Computer-Aided Design for FPGAs: Overview and Recent Research Trends

Jason H. Anderson and Tomasz S. Czajkowski

Abstract Computer-aided design (CAD) tools are an integral part of designing with FPGAs. The tools themselves have a considerable impact on the final FPGA circuit implementation, affecting circuit speed, logic density and power consumption. In this chapter, we survey the modern FPGA CAD flow used by most FPGA designers, giving a top-to-bottom description of the role of each stage of the flow. We also highlight recent research directions in FPGA CAD and give our thoughts on the future trends in the area.

1 Introduction

The importance of computer-aided design (CAD) tools for FPGAs cannot be underestimated. In the previous chapters, we have seen that modern FPGAs contain a fixed fabric of logic blocks, routing, hard IP blocks, and I/O blocks. FPGA CAD tools take a description of a digital circuit as input, along with constraints (e.g. on speed performance, area or power), and automatically map the circuit into the hardware blocks and routing available in the FPGA. A key point to recognize is that regardless of what features are incorporated into an FPGA, whether it be large blocks of RAM, processors or analog-to-digital converters, such features are useless unless they can be taken advantage of, and used effectively by the tools. Furthermore, the tools significantly affect the speed performance, area and power of circuits implemented in FPGAs. CAD tools alone can affect the speed performance of a circuit implemented in an FPGA by 30% or more [8]. Using an FPGA means using FPGA

Jason H. Anderson
Dept. of Electrical and Computer Engineering, University of Toronto,
e-mail: janders@eecg.toronto.edu

Tomasz S. Czajkowski
Altera Corporation,
e-mail: tczajkow@altera.com

CAD tools, and the tools are truly the “face” of the vendor seen by engineers and designers in the field.

This chapter gives an overview of the CAD flow used by modern FPGA designers, and along the way, highlights the recent developments in FPGA CAD and future trends in the field. The intent here is not to provide a detailed coverage of algorithmic aspects, but rather, to provide a tutorial-like treatment that will be useful to experienced designers and newcomers alike.

1.1 The Modern FPGA CAD Flow

The largest modern FPGAs contain billions of transistors and implement complete systems with hundreds of thousands of gates. Owing to the complexity of mapping a circuit into an FPGA, the CAD flow is broken into manageable steps. Fig. 1 shows the flow used by most modern FPGA designers. The input to the flow is a circuit described in either VHDL or Verilog at the register-transfer level (RTL)[1]. While not entirely independent of the target FPGA, the early steps of the flow tend to be more generic, while the later steps are more closely tied to the specific hardware available. The RTL synthesis step parses the input and transforms the VHDL/Verilog into a block-level circuit description, usually consisting of large blocks such as multipliers, adders, multiplexers, state machines, RAMs, and chunks of generic Boolean logic. Logic synthesis then optimizes the circuit at the level of Boolean equations.

In technology mapping, the circuit is mapped from a generic form into an equivalent circuit composed of basic logic elements available on the target device, e.g. LUTs, registers and multiplexers. As described in Chapter X, the logic blocks in modern FPGAs contain clusters of LUTs, registers and other circuitry. Packing, also referred to in the literature as *clustering*, is the step in which elements of the technology mapped circuit are packed into the logic blocks. The placement stage decides where each logic block should be located on the two-dimensional FPGA. Routing forms the desired connections between the placed logic blocks. Finally, the bitstream is generated for programming the FPGA device.

On the right side of Fig. 1, observe that timing analysis [49] and power analysis [84] feed into all stages of the CAD flow. All stages make decisions that ultimately impact circuit speed and power therefore, the tools must have access to such analysis data. Exact analysis of timing and power is impossible before routing is complete, so estimates are used at the earlier phases of the flow [10, 33, 55, 81]. The concept of physical synthesis, shown on the left of Fig. 1, has been developed to counter inaccuracies in timing estimates. In general, estimates of delay can be made more accurately at later stages of the flow. In physical synthesis, delay estimates made in placement are used to drive incremental/partial re-execution of earlier phases of the flow. Literature on physical synthesis has so far been aimed at improv-

[1] In RTL, the cycle-by-cycle behavior of the circuit’s functionality is completely specified by the designer.

ing speed performance, however, the objective of physical synthesis could equally well be power, routability or other criteria.

The two largest FPGA vendors, Xilinx and Altera, supply a complete tool flow from RTL-to-bits, often free-of-charge to their customers and to universities. Alternative third-party tools are also available for the initial stages of the flow, such as the popular Synopsys (Synplicity) and Magma Design Automation tools, and are known to produce excellent results. Historically, the use of FPGA vendor tools has been mandatory for the back-end of the flow, beginning with packing, however, that may be changing as Synplicity now offers a flow encompassing packing, placement and physical synthesis.

Though not shown in the flow of Fig. 1, simulation, test and verification of the design can be done at any stage. In practice, many customers simulate their initial design specification (RTL Verilog or VHDL), and then do not simulate again. Rather, designers leverage FPGA reconfigurability to accelerate their verification. After routing, a bitstream is generated for the design, the FPGA is programmed, and verification is done in the lab using the actual hardware. Such a verification flow is impossible for custom IC technologies, yet it is feasible for programmable logic where designs can be modified and devices reconfigured following the discovery of design flaws.

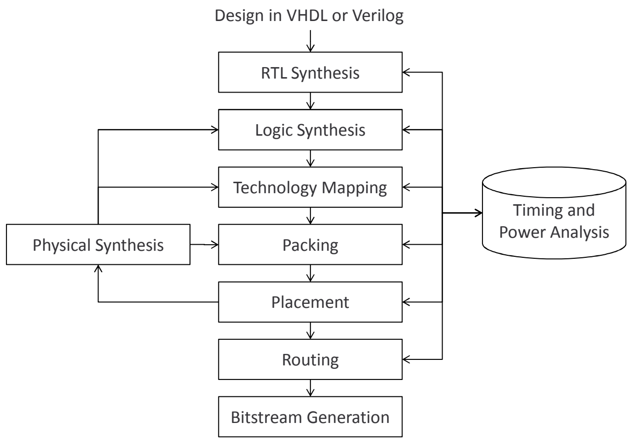


Fig. 1 FPGA CAD flow.

Without question, the majority of published research on FPGA CAD has been on the back-end of the flow, from technology mapping onwards and including physical synthesis. The subsequent sections outline the role of stage of the flow and highlight recent research results. It is worth mentioning the divide between academic and industrial research in FPGA CAD. None of the commercial FPGA vendors release the source code for their tools publicly, and as such, published research on FPGA CAD from academia is typically conducted using an entirely different CAD framework.

Academic work on synthesis and technolgy mapping has recently been conducted using the ABC synthesis framework, developed at UC Berkeley [82], while work on packing, placement and routing has been done using the VPR framework from the University of Toronto [15, 68]. Compounding the problems associated with using a different toolset, academic research often targets a simplified FPGA model that differs considerably from modern commercial FPGAs. The net effect of this is that at times, academic research results have not been directly transferrable to industry. The situation may be changing however, with the introduction of the Quartus University Interface Program (QUIP) from Altera, which permits academic and other researchers to interface their CAD tools with the Altera flow [30]. We highlight both academic and industrial FPGA CAD in this chapter.

2 RTL Synthesis

Fig. 2 shows the general approach taken in RTL synthesis. First, the input VHDL or Verilog design is parsed and analyzed. The circuit is represented internally as a parse tree. Next is elaboration, where the circuit netlist begins to take shape. The elaborated netlist may contain input and output ports, logic gates, registers, large blocks (e.g. multipliers, adders, RAMs) and state machines. These initial steps are loosely coupled to the target FPGA. In fact, many FPGA and ASIC vendors use the identical third-party software (Verific) for their front-end HDL parsing/analysis/elaboration [103], despite the fact that vendor architectures differ considerably from one another.

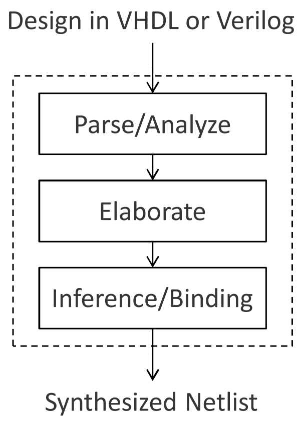


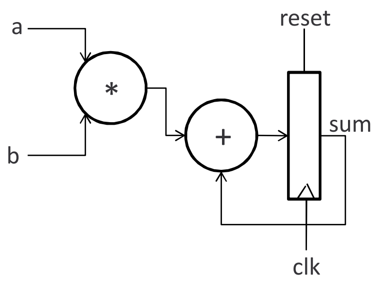
Fig. 2 Steps in RTL synthesis.

The final step of RTL synthesis, called inferencing/binding, is closely tied to the target FPGA. The concept here is to “infer” the hardware corresponding to code statements in the HDL. For example, a hardware multiplier would be inferred from the VHDL statement: `Z <= A * B`. The inputs to the multiplier would be attached to signals `A` and `B`; the output would be attached to `C`. Likewise, shifters, adders, dividers, state machines, other blocks, and generic logic gates would be inferred from HDL statements accordingly.

A hardware block inferred from HDL description is then configured to perform a specific function. This is called binding. For example, an inferred multiplier block might be bound to an implementation by a DSP block in the target FPGA. A shifter might be bound to a vendor-specific shifter implementation. Small RAMs inferred from HDL could be bound to LUT-based memories; large RAMs could be bound to block RAMs in the FPGA fabric.

An end-to-end example illustrating the action of RTL synthesis is shown in Fig. 3. Fig. 3(a) shows a section of VHDL code to implement a multiply-accumulate function. Fig. 3(b) shows an RTL netlist, produced by RTL synthesis. Line 1 of the code indicates that the functionality in lines 2-10 only "executes" when a change on the clock signal `clk` happens, i.e. this implements edge triggering. From lines 3-9, an 8-bit positive edge-triggered register file is inferred; the register file's output is called `sum`. The register file is reset synchronously when the `reset` signal is asserted (line 4); otherwise, the value in the register file is updated to the sum of its prior value and the product of signals `a` and `b` (line 7). For the synchronous reset, the RTL synthesis tool needs to know whether the flip-flops in the target FPGA have a built-in synchronous reset pin, or whether the synchronous reset must be implemented using generic logic gates. From line 7, the adder and multiplier blocks are inferred. The multiplier would likely be bound to a DSP block in the target FPGA. The adder would likely be bound to a logic-block implementation, utilizing the fast carry-chain arithmetic available in the hardware.

```
1:  process(clk)
2:  begin
3:  if (clk'event and clk = '1') then
4:    if (reset = '1') then
5:       sum <= "00000000";
6:    else
7:       sum <= sum + a*b;
9:    end if;
10: end if;
11: end process;
```



a) VHDL code b) RTL netlist

Fig. 3 Example of RTL synthesis.

The key to inferring and binding is awareness of the hardware blocks available in the target FPGA and taking advantage of the hardware to meet constraints on area, speed or power. A recent work explored area/delay tradeoffs in binding for FPGAs [111]. A paper by Tessier et al. described an approach for binding RAMs to hardware that the minimizes dynamic power consumption, at the cost of increased area [100]. Tessier's work is based on the property that block RAMs in FPGAs have configurable aspect ratio, leading to implementation alternatives having different area/power tradeoffs. Metzgen and Nancekievill studied the inference and binding of multiplexers with the objective of reducing area [74].

Recent work by Howland and Tessier describes a power optimization approach in FPGA RTL synthesis [50]. A classic "data guarding" approach is taken, shown in Fig. 4. Select signals on multiplexers can be used to deduce that the outputs of certain circuit blocks do not affect overall circuit outputs, and hence the inputs to such blocks can be gated to reduce dynamic power dissipation within the blocks. Power-aware RTL synthesis for FPGAs is also the basis of a start-up company called PwrLite [85].

Commerial RTL synthesis tools FPGAs have been available for over 15 years. Altera and Xilinx currently offer their own RTL synthesis tools, and third-party tools from Synopsys, Mentor Graphics and Magma are also popular. Despite this, the industrial work has been kept proprietary and there has been a lack of published research on RTL synthesis. A robust and modifiable publicly-available RTL synthesis framework has not been available to the research community and there has also been a lack of RTL benchmark circuits. Recently however, Jamieson and Rose released a Verilog-based RTL synthesis framework for FPGAs that correctly infers multipliers from the input HDL [53]. Jamieson's framework may well serve as a launch point for further development of a more comprehensive solution that infers more varied block types.

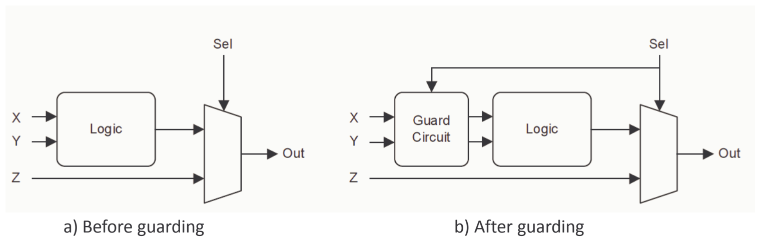


Fig. 4 Data guarding for power optimization in RTL synthesis (from [50]).

3 Logic Synthesis

The product of RTL Synthesis stage is a complete, though unoptimized, representation of a logic circuit as shown in Fig. 5. The circuit consists of input and output ports, inferred blocks such as adders, multipliers, memories and other specialized components found in the target FPGA device, as well as a clusters of registers and generic logic gates. The logic synthesis stage focuses on optimizing clusters of logic gates and registers in an effort to reduce the area they occupy, delay through the longest register-to-register path, and their power dissipation.

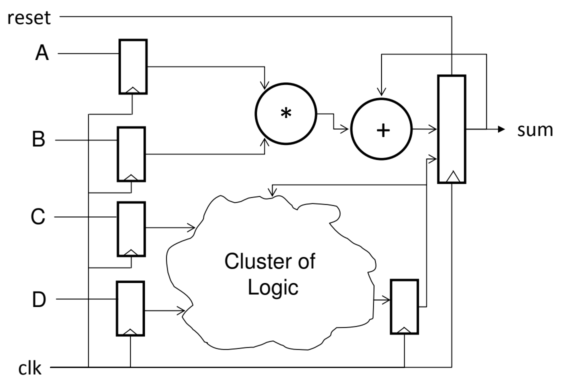


Fig. 5 Example output of RTL Synthesis.

In general, logic synthesis consists of two main parts: combinational logic synthesis and sequential optimization. Combinational logic synthesis looks at clusters of connected logic gates, referred to as logic cones, and applies algorithms that alter the structure of each logic cone without changing its logic function. Sequential optimization further improves the logic circuit by considering registers in the restructuring operations, allowing circuits such as finite state machines to be synthesized well. In the following, we describe combinational and sequential logic synthesis in more detail.

3.1 Combinational Logic Synthesis

Combinational logic synthesis is a process of optimizing logic functions in a circuit, without changing the logical behavior of the circuit. The simplest form of such optimization is two-level optimization, where a logic expression is implemented such that any path from the inputs of a logic expression to their outputs contains at most two gates (not counting including inverters). To illustrate this idea, consider the example in Fig. 6.

In Fig. 6(a) a logic function is expressed using a Karnaugh map. A Karnaugh map is a truth table that consists of rows and columns. The rows are indexed by variables x_1x_2 and the columns are indexed by variables x_3x_4. Both rows and columns are arranged such that adjacent rows/columns index values differ in exactly one bit position. This arrangement allows us to create a circuit for a logic function that consists of AND and OR gates. The AND gates are created by covering adjacent 1s in the table. For example, notice the encircled terms in column 01 in Fig. 6(b). We can

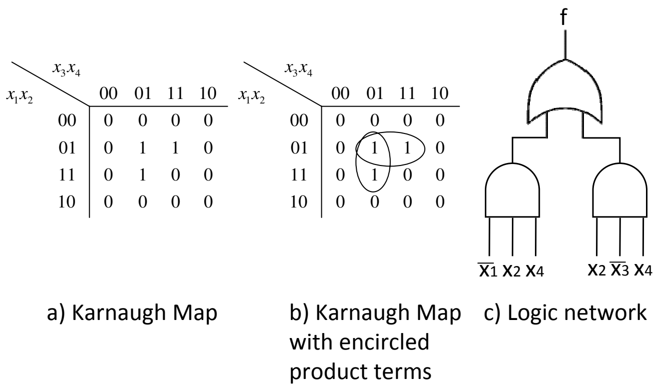


Fig. 6 Synthesis example for a simple logic function.

represent this group as a product term $x_2\overline{x}_3x_4$, because the logic function assumes a value of 1 when $x_2 = 1$, $x_3 = 0$, and $x_4 = 1$. Similarly, the terms in row 01 can be expressed as $\overline{x}_1x_2x_4$. Because the logic function is 1 when either of the two conditions are true, then the function can be synthesized as the logical OR of two AND gates. We thereby create a two-level representation of a logic function as shown in Fig. 6(c).

The above example illustrates the notion of taking a description of a logic function, in this case in a form of a Karnaugh map, and implementing it using logic gates. Although the example is simple, it shows the essence of logic synthesis. In practical applications, logic expressions that need to be implemented on an FPGA contain many more inputs and are much more difficult to synthesize. To address that problem, a wide array of methods have been developed to automate the process of implementing logic functions in FPGAs. In general, we distinguish three types of approaches that address this problem:

1. Tabular
2. Symbolic
3. Graph-based

Each of these approaches achieves the same goal (optimizing a logic function to minimize area or improve delay), however the approaches differ in how the initial logic expression is represented and how the optimization steps are performed on each logic function.

Tabular methods are based on the work of Ashenhurst [13] and Curtis [31]. In their work, they chose to represent logic functions using a table. A table has rows indexed by some subset of variables; the remaining variables are assigned to index the columns. An example table is shown in Fig. 7. The purpose of the table is to identify *compatible* columns. Such columns represent a function that could be extracted from a logic expression, permitting a simplified hardware implementation. For example, notice that column 001 is identical to columns 000, 010, and 011, save for the don't care entries (shown with a *d* in the table). We can express the first four columns of this table as a product of the column function, x_1x_2, and the selector function, $\overline{x}_3$, that determines where this column appears in the truth table. Applying similar reasoning to the remaining columns, we can implement the given function as shown in Fig. 8. Many works that followed performed a *decomposition* of logic expressions using tables to simplify logic circuits. In each instance, some relationship between columns was sought. Examples of such works include [52, 108, 32].

x_1x_2 \ $x_3x_4x_5$	000	001	010	011	100	101	110	111
00	0	0	0	*d*	0	0	0	1
01	0	0	0	0	0	0	0	0
10	*d*	0	*d*	0	1	0	1	0
11	1	1	1	1	1	0	1	1

Fig. 7 Example of an Ashenhurst-Curtis decomposition table.

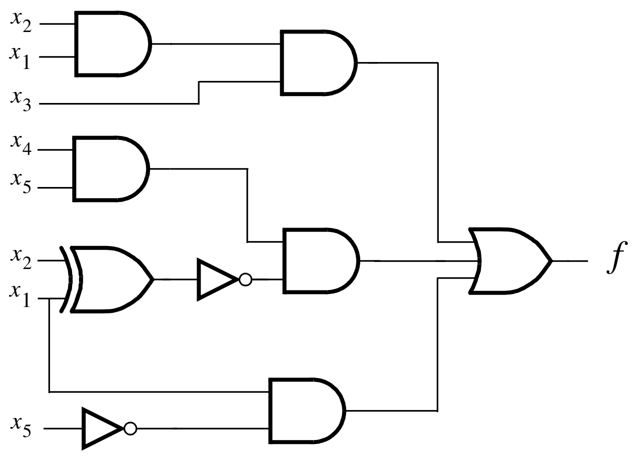


Fig. 8 Synthesized logic circuit for function in Fig. 7.

Symbolic methods focus on optimization of logic expressions. A classical problem in this context is one of decomposition of a sum-of-products logic expression. Such problems are addressed in the works on *kernel theory* and boolean or algebraic *division*, where subexpressions are extracted to simplify the final implementation of the logic function. For example, consider the goal of reducing the complexity of the following logic expression:

$$f = x_1x_2x_4 + x_1x_2x_5 + x_1x_3x_4 + x_1x_3x_5 + x_6$$

Using kernel theory, it is possible to process this equation to determine useful subexpressions, and implement the logic function as $f = x_1(x_2 + x_3)(x_4 + x_5) + x_6$. Examples of works that take advantage of symbolic manipulation include [90] and [101].

Finally, graph-based methods are techniques that operate on a graph representation of a logic function. Various graph methods have been proposed, but in all circumstances the graph consists of *nodes* that represent some logic function, and *edges* that connect the nodes to form a complete logic function. A seminal work in graph-based synthesis was that of Bryant [17], where he introduced the concept of binary decision diagrams (BDDs). Fig. 9 shows an example of a binary decision diagram.

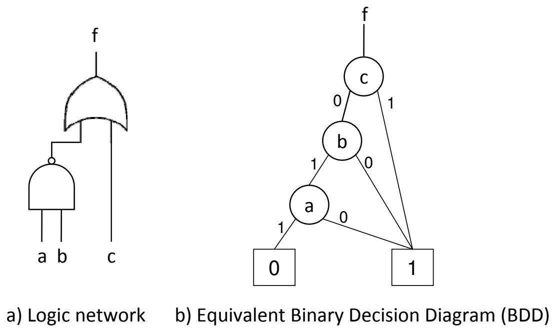


Fig. 9 Example of a binary decision diagram (BDD).

In this example, a logic function f is shown in Fig. 9(a). The same logic function is shown in Fig. 9(b) represented with BDDs. Nodes in a BDD correspond to input variables. To determine the value of a logic function for a given input pattern, one starts at the root (top) of the BDD graph, and traverses the graph one node at a time. At each node, a decision needs to be made as to which path to follow – the 0-path or the 1-path. This decision depends on the value of the variable the node represents. When the graph is traversed to one of the terminals (0 or 1 nodes), the function value can be determined. For example, to determine the value of f for the input pattern $abc = 010$ in the graph in Fig. 9(b), we proceed as follows. First, we begin at the root of the graph which is node c. Since variable $c = 0$, then we follow the path along the 0 edge to reach node b. Node b has a value of 1, which leads us to node a. Finally, by following the 0-edge from node a we reach a terminal node 1. Thus, for input pattern $abc = 010$, logic function value is $f = 1$.

There are numerous works which use BDDs to optimize logic functions. For FPGAs, and early work was by Lai et al. [58]; more recent works by Yang et al. [110], Vemuri et al. [102], and Cheng [23] show how BDDs can be utilized to synthesize various logic functions, finding ways to simplify them and thereby reduce their area.

A more recent work in graph-based techniques is the work of Mishchenko et al. [77], where AND-inverter graphs (AIGs) are used. In their work, a logic function is represented as a set of nodes that function as AND gates, connected by invertible edges. Invertible edges can be used to complement a logic expression that follows. An example of an AND-inverter graph is shown in Fig. 10.

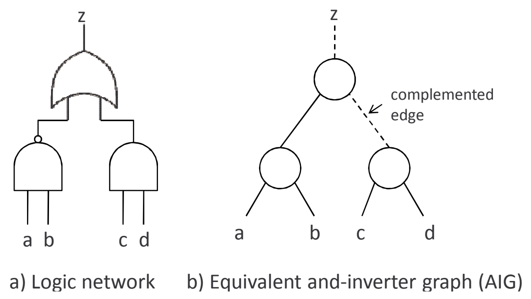


Fig. 10 Example AND-inverter graph used within the ABC synthesis tool.

Using AIGs, Mishchenko et al. used a *rewriting* technique [34] that can optimize logic functions rapidly. AIGs and rewriting optimizations became the basis for the ABC synthesis system, which recently became a popular research framework [82]. AIGs were also used by Ling et al. [65], where they use rewriting techniques to reduce the depth of a logic circuit in an effort to improve circuit performance, while maintaining circuit size.

The treatment above is only a basic introduction to logic synthesis of combinational circuits. Over the past decades, a wide array of techniques have been developed, addressing various types of logic functions. Comprehensive surveys of logic synthesis and decomposition techniques can be found in [28] and [83].

3.2 Sequential Optimization

Sequential optimization is a field of logic synthesis that deals sequential circuits – circuits that retain “state”. One class of such circuits is finite state machines (FSMs). FSMs are used to describe a sequence of events, where each event has a state associated with it. For example, consider an FSM with input w and output z, where z becomes high on a clock cycle following a sequence of 110 on input w. The state diagram for this FSM is shown in Fig. 11.

Implementing a state machine, such as the one in the above example, is a matter of selecting an encoding for each state, implementing logic to determine the state

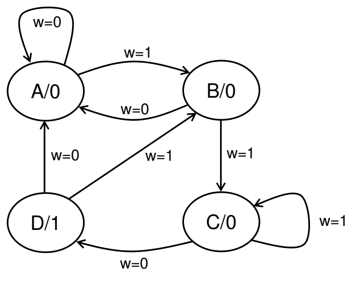


Fig. 11 A simple FSM example.

transitions and generating the output logic. For example, if we choose an encoding such that $A = 00$, $B = 01$, $C = 10$ and $D = 11$, then the circuit we generate for this FSM is as shown in Fig. 12(a). However, altering the state encoding such that $C = 11$ and $D = 10$ results in a smaller circuit, as shown in Fig. 12(b).

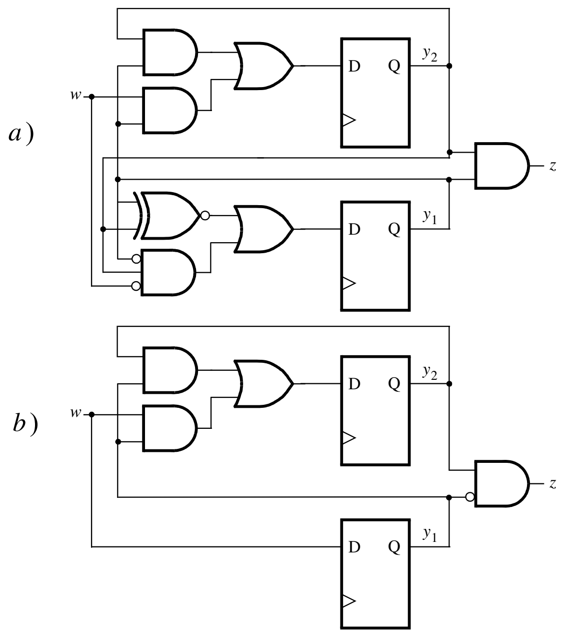


Fig. 12 Two possible implementations for the FSM in Fig. 11.

The above example highlights a problem with FSM implementation, namely, how to choose an encoding for each state to obtain the best solution for performance, area and power? This not only includes selecting the appropriate number of flip-flops to represent logic states, but also choosing an encoding of states so as to minimize next state and output logic. A recent work in [76] is an example of an area optimization of sequential circuits that leverages the notion that sequential circuits contain unreachable states, whereas power-oriented sequential optimizations are discussed in [78].

In addition to handling FSMs, sequential optimization algorithms also handle pipelined paths. One of the most popular algorithms is called *retiming*. Retiming is an operation that either pushes flip-flops forward or backward through the circuit, without altering the logical functionality of the circuit. The idea behind this operation is to allow CAD tools to rebalance path delays, with the ultimate objective being to improve circuit speed. Consider the high-level example in Fig. 13.

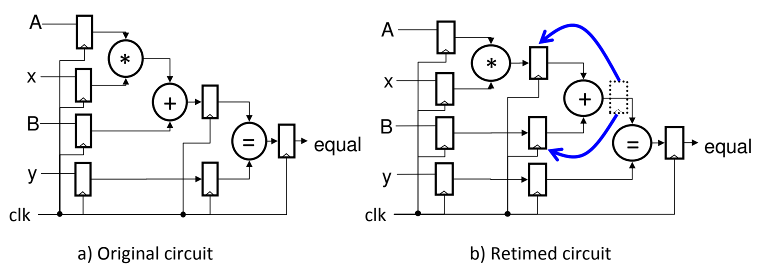


Fig. 13 A high-level example of retiming.

In this example, the circuit performs a check if $y = Ax + B$ and returns a result of 1 if the equation holds. Although it is conceptually easier to first evaluate the right hand side of the equation and then compare the result to y, as shown in Fig. 13(a), it is not necessarily the best approach from a performance standpoint. This is because a multiplier is much slower than an adder or a comparator. We can speed up the circuit by pushing back flip-flops at the output of the adder, as shown in Fig. 13(b). The resulting circuit will perform the same operation in the same number of clock cycles, however, the circuit will now be able to function at a higher clock frequency. The example demonstrates the concept of retiming. In practice, retiming occurs at the LUT level, where flip-flops are pushed backwards or forwards through a single LUT, which can provide a significant improvement in circuit performance [97]. Examples of works that take advantage of retiming include [96, 97, 29, 88, 1, 94]. It is worth noting that retiming is particularly well-suited for FPGAs, as FPGAs contain many registers (usually one register per LUT is provided), thereby permitting easy register insertion.

3.3 Remarks

The topic of logic synthesis is one of incredible depth. It has been researched vigorously for over 50 years, and despite the wealth of research, new and innovative approaches are still created today. To interested readers, who wish to explore this topic further, we recommend the following books: [46, 79, 36, 89].

4 Technology Mapping

Technology mapping transforms the circuit from a network of generic logic elements/gates into a network of the logic blocks available in the target FPGA. The majority of literature on FPGA technology mapping relates to mapping the circuit into look-up-tables (LUTs). Recall that a K-input LUT (K-LUT) can implement *any* logic function that uses up to K variables. Therefore, during technology mapping we only need be concerned with the number of inputs to each LUT and not its logic function. As such, most technology mapping algorithms first translate the circuit into a directed acyclic graph (DAG), and then map it into a network of input functions, each using no more than K inputs.

For example, consider a logic network in Fig. 14(a). First, the logic network is expressed as a DAG Fig. 14(b). The technology mapping task is to "cover" the DAG with LUTs, as shown in the 4-LUT mapping solution in Fig. 14(c). There are two LUTs in the mapping solution; observe that each LUT uses no more than 4 variables.

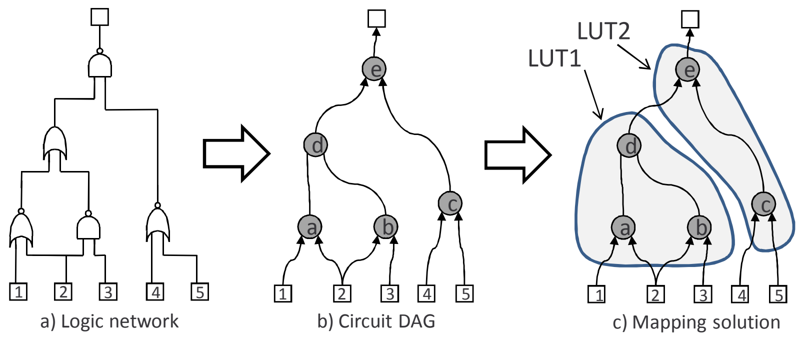


Fig. 14 Logic circuit, DAG and mapping solution.

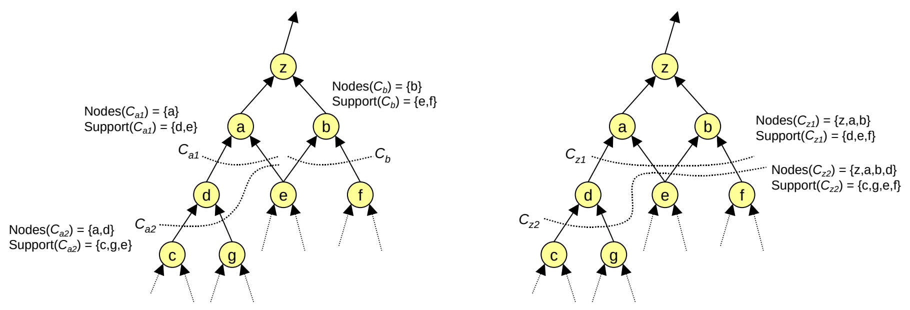
Research on technology mapping for FPGAs was active in the early 90s with a wide range of algorithms proposed [39, 41, 40, 25, 26]. Recent technology mappers are based on the notion of cuts [92, 27], which we delve into here. In the circuit DAG, $G(V,E)$, each node, $z \in V$, represents a single-output logic function and edges between nodes, $e \in E$, represent input/output dependencies between the corresponding logic functions. For a node z in the circuit DAG, let $inputs(z)$ represent the set of nodes that are fanins of z. For a subgraph, H, of a DAG, let $inputs(H)$ represent the set of nodes outside of H that are fanins of nodes in H. For example, in Fig. 14(c), $inputs(e) = \{d, c\}$ and $inputs(LUT2) = \{d, 4, 5\}$.

A node x is said to be a predecessor of node z if there exists a directed path in the graph from x to z. The subgraph consisting of a node z and all of its predecessors is referred to as the subgraph *rooted* at z. For any node z in a network, a *K-feasible cone* at z, N_z, is defined to be a subgraph consisting of z and some of its predecessors such that $|inputs(N_z)| \leq K$. Consequently, the technology mapping problem for K-LUTs can be thought of as covering an input Boolean network with K-feasible cones. Generally, there are many K-feasible cones for each node in the network, each having different area, delay, or power characteristics.

A concept closely related to K-feasible cone is that of *K-feasible cut*. A K-feasible cut for a node z is a partition, $(X, \overline{X})$, of the nodes in the subgraph rooted at z such that $z \in \overline{X}$, and the number of nodes in X that fanout to nodes in $\overline{X}$ is $\leq K$. There is a one-to-one correspondence between K-feasible cuts and K-feasible cones. Given a cut, $(X, \overline{X})$, the K-feasible cone is simply the subgraph induced by the nodes in $\overline{X}$. The key point to realize is that the problem of finding all possible K-LUTs that generate a logic function for node z is equivalent to the problem of enumerating all K-feasible cuts for node z. To simplify the presentation, for a K-feasible cut, $C_z = (X, \overline{X})$, for a node z, $Nodes(C_z)$ is used to represent the set $\overline{X}$, where $z \in \overline{X}$. $Support(C_z)$ is used to represent subset of nodes in X that fanout to nodes in $\overline{X}$. For example, for consider cut C_{z1} in Fig. 15(b), $Nodes(C_{z1}) = \{z, a, b\}$ and $Support(C_{z1}) = \{d,\ e,\ f\}$. Finally, $Cuts(z)$ is used to represent the set of all feasible cuts for a node z.

Traversing the circuit DAG in topological order (from inputs-to-outputs), the cuts for each node z are generated by merging cuts from its fanin nodes, using the method described in [27, 92] and outlined here. Consider generating the K-feasible cuts for a node z with two fanin nodes, a and b. The list of K-feasible cuts for a and b have already been computed, due to the graph traversal order. Say that node a has two K-feasible cuts, C_{a1} and C_{a2}, and node b has one K-feasible cut, C_b, as shown in Figure 15(a). We can merge C_{a1} and C_b to create a cut, C_{z1}, for node z, such that $Support(C_{z1}) = Support(C_{a1}) \cup Support(C_b)$ and $Nodes(C_{z1}) = z \cup Nodes(C_{a1}) \cup Nodes(C_b)$ [see Figure 15(b)]. If $|Support(C_{z1})| > K$, the resulting cut is not K-feasible, and it is therefore discarded. Similarly, one can merge C_{a2} and C_b to create another candidate cut, C_{z2}, for node z. This provides a general picture of how the cut generation procedure works; however, there are several special cases to consider, and the reader is referred to [92] for complete details.

Having computed the set of K-feasible cuts for each node in the DAG, the graph is traversed in topological order again. During this second traversal a “best cut” is chosen for each node. The best cut may be chosen based on any criteria, whether it be area, power, delay, routability or a combination of these. In technology mapping, the depth of the longest path in the mapped network is often used as a proxy for the critical path delay. As a concrete example, if optimizing the depth of the mapped


z
a
b
d
e
f
c
g
Nodes(C_b) = {b}
Support(C_b) = {e,f}
Nodes(C_{a1}) = {a}
Support(C_{a1}) = {d,e}
C_{a1}
C_b
Nodes(C_{a2}) = {a,d}
Support(C_{a2}) = {c,g,e}
C_{a2}
Nodes(C_{z1}) = {z,a,b}
Support(C_{z1}) = {d,e,f}
C_{z1}
Nodes(C_{z2}) = {z,a,b,d}
Support(C_{z2}) = {c,g,e,f}
C_{z2}

(a) cuts for nodes *a* and *b* (b) cuts for node *z*

Fig. 15 Generating the *K*-feasible cut sets.

network is desirable, then, for a node z with a K-feasible cut, C_z, the cost of the cut is defined as:

$$Cost(C_z) = 1 + \max_{v \in Support(C_z)} \{Cost[BestCut(v)]\} \tag{1}$$

Thus, to compute the depth cost of cut C_z, (1) considers the depth cost of the best cut for each node, v, that fans out to a node in $Nodes(C_z)$. The best cut has already been selected for each of these support nodes, since the network is being traversed in an input-to-output fashion.

The last part of technology mapping is to build the final LUT network. A FIFO queue is initialized to contain all output nodes in the circuit. A node, v, is removed from the queue and its best cut, $C_v = BestCut(v)$, is recovered. The subnetwork corresponding to $Nodes(C_v)$ is implemented as a LUT in the mapping solution. Each node in $Support(C_v)$ is then added to the end of the FIFO queue, if it is not already in the queue. The process of removing nodes from the queue, using their best cuts to establish LUTs in the mapping solution, and adding the support of these cuts to the end of the queue continues until the queue contains only primary inputs. When this condition is met, the input Boolean network has been fully mapped into LUTs.

The beauty of cut-based technology mapping is that any cost function can be applied to rank the cuts and thus it is relatively easy to adapt cut-based mapping to optimize for any objective. Only the costing of cuts need be changed; the process of computing the cuts and generating the final mapped network remain the same. Cut-based mapping has been used extensively in many works to optimize for depth, power, area and routability [21, 9, 59, 92].

Despite the relative maturity of the topic, there have been a number of important breakthroughs in FPGA technology mapping in recent years. As noted in Chapter X, modern FPGAs contain LUTs with 6-inputs ($K = 6$). An upper bound on the number of cuts for a node is $O(n^K)$, where n is the number of nodes in the circuit. Thus, with $K = 6$, there can be a large number of cuts per node, increasing algorithm run-time and memory consumption. Mishchenko et al. address the cut explosion using the notion of *priority cuts* [75]. The idea is that instead of storing all possible cuts for each node, only a subset of “priority cuts” is stored, based on a cost function. When generating the set of cuts for a node, only the priority cuts of its fanin nodes are considered for merging. Despite the fact that many cuts are pruned with this technique, very little quality degradation is observed in practice, and the idea has been picked up and applied by industry [56].

Another recent concept, called *area flow*, was introduced by Manohararajah et al. and shown to significantly reduce the area (# of LUTs) of the mapped network [70]. The innovation in area flow is to divide the cost of a multi-fanout LUT equally across its fanout LUTs. For example, consider a LUT *A* with two fanin LUTs *B* and *C*. Furthermore, consider that LUT *C* has some other fanout LUT *D* (besides LUT *A*). Area flow recognizes that LUT *A* should not incur the full “charge” for LUT *C*, as *C* also fans out to another LUT, *D*. The costing strategy has implications on how multi-fanout nodes in the DAG are mapped, ultimately impacting area. Furthermore, in [70], the costing and mapping steps execute several times, where one iteration of costing and mapping uses fanout and depth information from the prior iteration to make better decisions. Manohararajah’s work has traction and was adapted for commercial application by Xilinx to reduce the number of connections in the mapped network [54].

A recent contribution to technology mapping is the use of Boolean satisfiability (SAT). Recent developments in SAT solver technology have enabled the use of the concept for practical purposes. An important work on technology mapping for FPGAs using boolean satisfiability was published by Ling et al. [64]. In their work, both the functional capabilities of the logic block, as well as a logic expresson to be potentially implemented in the logic block, are expressed as a Boolean equation (in conjunctive normal form). The equation is formulated such that if there is a satisfying assignment to the variables in the equation (equation output value is 1) then the logic expression can indeed be implemented by the logic block. SAT is used to determine if there is a satisfying variable assignment. Ling’s work considered logic blocks with different LUT configurations, however, the concept is easily extendible to target logic elements in commercial architectures, such as the Altera Stratix III logic element shown in Fig. 16. The appeal of using SAT is the ease with which complex logic block architectures can be described to a SAT solver, allowing researchers and designers to explore non-trivial mapping solutions for arbitrary logic functions.

5 Packing

Packing, also known as *clustering*, is the step wherein the elements of the technology mapped circuit are packed into the available FPGA hardware resources. Clusters of LUTs and flip-flops form the basis for logic blocks in today’s FPGAs, with fast local interconnect available for intra-logic block connectivity. Most commonly,

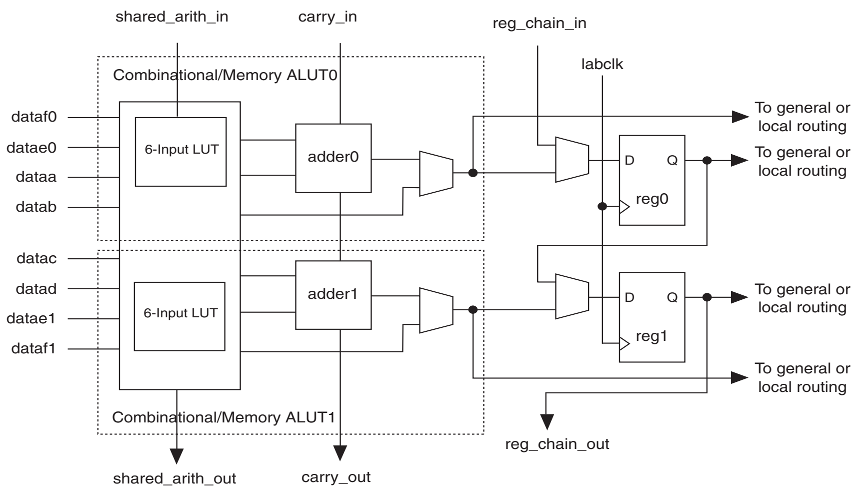


Fig. 16 High-level diagram of the Altera Stratix III adaptive logic module (ALM) [5].

the packing step combines LUTs and flip-flops in a design together to form logic blocks.

Fig. 17 depicts the classic FPGA logic block model used in the vast majority of academic research. It consists of a cluster of LUTs and flip-flops, where each flip-flop can be bypassed for implementing combinational logic. Inputs to the logic block come from the FPGA's general interconnect: horizontal and vertical channels of FPGA routing. Local interconnect inside the logic block is available for realizing fast paths within the logic block. Observe that each LUT/FF pair drives both local interconnect, as well as general interconnect. Most prior packing work assumes the local interconnect to be a full crossbar switch matrix – every input can be programmably connected to any output. In this logic block model, connections *within* the logic block are fast, and connections *between* logic blocks, routed through the general interconnect, are slow in comparison. The packing step decides which LUTs to put together into a single logic block, and therefore, packing has a significant impact on circuit speed. The model of Fig. 17 is representative of early Altera FLEX FPGAs [2], however, it has become out-of-step with the logic blocks present in modern Xilinx and Altera FPGAs. Modern logic blocks present a more complex packing problem, new optimization opportunities and impose different constraints.

Much research has been published on packing for the logic block in Fig. 17. Perhaps the most cited work is that of Betz and Rose who proposed an area-driven packing algorithm, and showed that the number of inputs to a logic block can be much smaller than the total number of LUT inputs within a cluster, due inherent locality in circuits [14]. In particular, for a logic block with N 4-input LUTs, [14] showed that only $2N+2$ inputs to the cluster are needed – a number much smaller than $4N$. Marquardt extended the work to perform timing-driven packing and demonstrated the impact of packing on critical path delay [71].

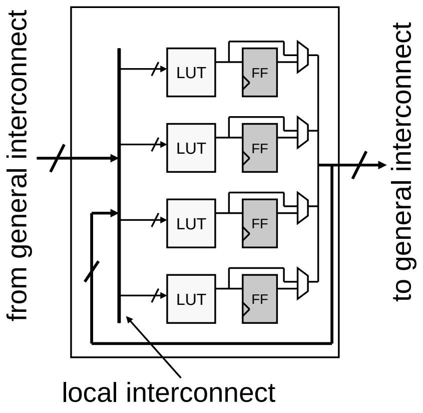


Fig. 17 Classic FPGA logic block targeted by most academic research.

Packing affects power consumption as intra-logic block connections will have lower capacitance than inter-logic block connections. A natural approach is to attempt to keep nets with high switching activity contained within logic blocks, as was proposed in [59]. An entirely different approach for power-driven packing was shown in [95], where Rent's rule was used to establish a preference for how many logic block inputs should be used during packing, leading to lower overall interconnect usage, capacitance and power. Although not yet available commercially, dual-V_{DD} FPGAs have been proposed by academia, where the idea is to programmably allow logic blocks to operate at reduced supply voltage (slower but lower power). Researchers at UCLA developed a complete CAD flow for a proposed dual-V_{DD} FPGA, including new packing techniques [22]. The aim of packing in this context is to pack LUTs based on their timing-criticality, placing non-critical LUTs together into logic blocks that will be operated at low V_{DD}. The work in [48] dealt with packing for a low-power FPGA having logic blocks that when idle, can be placed into a low leakage sleep state.

On the speed axis, more recent work includes [35] which uses a Rent's rule-based algorithm, and prevents loosely connected, or unrelated, LUTs from being packed together. Other papers tie together packing with other phases of the FPGA CAD flow. For example, [91] looked at packing in the context of logic replication for performance; a subset of LUTs are deliberately left empty by the packer to accomodate later LUT replications during placement. An interesting recent work by Lin et al. brought together packing and technology mapping and showed that higher speed can be attained using a unified algorithm for concurrent packing and technology mapping [62].

Relative to the block in Fig. 17, modern FPGAs have more complex logic blocks containing multi-output fracturable LUTs, multiplexers, gates, carry chains, and configurable registers. Little has been published on packing for commercial chips. To illustrate, Fig. 18 shows a quarter of a logic block (called a SLICE) in the Xilinx

Virtex-5 FPGA. A recent work by Ahmed et al. considered packing for Virtex-5 [3]. Observe in Fig. 18 that the LUTs in Virtex-5 have six inputs and two outputs, and can implement a single 6-input logic function or any two functions that together use no more than 5 inputs. The authors pack LUTs into the dual-output LUT during an integrated packing/placement phase, improving logic density in the FPGA while maintaining speed performance.

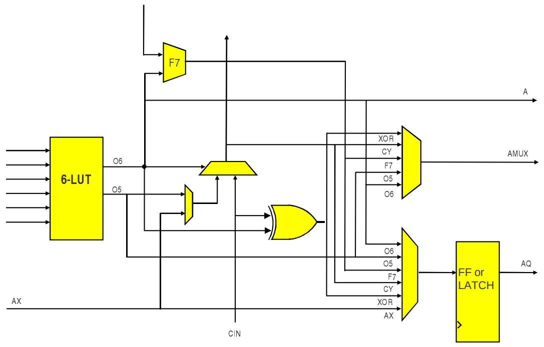


Fig. 18 Quarter of logic block (SLICE) in Xilinx Virtex-5 FPGA.

6 Placement

The result of technology mapping is a network of logic blocks that are ready to be located on the target two-dimensional FPGA device. There are two popular approaches to FPGA placement: one is based on the simulating annealing algorithm, and the other uses analytical placement techniques. We briefly outline both approaches here.

Simulated annealing is an optimization strategy that has proven effective for FPGA placement, owing to its flexibility to incorporate virtually any objective or constraint. Annealing is used in the VPR placer [15, 72], and in the Altera commercial placer, as well as in prior academic work . In annealing-based placement, an initial placement is first constructed, possibly randomly. The entire placement is assigned a numerical cost, reflecting estimated wirelength, speed performance and other criteria:

$$C = \alpha \cdot WL + \beta \cdot PERF + \gamma \cdot OTHER \tag{2}$$

where the wirelength and performance costs, WL and $PERF$, must be estimated, as precise routes are unavailable. The weights α, β and γ, are chosen to reflect the importance of each term. The advantage of simulated annealing is that $OTHER$ in (2) can be designed to represent any other objective criteria. For example, it could represent the chip power or legality constraints on the placement. As a concrete example, in VPR, the wirelength cost is:

$$WL = \sum_{n=1}^{N_{nets}} q(n) \cdot [bb_x(n) + bb_y(n)] \tag{3}$$

where $bb_x(n)$ and $bb_y(n)$ represent the span of net n in the x and y dimensions, respectively, and the $q(n)$ factor is 1 for nets with 3 or fewer terminals, and increases to 2.79 for nets with 50 terminals. The $q(n)$ factor addresses the problem that the sum of a net's x and y span is an underestimate of its total wirelength for nets with many terminals. The performance cost is:

$$PERF = \sum_{conn \in Circuit} d(conn) \cdot crit(conn)^e \tag{4}$$

where $conn$ is a connection in the circuit, $d(conn)$ is the estimated delay of the connection and $crit(conn)$ is the connection's timing criticality in the range of 0 to 1, with 1 meaning the connection is on the critical path, and 0 meaning the connection is non-critical. Parameter e in (4) is a tuning parameter.

Given an initial placement and a cost function, an annealing-based placer attempts random perturbations to the placement, and for each attempt, a change in cost, ΔC, is computed. A random perturbation typically comprises moving a single logic block to a new location or swapping one logic block with another. Perturbations that improve the cost ($\Delta C < 0$) are always accepted, while perturbations that worsen cost *may* be accepted with a probability:

$$P(Accept) = e^{\frac{-\Delta C}{T}} \tag{5}$$

where T is a parameter called *temperature* that decreases throughout the placement process. Initially, with T high, perturbations that worsen the placement are more likely to be accepted. Many perturbations are attempted at each temperature (thousands or tens of thousands at each temperature is not uncommon). As T is gradually decreased, perturbations that increase cost become less likely to be accepted. The value of accepting some perturbations that worsen cost is known as *hill climbing*; in essence, there exist scenarios where taking a few "bad" (uphill) moves can lead to a lower overall cost later in placement. Fig. 19(a) plots (5) for several temperatures. Fig. 19(b) shows how the wirelength cost value in the VPR placer changes across the placement iterations, where one iteration corresponds to one temperature. Observe that from the initial random placement, a 2/3 reduction in estimated wirelength is observed. Altering the initial placement, temperature, or the rate of temperature decrease has a drastic effect on the run-time and quality of annealing-based placement [87].

An alternative to annealing is to use analytical placement techniques, as in the Xilinx commercial placer. Analytical placement normally begins with a fixed placement of the I/O objects. A placement for the core objects is computed mathematically, such the squared wirelength is minimized:

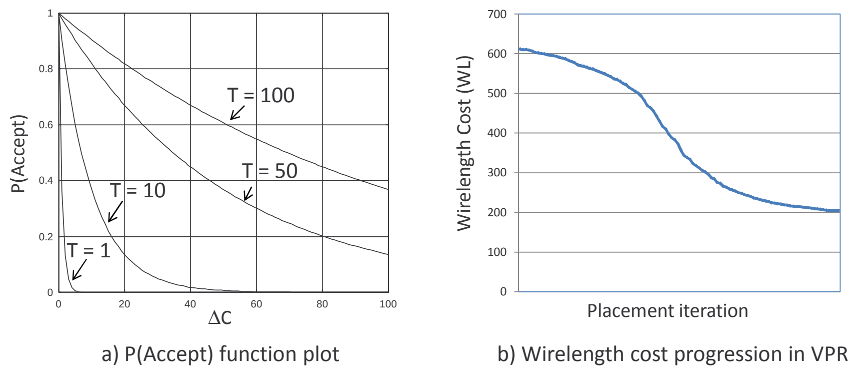


Fig. 19 Annealing cost function and placement cost progression.

$$\Phi = \sum_{j=1}^{N_{LogicBlocks}} \sum_{i=1}^{N_{LogicBlocks}} w_{i,j} \cdot [(x_i - x_j)^2 + (y_i - y_j)^2] \qquad (6)$$

where $w_{i,j}$ is a positive weight if logic block i connects to logic block j, and is zero if i and j are not connected to one another. The variables x_i and y_i represent the x and y positions of logic block i in the placement. The general approach is to find values for the x_i, y_i variables such that Φ is minimized. Since (6) is a quadratic function, it can be minimized by solving a linear system with standard solvers. Placement is done in the real-valued domain, and therefore placement results must be *snapped* onto the FPGA grid.

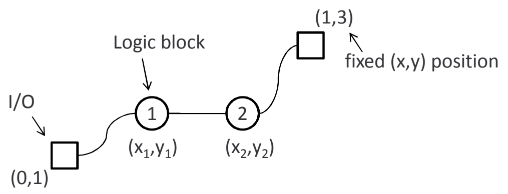


Fig. 20 Toy analytical placement example.

The approach is best illustrated by an example. Consider the circuit shown in Fig. 20, with two logic blocks and two I/O blocks (whose placement is fixed). We assume the edge weights are 1 in this example. The optimization function, Φ, can be broken into separate x and y components that can be minimized separately to find the values of unknowns x_1, x_2, y_1, and y_2:

$$\Phi_x = 1 \cdot (x_1 - 0)^2 + 1 \cdot (x_1 - x_2)^2 + 1 \cdot (x_2 - 2)^2 \qquad (7)$$

and

$$\Phi_y = 1 \cdot (y_1 - 1)^2 + 1 \cdot (y_1 - y_2)^2 + 1 \cdot (y_2 - 3)^2 \quad (8)$$

To minimize these equations, we need to solve two linear systems:

$$\begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \text{ and } \begin{bmatrix} 2 & -1 \\ -1 & 2 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \quad (9)$$

These systems can be solved using standard techniques, such as Gauss-Siedel or the conjugate gradient method [44].

Observe that the formulation described above does not incorporate constraints that prohibit logic blocks from overlapping with one another. Indeed, analytical placement initially produces an overlapped, infeasible placement. Fig. 21 shows an example of initial analytical placement results for a circuit. The figure shows I/O blocks, placed on the periphery, surrounding an overlapped placement of core logic blocks. Based on the initial placement, the formulation is modified to reduce overlaps, while at the same time keeping connected blocks close to one another. Much research has been published on how best to modify the formulation, with popular approaches being [104, 105, 106, 38]. The revised formulation is then re-solved and a new placement is produced. The process of solving, re-formulating, and re-solving progresses iteratively, gradually reducing the number of overlaps, eventually producing a placement of logic blocks that is fairly free of overlaps. Finally, the logic blocks are snapped onto the discrete placement slots available on the target FPGA grid.

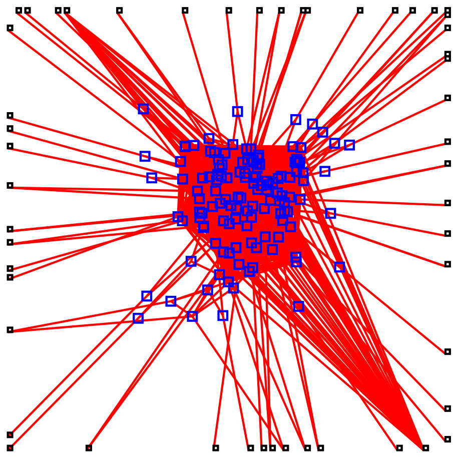


Fig. 21 Initial analytical placement of a circuit.

Following placement with either simulated annealing or analytical techniques, a greedy optimization is typically executed. Pairwise swaps of logic blocks are attempted and accepted if the placement is improved. This optimization can be done in a windowed fashion, where, within a window of the placement area, all possible logic block swaps are considered. Subsequently, the window is shifted to another region on the chip.

Placement and routing are the most time consuming phases of the CAD flow, requiring hours or even days for the largest commercial designs. Research on run-time reduction is therefore paramount, and a promising direction is through parallelization of placement algorithms. Chan and Schlag considered parallelizing VPR across a network of computers [19], showing considerable speed-up. More recently, Altera released a parallel placer targeted to modern multi-core microprocessors [67]. One core proposes the moves (perturbations), while the other cores evaluate moves concurrently. Backtracking is required in some cases to maintain deterministic results. This happens when moves evaluated concurrently are interdependent, e.g. the ΔC value for the n^{th} move depends on the ΔC value for the $n-1^{th}$ move.

Another important direction in FPGA placement research is consideration of architecture-specific placement constraints. I/O objects in FPGAs are organized into banks, with constraints on the I/O signaling standards may be placed together in a single bank. The constrained I/O placement problem has been handled through a new term in simulated annealing-based placement [12] and also through integer linear programming techniques [69]. Recently, the importance of recognizing the structure of the pre-fabricated clock network in placement has been shown [60, 107, 109]. The logic blocks in modern FPGAs are partitioned into clock regions where blocks in a region have access to the same set of clock signals. Power consumption can be reduced by limiting the number of regions spanned by the logic blocks belonging to a single clock domain. Power consumption can also be reduced by incorporating signal capacitance estimation into the placer, and including power estimates into the annealing cost function [86, 59, 45].

7 Routing

The role of the router is to form the desired electrical connections between the logic blocks in a placed design. FPGA routing differs considerably from routing in custom ICs. In custom ICs, wires, vias and repeaters (buffers) may be located anywhere by the router, as allowed by layout design rules. In FPGAs, however, the metal routing wires are fixed, as are the repeaters and routing switches. Programmable switches permit wires to be programmably connected to one another and permit pins on logic blocks to be connected to wires. The router's job is ultimately to decide which switches to turn on to make the desired connections between logic blocks, while meeting speed and power constraints.

An example of a simplified programmable routing network is shown in Fig. 22. The figure shows 4 logic blocks, each with 4 pins. Each pin can be programmably connected to two neighboring wires, illustrated by X in the figure. Metal wires can also be connected to other metal wires, using switch blocks. For clarity, in Fig. 22,

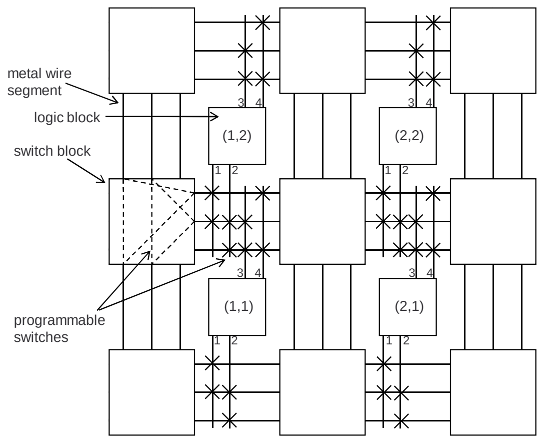


Fig. 22 Abstract FPGA routing fabric.

programmable connections are shown in only one of the switch blocks, illustrated using dashed lines. A search-based algorithm is used in FPGA routers to route a load pin on a signal. Beginning with the signal's source pin, a greedy search algorithm, similar to Dijkstra's algorithm [37], is used to traverse the interconnect network towards the load pin. In this search, each of the routing resources (wires, pins and switches) is assigned a cost corresponding to delay, capacitance, length or other criteria. The router's objective is to find a low-cost path for each load pin. Fig. 23 shows an example routing solution for 3 connections. For example, one of the connections is between pin 4 on the logic block at (1,2), which connects to pin 3 on the logic block at (1,1). Modern FPGAs contain metal wire segments of varied lengths, allowing long distance connections to be made using fewer programmable switches, reducing interconnect delay. Routers must also handle multi-fanout signals, where it is normally advantageous to share partial routing paths between loads, reducing overall capacitance and power.

While FPGA routing research has been active since the early 1990s (e.g., [16, 61, 4]), a breakthrough occured in 1995, with the publication of the PathFinder algorithm [73] by McMurchie and Ebeling, based partly on prior work in the ASIC domain by Nair [80]. Nair noted that given a set of connections to route, the first connections routed create blockages for the later connections, making routing solutions dependent on connection order. His innovation was to reduce order dependence by routing all connections multiple times, in the same order. Consider a scenario where there are n connections to route. After routing all connections once, a second pass begins wherein the first connection is ripped-up and re-routed; however, in this second pass, while routing the first connection, the connections $2-n$ are seen as blockages.

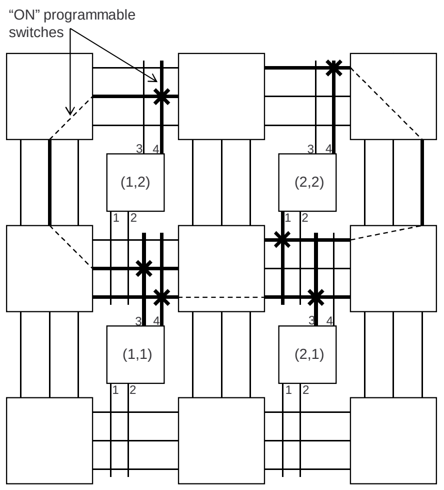


Fig. 23 Example routing solution for 3 connections.

PathFinder borrows Nair's iterative routing concept and also allows interconnect resources (i.e. wires, pins and sets of wires) to be *over-subscribed* in the early steps of routing. This means, for example, that a single metal wire segment can be shared by multiple different signals. Such signal shorts are permitted initially, and then removed gradually by a rip-up and re-route mechanism, eventually producing a feasible short-free routing.

Fig. 24 gives the flow of the PathFinder algorithm. Initially all signals are routed in the best-possible manner, without concern for shorts between signals. Timing-critical signals will be routed with low-delay; non-timing-critical signals will be routed to minimize resource usage. Since shorts are ignored initially, the initial routing solution will be independent of the order in which signals are routed. After initial routing, assuming the presence of shorts, some or all signals are selected, ripped-up and re-routed. The penalties for creating new shorts are then increased. In essence, signals that are shorted together on a wire *negotiate* among themselves for that wire; hence, the label applied to the algorithm: *negotiated congestion routing*. The process continues iteratively, until either the routing is feasible, or else a fixed number of loop iterations is exceeded and the design is deemed unroutable. PathFinder has proven to be robust in practice and it produces good routing solutions. The two largest FPGAs vendors, Xilinx and Altera, both use variations of PathFinder in their commercial routers.

Many improvements to PathFinder have appeared in the literature. Swartz et al. enhanced PathFinder from the run-time viewpoint, by pruning the router's search when routing a signal load pin [99]. Two pruning techniques are proposed: 1) instead of applying a breadth-first exploration of the interconnection network, a directed search toward the load pin is used, and 2) when routing a load pin, a bounding rectangle

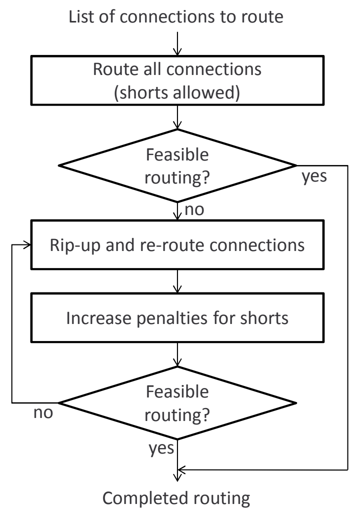

List of connections to route
Route all connections (shorts allowed)
Feasible routing?
yes
no
Rip-up and re-route connections
Increase penalties for shorts
Feasible routing?
no
yes
Completed routing

Fig. 24 Flow of PathFinder algorithm.

is drawn, encompassing the load and the signal's source pin, and only those routing resources falling within the rectangle are considered (the rest are pruned). Several researchers have attempted to parallelize PathFinder, using multple CPUs to route different sets of signals [18, 20] concurrently; however, the parallelization research was conducted in a distributed computing framework and not on the multi-core processors available today.

Timing-driven routing has typically meant minimizing the delay of the longest critical path, however, modern routers must also handle the scenario where paths are too fast. Interconnect delays in FPGAs are decreasing due to technology scaling and architectural improvements, making hold time violations a concern, leading to minimum delay constraints on connections. Altera tackled the problem of handling short and long path constraints within a PathFinder-based framework [43]: given short and long path delay constraints, a maximum and a minimum delay can be computed for each connection [42], yielding an acceptable delay window. The router's cost function is adapted to have a "valley" shape, where the lowest (best) cost (base of valley) corresponds to the center of the delay window, and cost escalates as connection delay becomes either too short or too long.

Growing device sizes necessitate concentration on the scalability of CAD algorithms. Several recent works deal with router memory consumption. Sharma and Hauck used a clustering approach to reduce the size of the table holding cost estimates for router search pruning [93]. Chin and Wilton noted that since the FPGA's interconnection fabric is regular, a data model representing the entire interconnection network need not be kept in memory at once [24]. Rather, the fabric can be computed "on the fly" during the routing of an individual pin, with special handling for irregularities in the FPGA fabric, e.g. routing on the edge of the chip [24].

8 Physical Synthesis

Following routing, the circuit can be implemented on an FPGA. However, recent research has shown that a logic circuit implementation can still be significantly improved post-routing. The main reason for such improvement is that complete design implementation data is only available post-routing. Algorithms can utilize this data to further improve the circuit. In physical synthesis, a feedback loop is introduced into the FPGA CAD flow. The feedback starts after routing (or perhaps placement) and leads back to any of the earlier stages in the flow. The core idea is provide physical information, primarily delay information, back to earlier phases of the flow, to allow better optimizations to be applied based on more accurate information. Both Altera and Xilinx CAD tools incorporate physical synthesis optimizations.

Within the physical synthesis flow, we can distinguish several different approaches. The first type applies synthesis, technology mapping and placement in an iterative process. The second type uses the synthesizer to specify to the placer where to place logic elements. This enables the placer to better understand the decisions made by the synthesizer, and possibly to accommodate them. A third type permits the placer to consider several alternative logic mappings for placement.

An example of the iterative approach was proposed by Lin et al. [63]. During each iteration, the mapping algorithm takes some of the gates from one LUT and places them in another, basing its decisions on net delays between the gates. The new mapping is then placed again, using the prior placement as a guide. The work of Singh and Brown [96] proposes that the placer be provided with an incentive to situate logic elements in a specific location on the device. Their approach starts by executing the normal CAD flow to obtain a synthesized and placed implementation. Then, placement-driven optimization techniques are invoked to reduce the delay on critical paths. Each new logic element (LUT) created in the process is assigned a location that aims to minimize the disruption to the entire logic circuit. The key contribution of the work is that the synthesizer communicates to the placer the intended location of synthesized logic elements, allowing the placer to respond accordingly.

The approach proposed by Lou et al. [66] relies on the synthesis tool to aid the placer by providing several mapping solutions for each logic function. Based on the performance of the circuit, the placer may choose one of several mapping solutions to use during placement. Since it is easier to estimate circuit delay during placement, each mapping solution can be better examined and the best available mapping solution is used.

Physical synthesis has also received a lot of attention from both Altera and Xilinx, in their respective commercial tools. Altera's CAD tools include physical synthesis techniques such as the ones described by Singh et al. [98]. Post-technology mapping optimizations consider logic restructuring at a coarse granularity, but require accurate timing models. For timing critical paths, it is possible to attain reasonable delay estimates, because these paths have priority to use fast routing resources. However, for non-critical paths it is not always possible to predict the wiring delay. Once the post-technology mapping optimizations have been applied, post-placement optimization techniques in [98] can fine-tune the circuit implementation to provide a good final solution.

The commercial tools from Xilinx incorporate physical synthesis to improve speed performance, where the idea is to identify connections in the design that are on timing-critical paths not meeting user performance constraints [7, 6]. The design objects on this connections are incrementally re-placed and the design is incrementally re-routed. The incremental changes take considerably less time than a full placement and routing. If the new incrementally-generated solution has superior performance, it is accepted, otherwise, the tools revert back to the previously-observed best-performance solution.

It is also possible to improve the power consumption of the circuit implementation without "touching" the placement and routing solution. One idea has been to use the concept of SPFDs – *Sets of Pairs of Functions to be Distinguished,* which is an approach for computing the don't-cares in logic functions [51, 57]. Some LUT functions can be changed in ways that do not affect the circuit's functional correctness, yet may reduce toggling on the connections between LUTs, thereby reducing power. While SPFDs have been used to reduce power on signals between LUTs, the work in [45] shows how to reduce power *within* LUTs.

Leakage power reduction also can be reduced at the post-routing stage by recognizing that FPGA routing hardware consumes more leakage when signals are in the logic-0 state versus the logic-1 state[2]. The work in [11] inverts the logic functions implemented by LUTs so that the signals produced by the LUTs spend more time in the low-leakage (logic-1) state.

9 Future Trends

Research on power optimization has been vigorous in recent years and is likely to continue to be active, given technology scaling trends and the desire of the commercial vendors to broaden the usage of FPGAs in low-power mobile markets. CAD techniques for dealing with process variations and reliability, while well-studied for custom ICs, are largely unexplored for FPGAs and will be of rising importance in the future.

Looking forward, one way to expand the usage and market of programmable hardware is for FPGAs to be adopted by the software development community and used for computing applications. Today's FPGA CAD tools represent perhaps the most significant obstacle to achieving that goal. First, the run-time of the CAD tools is simply too long, taking hours or days for the largest circuits. More scalable CAD algorithms for FPGAs need to be developed. Run-times need to be brought closer to those required for compiling software programs, perhaps through parallelization of CAD algorithms to take advantage of multi-core CPUs. Or perhaps, new FPGA

[2] The dependence of leakage power on logic state has been exploited for power in the custom IC domain [47]

architectures specifically designed to allow fast tool run-time could be devised. Second, the complexity of the CAD tools is an insurmountable barrier for many software programmers. Needed are tools which allow designers to operate at a higher level of abstraction, writing code in variants of C or streaming languages. Without these, FPGAs may well lose out to other computing platforms coming onto the market today, such as Graphics Processing Units (GPUs) or many-core computers.

References

1. Continuous retiming: algorithms and applications. In *ICCD '97: Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, page 116, Washington, DC, USA, 1997. IEEE Computer Society.
2. *FLEX 10K Programmable Logic Device Datasheet*. Altera, Corp., San Jose, CA, 2003.
3. T. Ahmed, P. Kundarewich, J. Anderson, B. Taylor, and R Aggarwal. Architecture-specific packing for Virtex-5 FPGAs. In *ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays*, pages 5–13, Monterey, CA, 2008.
4. Michael J. Alexander and Gabriel Robins. New performance-driven fpga routing algorithms. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 562 – 567, 1995.
5. Altera, Corp., San Jose, CA. *Stratix-III FPGA Family Data Sheet*, 2008.
6. J.H. Anderson. Incremental placement of design objects in an integrated circuit design. *U.S. Patent #6,871,336*, 2005.
7. J.H. Anderson, S. Kalman, and V. Verma. Post-layout optimization in integrated circuit design. *U.S. Patent #7,111,268*, 2006.
8. J.H. Anderson, S. Nag, K. Chaudhary, S. Kalman, C. Madabhushi, and P. Cheng. Run-time-conscious automatic timing-driven FPGA layout synthesis. In *Int'l Conf. on Field-Programmable Logic and Applications*, pages 168–178, Antwerp, Belgium, 2004.
9. J.H. Anderson and F.N. Najm. Power-aware technology mapping for LUT-based FPGAs. In *IEEE International Conference on Field-Programmable Technology*, pages 211–218, Hong Kong, 2002.
10. J.H. Anderson and F.N. Najm. Power estimation techniques for FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(10):1015–1027, October 2004.
11. J.H. Anderson, F.N. Najm, and T. Tuan. Active leakage power optimization for FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 33–41, Monterey, CA, 2004.
12. J.H Anderson, J. Saunders, S. Nag, C. Madabhushi, and R. Jayaraman. A placement algorithm for fpga designs with multiple I/O standards. In *FPL*, pages 211–220, 2000.
13. R.L. Ashenhurst. The decomposition of switching functions. In *International Symposium on the Theory of Switching*, pages 74 – 116, 1959.
14. 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, Santa Clara, CA, 1997.
15. 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, London, UK, 1997.
16. Stephen Brown, Jonathan Rose, and Zvonko G. Vranesic. A detailed router for field-programmable gate arrays. *IEEE Trans. on CAD*, 11:620–628, 1992.
17. Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

18. L. Cabral, J. Aude, and N. Maculan. Tdr: A distributed-memory parallel routing algorithm for FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 227–240, 2002.
19. P.K. Chan and M.D.F. Schlag. Parallel placement for field-programmable gate arrays. In *ACM International Symposium on Field Programmable Gate Arrays*, Monterey, CA, 2003.
20. P.K. Chan, M.D.F. Schlag, C. Ebeling, and L. McMurchie. Distributed-memory parallel routing for field-programmable gate arrays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(8):850–862, Aug 2000.
21. D. Chen and J. Cong. Daomap: a depth-optimal area optimization mapping algorithm for fpga designs. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 752–759, 2004.
22. D. Chen and J. Cong. Delay optimal low-power circuit clustering for FPGAs with dual supply voltages. In *ACM/IEEE International Symposium on Low-Power Electronics and Design*, pages 70–73, Newport Beach, CA, 2004.
23. Lei Cheng, Deming Chen, and Martin D. F. Wong. Ddbdd: delay-driven bdd synthesis for fpgas. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 910–915, New York, NY, USA, 2007. ACM.
24. S.Y.L. Chin and S.J.E. Wilton. Memory footprint reduction for fpga routing algorithms. *Field-Programmable Technology, 2007. ICFPT 2007. International Conference on*, pages 1–8, Dec. 2007.
25. J. Cong and Y. Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in look-up-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, 1994.
26. J. Cong and Y. Ding. On area/depth trade-off in LUT-based FPGA technology mapping. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(2):137–148, 1994.
27. J. Cong, C. Wu, and E. Ding. Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 29–35, Monterey, CA, 1999.
28. Jason Cong and Yuzheng Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Transactions on Design Automation of Electronic Systems*, 1:145–204, 1996.
29. Jason Cong and Sung Kyu Lim. Physical planning with retiming. In *ICCAD '00: Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 2–7, Piscataway, NJ, USA, 2000. IEEE Press.
30. Altera Corp. Quartus university interface program. *http://www.altera.com/education/univ/research/unv-quip.html*, 2009.
31. H.A. Curtis. A new approach to the design of switching circuits. 1962.
32. T. S. Czajkowski and S. D. Brown. Functionally linear decomposition and synthesis of logic circuits for fpgas. In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 18–23, New York, NY, USA, 2008. ACM.
33. T.S. Czajkowski and S.D. Brown. Fast toggle rate computation for FPGA circuits. In *IEEE International Conference on Field Programmable Logic and Applications*, pages 65–70, Heidelberg, Germany, 2008.
34. J.A. Darringer, W.H. Joyner, C.L. Berman, and L. Trevillyan. Logic synthesis through local transformations. *IBM Journal of Research and Development*, 25(4):272 – 280, 1981.
35. M.E. Dehkordi and S.D. Brown. The effect of cluster packing and node duplication control in delay driven clustering. In *IEEE International Conference on Field-Programmable Technology*, pages 227–233, Hong Kong, 2002.
36. G. DeMicheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
37. E.W. Dijkstra. A note on two problems in connxion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
38. Hans Eisenmann and Frank M. Johannes. Generic global placement and floorplanning. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 269–274, 1998.

39. R.J. Francis, J. Rose, and K. Chung. Chortle: A technology mapping program for lookup table-based field programmable gate arrays. In *ACM/IEEE Design Automation Conference*, pages 613–619, Orlando, FL, 1990.
40. R.J Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *ACM/IEEE Design Automation Conference*, pages 227–233, San Francisco, CA, June 1991.
41. R.J Francis, J. Rose, and Z. Vranesic. Technology mapping for lookup table-based FPGAs for performance. In *IEEE International Conference on Computer-Aided Design*, pages 568–571, 1991.
42. J. Frankle. Iterative and adaptive slack allocation for performance-driven layout and FPGA routing. In *ACM/IEEE Design Automation Conference*, pages 536–542, 1992.
43. R. Fung, V. Betz, and W. Chow. Simultaneous short-path and long-path timing optimization for fpgas. *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 838–845, Nov. 2004.
44. G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1996.
45. S. Gupta, J. Anderson, L. Farragher, and Q. Wang. CAD techniques for power optimization in Virtex-5 FPGAs. In *IEEE Custom Integrated Circuits Conference*, pages 85–88, San Jose, CA, 2007.
46. G.D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Springer, 1996.
47. J.P. Halter and F.N. Najm. A gate-level leakage power reduction method for ultra-low-power CMOS circuits. In *IEEE Custom Integrated Circuits Conference*, pages 475–478, Santa Clara, CA, 1997.
48. H. Hassan, M. Anis, and M. Elmasry. Lap: A logic activity packing methodology for leakage power-tolerant FPGAs. In *ACM International Symposium on Low Power Electronics and Design*, pages 257–262, San Diego, CA, 2005.
49. R. B. Hitchcock, G. L. Smith, and D. D. Cheng. Timing analysis of computer hardware. *IBM Jour. of Research and Development*, 26(1):100–105, January 1982.
50. D. Howland and R. Tessier. RTL dynamic power optimization for FPGAs. In *IEEE Midwest Symposium on Circuits and Systems*, pages 714–717, Nashville, TN, 2008.
51. J-M. Hwang, F-Y. Chiang, and T-T. Hwang. A re-engineering approach to low power FPGA design using SPFD. In *ACM/IEEE Design Automation Conference*, pages 167–174, San Francisco, CA, 1998.
52. T.-T. Hwang, R.M. Owens, and M.J. Irwin. Exploiting communication complexity for multilevel logic synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(10):1017–1027, Oct 1990.
53. P. Jamieson and J. Rose. A verilog RTL synthesis tool for heterogeneous FPGAs. In *International Conference on Field Programmable Logic and Applications*, pages 305–310, Tampere, Finland, 2005.
54. Stephen Jang, Billy Chan, Kevin Chung, and Alan Mishchenko. Wiremap: Fpga technology mapping for improved routability. In *FPGA '08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 47–55, 2008.
55. T. Karnik and S.-M. Kang. An empirical model for accurate estimation of routing delay in FPGAs. In *IEEE International Conference on Computer-Aided Design*, pages 328–331, 1995.
56. A. Kennings, K. Vorwerk, A. Kundu, V. Pevzner, and A. Fox. FPGA technology mapping with encoded libraries and staged priority cuts. In *ACM International Symposium on Field-Programmable Gate Arrays*, 2009.
57. B. Kumthekar, L. Benini, E. Macii, and F. Somenzi. Power optimisation in FPGA-based design without rewiring. *IEE Proc. Comput. Digit. Tech.*, 147(3):167–174, May 2000.
58. Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula. Bdd based decomposition of logic functions with application to fpga synthesis. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 642–647, New York, NY, USA, 1993. ACM.

59. J. Lamoureux and S.J.E. Wilton. On the interaction between power-aware FPGA CAD algorithms. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 701–708, San Jose, CA, 2003.
60. J. Lamoureux and S.J.E. Wilton. Clock-aware placement for FPGAs. In *IEEE International Conference on Field-Programmable Logic and Applications*, pages 124–131, Amsterdam, The Netherlands, 2007.
61. Guy G. Lemieux and Stephen D. Brown. A detailed routing algorithm for allocating wire segments in field-programmable gate arrays. In *ACM/SIGDA Physical Design Workshop*, pages 215–226, 1993.
62. J. Lin, D. Chen, and J. Cong. Optimal simultaneous mapping and clustering for FPGA delay optimization. In *ACM/IEEE Design Automation Conference*, pages 472–477, San Francisco, CA, 2006.
63. J. Y. Lin, A. Jagannathan, and J. Cong. Placement-driven technology mapping for LUT-based fpgas. In *ACM International Symposium on Field-Programmable Gate Arrays*, pages 121–126, Monterey, CA, 2003.
64. A.C. Ling, D.P. Singh, and S.D. Brown. Fpga plb architecture evaluation and area optimization techniques using boolean satisfiability. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1196–1210, July 2007.
65. Andrew C. Ling, Jianwen Zhu, and Stephen D. Brown. Delay driven aig restructuring using slack budget management. In *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 163–166, 2008.
66. Jinan Lou, Wei Chen, and Massoud Pedram. Concurrent logic restructuring and placement for timing closure. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 31–36, Piscataway, NJ, USA, 1999. IEEE Press.
67. A. Ludwin, V. Betz, and K. Padalia. High-quality, determinstic parallel placement for FPGAs on commodity hardware. In *ACM/SIGDA Int'l Symposium on Field Programmable Gate Arrays*, pages 14–23, Monterey, CA, 2008.
68. Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Mark Fang, and Jonathan Rose. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages ?–?, 2009.
69. Wai-Kei Mak. I/o placement for fpgas with multiple i/o standards. In *FPGA '03: Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 51–57, 2003.
70. V. Manohararajah, S.D. Brown, and Z.G. Vranesic. Heuristics for area minimization in LUT-based FPGAs. In *International Workshop on Logic and Synthesis*, pages 14–21, 2004.
71. 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, Monterey, CA, 1999.
72. A. Marquardt, V. Betz, and J. Rose. Timing-driven placement for FPGAs. In *ACM International Symposium on Field-Programmable Gate Arrays*, pages 203–213, Monterey, CA, 2000.
73. L. McMurchie and C. Ebeling. Pathfinder: A negotiation-based performance-driven router for FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 111–117, Monterey, CA, 1995.
74. Paul Metzgen and Dominic Nancekievill. Multiplexer restructuring for fpga implementation cost reduction. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 421–426, 2005.
75. A. Mishchenko, Sungmin Cho, S. Chatterjee, and R. Brayton. Combinational and sequential mapping with priority cuts. *Computer-Aided Design, 2007. ICCAD 2007. IEEE/ACM International Conference on*, pages 354–361, Nov. 2007.
76. Alan Mishchenko, Michael Case, Robert Brayton, and Stephen Jang. Scalable and scalably-verifiable sequential synthesis. *Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on*, pages 234–241, Nov. 2008.

77. Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting: A fresh look at combinational logic synthesis. In *In DAC 06: Proceedings of the 43rd annual conference on Design automation*, pages 532–536. ACM Press, 2006.
78. J. Monteiro, J. Rinderknecht, S. Devadas, and A. Ghosh. Optimization of combinational and sequential logic circuits for low power using precomputation. In *ARVLSI '95: Proceedings of the 16th Conference on Advanced Research in VLSI (ARVLSI'95)*, page 430, Washington, DC, USA, 1995. IEEE Computer Society.
79. R. Murgai, R.K. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis for Field-Programmable Gate Arrays*. Kluwer Academic Publishers, 1995.
80. R. Nair. A simple yet effective technique for global wiring. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(2):165–172, March 1987.
81. F. Najm. Transition density: A new measure of activity in digital circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12:310–323, February 1993.
82. UC Berkeley Department of EECS. ABC – a system for sequential synthesis and verification. *http://www.eecs.berkeley.edu/∼alanmi/abc/*, 2009.
83. Marek A. Perkowski and Stanislaw Grygiel. A survey of literature on function decomposition. Technical report, Portland State University, 1995.
84. K.W. Poon, A. Yan, and S. J. E. Wilton. A flexible power model for FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 312–321, Montpellier, France, 2002.
85. PwrLite, Inc., Santa Clara, CA. *CooolGate RTL Synthesis*, 2009.
86. K. Roy. Power-dissipation driven FPGA place and route under timing constraints. *IEEE Transactions On Circuits and Systems*, 46(5):634–637, May 1999.
87. Y. Sankar and J. Rose. Trading quality for compile time: ultra-fast placement for FPGAs. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 157–166, 1999.
88. Sachin S. Sapatnekar and Rahul B. Deokar. Efficient retiming of large circuits. *IEEE Transactions on VLSI Systems*, 6:74–83, 1998.
89. T. Sasao. *Switching Theory for Logic Synthesis*. Kluwer Academic Publishers, 1999.
90. Tsutomu Sasao and Jon T. Butler. A design method for look-up table type fpga by pseudo-kronecker expansion. In *In Int'l Symp. on Multi-Valued Logic*, pages 97–106, 1994.
91. K. Schabas and S. D. Brown. Using logic duplication to improve performance in fpgas. In *ACM International Symposium on Field-Programmable Gate Arrays*, pages 136–142, Monterey, CA, 2003.
92. M. Schlag, J. Kong, and P.K. Chan. Routability-driven technology mapping for lookup table-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):13–26, 1994.
93. A. Sharma and S. Hauck. Accelerating FPGA routing using architecture-adaptive a* techniques. *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pages 225–232, Dec. 2005.
94. Narendra Shenoy and Richard Rudell. Efficient implementation of retiming. In *ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 226–233, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
95. A. Singh and M. Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 59–66, Monterey, CA, February 2002.
96. Deshanand P. Singh and Stephen D. Brown. Integrated retiming and placement for field programmable gate arrays. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 67–76, New York, NY, USA, 2002. ACM.
97. Deshanand P. Singh, Valavan Manohararajah, and Stephen D. Brown. Incremental retiming for fpga physical synthesis. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 433–438, New York, NY, USA, 2005. ACM.

98. D.P. Singh, V. Manohararajah, and S.D. Brown. Two-stage physical synthesis for fpgas. *Custom Integrated Circuits Conference, 2005. Proceedings of the IEEE 2005*, pages 171–178, Sept. 2005.
99. J. Swartz, V. Betz, and J. Rose. A fast routability-driven router for FPGAs. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 140–149, Monterey, CA, 1998.
100. R. Tessier, V. Betz, D. Neto, A. Egier, and T. Gopalsamy. Power-efficient RAM mapping algorithms for FPGA embedded memory blocks. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):278–290, Feb. 2007.
101. Chien-Chung Tsai and Malgorzata Marek-Sadowska. Multilevel logic synthesis for arithmetic functions. In *DAC '96: Proceedings of the 33rd annual conference on Design automation*, pages 242–247, New York, NY, USA, 1996. ACM.
102. Navin Vemuri, Priyank Kalla, and Russell Tessier. BDD-based logic synthesis for lut-based fpgas. *ACM Transasctions on Design Automation of Electronic Systems*, 7:501–525, 2002.
103. Verific Design Automation, Inc., Alameda, CA. *HDL Component Software*, 2009.
104. N. Viswanathan and C. C.-N. Chu. Fastplace: Efficient analytical placement using cell shifting, iterative local refinement and a hybrid net model. In *ACM/IEEE International Symposium on Physical Design*, pages 26–33, Phoenix, AZ, 2004.
105. N. Viswanathan, Min Pan, and C. Chu. Fastplace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. In *ASP-DAC '07: Proceedings of the 2007 conference on Asia South Pacific design automation*, pages 135–140, 2007.
106. K. Vorwerk, A. Kennings, and A. Vannelli. Engineering details of a stable force-directed placer. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 573–580, 2004.
107. K. Vorwerk, M. Rahman, J. Dunoyer, Y.-C. Hsu, A. Kundu, and A. Kennings. A technique for minimizing power during FPGA placement. In *IEEE International Conference on Field Programmable Logic and Applications*, pages 233–238, Heidelberg, Germany, 2008.
108. Wei Wan and Marek A. Perkowski. A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to fpga mapping. In *EURO-DAC '92: Proceedings of the conference on European design automation*, pages 230–235, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
109. Q. Wang, S. Gupta, and J. Anderson. Clock power reduction for Virtex-5 FPGAs. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages ?–?, 2009.
110. Congguang Yang and M. Ciesielski. Bds: a bdd-based logic optimization system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(7):866–876, Jul 2002.
111. Jian Zhang, Jinian Bian, and Qiang Zhou. Area and delay driven binding algorithm of rtl tech. mapping for heterogeneous fpgas. *Communications, Circuits and Systems, 2008. ICCCAS 2008. International Conference on*, pages 1231–1235, May 2008.