGA technology mapping has been a subject of extensive research, recent studies indicate that there is considerable room to improve [66]. This is especially true in the combined solution space of logic optimization and technology mapping. There were some initial efforts in integrating logic optimization with mapping. However, more research is needed to find effective and efficient ways to combine the two to arrive at better mapping solutions. As semiconductor technologies advance, new FPGA architecture features are being introduced to improve area utilization, performance, and/or power. For example, architectures have been introduced or proposed to use LUTs with large number of inputs or multiple supply voltages. New mapping techniques are needed to utilize these new architecture features.
2. Physical synthesis for FPGAs needs to consider the impact on routing and routability. Most existing techniques stay at the placement level and pay little attention to routing. As a result, for high utilization situations (common in practice), the predicted performance gain may not be realizable after routing. Next generation interconnect-centric physical synthesis techniques are needed to further improve predictability of results. Another direction of research is to consider high-level optimizations in physical synthesis. Local and simple transformations such as logic replication and local remapping can explore only a limited solution space during physical synthesis. It is desirable to use physical information to influence high-level optimizations/transformations, for example, determining datapath architectures, selecting on-chip resources to implement memory, to name a few. Such physical synthesis techniques can potentially bring in more improvement in quality of results.

Behavioral Synthesis

Nowadays, high-end FPGAs can implement large and complex SoC designs which were originally only possible through ASIC implementation. These design applications may have dramatically different user requirements in terms of performance, power, memory bandwidth, computational throughput, etc. To meet these demands, we foresee that next-generation FPGA chips will encompass a large number (in the order of hundreds) of heterogenous cores (soft and hard) with supporting memory hierarchy and interconnect hierarchy. Sophisticated architecture features will surface as well, such as the architecture supports of multi-cycle interconnect communication, core-level and/or logic block-level multi-Vdd, core-level and/or logic block-level power/clock gating, full-chip adaptive threshold voltage, full-chip error checking capability, etc. Mapping different applications onto these architectures to fulfill various design requirements will become a dauntingly complex task. Behavioral synthesis will be in a critical need to tackle the design complexity problem and improve design productivity. It will be an important component of so called electronic system level design (ESL) methodology to speed up high-quality hardware implementation and enable fast and accurate design space exploration. We would like to highlight two research directions in this context.

1. To address the speed, power, and interconnect challenges, behavioral synthesis has to take meaningful physical information from potential hardware implementation and layout (physical planning). One direction is to incorporate physical planning in as early as possible, thus performing a combined synthesis and layout optimization. The layout needs to deal with the heterogeneity of the FPGA chip, core topology, and the specific memory and interconnect structures. The abundant silicon capacity can be utilized to improve the interconnect performance and integrity either by interconnect pipelining or resource redundancy, which can also be combined with system reliability design strategies. Layout information will serve as a guideline for the synthesis engine to determine the scheduling and binding solutions to

exploit specific architecture features (such as power gating and multi-Vdd) for better quality of design. The synthesis engine will generate RTL implementations together with physical and timing constraints, which can serve as guidelines for the downstream physical design tools.

2. We need to model communication interfaces between different components within the FPGA chip under different communication protocols and topologies, such as point-to-point connections, buses, or network-on-chips. A deeper understanding of efficient and robust communication synthesis is much needed.

References

[1] International Technology Roadmap for Semiconductors, Executive Summary, 2003. http://public.itrs.net/ Files/2003ITRS/Home2003.htm.

[2] E. Ahmed and J. Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *ACM International Symposium on FPGA*, February 2000.

[3] C. Albrecht. Provably good global routing by a new approximation algorithm for multicommodity flow. In *Proc. International Symposium on Physical Design*, pages 19–25, March 2000.

[4] M. J. Alexander, J. P. Cohoon, J. L. Ganley, and G. Robins. Placement and routing for performance-oriented FPGA layout. *VLSI Design: An International Journal of Custom-Chip Design, Simulation, and Testing*, 7(1), 1998.

[5] M. J. Alexander and G. Robins. New performance-driven FPGA routing algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1505–1517, December 1996.

[6] Altera. MAX 7000B Data Sheet. http://www.altera.com/literature/ds/m7000b.pdf.

[7] Altera. PowerPlay Early Power Estimator. http://www.altera.com/support/devices/estimator/pow-powerplay.html.

[8] Altera. PowerPlay Power Analyzer. http://www.altera.com/support/devices/estimator/pow-powerplay.html.

[9] Altera. Stratix II 90-nm Silicon Power Optimization. http://www.altera.com/products/devices/stratix2/features/st2-90nmpower.html.

[10] Altera. Stratix II Device Handbook. http://www.altera.com/literature/hb/stx2/stratix2_handbook.pdf.

[11] Altera. White paper, "Stratix II vs. Virtex-4 Power Comparison & Estimation Accuracy White Paper. http://altera.com/literature/wp/wp_s2v4_pwr_acc.pdf.

[12] Altera, August 2002. Stratix Programmable Logic Device Family Data Sheet.

[13] J. Anderson and F. N. Najm. Power-aware technology mapping for LUT-based FPGAs. In *IEEE International Conference on Field-Programmable Technology*, 2002.

[14] J. Anderson and F. N. Najm. Interconnect capacitance estimation for FPGAs. In *IEEE/ACM Asia and South Pacific Design Automation Conference*, Yokohama, Japan, 2004.

[15] J. H. Anderson and S. D. Brown. Technology mapping for large complex PLDs. In *Design Automation Conf.*, 1998.

[16] J. H. Anderson, F. N. Najm, and T. Tuan. Active leakage power optimization for FPGAs. *International Symposium on Field Programmable Gate Arrays*, February 2004.

[17] F. Assaderaghi, D. Sinitsky, S. A. Parke, J. Bokor, P. K. Ko, and C. Hu. Dynamic threshold-voltage MOSFET (DTMOS) for ultra-low voltage VLSI. *IEEE Transactions Electron Devices*, 44:414–422, March 1997.

[18] B. Awerbuch, A. Bar Noy, N. Linial, and D. Peleg. Improved routing strategies with succinct tables. *J. Algorithms*, 11(3):307–341, 1990.

[19] P. Banerjee. *Parallel Algorithms for VLSI Computer-Aided Design*. Prentice-Hall, Inc., Englewoods-Cliffs, NJ, 1994.

[20] G. Beraudo and J. Lillis. Timing optimization of FPGA placements by logic replication. In *ACM/IEEE Design Automation Conference*, pages 96–201, 2003.

[21] V. Betz and J. Rose. Cluster-based logic blocks for FPGAs: area-efficiency vs. input sharing and size. In *IEEE Custom Integrated Circuits Conference*, pages 551–554, 1997.

[22] V. Betz and J. Rose. VPR: a new packing, placement and routing tool for FPGA research. In *International Workshop on Field-Programmable Logic and Applications*, pages 213–222, 1997.

[23] V. Betz, J. Rose, and A. Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, 1999.

[24] N. Bhat and D. D. Hill. Routable technology mapping for LUT FPGAs. In *IEEE International Conference on Computer Design*, pages 95–98, 1992.

[25] E. Bozorgzadeh, S. Ogrenci, and M. Sarrafzadeh. Routability-driven packing for cluster-based FPGAs. In *Asia South Pacific Design Automation Conf.*, 2001.

[26] R. K. Brayton. Understanding SPFDs: A new method for specifying flexibility. In *International Workshop on Logic Synthesis*, 1997.

[27] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays*. Kluwer Academic Publishers, May 1992.

[28] S. Brown and J. Rose. FPGA and CPLD architectures: A tutorial. *IEEE Design and Test of Computers*, 12(2):42–57, 1996.

[29] S. Brown, J. Rose, and Z. G. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Trans. on Computer-Aided Design*, 11(5):620–628, May 1992.

[30] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek. Fast module mapping and placement for datapaths in FPGAs. In *International Symposium on Field Programmable Gate Arrays*, 1998.

[31] T. Chan, J. Cong, and K. Sze. Multilevel generalized force-directed method for circuit placement. In *Proceedings of the Int'l Symposium on Physical Design*. San Francisco, CA, April 2005.

[32] H. Chang and S. Sapatnekar. Impact of process variations on power: full-chip analysis of leakage power under process variations, including spatial correlations. In *Proc. of Design Automation Conf.*, June 2005.

[33] S. C. Chang, K. T. Cheng, N.-S. Woo, and M. Marek Sadowska. Postlayout rewiring using alternative wires. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 16(6):587–96, June 1997.

[34] S. C. Chang, L. V. Ginneken, and M. Marek-Sadowska. Circuit optimization by rewiring. *IEEE Transaction on Computers*, 48(9):962–970, September 1999.

[35] Y. W. Chang and Y. T. Chang. An architecture-driven metric for simultaneous placement and global routing for FPGAs. In *Proc. Design Automation Conf.*, pages 567–572, 2000.

[36] Y. W. Chang, K. Zhu, and D. F. Wong. Timing-driven routing for symmetrical array-based FPGAs. *ACM Trans. on Design Automation of Electronic Systems*, 5(3), July 2000.

[37] C. Chen, Y. Tsay, Y. Hwang, T. Wu, and Y. Lin. Combining technology mapping and placement for delay-optimization in FPGA designs. In *Int'l Conf. Computer Aided Design*, 1993.

[38] D. Chen and J. Cong. DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs. In *Int'l Conf. Computer Aided Design*, 2004.

[39] D. Chen and J. Cong. Delay optimal low-power circuit clustering for FPGAs with dual supply voltages. *International Symposium on Low Power Electronics and Design*, August 2004.

[40] D. Chen, J. Cong, M. Ercegovac, and Z. Huang. Performance-driven mapping for CPLD architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(10):1424–1431, October 2003.

[41] D. Chen, J. Cong, and Y. Fan. Low-power high-level synthesis for FPGA architectures. *International Symposium on Low Power Electronics and Design*, August 2003.

[42] D. Chen, J. Cong, F. Li, and L. He. Low-power technology mapping for FPGA architectures with dual supply voltages. *International Symposium on Field Programmable Gate Arrays*, February 2004.

[43] G. Chen and J. Cong. Simultaneous logic decomposition with technology mapping in FPGA designs. *International Symposium on Field-Programmable Gate-Arrays*, 2001.

[44] G. Chen and J. Cong. Simultaneous timing driven clustering and placement for FPGAs. In *International Conference on Field Programmable Logic and Its Applications*, pages 158–167, August 2004.

[45] G. Chen and J. Cong. Simultaneous timing-driven placement and duplication. *International Symposium on Field-Programmable Gate-Arrays*, 2005.

[46] K. C. Chen, et al. DAG-map: graph-based FPGA technology mapping for delay optimization. *IEEE Design and Test of Computers*, 9(3):7–20, September 1992.

[47] A. Chowdary and J. P. Hayes. Technology mapping for field-programmable gate arrays using integer programming. In *Int'l Conf. Computer Aided Design*, November 1995.

[48] J. Cong and Y. Ding. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *Int'l Conf. Computer Aided Design*, November 1992.

[49] J. Cong and Y. Ding. Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs. In *Int'l Conf. Computer Aided Design*, 1993.

[50] J. Cong and Y. Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, 13(1):1–12, January 1994.

[51] J. Cong and Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Transactions on VLSI Systems*, 2(2):137–148, 1994.

[52] J. Cong and Y. Ding. Combinational logic synthesis for LUT based field programmable gate arrays. *ACM Trans. on Design Automation of Electronic Systems*, 1(2):145–204, April 1996.

[53] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Behavior and communication co-optimization for systems with sequential communication media. In *IEEE/ACM Design Automation Conference*, 2006.

[54] J. Cong, Y. Fan, G. Han, X. Yang, and Z. Zhang. Architecture and synthesis for on-chip multi-cycle communication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 550–564, April 2004.

[55] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *International Symposium on Field-Programmable Gate Arrays*, February 2004.

[56] J. Cong, J. Fang, M. Xie, and Y. Zhang. MARS—a multilevel full-chip gridless routing system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):382–394, March 2005.

[57] J. Cong, H. Huang, and X. Yuan. Technology mapping and architecture evaluation for k/m-macrocell-based FPGAs. *ACM Trans. on Design Automation of Electronic Systems*, 10:3–23, January 2005.

[58] J. Cong and Y. Hwang. Simultaneous depth and area minimization in LUT-based FPGA mapping. *International Symposium on Field-Programmable Gate-Arrays*, February 1995.

[59] J. Cong and Y. Hwang. Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA design. In *ACM/IEEE Design Automation Conference*, 1996.

[60] J. Cong, A. Kahng, G. Robins, M. Sarrafzadeh, and C. K. Wong. Provably good performance-driven global routing. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):739–752, June 1992.

[61] J. Cong, T. Kong, J. Shinnerl, M. Xie, and X. Yuan. Large scale circuit placement. *ACM Transactions on Design Automation of Electronic Systems*, 10(2):389–430, April 2005.

[62] J. Cong, K. S. Leung, and D. Zhou. Performance-driven interconnect design based on distributed RC delay model. In *Proc. ACM/IEEE 30th Design Automation Conference*, pages 606–611, June 1993.

[63] J. Cong and S. K. Lim. Physical planning with retiming. In *IEEE International Conference on Computer Aided Design*, pages 2–7, 2000.

[64] J. Cong, Y. Lin, and W. Long. SPFD-based global reviewing. In *International Symposium on Field-Programmable Gate Arrays*, 2002.

[65] J. Cong and W. Long. Theory and algorithm for SPFD-based global rewiring. In *International Workshop on Logic Synthesis*, 2001.

[66] J. Cong and K. Minkovich. Optimality study of logic synthesis for LUT-based FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, February 2006.

[67] J. Cong and B. Preas. A new algorithm for standard cell global routing. In *Proc. Int'l Conf. on Computer-Aided Design*, pages 176–179, November 1988.

[68] J. Cong and M. Romesis. Performance-driven multi-level clustering with application to hierarchical FPGA mapping. In *Design Automation Conference*, 2001.

[69] J. Cong and J. Shinnerl, editors. *Multilevel Optimization in VLSI CAD*. Kluwer Academic Publishers, 2003.

[70] J. Cong and C. Wu. FPGA synthesis with retiming and pipelining for clock period minimization of sequential circuits. In *Design Automation Conference*, 1997.

[71] J. Cong and C. Wu. Optimal FPGA mapping and retiming with efficient initial state computation. *Design Automation Conference*, 1997.

[72] J. Cong, C. Wu, and Y. Ding. Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. *International Symposium on Field-Programmable Gate Arrays*, February 1999.

[73] J. Cong and S. Xu. Delay-optimal technology mapping for FPGAs with heterogeneous LUTs. In *Design Automation Conference*, 1998.

[74] J. Cong and S. Xu. Delay-oriented technology mapping for heterogeneous FPGAs with bounded resources. In *Int'l Conf. Computer Aided Design*, 1998.

[75] J. Cong and S. Xu. Technology mapping for FPGAs with embedded memory blocks. In *International Symposium on Field-Programmable Gate Arrays*, 1998.

[76] J. Cong and S. Xu. Performance-driven technology mapping for heterogeneous FPGAs. *IEEE Trans. on Computer-aided Design of Integrated Circuits and Systems*, 19(11):1268–1281, November 2000.

[77] Ph.D. Dissertation D. Chen. Design and synthesis for low-power FPGAs. *Computer Science Department, University of California*, December 2005.

[78] J. A. Davis, V. K. De, and J. Meindl. A stochastic wire-length distribution for gigascale integration (GSI)—Part I: derivation and validation. 45(3):580–589, March 1998.

[79] G. De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, Inc., 1994.

[80] Y. Ding, P. Suaris, and N. Chou. The effect of post-layout pin permutation on timing. In *Int'l Symposium on Field Programmable Gate Arrays*, 2005.

[81] A. Duncan, D. Hendry, and P. Gray. An overview of the COBRA-ABS high level synthesis system for multi-FPGA systems. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 106–115, April 1998.

[82] J. M. Emmert and D. Bhatia. A methodology for fast FPGA floorplanning. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 47–56, February 21-23 1999.

[83] L. A. Entrena and K. T. Cheng. Combinational and sequential logic optimization by redundancy addition and removal. *IEEE Transaction on Computer Aided Design of Integrated Circuits and Systems*, 14(7):909–916, 1995.

[84] W. Fang and A. Wu. Multi-way FPGA partitioning by fully exploiting design hierarchy. *ACM Transactions on Design Automation of Electronic Systems*, 5(1):34–50, January 2000.

[85] A. Farrahi and M. Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Tran. on Computer Aided Design of Integrated Circuits and Systems*, 13(11):1319–1332, November 1994.

[86] A. H. Farrahi and M. Sarrafzadeh. FPGA technology mapping for power minimization. In *International Workshop in Field Programmable Logic and Applications*, 1994.

[87] FishTail. Design Automation. http://www.fishtail-da.com/.

[88] R. J. Francis, J. Rose, and Z. Vranesic. Technology mapping for lookup table-based FPGA's for performance. In *Int'l Conf. Computer-Aided Design*, November 1991.

[89] R. J. Francis, et al. Chortle-crf: fast technology mapping for lookup table-based FPGAs. In *Design Automation Conference*, 1991.

[90] J. Frankle. Iterative and adaptive slack allocation for performance-driven layout and FPGA routing. In *Proceedings of Design Automation Conference*, pages 536–542, 1992.

[91] A. Gayasen, Y. Tsai, N. Vijaykrishnan, M. Kandemir, M. Irwin, and T. Tuan. Reducing leakage energy in FPGAs using region-constrained placement. In *ACM International Symposium on Field Programmable Gate Arrays*, February 2004.

[92] V. George and J. Rabaey. *Low-energy FPGAs—Architecture and Design*. Kluwer Academic Publishers, 2001.

[93] V. George and J. Rabaey. *Low-Energy FPGAs: Architecture and Design*. Springer, June 2001.

[94] S. Ghiasi, E. Bozorgzadeh, S. Choudhury, and M. Sarrafzadeh. A unified theory of timing budget management. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 653–659, November 2004.

[95] M. Gokhale and J. Stone. Automatic allocation of arrays to memories in FPGA processors with multiple memory banks. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 63–69, April 1999.

[96] P. Gopalakrishnan, X. Li, and L. Pileggi. Architecture-aware FPGA placement using metric embedding. In *IEEE/ACM Design Automation Conference*, pages 460–465, 2006.

[97] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee. Parallel algorithms for FPGA placement. In *Proc. Great Lakes Symposium on VLSI (GVLSI 2000)*, March 2000.

[98] Z. Hasan, D. Harrison, and M. Ciesielski. A fast partition method for PLA-based FPGAs. *IEEE Design and Test of Computers*, December 1992.

[99] G. D. Hatchel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 1996.

[100] P. S. Hauge, R. Nair, and E. J. Yoffa. Circuit placement for predictable performance. In *International Conference of Computer Aided Design*, pages 88–91, 1987.

[101] J. He and J. Rose. Technology mapping for heterogeneous FPGAs. In *International Symposium on Field Programmable Gate Arrays*, 1994.

[102] M. Hrkic and J. Lillis. S-Tree: a technique for buffered routing tree synthesis. In *Design Automation Conference*, 2002.

[103] M. Hrkic, J. Lillis, and G. Beraudo. An approach to placement-coupled logic replication. In *ACM/IEEE Design Automation Conference*, pages 711–716, June 2004.

[104] Y. Hu, Y. Lin, L. He, and T. Tuan. Simultaneous time slack budgeting and retiming for Dual-Vdd FPGA power reduction. In *IEEE/ACM Design Automation Conference*, 2006.

[105] S. W. Hur and J. Lillis. Mongrel: hybrid techniques for standard cell placement. In *International Conference of Computer Aided Design*, 2000.

[106] J. Hwang, F. Chiang, and T. Hwang. A re-engineering approach to low power FPGA design using SPFD. In *Design Automation Conference*, 1998.

[107] M. Inuani and J. Saul. Re-synthesis in technology mapping for heterogeneous FPGAs. In *International Conference on Computer Design*, 1998.

[108] P. Jamieson and J. Rose. A verilog RTL synthesis tool for heterogeneous FPGAs. In *International Conference on Field Programmable Logic and Applications*, August 2005.

[109] A. B. Kahng, S. Reda, and Q. Wang. Architecture and details of a high quality, large-scale analytical placer. In *ACM/IEEE Intl. Conf. on Computer-Aided Design*, pages 891–898, November 2005.

[110] G. Karypis and V. Kumar. Multilevel hypergraph partitioning. In *Design Automation Conference*, 1997.

[111] A. Kaviani and S. Brown. Technology mapping issues for an FPGA with lookup tables and PLA-like blocks. In *International Symposium on Field Programmable Gate Arrays*, 2000.

[112] K. Keutzer. DAGON: technology binding and local optimization by DAG matching. *ACM/IEEE Design Automation Conference*, pages 341–347, 1987.

[113] D. Kim, J. Jung, S. Lee, J. Jeon, and K. Choi. Behavior-to-placed RTL synthesis with performance-driven placement. In *Int. Conf. on Computer Aided Design*, pages 320–326, November 2001.

[114] A. Koch. Structured design implementation—a strategy for implementing regular datapaths on FPGAs. In *International Symposium on Field Programmable Gate Arrays*, pages 151–157, 1996.

[115] T. Kong. A novel net weighting algorithm for timing-driven placement. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 172–176, 2002.

[116] M. R. Korupolu, K. K. Lee, and D. F. Wong. Exact tree-based FPGA technology mapping for logic blocks with independent LUTs. In *Design Automation Conference*, 1998.

[117] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15:141–145, 1981.

[118] J. L. Kouloheris. *Empirical Study of the Effect of Cell Granularity on FPGA Density and Performance*. Ph.D. Thesis, Stanford University, 1993.

[119] S. Krishnamoorthy and R. Tessier. Technology mapping algorithms for hybrid FPGAs containing lookup tables and PLAs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 22(5), May 2003.

[120] E. Kusse and J. Rabaey. Low-energy embedded FPGA structures. In *Proc. of International Symposium on Low Power Electronics and Design*, August 1998.

[121] J. Lamoureux and S. J. E. Wilton. On the interaction between power-aware FPGA CAD algorithms. In *IEEE International Conference on Computer-Aided Design*, November 2003.

[122] Lattice. Power estimation in ispMACH 5000B devices, May 2002.

[123] E. L. Lawler, K. N. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. *Trans. On Computer*, C18(1), 1969.

[124] S. Lee and D. F. Wong. Timing-driven routing for FPGAs based on Lagrangian relaxation. In *Proc. of International Symposium on Physical Designs*, pages 176–181, April 2002.

[125] Y. S. Lee and C. H. Wu. A performance and routability-driven router for FPGAs considering path delay. In *Proc. of Design Automation Conference*, pages 557–561, 1995.

[126] C. Legl, B. Wurth, and K. Eckl. A Boolean approach to performance-directed technology mapping for LUT-based FPGA designs. In *Design Automation Conference*, June 1996.

[127] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 264–271, 1995.

[128] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):813–834, 1997.

[129] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.

[130] G. Lemieux and S. D. Brown. A detailed routing algorithm for allocating wire segments in FPGAs. *ACM/SIGDA Physical Design Workshop*, 1993.

[131] F. Li, D. Chen, L. He, and J. Cong. Architecture evaluation for power-efficient FPGAs. In *ACM International Symposium on Field Programmable Gate Arrays*, pages 175–184, Monterey, California, 2003.

[132] F. Li, Y. Lin, and L. He. FPGA power reduction using configurable Dual-Vdd. In *IEEE/ACM Design Automation Conference*, pages 735–740, June 2004.

[133] F. Li, Y. Lin, and L. He. Vdd programmability to reduce FPGA interconnect power. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 760–765, San Jose, November 2004.

[134] F. Li, Y. Lin, L. He, D. Chen, and J. Cong. Power modeling and characteristics of field programmable gate arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(11), November 2005.

[135] F. Li, Y. Lin, L. He, and J. Cong. Low-power FPGA using dual-Vdd/Dual-Vt techniques. In *International Symposium on Field Programmable Gate Arrays*, pages 42–50, February 2004.

[136] H. Li, S. Katkoori, and W. K. Mak. Power minimization algorithms for LUT based FPGA technology mapping. *ACM Transactions on Design Automation of Electronic Systems*, 9(1):33–51, January 2004.

[137] R. Liberskind Hadas, N. Hasan, J. Cong, P. Mckinley, and C. L. Liu. Kluwer Academic Publishers, 1992.

[138] E. Lin and S. Wilton. Macrocell architectures for product term embedded memory arrays. *Field Programmable Logic Applications*, pages 48–58, August 2001.

[139] J. Lin, D. Chen, and J. Cong. Optimal simultaneous mapping and clustering for FPGA delay optimization. In *IEEE/ACM Design Automation Conference*, 2006.

[140] J. Lin, A. Jagannathan, and J. Cong. Placement-driven technology mapping for LUT-based FPGAs. In *International Symposium on Field Programmable Gate Arrays*, pages 121–126, February 2003.

[141] A. Ling, D. Singh, and S. Brown. FPGA technology mapping: a study of optimality. In *Design Automation Conference*, 2005.

[142] P. Maidee, C. Ababei, and K. Bazarga. Timing-driven partitioning-based placement for Island style FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):395–406, March 2005.

[143] V. Manohararajah, S. D. Brown, and Z. G. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. In *International Workshop of Logic Synthesis*, 2004.

[144] A. Marquardt, V. Betz, and J. Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 37–46, 1999.

[145] A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for FPGAs. In *International Symposium on Field Programmable Gate Arrays*, pages 203–213, Monterey, Ca., February 2000.

[146] A. Mathur and C. L. Liu. Performance-driven technology mapping for lookup-table based FPGAs using the general delay model. In *International Workshop on Field Programmable Gate Arrays*, February 1994.

[147] L. Mcmurchie and C. Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs. In *Proceedings of International Symposium on Field-Programmable Gate Arrays*, February 1995.

[148] A. Mishchenko, S. Chatterjee, and R. Brayton. Improvements to technology mapping for LUT-based FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, 2006.

[149] R. Mukherjee and S. Ogrenci Memik. Evaluation of Dual Vdd fabrics for low power FPGAs. In *Asia-South Pacific Design Automation Conference*, January 2005.

[150] R. Murgai, R. Brayton, and A. Sangiovanni Vincentelli. On clustering for minimum delay/area. In *Int'l Conf. Computer Aided Design*, November 1991.

[151] R. Murgai, R. Brayton, and A. Sangiovanni Vincentelli. *Logic Synthesis for Field-Programmable Gate Arrays*. Springer, July 1995.

[152] R. Murgai, et al. Improved logic synthesis algorithms for table look up architectures. In *Int'l Conf. Computer Aided Design*, November 1991.

[153] S. Mutoh, et al. 1-V Power supply high-speed digital circuit technology with multi-threshold-voltage CMOS. *IEEE Journal of Solid-State Circuits*, 30(8):847–854, August 1995.

[154] S. K. Nag and R. A. Rutenbar. Performance-driven simultaneous placement and routing for FPGAs. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 17(6):499–518, June 1998.

[155] R. Nair. A simple yet effective technique for global wiring. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):165–172, March 1987.

[156] G. J. Nam, F. Aloul, K. A. Sakallah, and R. A. Rutenbar. A comparative study of two Boolean formulations of FPGA detailed routing constraints. *IEEE Transactions on Computers*, 53(6):688–696, June 2004.

[157] G. J. Nam, K. A. Sakallah, and R. A. Rutenbar. Satisfiability-based layout revisited: detailed routing of complex FPGAs via search-based Boolean SAT. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 167–175, February 1999.

[158] P. Pan and C. C. Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, 1998.

[159] P. Pan and C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. In *Design Automation Conf.*, June 1996.

[160] P. Pan and C. L. Liu. Technology mapping of sequential circuits for LUT-based FPGAs for performance. In *International Symposium on Field-Programmable Gate Arrays*, 1996.

[161] P. Pan and C. L. Liu. Optimal clock period FPGA technology mapping for sequential circuits. *ACM Transactions on Design Automation of Electronic Systems*, 3(3):437–462, 1998.

[162] K. Poon, S. J. E. Wilton, and A. Yan. A detailed power model for field programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems*, 10(2):279–302, April 2005.

[163] A. Rahman and V. Polavarapuv. Evaluation of low-leakage design techniques for field programmable gate arrays. In *International Symposium on Field Programmable Gate Arrays*, February 2004.

[164] R. Rajaraman and D. F. Wong. Optimal clustering for delay minimization. In *Design Automation Conference*, June 1993.

[165] J. Rose. Parallel global routing for standard cells. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 9(10):1085–1095, October 1990.

[166] Y. Sankar and J. Rose. Trading quality for compile time: ultra-fast placement for FPGAs. In *International Symposium on Field Programmable Gate Arrays*, pages 157–166, 1999.

[167] M. Schlag, J. Kong, and P. K. Chan. Routability-driven technology mapping for lookup table-based FPGAs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):13–26, 1994.

[168] H. Schmit, L. Arnstein, D. Thomas, and E. Lagnese. Behavioral synthesis for FPGA-based computing. In *Workshop on FPGAs for Custom Computing Machines*, pages 125–132, 1994.

[169] E. M. Sentovich, et al. *SIS: A system for sequential circuit synthesis.* Berkeley, CA, University of California, 1992. Dept. of Electrical Engineering and Computer Science.

[170] L. Shang and N. K. Jha. High-level power modeling of CPLDs and FPGAs. In *IEEE International Conference on Computer Design*, September 2001.

[171] L. Shang, A. Kaviani, and K. Bathala. Dynamic power consumption in virtex-II FPGA family. In *ACM International Symposium on Field Programmable Gate Arrays*, February 2002.

[172] K. R. Shayee, J. Park, and P. Diniz. Performance and area modeling of complete FPGA designs in the presence of loop transformations. *IEEE Transactions on Computers*, 53(11):1420–1435, November 2004.

[173] J. P. M. Silva and K. A. Sakallah. GRASP—a new search algorithm for satisfiability. In *Proc. ACM/IEEE Int'l Conf. Computer Aided Design*, November 1997.

[174] A. Singh and M. Marek Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. In *ACM International Symposium on Field Programmable Gate Arrays*, February 2002.

[175] D. Singh and S. Brown. Integrated retiming and placement for field programmable gate arrays. In *International Symposium on Field Programmable Gate Arrays*, pages 67–76, February 2002.

[176] A. Srivastava, D. Sylvester, and D. Blaauw. Power minimization using simultaneous gate sizing, Dual-Vdd and Dual-Vth assignment. In *Design Automation Conference*, 2004.

[177] H. Styles and W. Luk. Branch optimization techniques for hardware compilation. In *International Conference on Field Programmable Logic and Applications*, 2003.

[178] P. Suaris, L. Liu, Y. Ding, and N. Chou. Incremental physical resynthesis for timing optimization. In *Int'l Symposium on Field Programmable Gate Arrays*, 2004.

[179] W. Sun, M. Wirthlin, and S. Neuendorffer. Combining module selection and resource sharing for efficient FPGA pipeline synthesis. In *Int'l Symposium on Field Programmable Gate Arrays*, 2006.

[180] J. Swartz, V. Betz, and J. Rose. A fast routability-driven router for FPGAs. In *Int'l Symposium on Field Programmable Gate Arrays*, pages 140–149, Monterey, CA, 1998.

[181] Synopsys. http://www.synopsys.com/products/products_matrix.html.

[182] SystemC. http://www.systemc.org.

[183] M. Takahashi, et al. A 60mW MPEG4 video codec using clustered voltage scaling with variable supply-voltage scheme. *Journal of Solid-State Circuits*, 1998.

[184] R. Tessier. Fast placement approaches for FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, 7(2):284–305, April 2002.

[185] The MathWorks. http://www.mathworks.com/.

[186] N. Togawa, M. Sato, and T. Ohtsuki. Maple: a simultaneous technology mapping, placement, and global routing algorithm for field-programmable gate arrays. In *Int'l Conf. Computer Aided Design*, 1994.

[187] N. Togawa, M. Sato, and T. Ohtsuki. A simultaneous placement and global routing algorithm with path length constraints for transport-processing FPGAs. In *Asia South Pacific Design Automation Conf.*, pages 569–578, 1997.

[188] S. Trimberger. *Field-Programmable Gate Array Technology*. Springer, January 1994.

[189] T. Tuan, S. Kao, A. Rahman, S. Das, and S. Trimberger. A 90nm low-power FPGA for battery-powered applications. In *International Symposium on Field Programmable Gate Arrays*, 2006.

[190] T. Tuan and B. Lai. Leakage power analysis of a 90nm FPGA. In *Custom Integrated Circuits Conference*, 2003.

[191] K. Usami and M. Horowitz. Clustered voltage scaling for low-power design. In *International Symposium on Low Power Design*, April 1995.

[192] H. Vaishnav and M. Pedram. Delay optimal clustering targeting low-power VLSI circuits. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(6), June 1999.

[193] K. Wakabayashi. C-based behavioral synthesis and verification analysis on industrial design examples. In *Asian and South Pacific Design Automation Conference*, pages 344–348, January 2004.

[194] K. Wakabayashi and T.Okamoto. C-based SoC design flow and EDA tools: an ASIC and system vendor perspective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1507–1522, December 2000.

[195] Z. H. Wang, E. C. Liu, J. Lai, and T. C. Wang. Power minimization in LUT-based FPGA technology mapping. In *Asia South Pacific Design Automation Conference*, 2001.

[196] L. Wei, Z. Chen, M. Johnson, K. Roy, and V. De. Design and optimization of low voltage high performance dual threshold CMOS circuits. In *Design Automation Conference*, 1998.

[197] K. Weiß, C. Oetker, I. Katchan, T. Steckstor, and W. Rosenstiel. Power estimation approach for SRAM-based FPGAs. In *ACM International Symposium on Field Programmable Gate Arrays*, February 2000.

[198] S. Wilton. SMAP: heterogeneous technology mapping for area reduction in FPGAs with embedded memory arrays. In *International Symposium on Field Programmable Gate Arrays*, 1998.

[199] S. Wilton. Heterogeneous technology mapping for area reduction in FPGAs with embedded memory arrays. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(1), January 2000.

[200] S. Wilton. Heterogeneous technology mapping for FPGAs with dual-port embedded memory arrays. In *International Symposium on Field Programmable Gate Arrays*, 2000.

[201] F. G. Wolff, M. J. Knieser, D. J. Weyer, and C. A. Papachristou. High-level low power FPGA design methodology. In *IEEE National Aerospace Conference*, 2000.

[202] P. Wong, L. Cheng, Y. Lin, and L. He. FPGA device and architecture evaluation considering process variation. In *Proc. IEEE/ACM International Conf. on Computer-Aided Design*, November 2005.

[203] Y. L. Wu and M. Marek Sadowska. An efficient router for 2-D field programmable gate arrays. In *Proc. of European Design Automation Conference*, pages 412–416, 1994.

[204] Y. L. Wu and M. Marek Sadowska. Orthogonal greedy coupling—a new optimization approach to 2-D FPGA routing. In *Proc. of Design Automation Conference*, June 1995.

[205] Xilinx. Website, http://www.xilinx.com.

[206] Xilinx. Spartan-3E Data Sheets. http://direct.xilinx.com/bvdocs/publications/ds312.pdf.

[207] Xilinx. Virtex-4 Data Sheet. http://www.xilinx.com.

[208] Xilinx. Virtex-4 Web Power Tool Version 8.1.01. http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex4/index.htm.

[209] Xilinx. Virtex-5 Data Sheet. http://www.xilinx.com.

[210] Xilinx. white paper 205: hardware/software codesign for platform FPGAs. http://www.xilinx.com/products/design_resources/proc_central/resource/hardware_software_codesign.pdf.

[211] Xilinx. XPower Tool. http://www.xilinx.com/products/design_resources/design_tool/grouping/power_tools.htm.

[212] Xilinx. A simple method of estimating power in XC4OOOXL/EX/E FPGAs, June 30 1997. Application Brief X014.

[213] M. Xu and F. J. Kurdahi. ChipEst-FPGA: a tool for chip level area and timing estimation of lookup table based FPGAs for high level applications. In *Asia and South Pacific Design Automation Conference*, January 1997.

[214] M. Xu and F.J. Kurdahi. *Design Automation and Test in Europe*. Layout-driven high level synthesis for FPGA based architectures. 1998.

[215] S. Yamshita, H. Sawada, and A. Nagoya. A new method to express functional permissibilities for LUT based FPGAs and its applications. In *International Conference on Computer Aided Design*, pages 254–261, 1996.

[216] H. Yang and D. F. Wong. Edge-map: optimal performance driven technology mapping for iterative LUT based FPGA designs. In *Int'l Conf. Computer Aided Design*, November 1994.

[217] A. G. Ye, J. Rose, and D. Lewis. Synthesizing datapath circuits for FPGAs with emphasis on area minimization. In *International Conference on Field-Programmable Technology*, December 2002.

[218] A. Z. Zelikovsky. An 11/6 approximation algorithm for the network Steiner problem. *Algorithmica*, 9:463–470, 1993.

[219] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer-Aided Design*, pages 279–285, 2001.

[220] Y. Zhan, et al. Statistical timing analysis: correlation-aware statistical timing analysis with non-gaussian delay distributions. In *Proc. of Design Automation Conf.*, June 2005.

[221] P. S. Zuchowski, et al. A hybrid ASIC and FPGA architecture. In *International Conference on Computer-Aided Design*, November 2002.