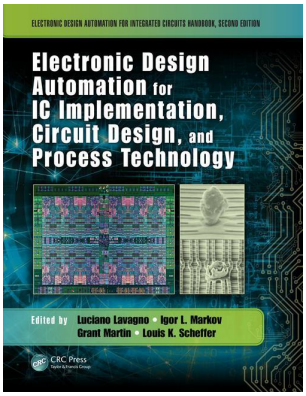

Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology

Luciano Lavagno, Igor L. Markov, Grant Martin, Louis K. Scheffer

Design Flows

Publication details
https://test.routledgehandbooks.com/doi/10.1201/b19714-3
David Chinnery, Leon Stok, David Hathaway, Kurt Keutzer
Published online on: 26 Apr 2016

1

Design Flows

David Chinnery, Leon Stok, David Hathaway, and Kurt Keutzer

CONTENTS

1.1 EVOLUTION OF DESIGN FLOWS

Scaling has driven digital integrated circuit (IC) implementation from a design flow that uses primarily stand-alone synthesis, placement, and routing algorithms to an integrated construction and analysis flow for design closure. This chapter will outline how the challenges of rising interconnect delay led to a new way of thinking about and integrating design closure tools (see Chapter 13). Scaling challenges such as power, routability, variability, reliability, ever-increasing design size, and increasing analysis complexity will keep on challenging the current state of the art in design closure.

A modern electronic design automation (EDA) flow starts with a high-level description of the design in a register-transfer-level (RTL) language, such as Verilog or VHDL, reducing design work by abstracting circuit implementation issues. Automated tools synthesize the RTL to logic gates from a standard cell library along with custom-designed macro cells, place the logic gates on a floor plan, and route the wires connecting them. The layout of the various diffusion, polysilicon, and metal layers composing the circuitry are specified in GDSII database format for fabrication.

The RTL-to-GDSII flow has undergone significant changes in the past 30 years. The continued scaling of CMOS technologies significantly changed the objectives of the various design steps. The lack of good predictors for delay has led to significant changes in recent design flows. In this chapter, we will describe what drove the design flow from a set of separate design steps to a fully integrated approach and what further changes we see are coming to address the latest challenges.

Similar to the eras of EDA identified by Alberto Sangiovanni-Vincentelli in "The Tides of EDA" [1], we distinguish three main eras in the development of the RTL-to-GDSII computer-aided design flow: the Age of Invention, the Age of Implementation, and the Age of Integration. During the invention era, logic synthesis, placement, routing, and static timing analysis were invented. In the age of implementation, they were drastically improved by designing sophisticated data structures and advanced algorithms. This allowed the software tools in each of these design steps to keep pace with the rapidly increasing design sizes. However, due to the lack of good predictive cost functions, it became impossible to execute a design flow by a set of discrete steps, no matter how efficiently implemented each of the steps was, requiring multiple iterations through the flow to close a design. This led to the age of integration where most of the design steps are performed in an integrated environment, driven by a set of incremental cost analyzers.

Let us look at each of the eras in more detail and describe some of their characteristics and changes to steps within the EDA design flow.

1.2 THE AGE OF INVENTION

In the early days, basic algorithms for routing, placement, timing analysis, and synthesis were *invented*. Most of the early invention in physical design algorithms was driven by package and board designs. Real estate was at a premium, and only a few routing layers were available and pins were limited. Relatively few discrete components needed to be placed. Optimal algorithms of high complexity were not a problem since we were dealing with few components.

In this era, basic partitioning, placement, and routing algorithms were invented. A fundamental step in the physical design flow is partitioning to subdivide a circuit into smaller portions to

simplify floor planning and placement. Minimizing the wires crossing between circuit partitions is important to allow focus on faster local optimizations within a partition, with fewer limiting constraints between partitions, and to minimize the use of limited global routing resources. In 1970, Kernighan and Lin [2] developed a minimum cut partitioning heuristic to divide a circuit into two equal sets of gates with the minimum number of nets crossing between the sets. Simulated annealing [3] algorithms were pioneered for placement and allowed for a wide range of optimization criteria to be deployed. Basic algorithms for channel, switch box, and maze routing [4] were invented. By taking advantage of restricted topologies and design sizes, optimal algorithms could be devised to deal with these particular situations.

1.3 THE AGE OF IMPLEMENTATION

With the advent of ICs, more and more focus shifted to design automation algorithms to deal with them rather than boards. Traditional CMOS scaling allowed the sizes of these designs to grow very rapidly. As design sizes grew, design tool implementation became extremely important to keep up with the increasingly larger designs and to keep design time under control. New implementations and data structures were pioneered and algorithms that scaled most efficiently became the standard.

As design sizes started to pick up, new layers of abstraction were invented. The invention of standard cells allowed one to separate the detailed physical implementation of the cells from the footprint image that is seen by the placement and routing algorithms. Large-scale application of routing, placement, and later synthesis algorithms took off with the introduction of the concept of standard cells.

The invention of the standard cell can be compared to the invention of the printing press. While manual book writing was known before, it was a labor-intensive process. Significant automation in the development of printing was enabled by keeping the height of the letters fixed and letting the width of the base of each of the letters vary according to the letter's size. Similarly, in standard cell application-specific IC (ASIC) design, one uses standard cells of common height but varying widths depending on the complexity of the single standard cell. These libraries (Chapter 12) created significant levels of standardization and enabled large degrees of automation. The invention of the first gate arrays took the standardization to an even higher level.

This standardization allowed the creation of the ASIC business model, which created a huge market opportunity for automation tools and spawned a number of innovations. Logic synthesis [5] was invented to bridge the gap from language descriptions to these standard cell implementations.

In the implementation era, a design flow could be pasted together from a sequence of discrete steps (see Figure 1.1). RTL logic synthesis translated a Verilog or VHDL description, performing technology-independent optimizations of the netlist and then mapping it into a netlist with technology gates, followed by further technology-dependent gate-level netlist optimization.

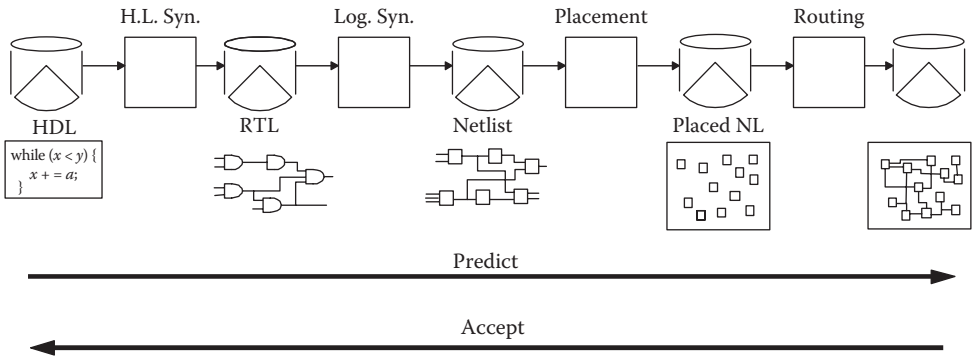

FIGURE 1.1 Sequential design flow.

This was followed by placement of the gates and routing to connect them together. Finally, a timing simulator was used to verify the timing using a limited amount of extracted wiring parasitic capacitance data.

Digital circuit sizes kept on increasing rapidly in this era, doubling in size every 2 years per Moore's law [6]. Logic synthesis has allowed rapid creation of netlists with millions of gates from high-level RTL designs. ICs have grown from 40,000 gates in 1984 to 40,000,000 gates in 2000 to billion gate designs in 2014.

New data structures like Quadtrees [7] and R-trees [8] allowed very efficient searches in the geometric space. Applications of Boolean Decision Diagrams [9] enabled efficient Boolean reasoning on significantly larger logic partitions.

Much progress was made in implementing partitioning algorithms. A more efficient version of Kernighan and Lin's partitioning algorithm was given by Fiduccia and Mattheyses [10]. They used a specific algorithm for selecting vertices to move across the cut that saved runtime and allowed for the handling of unbalanced partitions and nonuniform edge weights. An implementation using spectral methods [11] proved to be very effective for certain problems. Yang [12] demonstrated results that outperformed the two aforementioned methods by applying a network flow algorithm iteratively.

Optimizing quadratic wire length became the holy grail in placement. Quadratic algorithms took full advantage of this by deploying efficient quadratic optimization algorithms, intermixed with various types of partitioning schemes [13].

Original techniques in logic synthesis, such as kernel and cube factoring, were applied to small partitions of the network at a time. More efficient algorithms like global flow [14] and redundancy removal [15] based on test generation could be applied to much larger designs. Complete coverage of all timing paths by timing simulation became too impractical due to its exponential dependence on design size, and static timing analysis [16] based on early work in [17] was invented.

With larger designs came more routing layers, allowing over-the-cell routing and sharing of intracell and intercell routing areas. Gridded routing abstractions matched the standard cell templates well and became the base for much improvement in routing speed. Hierarchical routing abstractions such as global routing, switch box, and area routing were pioneered as effective ways to decouple the routing problem.

Algorithms that are applicable to large-scale designs without partitioning must have order of complexity less than $O(n^2)$ and preferably not more than $O(n \log n)$. These complexities were met using the aforementioned advances in data structures and algorithms, allowing design tools to handle large real problems. However, it became increasingly difficult to find appropriate cost functions for such algorithms. Accurate prediction of the physical effects earlier in the design flow became more difficult.

Let us discuss how the prediction of important design metrics evolved over time during the implementation era. In the beginning of the implementation era, most of the important design metrics such as area and delay were quite easy to predict. (Performance, power, and area are the corresponding key metrics for today's designs.) The optimization algorithms in each of the discrete design steps were guided by objective functions that relied on these predictions. As long as the final values of these metrics could be predicted with good accuracy, the RTL-to-GDSII flow could indeed be executed in a sequence of fairly independent steps. However, the prediction of important design metrics was becoming increasingly difficult. As we will see in the following sections, this led to fundamental changes in the design closure flow. The simple sequencing of design steps was not sufficient anymore.

Let us look at one class of the prediction functions, estimates of circuit delay, in more detail. In the early technologies, the delay along a path was dominated by the delay of the gates. In addition, the delay of most gates was very similar. As a result, as long as one knew how many gates there were on the critical path, one could reasonably predict the delay of the path by counting the number of levels of gates on a path and multiplying that with a typical gate delay. The delay of a circuit was therefore known as soon as logic synthesis had determined the number of logic levels on each path. In fact, in early timing optimization, multiple gate sizes were used to keep delays reasonably constant across different loadings rather than to actually improve the delay of a gate. Right after the mapping to technology-dependent standard cells, the area could be reasonably

TABLE 1.1 Gate Delays

Gate	Logical Effort	Intrinsic Delay	FO4 Delay
INV	1.00	1.00	5.00
NAND2	1.18	1.34	6.06
NAND3	1.39	2.01	7.57
NAND4	1.62	2.36	8.84
NOR2	1.54	1.83	7.99
NOR3	2.08	2.78	11.10
NOR4	2.63	3.53	14.05

well predicted by adding up the cell areas. Neither subsequent placement nor routing steps would change these two quantities significantly. Power and noise were not of very much concern in these times.

In newer libraries, the delays of gates with different logic complexities started to vary significantly. Table 1.1 shows the relative delays of different types of gates. The logical effort characteristic indicates how the delay of the gate increases with load, and the intrinsic delay is the load-independent contribution of the gate delay [18]. The fourth column of the table shows that the fanout-of-4 (FO4) delay of a more complex NOR4 logic gate can be three times the delay of a simple inverter. (The FO4 delay is the delay of an inverter driving a load capacitance that has four times the inverter's input pin capacitance. An FO4 delay is a very useful metric to measure gate delay as it is mostly independent of process technology and operating conditions. Static gates vary only 20% in FO4 delay over a variety of such conditions [19].)

Simple addition of logic levels is therefore insufficient to estimate path delay. Predicting the delay of a design with reasonable accuracy requires knowing what gates the logic is actually mapped to. It became necessary to include a static timing analysis engine (Chapter 6) in the synthesis system to calculate these delays. The combination of timing and synthesis was the first step on the way to the era of integration. This trend started gradually in the 1980s; but by the beginning of the 1990s, integrated static timing analysis tools were essential to predict delays accurately. Once a netlist was mapped to a particular technology and the gate loads could be approximated, a pretty accurate prediction of the delay could be made by the timing analyzer.

At that time, approximating the gate load was relatively easy. The load was dominated by the input capacitances of the gates that were driven. The fact that the capacitance of the wire was estimated by an inaccurate wire load model was hardly an issue. Therefore, as long as the netlist was not modified in the subsequent steps, the delay prediction was quite reliable.

Toward the mid-1990s, these predictions based on gate delays started losing accuracy. Gate delays became increasingly dependent on the load capacitance driven as well as on the rise and fall rates of the input signals (input slew) to the gates. At the same time, the fraction of loads due to wire capacitance started to increase. Knowledge of the physical design became essential to reasonably predict the delay of a path. Initially, it was mostly just the placement of the cells that was needed. The placement affected the delay, but the wire route had much less impact, since any route close to the minimum length would have similar load.

In newer technologies, more and more of the delay started to shift toward the interconnect. Both gate and wire (RC) delay really began to matter. Figure 1.2 shows how the gate delay and interconnect delay compare over a series of technology generations. With a Steiner tree approximation of the global routing, the lengths of the nets could be reasonably predicted. Using these lengths, delays could be calculated for the longer nets. The loads from the short nets were not very significant, and a guesstimate of these was still appropriate. Rapidly, it became clear that it was very important to buffer the long nets really well. In Figure 1.2, we see that around the 130 nm node, the difference between a net with repeaters inserted at the right places and an unbuffered net starts to have a significant impact on the delay. In automated place and route, buffering of long wires became an integral part of physical design. Today's standard approach for repeater (buffer or inverter) insertion on long wires is van Ginneken's dynamic programming buffering algorithm [20] and derivatives thereof. Van Ginneken's buffering algorithm has

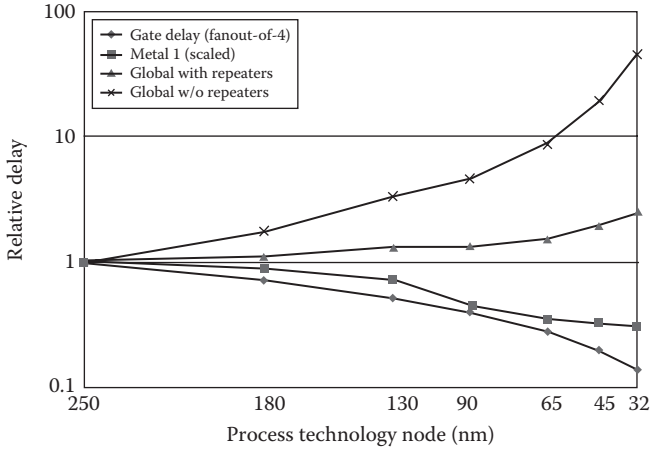

FIGURE 1.2 Gate and interconnect delay.

runtime complexity of $O(n^2)$, where n is the number of possible buffer positions, but this has been improved to near-linear [21].

Recently, a wire's physical environment has become more significant. Elmore RC wire delay models are no longer accurate, and distributed RC network models must be used, accounting for resistive shielding [22] and cross coupling. The cross-coupling capacitance between wires has increased as the ratio between wire spacing and wire height decreases. Furthermore, optical proximity effects when exposing the wafer vary depending on the distance between neighboring wires and the resulting impact when etching the wires also affects their resistance. Therefore, the actual routes for the wires from detailed routing are now very significant in predicting the delays. Moreover, global routing estimates of routing congestion can differ significantly from the actual routes taken, and global routing may mispredict where routing violations occur due to severe detailed routing congestion [23].

Netlist optimizations, traditionally done in logic synthesis, became an important part of place and route. Placement and routing systems that were designed to deal with static (not changing) netlists had to be reinvented.

Until recently, postroute optimizations were limited to minor changes that would cause minimal perturbation to avoid requiring a full reroute. These included footprint compatible cell swaps, particularly for critical path delay fixing by swapping cells to low threshold voltage, and leakage power minimization by swapping to high threshold voltage for paths with timing slack; manual reroutes to overcome severe congestion; manual engineering change orders (ECOs); and the use of metal programmable spare cells for fixes very late in the design cycle, avoiding the need to redo base layers. However, the gap between global routing estimates and detailed routing extracted capacitance and resistance have forced additional conservatism before detailed routing, for example, avoidance of small drive strength cells that will be badly affected by additional wire capacitance, leaving significant optimization opportunities in postroute.

1.4 THE AGE OF INTEGRATION

This decrease in predictability continued and firmly planted us in the age of *integration*. The following are some of the characteristics of this era:

- The impact of later design steps is increasing.
- Prediction is difficult.
- Larger designs allow less manual intervention.
- New cost functions are becoming more important.
- Design decisions interact.
- Aggressive designs allow less guardbanding.

The iterations between sequential design steps such as repeater insertion, gate sizing, and placement steps not only became cumbersome and slow but often did not even converge. EDA researchers and developers have explored several possible solutions to this convergence problem:

- Insert controls for later design steps into the design source.
- Fix problems at the end.
- Improve prediction.
- Concurrently design in different domains.

The insertion of controls proved to be very difficult. As illustrated in Figure 1.3, the path through which source modifications influence the final design result can be very indirect, and it can be very hard to understand the effect of particular controls with respect to a specific design and tools methodology. Controls inserted early in the design flow might have a very different effect than desired or anticipated on the final result. A late fix in the design flow requires an enormous increase in manual design effort. Improving predictions has proven to be very difficult. Gain-based delay models traded off area predictability for significant delay predictability and gave some temporary relief, but in general it has been extremely hard to improve predictions. The main lever seems to be concurrent design by integrating the synthesis, placement, and routing domains and coupling them with the appropriate design analyzers.

After timing/synthesis integration, placement-driven (physical) synthesis was the next major step on the integration path. Placement algorithms were added to the integrated static timing analysis and logic synthesis environments. Well-integrated physical synthesis systems became essential to tackle the design closure problems of the increasingly larger chips.

This integration trend is continuing. Gate-to-gate delay depends on the wire length (unknown during synthesis), the layer of the wire (determined during routing), the configuration of the neighboring wires (e.g., distance—near/far—which is unknown before detailed routing), and the signal arrival times and slope of signals on the neighboring wires. Therefore, in the latest technologies, we see that most of the routing needs to be completed to have a good handle on the timing of a design. Local congestion issues might force a significant number of routing detours. This needs to be accounted for and requires a significantly larger number of nets to be routed earlier for design flow convergence. Coupling between nets affects both noise and delay in larger portions of the design. Therefore, knowledge of the neighbors of a particular net is essential to carry out the analysis to the required level of detail and requires a significant portion of the local routing to be completed. In addition, power has become a very important design metric, and noise issues are starting to significantly affect delays and even make designs function incorrectly. In addition to integrated static timing analysis, power and noise analyses need to be included as well.

For example, white space postroute optimization to fix timing violations after routing is now in common use. Added cells and resized cells are placed at a legal location in available white space to avoid perturbing placement of the other cells and to minimize any changes to detailed routing

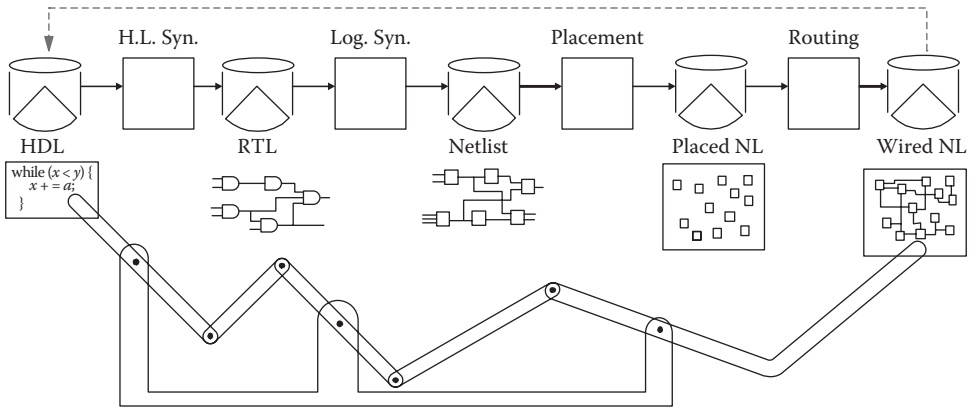

FIGURE 1.3 Controlled design flow.

between the other cells but allowing significant rerouting of nets connected to the modified cells. Larger cells, particularly those of multiple standard cell row heights, may still not be permitted to move in postroute optimization to limit the perturbation.

Consider the analysis of upsizing a cell to reduce short circuit power, which is determined by input slews and load capacitances, in postroute white space dynamic power optimization. This requires incremental evaluation of a new cell location with corresponding changes to detailed routing. The setup and hold timing impact is checked across multiple corners and modes with signal integrity cross talk analysis. Input slews to the cell are determined accounting for the impact on fanin load capacitance and the cell's output slew that affects its fanouts. The changed slews and extracted wire loads impact dynamic power consumption, and the tool must also ensure no degradation in total power, which includes both dynamic and leakage power in active mode. Such automated postroute optimizations can reduce cell area by up to 5%, helping yield and reducing fabrication cost; postroute dynamic power reductions of up to 12% have been achieved with Mentor Graphics' Olympus-SoC™ place-and-route tool [24], reducing total power consumption and extending battery life. Significant setup and hold timing violations can also be fixed in an automated fashion. These are remarkable improvements given how late in the design flow they are achieved, providing significant value to design teams.

These analysis and optimization algorithms need to work in an incremental fashion, because runtime constraints prevent us from recalculating all the analysis results when frequent design changes are made by the other algorithms. In the age of integration, not only are the individual algorithms important, but the way they are integrated to reach closure on the objectives of the design has become the differentiating factor. A fine-grained interaction between these algorithms, guided by fast incremental timing, power, and area calculators, has become essential.

In the era of integration, most progress has been made in the way the algorithms cooperate with each other. Most principles of the original synthesis, placement, and routing algorithms and their efficient implementations are still applicable, but their use in EDA tools has changed significantly. In other cases, the required incrementality has led to interesting new algorithms—for example, trialing different local optimizations with placement and global routing to pick the best viable solution that speeds up a timing critical path [25]. While focusing on the integration, we must retain our ability to focus on advanced problems in individual tool domains in order to address new problems posed by technology and to continue to advance the capabilities of design algorithms.

To achieve this, we have seen a shift to the development of EDA tools guided by four interrelated principles:

1. Tools must be integrated.
2. Tools must be modular.
3. Tools must operate incrementally.
4. Tools must sparsely access only the data that they need to reduce memory usage and runtime.

Let us look at each of these in more detail in the following sections.

1.4.1 TOOL INTEGRATION

Tool integration allows them to directly communicate with each other. A superficial form of integration can be provided by initiating, from a common user interface, the execution of traditional point tools that read input from files and save output to files. A tighter form of integration allows tools to communicate while concurrently executing rather than only through files. This tight integration is generally accomplished by building the tools on a common database (see Chapter 15) or through a standardized message passing protocol.

Tool integration enables the reuse of functions in different domains because the overhead of repeated file reading and writing is eliminated. This helps to reduce development resource requirements and improve consistency between applications. Although tool

integration enables incremental tool operation, it does not require it. For example, one could integrate placement and routing programs and still have the placement run to completion before starting routing. Careful design of an application can make it easier to integrate with other applications on a common database model, even if it was originally written as a stand-alone application.

Achieving tool integration requires an agreed-upon set of semantic elements in the design representation in terms of which the tools can communicate. These elements generally consist of cells, pins, and nets and their connectivity, cell placement locations, and wire routes. Cells include standard cells, macros, and hierarchical design blocks. Individual applications will augment this common data model with domain-specific information. For example, a static timing analysis tool will typically include delay and test edge data structures. In order for an integrated tool to accept queries in terms of the common data model elements, it must be able to efficiently find the components of the domain-specific data associated with these elements. Although this can be accomplished by name lookup, when integrated tools operate within a common memory space, it is more efficient to use direct pointers. This, in turn, requires that the common data model provide methods for applications to attach private pointers to the common model elements and callbacks to keep them consistent when there are updates to the elements (see Section 1.4.3.3).

1.4.2 MODULARITY

Modular tools are developed in small, independent pieces. This gives several benefits. It simplifies incremental development because new algorithms can more easily be substituted for old ones if the original implementation was modular. It facilitates reuse. Smaller modular utilities are easier to reuse, since they have fewer side effects. It simplifies code development and maintenance, making problems easier to isolate. Modules are easier to test independently. Tool modularity should be made visible to and usable by application engineers and sophisticated users, allowing them to integrate modular utilities through an extension language.

In the past, some projects have failed largely due to the lack of modularity [26]. To collect all behavior associated with the common data model in one place, they also concentrated control of what should be application-specific data. This made the data model too large and complicated and inhibited the tuning and reorganization of data structures by individual applications.

1.4.3 INCREMENTAL TOOLS

Tools that operate incrementally update design information, or the design itself, without revisiting or reprocessing the entire design. This enables fine-grained interaction between integrated tools. For example, incremental processing capability in static timing analysis helps logic synthesis to change a design and evaluate the effect of that change on timing, without requiring a complete timing analysis to be performed.

Incremental processing can reduce the time required to cycle between analysis and optimization tools. As a result, it can improve the frequency of tool interaction, allowing a better understanding of the consequences of each optimization decision.

The ordering of actions between a set of incremental applications is important. If a tool like synthesis invokes another incremental tool like timing analysis, it needs immediate access to the effects of its actions as reported by the invoked tool. Therefore, the incremental invocation must behave as if every incremental update occurs immediately after the event that precipitates it.

An example of incremental operation is fast what-if local timing analysis for a cell resize that can be quickly rolled back to the original design state and timing state if the change is detrimental. The local region of timing analysis may be limited to the cell, the cell's fanouts because of output slew impacting their delay, the cell's fanins as their load is affected, and the fanins' fanouts, as fanin slews changed with the change in load. Timing changes are not allowed to propagate beyond that region until the resize is committed or until global costing analysis is performed. Fully accurate analysis would require propagating arrival times to the timing endpoints and then

propagating back required times, which can be very runtime expensive. Even in a *full* incremental timing analysis update, only those affected paths are traversed and updated, and timing is not propagated further where the change is determined to have no impact, for example, if it remains a subcritical side path.

There are four basic characteristics desirable in an incremental tool:

1. Autonomy to avoid explicit invocation and control of other engines, for example, by registering callbacks so that an engine will be notified of events that impact it
2. Lazy evaluation to defer and minimize additional computation needed
3. Change notification to inform other engines of changes so that they may make associated updates where necessary
4. Reversibility to quickly undo changes, save, and restore

We shall explain these further in the following sections.

1.4.3.1 AUTONOMY

Autonomy means that applications initiating events that precipitate changes in another incremental tool engine do not need to notify explicitly the incremental tool of those events and that applications using results from an incremental tool do not need to initiate explicitly or control the incremental processing in that tool. Avoiding explicit change notification is important because it simplifies making changes to a design and eliminates the need to update the application when new incremental tools are added to the design tool environment. After registering callbacks, incremental applications can be notified of relevant events.

Avoiding explicit control of other incremental tools is important so that the caller does not need to understand the details of the incremental algorithm, facilitating future algorithmic improvements and making mistakes less likely. It also simplifies the use of the incremental tool by other applications.

1.4.3.2 LAZY EVALUATION (FULL AND PARTIAL)

Lazy evaluation means that an incremental tool should try, to the greatest extent possible, to defer processing until the results are needed. This can save considerable processing time when some results are never used. For example, consider a logic synthesis application making a series of individual disconnections and reconnections of pins to accomplish some logic transformation. If an incremental timing analyzer updates the timing results after each of these individual actions, it will end up recomputing time values many times, while only the last values computed are actually used.

Lazy evaluation simplifies the flow of control when recalculating values. If we have interdependent analysis functions that all try to update results when notified of a netlist change, then the callbacks would have to be ordered to ensure that updates occur in the correct order. For example, if timing updates are made before extraction updates, the timing results will be incorrect as stale extraction results are being used. With lazy evaluation, each application performs only invalidation when notified of a design change, then updates are ordered correctly through demand-driven recomputation. After a netlist change, when timing is requested, the timing engine requests extraction results before performing the delay computation. The extraction engine finds that the existing extraction results have been invalidated by the callbacks from the netlist change, so it updates those results, and only then is the timing analysis updated by the timing engine. An example of this is shown in Figure 1.4.

Lazy evaluation may be full, if all pending updates are performed as soon as any information is requested, or partial, if only those values needed to give the requested result are updated. If partial lazy evaluation is used, the application must still retain enough information to be able to determine which information has not yet been updated, since this information may be requested later. Partial lazy evaluation is employed in some timing analyzers [27] by levelizing the design and limiting the propagation of arrival and required times based on this levelization, providing significant benefits in the runtime of the tool.

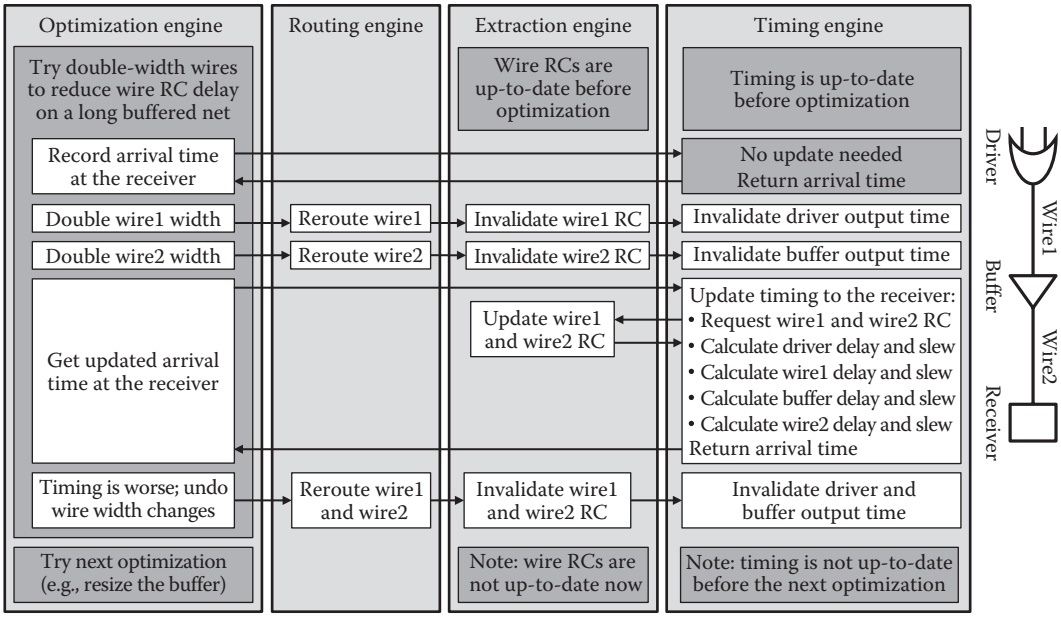

FIGURE 1.4 This example shows change notification callbacks from the routing engine to the extraction and timing analysis engines. There are lazy updates to the wire RC extraction and the timing analysis upon request from the timing engine and optimization engine, respectively.

1.4.3.3 CHANGE NOTIFICATION (CALLBACKS AND UNDIRECTED QUERIES)

With change notification, the incremental tool notifies other applications of changes that concern them. This is more than just providing a means to query specific pieces of information from the incremental tool, since the application needing the information may not be aware of all changes that have occurred. In the simplest situations, a tool initiating a design change can assume that it knows where consequent changes to analysis results (e.g., timing) will occur. In this case, no change notification is required. But in other situations, a tool may need to respond not only to direct changes in the semantic elements of the common data model but also to secondary changes within specific application domains (e.g., changes in timing).

Change notification is important because the applications may not know where to find all the incremental changes that affect them. For example, consider an incremental placement tool used by logic synthesis. Logic synthesis might be able to determine the blocks that will be directly replaced as a consequence of some logic change. But if the replacement of these blocks has a ripple effect, which causes the replacement of other blocks (e.g., to open spaces for the first set of replaced blocks), it would be much more difficult for logic synthesis to determine which blocks are in this second set. Without change notification, the logic synthesis system would need to examine all blocks in the design before and after every transformation to ensure that it has accounted for all consequences of that transformation.

Change notification may occur immediately after the precipitating event or may be deferred until requested. Immediate change notification can be provided through callback routines that applications can register with and that are called whenever certain design changes occur.

Deferred change notification requires the incremental tool to accumulate change information until a requesting application is ready for it. This requesting application will issue an undirected query to ask for all changes of a particular type that have occurred since some checkpoint (the same sort of checkpoint required for undoing design changes). It is particularly important for analysis results, since an evaluation routine may be interested only in the cumulative effect of a series of changes and may neither need nor want to pay the price (in nonlazy evaluation) of immediate change notification.

A typical use of undirected queries is to get information about changes that have occurred in the design, in order to decide whether or not to reverse the actions that caused the changes.

For this purpose, information is needed not only about the resultant state of the design but also about the original state so that the delta may be determined. (Did things get better or worse?) Thus, the application making an undirected query needs to specify the starting point from which changes will be measured. This should use the same checkpoint capability used to provide reversibility.

1.4.3.4 REVERSIBILITY (SAVE/RESTORE AND RECOMPUTE)

Trial evaluation of candidate changes is important because of the many subtle effects any particular design change may cause and because of the order of complexity of most design problems. An application makes a trial design change, examines the effects of that change as determined in part by the incremental tools with which it is integrated, and then decides whether to accept or reject the change. If such a change is rejected, we need to make sure that we can accurately recover the previous design state, which means that all incremental tool results must be reversible.

Each incremental tool should handle the reversal of changes to the data for which it is responsible. In some cases such as timing analysis, the changed data (e.g., arrival and required times) can be deterministically derived from other design data, and it may be more efficient to recompute them rather than to store and recover them. In other cases such as incremental placement, changes involve heuristics that are not reversible, and previous state data must be saved, for example, in a local stack. The incremental tool that handles the reversal of changes should be the one actually responsible for storing affected data. Thus, changes to the netlist initiated by logic synthesis should be reversed by the data model (or an independent layer built on top of it) and not by logic synthesis itself.

All such model changes should be coordinated through a central undo facility. Such a facility allows an application to set checkpoints to which it could return. Applications, which might have information to undo, register callbacks with the facility to be called when such a checkpoint is established or when a request is made to revert to a previous checkpoint.

Ordering of callbacks to various applications to undo changes requires care. One approach is to examine the dependencies between incremental applications and decide on an overall ordering that would eliminate conflicts. A simpler solution is to ensure that each atomic change can be reversed and to have the central undo facility call for the reversal of all changes in the opposite order from that in which they were originally made.

Applications can undo changes in their data in one of two ways, as outlined earlier. Save/restore applications (e.g., placement) may merely store the previous state and restore it without requiring calls to other applications. Recompute applications (e.g., timing analysis) may recompute previous state data that can be uniquely determined from the state of the model. Recompute applications generally do not have to register specific undo callbacks, but to allow them to operate, model change callbacks (and all other change notification callbacks) must be issued for the reversal of model changes just as they are for the original changes.

Such a central undo facility can also be used to help capture model change information. This can be either to save to a file, for example, an ECO file, or to transmit to a concurrently executing parallel process, on the same machine or another one, with which the current applications need to synchronize.

Even when we choose to keep the results of a change, we may need to undo it and then redo it. A typical incremental processing environment might first identify a timing problem area and then try and evaluate several alternative transformations. The original model state must be restored before each alternative is tried, and after all alternatives have been evaluated, the one that gives the best result is then reapplied.

To make sure that we get the same result when we redo a transformation that we got when we did it originally, we also need to be able to reverse an undo. Even for deterministic transformations, the exact result may depend on the order in which certain objects are visited, and such orderings may not be semantic invariants and thus may be altered when undoing a change. Also, a considerable amount of analysis may be required by some transformations to determine the exact changes that are to be made, and we would like to skip this analysis when we redo the transformation. Avoiding nondeterministic behavior is also highly desirable; otherwise, reproducing faulty behavior during debug is fraught with complications.

For these reasons, the central undo/ECO facility should be able to store and manage a tree of checkpoints rather than just a single checkpoint. Trial transformations to a circuit may also be nested, so we need to be able to stack multiple checkpoints.

It is important to remember that there is no magic bullet that will turn a nonincremental tool into an incremental one. In addition to having a common infrastructure, including such things as a common data model and a callback mechanism, appropriate incremental algorithms need to be developed.

Following some of these guidelines also encourages incremental design automation tool development. Rewriting tools is an expensive proposition. Few new ideas change all aspects of a tool. Incremental development allows more stability in the tool interface, which is particularly important for integrated applications. It also allows new ideas to be implemented and evaluated more cheaply, and it can make a required function available more quickly.

1.4.4 SPARSE ACCESS

Sparse access refers to only loading the portion of the design and associated files that are necessary to perform a given task and deferring loading additional data until needed. This can reduce both memory usage and runtime by an order of magnitude in some cases.

For example, a design check to verify correct connectivity to sleep header cells for power gating needs to only traverse the cell pin and net connections and may avoid loading libraries if cell types are identified by a naming convention. The netlist portion traversed might be further limited to just the nets connecting to the sleep header cells and the cells driving the sleep control signal, ensuring that logic is connected to the always-on power supply.

Sparse access to netlist or parasitic data typically requires paging memory to disk with an index to specify where to find data for a given cell or net. Likewise, cells within a library can be indexed to provide individual access rather than loading the entire library, which can reduce the memory overhead for loading composite current source (CCS) libraries by an order of magnitude or more as only a few hundred cells are used out of several thousand. This may require precharacterization of libraries, for example, to provide a cached list of which delay buffers are available to fix hold violations and the subset of those that are Pareto optimal in terms of area or power versus delay trade-off.

1.4.4.1 MONOLITHIC EDA TOOLS AND MEMORY LIMITS

In-house tools at design companies are often fast point tools that load only the necessary portion of the design and associated technology files or libraries to minimize the start-up runtime overhead, as the task being performed may run in a minute to an hour. This can be illustrated by the example of running tens of design checks across all the design blocks in parallel on a server farm on a nightly basis. Such point tools are very useful to analyze and check designs for any significant issues before launching further runs that will take significant time.

In contrast, commercial EDA tools have typically been designed for use in a monolithic manner. One or more major flow steps would be performed within the tool, such as synthesis, placement, and optimization; clock tree synthesis (CTS); post-CTS optimization; detailed routing; and postroute optimization. The entire design, technology files, and libraries are all loaded taking significant memory. The initial loading runtime of several minutes is smaller compared to that of the larger flow step being performed.

Commercial EDA tools are now facing tight memory limits when analyzing and optimizing large designs across multiple processes and operating corners. A combination of 30 corners and modes or more is not uncommon—for example, slow, typical, and fast process corners; low temperature, room temperature, and high temperature; –5% voltage, typical voltage, and +5% voltage; supply voltage modes such as 0.7, 0.9, and 1.1 V; and asleep, standby, awake, and scan operating modes. CCS libraries can be 10 GB each, so just loading the libraries for all these corners can be 100 GB or more of memory. Today's designs are typically anywhere between blocks of several hundred thousand gates to hundreds of millions of gates when analyzed in a flat manner at the top level and can also take 100 GB or more of memory, particularly with standard parasitic

exchange format (SPEF)—annotated parasitic capacitances. When running multithreaded across a high-end 16 CPU core server to reduce runtimes that would otherwise take days, some data must also be replicated to run multithreaded to avoid memory read/write collisions and to limit pauses needed to synchronize data. Today's server farms only have a few machines with 256GB of memory, and servers with higher memory than that are very expensive.

EDA developers use a variety of standard programming techniques to reduce memory usage. For example, a vector data structure takes less memory than a linked list (to store the same data), whereas multiple Boolean variables may be packed as bits in a single CPU word and extracted with bitmasks when needed. Duplicated data can be compressed, for example, during layout versus schematic verification [28] where hierarchical representations of layout data represent individual library cells containing many repeated geometric shapes. Memory usage, runtime, and quality of results, such as circuit timing, area, and power, are carefully monitored across regression suites to ensure that code changes in the EDA tool do not degrade results or tool performance.

1.4.4.2 PARALLEL AND DISTRIBUTED COMPUTING FOR EDA

EDA vendors are migrating tools to work in a distributed manner with a controller on one machine distributing task to workers on other machines to perform in parallel. Multithreading and fork–join [28] are also common parallelism techniques used in EDA software, and they can be combined with distributed computation. These methods can reduce runtime for analysis, optimization, and verification by an order of magnitude. Shared resources can also be more effectively used with appropriate queuing.

Many of the algorithms used in the RTL-to-GDSII flow are graph algorithms, branch-and-bound search, or linear algebra matrix computations [29]. While most of these algorithms can be parallelized, Amdahl's law [30] limits the speedup due to the serial portion of the computation and overheads for communication, cache coherency, and bottlenecks where synchronization between processes is needed. Performance improvements with parallelism are also often overstated [31].

Much of the development time in parallelizing software actually focuses on reducing the runtime of the serial portion, for example, avoiding mutually exclusive locks on shared resources to minimize stalls where one process has to wait for another to complete. Fast and memory-efficient implementations of serial algorithms, such as Dijkstra's shortest-path algorithm [32], are still very important in EDA as there is often no better parallel algorithm [31].

As an example of how distributed computing is used in EDA, optimization at several important timing corners may require a server with a large amount of memory, but lower-memory servers can verify that timing violations do not occur at other corners. Analysis at dominant corners causing the majority of setup and hold timing violations reduces the need for analysis at other corners. However, there are still cases where a fix at a dominant corner can cause a violation at another corner, so it cannot be entirely avoided. The controller and the workers cannot afford the memory overhead of loading libraries for all the corners, nor the additional analysis runtime. Distributed optimization allows staying within the server memory limits. One worker performs optimization with analysis at dominant corners only, with other workers verifying that a set of changes do not violate constraints at the other corners.

Partitioning large designs into smaller portions that can be processed separately is a common approach to enabling parallelism [29]. This may be done with min-cut partitioning [12] to minimize cross-border constraints where there will be suboptimality. However, some regions may be overly large due to reconvergent timing paths, latch-based timing, cross coupling between wires, and other factors that limit a simple min-cut partitioning approach. The regions need to be sized so as to balance loads between workers. How best to subpartition a design is a critical problem in EDA [29] and varies by application. For example, timing optimization may consider just critical timing paths, whereas area optimization may try to resize every cell that is not fixed.

Portions of a design can be rapidly optimized in parallel in a distributed manner. Border constraints imposed to ensure consistency between workers' changes to the design do reduce the optimality. Individual workers use sparse access to just read the portion of the design that they are optimizing, though logical connectivity may not be sufficient as cross-coupling analysis considers physically nearby wires.

Parallel computation can be a source of nondeterminism when multiple changes are performed in parallel. To avoid this nondeterminism, incremental analysis on distributed workers may need to use a consistent snapshot of the circuit with periodic synchronizations points at which changes and analysis updates are done or undone in a fixed deterministic order. Some operations may also need to be serial to ensure consistent behavior.

1.5 FUTURE SCALING CHALLENGES

In the previous sections, we mainly focused on how continued scaling changed the way we are able to predict delay throughout the design flow. This has been one of the main drivers of the design flow changes over the past two decades. However, new challenges require rethinking the way we automate the design flow. In the following sections, we will describe some of these in more detail and argue that they are leading to a design closure that requires an integrated, incremental analysis of routability, power, noise, and variability (with required incremental extraction tools) in addition to the well-established incremental timing analysis.

1.5.1 DYNAMIC AND LEAKAGE POWER

Tools have traditionally focused on minimizing both critical path delay and circuit area. As technology dimensions have scaled down, the density of transistors has increased, mitigating the constraints on area, but increasing the power density (power dissipated per unit area). Heat dissipation limits the maximum chip power, which, in turn, limits switching frequency and hence how fast a chip can run. By 90 nm technology, even some high-end microprocessor designers found power to be a major constraint on performance [33]. Chapter 3 provides further detail on power analysis and optimization beyond the discussion in this subsection.

The price of increasing circuit speed with Moore's law has been an even faster increase in dynamic power. The dynamic power due to switching a capacitance C with supply voltage V_{dd} is $fCV^2_{dd}/2$. Increasing circuit speed increases switching frequency f proportionally, and capacitance per unit area also increases.

Transistor capacitance varies inversely to transistor gate oxide thickness t_{ox} ($C_{gate} = \varepsilon_{ox}WL/t_{ox}$, where ε_{ox} is the gate dielectric permittivity and W and L are transistor width and length). Gate oxide thickness has been scaled down linearly with transistor length so as to limit short channel effects and maintain gate drive strength to increase circuit speed [34]. As device dimensions scale down linearly, transistor density increases quadratically and the capacitance per unit area increases linearly. Additionally, wire capacitance has increased relative to gate capacitance, as wires are spaced closer together with taller aspect ratios to limit wire resistance and thus to limit corresponding wire RC delays.

If the supply voltage is kept constant, the dynamic power per unit area increases slower than quadratically due to increasing switching frequency and increasing circuit capacitance. To reduce dynamic power, supply voltage has been scaled down. However, this reduces the drive current, which reduces the circuit speed. The saturation drive current $I_{D,sat} = kW(V_{dd} - 2V_{th})^a / Lt_{ox}$, where V_{th} is the transistor threshold voltage and the exponent a is between 1.2 and 1.3 for recent technologies [35]. To avoid reducing speed, threshold voltage has been scaled down with supply voltage.

Static power has increased with reductions in transistor threshold voltage and gate oxide thickness. The subthreshold leakage current, which flows when the transistor gate-to-source voltage is below V_{th} and the transistor is nominally off, depends exponentially on V_{th} ($I_{subthreshold} = ke^{-qV_{th}/nkT}$, where T is the temperature and n, q, and k are constants). As the gate oxide becomes very thin, electrons have a nonzero probability of quantum tunneling through the thin gate oxide. While gate tunneling current was smaller by orders of magnitudes than subthreshold leakage, it was increasing much faster due to the reduction in gate oxide thickness [36].

Automated logic synthesis and place-and-route tools have focused primarily on dynamic power when logic evaluates within the circuit, as static power was a minimal portion of the total power when a circuit was active. For example, automated clock gating reduces unnecessary

switching of logic. Managing standby static power was left for the designers to deal with by using methods such as powering down modules that are not in use or choosing standard cell libraries with lower leakage.

1.5.1.1 LEAKAGE POWER

Standby power may be reduced by using high threshold voltage sleep transistors to connect to the power rails [37]. In standby, these transistors are switched off to stop the subthreshold leakage path from supply to ground rails. This technique is known as power gating. When the circuit is active, these sleep transistors are on, and they must be sized sufficiently wide to cause only a small drop in voltage swing, but not so wide as to consume excessive power. To support this technique, place-and-route software must support connection to the *virtual* power rail provided by the sleep transistors and clustering of cells that enter standby at the same time so that they can share a common sleep transistor to reduce overheads.

In recent 45–28 nm process technologies, leakage can contribute up to about 40% of the total power when the circuit is active. There is a trade-off between dynamic power and leakage power. Reducing threshold voltage and gate oxide thickness allows the same drive current to be achieved with narrower transistors, with correspondingly lower capacitance and reduced dynamic power, but this increases leakage power. Increasing transistor channel length provides an alternate way to reduce leakage at the cost of increased gate delay, with somewhat higher gate capacitance and thus higher dynamic power. Alternate channel lengths are cheaper as alternate threshold voltages require separate implant masks that increase fabrication expense.

Designers now commonly use standard cell libraries with multiple threshold voltages and multiple channel lengths, for example, a combination of six threshold voltage and channel length alternatives: low/nominal/high V_{th}, with regular and +2 nm channel lengths. Low threshold voltage and shorter channel length transistors reduce the delay of critical paths. High threshold voltage and longer channel length transistors reduce the leakage power of gates that have slack. EDA tools must support a choice between cells with smaller and larger leakage as appropriate for delay requirements on the gate within the circuit context.

Significant improvements to leakage power minimization have been achieved in the last couple of years using fast global optimization approaches such as Lagrangian relaxation. This has made it possible to optimize a million-gate netlist in an hour [38], and significant further speedups may be achieved by multithreading.

It is essential that tools treat leakage power on equal terms with dynamic power when trying to minimize the total power. Preferentially optimizing leakage power or dynamic power is detrimental to the total power consumption in active mode. Other operating corners and modes must also be considered during power optimization, as there are also design limits imposed on standby power consumption as not all gates can be power-gated off.

1.5.1.2 DYNAMIC POWER

Leakage power has been significantly reduced with multigated devices, such as triple-gated FinFETs [39]. With 22–14 nm FinFET process technologies, we have seen that leakage power contributes 10%–30% of the total power when the circuit is active, depending on switching activity and threshold voltage choices. So, designers have increased the use of dynamic power minimization techniques.

There has been significant development in industry and academia on register clumping and swapping of multibit flops to reduce the clock load and clock power. Other research has focused on optimizing the clock trees to trade-off latency, clock skew, and clock power. Clock skews in the range of 30–50 ps can be achieved with multisource CTS and in the range of 10–30 ps with automated clock mesh synthesis [40], providing further opportunity for designs to achieve higher performance with trade-offs between clock skew and clock power.

Some industry designs have used fine-grained voltage islands to reduce power consumption [41], though this is yet to be supported by EDA vendor tools due to the complicated design methodology and limited market thus far.

There is further opportunity to automatically optimize wire width and wire spacing in conjunction with gate sizing/V_{th} assignment and buffering to optimize delay versus power consumption. To reduce the dynamic power of high-activity wires, they can be preferentially routed to shorten them, and wider spacing can be used to reduce their coupling capacitance to neighbors. Downsizing a gate reduces the switching power, but in some cases, it may be preferential to upsize the gate or insert a buffer to reduce slew and short circuit power. Vendor EDA tools still perform little optimization of wire spacing or wire width, overlooking trade-offs between (1) increased wire capacitance and (2) reduced wire resistance and reduced RC delay. Nondefault rules (NDRs) for wire width and spacing are typically manually provided by designers, with some limited tool support for choosing between the available NDRs.

Techniques to reduce peak power and glitching, for both peak and average power reductions, are also of interest to designers with little automation support as yet. Glitching can be reduced by insertion of deglitch latches or by balancing path delays, but dynamic timing analysis is runtime expensive, so simplified fast approaches are needed. Logic could also be remapped to better balance path delays or reduce switching activity within the logic, but remapping is quite runtime expensive.

1.5.2 PLACEMENT AND ROUTABILITY WITH COMPLEX DESIGN RULES

Multiple patterning to fabricate smaller feature sizes has added significant design rule complexity that has complicated both placement and routing. The design rules are more complex due to interaction between features on the same layer that must be on different masks to achieve higher resolution and due to potential misalignment of the multipatterning masks. For further details on design rules, please see Chapter 20.

The wavelength of the lithography light source imposes a classical Rayleigh limit on the smallest linewidth of features that can be resolved in IC fabrication, as detailed in Chapter 21, which provides a deeper discussion on multiple patterning. Resolution beyond 65 nm process technologies is limited by the 193 nm wavelength provided by argon fluoride lasers. To scale further, either immersion lithography or double patterning is needed for 45 nm devices, and both immersion lithography and double patterning are required for technologies from 32 to 14 nm [42]. Triple patterning will first be used for 10 nm process technology [43]. Multipatterning has additional fabrication costs for the additional masks to expose layers requiring smaller resolutions. Higher metal layers, which are wider, are more cheaply added without multipatterning or immersion lithography.

Other solutions for fabricating smaller features have not yet been practical for large-scale production. Extreme ultraviolet (EUV) lithography with 13.5 nm wavelength holds the promise of being able to print much smaller transistors at lower cost with simpler design rules [44]. However, the introduction of EUV has been further delayed to at least 7 nm process technology due to a variety of issues, in particular difficulty in achieving a 100 W power source to provide good exposure throughput [45]. Recently, record throughput of 1022 wafers in 24 hours at Taiwan Semiconductor Manufacturing Company and 110 W EUV power capability has been reported by ASML [46]. Electron beams can be used to write small patterns but are far too slow to be practical, except that they have been used extensively in testing fabricated ICs [47]. Double patterning may be needed with EUV at 7 nm and such resolution enhancement techniques will be required with EUV for 5 nm technology [43]. Consequently, foundries are committed to the use of multipatterning to fabricate smaller process technologies, and they are preparing alternative solutions such as self-aligned quadruple patterning in case EUV is not ready in time for 7 nm process technology. The multipatterning design-rule constraints for placement and routing are now major issues.

Since 32nm, horizontal edge-type placement restrictions [48] have been required to improve yield and for design for manufacturability. For example, if the layout shapes of polysilicon on adjacent cell edges will be detrimental to yield, spacing may be required to allow the insertion of dummy poly to minimize the optical interference between the adjacent cells [49]. This results in different horizontal edge-spacing restrictions depending on the layout of the cells.

In 10 nm process technology, we now also see vertical edge abutment restrictions. Such restrictions are introduced to comply with more conservative design rules, for example, to provide

sufficient space for nearby vias to connect to cell pins or the power/ground metal stripes. Another objective is to prevent conflict between color choices for wires within cells as adjacent wires may not have the same coloring assignment. (In the context of multipatterning, coloring refers to which pattern a portion of routing is assigned to.) Vertical restrictions are more complicated, specifying distance ranges so that portions of cells may not vertically abut. Early tool support has been provided for these vertical restrictions, but further updates are needed to placers and routers to account for these better during optimization. The Library Exchange Format (LEF) [48] also needs to be updated to add specification of these vertical placement constraints.

In global placement, temporarily increasing the size of (bloating) the cells with an estimate of the additional spacing needed can help account for edge-type placement restrictions and reduce routing congestion [50,51]. Routing congestion is increased by routing blockages, routing NDRs, and preroutes such as the power, ground, and clock grids. Detailed placement must also consider these during placement legalization, as cell pins may be inaccessible to connect to if they are next to or under a preroute, or due to the additional spacing needed for an NDR [51].

With increasing design rule complexity for multipatterning, patterns must be more regular and may be restricted to unidirectional segments at each layer to improve manufacturability [52], disallowing wrong-way routing and requiring more vias, adding to routing congestion. Where wrong-way routing is allowed, there are different design rules with much wider spacing constraints between wires for the nonpreferred direction. Routers have been updated to support these more complex design rules, and both the resistance and the capacitance of vias must also now be considered for accurate parasitic extraction.

There are also rules to insert dummy metal fill between cells to reduce variation in dielectric thickness, but metal fill adds to wire coupling capacitance and thus increases wire delay [53]. Metal fill adds the complication that downsizing a cell can introduce a spacing violation as a gap may be introduced between cells with insufficient space for metal fill. Similarly, a downsized cell may have different edge types, due to different layouts within the cell, which can introduce a placement violation with neighboring cells if the different edge type requires wider spacing.

When performing multiple rounds of patterning (usually two or three), the shapes for each round are identified by *color*. There is different wire coupling capacitance in the same metal layer with the *color choice* due to misalignment error between the masks of different colors for multipatterning [54]. Depending on the foundry and fabrication process, some EDA flows preassign routing tracks a given color, whereas other design flows are colorless with the foundry coloring the provided GDSII layout. When wire color is not determined, as in a colorless design flow, it can be considered in timing analysis by adding two additional minimum and maximum corners for wire resistance and capacitance extraction [55]. The color assignment for wires within cells affects the cell delay characteristics, so EDA tools may select different cell implementations that differ only in the color assignment of wires within the cells, if standard cell libraries provide the different coloring alternatives. Global and detailed routers also need to be aware of wire coloring, either conservatively estimating capacitance and resistance for the worst-case color choices or preferably optimizing the wire color choice based on timing criticality.

Some spacing restrictions do not need to be strict but are there to improve yield, so they may be satisfied where opportunity permits to do so without degrading critical path delay and hence timing yield. Today's EDA tools strictly enforce spacing restrictions. Design groups have built fast incremental tools to provide this capability using internal database implementations with sparse access or scripted slower TCL implementations in vendor tools [56]. There remain further opportunities for native EDA tool support of yield analysis and optimization during placement.

1.5.3 VARIABILITY IN PROCESS AND OPERATING CONDITIONS

Historically, it was assumed that static timing analysis at one slow corner and one fast corner was sufficient to ensure that timing constraints were met. Worst-case circuit delay was estimated from the slow corner for the standard cell library: This was the slow process corner, with a slow operating corner with lower supply voltage (e.g., –10% V_{dd}) and high temperature (e.g., 100°C). Hold-time violations by fast timing paths and worst-case dynamic power were estimated from the fast process corner, with a fast operating corner with higher supply voltage (e.g., +10% V_{dd}) and

low temperature (e.g., 0°C). The slow (fast) process corner would be for both p-type and n-type MOSFETs due to higher (lower) threshold voltages and longer (shorter) channel lengths due to process variability, which also reduces (increases) subthreshold leakage current. Worst-case leakage power can be estimated at the fast process and high-temperature corner.

In practice, a mix of *slow* and *fast* process variations and differences in spot temperatures on a chip may lead to unforeseen failures not predicted by a single process corner. Also in 90 nm and smaller process technologies, inverted temperature dependence of delay has added an additional slower operating corner at lower temperature and further complications for static timing analysis [57]. Temperature inversion is due to the competing temperature dependence of drain current with carrier mobility and threshold voltage. The impact of mobility is more significant at higher supply voltage, and the impact of threshold voltage is more significant at lower supply voltage. Mobility decreases with increasing temperature that reduces drain current slowing devices. Threshold voltage decreases with increasing temperature, making the circuit faster. Due to these competing effects, the path delay may no longer monotonically increase with temperature and may instead first decrease and then increase with temperature [57].

Both optimization and timing sign-off with a combination of many corners and modes have become common today. Additionally, the majority of chips fabricated may be substantially faster and have lower average power under normal circuit conditions. The design costs are significant for this overly conservative analysis.

Although some circuit elements are under tighter control in today's processes, the variation of other elements has increased. For example, a small reduction in transistor threshold voltage or channel length can cause a large increase in leakage. The layout of a logic cell's wires and transistors has become more complicated. Optical interference changes the resulting layout. Phase-shift lithography attempts to correct this. Varying etch rates have a greater impact on narrower wires.

Certain cell layouts are more likely to reduce yield, for example, due to increased gate delay or even a complete failure such as a disconnected wire. Ideally, cell layouts with lower yield would not be used, but these cells may be of higher speed. A small decrease in yield may be acceptable to meet delay requirements. To support yield trade-offs, foundries must share some yield information with customers along with corresponding price points.

Yield and variability data can be annotated to standard cell libraries, enabling software to perform statistical analysis of timing, power, and yield. This requires a detailed breakdown of process variability into systematic (e.g., spatially correlated) and random components. With this information, tools can estimate the yield of chips that will satisfy delay and power constraints. This is not straightforward, as timing paths are statistically correlated, and variability is also spatially correlated. However, these correlations can be accounted for in a conservative fashion that is still less conservative than worst-case corner analysis.

Variation in process and operating conditions as they pertain to static timing analysis are examined in Chapter 6. Chapter 22 discusses in detail process variation and design for manufacturability.

1.5.4 CIRCUIT RELIABILITY AND AGING

Transient glitches in a circuit's operation can be caused by cross-coupling noise as well as alpha-particle and neutron bombardment. There are also circuit aging issues that are of increasing concern for reliability, including electromigration, hot-carrier injection, and negative-bias threshold instability. These are discussed in more detail in Chapter 13.

Cross-coupling noise has become more significant with higher circuit frequencies and greater coupling capacitance between wires. Wire cross-coupling capacitance has increased with wires spaced closer together and with higher aspect ratios, which have been used to reduce wire RC delays as dimensions scale down. Wires can be laid out to reduce cross-coupling noise (e.g., by twizzling or shielding with ground wires). There are tools for analyzing cross-coupling noise, and some support for half shielding or full shielding of wires by routing them with one or both sides next to a power or ground net at the cost of additional routing congestion and detours.

As gate capacitance decreases with device dimensions and supply voltages are reduced, smaller amounts of charge are stored and are more easily disrupted by an alpha-particle or neutron strike.

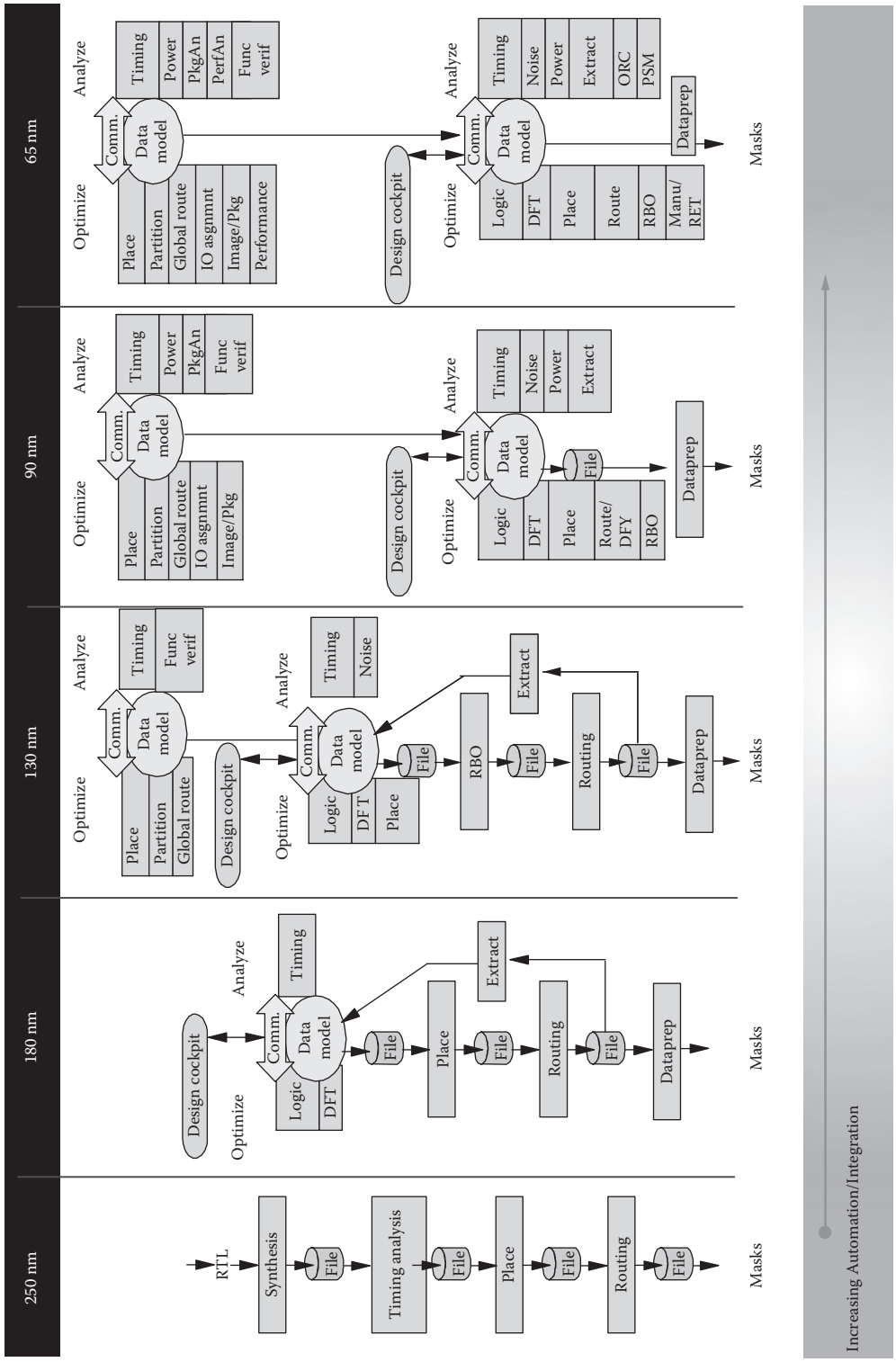

FIGURE 1.5 Integrated design flow.

Soft error rates due to alpha particles increase by a large factor as device dimensions are scaled down [58]. Soft error rates can be reduced by using silicon on insulator technology and other manufacturing methods or by using latches that are more tolerant to transient pulses [58].

For fault tolerance to glitches, circuits can have error detection and correction or additional redundancy. Tool support for logic synthesis and mapping to such circuits will simplify a designer's task.

Electromigration can cause open or short circuits [59,60] preventing correct circuit operation, which is becoming more problematic as current densities increase with narrower wires. Designers now need greater EDA tool support for the analysis of electromigration [61] and automated solutions, such as widening wires with high switching activity and additional spacing around clock cells. In a 10 nm technology, clock cells may also require decoupling capacitors inserted next to them to limit voltage drop.

1.6 CONCLUSION

In this chapter, we looked at how the RTL-to-GDSII design flow has changed over time and will continue to evolve. As depicted in Figure 1.5, we foresee continuing integration of more analysis functions into the integrated environment to cope with design for robustness and design for power. We provided an overview of a typical EDA flow and its steps and outlined the motivation for further EDA tool development.

There is high demand from IC design companies for faster turnaround to iterate through the EDA flow. Designs continue to increase rapidly in number of transistors, integrating more components, with smaller 10 and 7 nm process technologies actively being researched and developed, with 3D ICs on the horizon. More complex design rules for these newer technologies drive major EDA tool development in place and route. To achieve higher performance and lower power, the correlation must be improved between interrelated flow steps from RTL estimation and synthesis through to postroute optimization. This necessitates further tool integration, with analysis across more process corners and more operating modes and choices as to what are appropriate levels of abstraction to achieve better accuracy earlier in the design flow. All of these increase the computational demand, which motivates increasing the parallelism of EDA tools with support for larger sets of data, but still within relatively tight memory constraints. Much research and development continues within the EDA industry and academia to meet these difficult design challenges, as the EDA market continues to grow, albeit maturing with further consolidation within the industry.

The following chapters describe the individual RTL-to-GDSII flow steps in more detail, examining the algorithms and data structures used in each flow step. The additional infrastructure for design databases, cell libraries, and process technology is also discussed in the later chapters. We hope these encourage readers to innovate further with a deeper understanding of EDA.

ACKNOWLEDGMENT

The authors thank Vladimir Yutsis for his helpful feedback on Section 1.5.2.

REFERENCES

1. A. Sangiovanni-Vincentelli, The tides of EDA, *IEEE Design and Test of Computers*, 20, 59–75, 2003.
2. B.W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *Bell System Technical Journal*, 49, 291–308, 1970.
3. S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vevvhi, Optimization by simulated annealing, *Science*, 220, 671–680, 1983.
4. K.A. Chen, M. Feuer, K.H. Khokhani, N. Nan, and S. Schmidt, The chip layout problem: An automatic wiring procedure, *Proceedings of the 14th Design Automation Conference*, New Orleans, LA, 1977, pp. 298–302.
5. J.A. Darringer and W.H. Joyner, A new look at logic synthesis, *Proceedings of the 17th Design Automation Conference*, Minneapolis, MN, 1980, pp. 543–549.
6. D.C. Brock, *Understanding Moore's Law: Four Decades of Innovation*, Chemical Heritage Foundation, Philadelphia, PA, 2006.

7. J.L. Bentley, Multidimensional binary search trees used for associative searching, *Communication of the ACM*, 18, 509–517, 1975.
8. A. Guttman, R-trees: A dynamic index structure for spatial searching, *SIGMOD International Conference on Management of Data*, Boston, MA, 1984, pp. 47–57.
9. R.E. Bryant, Graph-based algorithms for Boolean function manipulation, *IEEE Transactions on Computer*, C-35, 677–691, 1986.
10. C.M. Fiduccia and R.M. Mattheyses, A linear time heuristics for improving network partitions, *Proceedings of the 19th Design Automation Conference*, Las Vegas, NV, 1982, pp. 175–181.
11. L. Hagen and A.B. Kahng, Fast spectral methods for ratio cut partitioning and clustering, *Proceedings of the International Conference on Computer Aided Design*, Santa Clara, CA, 1991, pp. 10–13.
12. H. Yang and D.F. Wong, Efficient network flow based min-cut balanced partitioning, *Proceedings of the International Conference on Computer Aided Design*, San Jose, CA, 1994, pp. 50–55.
13. J.M. Kleinhans, G. Sigl, and F.M. Johannes, Gordian: A new global optimization/rectangle dissection method for cell placement, *Proceedings of the International Conference on Computer Aided Design*, San Jose, CA, 1999, pp. 506–509.
14. C.L. Berman, L. Trevillyan, and W.H. Joyner, Global flow analysis in automatic logic design, *IEEE Transactions on Computer*, C-35, 77–81, 1986.
15. D. Brand, Redundancy and don't cares in logic synthesis, *IEEE Transactions on Computer*, C-32, 947–952, 1983.
16. R.B. Hitchcock Sr., Timing verification and the timing analysis program, *Proceedings of the 19th Design Automation Conference*, Las Vegas, NV, 1982, pp. 594–604.
17. T.I. Kirkpatrick and N.R. Clark, PERT as an aid to logic design, *IBM Journal of Research and Development*, 10, 135–141, 1966.
18. I. Sutherland, and R. Sproull, Logical effort: Designing for speed on the back of an envelope, *Proceedings of the 1991 University of California/Santa Cruz conference on Advanced Research in VLSI*, Santa Cruz, CA, 1991, pp. 1–16.
19. D. Harris et al., The fanout-of-4 inverter delay metric, unpublished manuscript, 1997. http://odin.ac.hmc.edu/~harris/research/FO4.pdf.
20. L.P.P.P. van Ginneken, Buffer placement in distributed RC-tree networks for minimal Elmore delay, *International Symposium on Circuits and Systems*, New Orleans, LA, 1990, pp. 865–868.
21. W. Shi and Z. Li, A fast algorithm for optimal buffer insertion, *IEEE Transactions on Computer-Aided Design*, 24(6), 879–891, June 2005.
22. C. Kashyap et al., An "effective" capacitance based delay metric for RC interconnect, *International Conference on Computer-Aided Design*, 2000, pp. 229–234.
23. I. Bustany et al., ISPD 2014 Benchmarks with sub-45 nm technology rules for detailed-routing-driven placement, *Proceedings of the 2014 International Symposium on Physical Design*, Petaluma, CA, 2014, pp. 161–168.
24. V. Lebars, Data path optimization: The newest answer to dynamic power reduction, *EDN Magazine*, May 2015. http://www.edn.com/electronics-blogs/eda-power-up/4439458/Data-path-optimization--The-newest-answer-to-dynamic-power-reduction.
25. Y. Luo, P.V. Srinivas, and S. Krishnamoorthy, Method and apparatus for optimization of digital integrated circuits using detection of bottlenecks, US Patent 7191417, 2007.
26. J. Lakos, *Large-Scale C++ Software Design*, Addison-Wesley, Reading, MA, 1996.
27. R.P. Abato, A.D. Drumm, D.J. Hathaway, and L.P.P.P. van Ginneken, Incremental timing analysis, US Patent 5 508 937, 1996.
28. S. Hirsch, and U. Finkler, To Thread or not to thread, *IEEE Design and Test*, 30(1), 17–25, February 2013.
29. B. Catanzaro, K. Keutzer, and B. Su, Parallelizing CAD: A timely research agenda for EDA, *Proceedings of the Design Automation Conference*, Anaheim, CA, 2008, pp. 12–17.
30. G. Amdahl, Validity of the single-processor approach to achieving large scale computing capabilities, *Proceedings of the AFIPS Spring Joint Computer Conference*, Atlantic City, NJ, 1967, pp. 483–485.
31. P. Madden, Dispelling the myths of parallel computing, *IEEE Design and Test*, 30(1), 58–64, February 2013.
32. E. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik*, 1, 269–271, 1959.
33. A. Wolfe, Intel clears up post-Tejas confusion, *CRN Magazine*, May 17, 2004. http://www.crn.com/news/channel-programs/18842588/intel-clears-up-post-tejas-confusion.htm.
34. S. Thompson, P. Packan, and M. Bohr, MOS scaling: Transistor challenges for the 21st century, *Intel Technology Journal*, Q3, 1–19, 1998.
35. T. Sakurai and R. Newton, Delay analysis of series-connected MOSFET circuits, *IEEE Journal Solid-State Circuits*, 26, 122–131, 1991.
36. D. Lee, W. Kwong, D. Blaauw, and D. Sylvester, Analysis and minimization techniques for total leakage considering gate oxide leakage, *Proceedings of the 40th Design Automation Conference*, Anaheim, CA, 2003, pp. 175–180.

37. S. Mutoh et al., 1-V power supply high-speed digital circuit technology with multithreshold—voltage CMOS, *IEEE Journal Solid-State Circuits*, 30, 847–854, 1995.
38. G. Flach et al., Effective method for simultaneous gate sizing and Vth assignment using Lagrangian relaxation, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4), 546–557, April 2014.
39. C. Jan et al., A 22 nm SoC platform technology featuring 3-D tri-gate and high-k/metal gate, optimized for ultra low power, high performance and high density SoC applications, *Proceedings of the IEEE International Electron Devices Meeting*, San Francisco, CA, 2012, pp. 3.1.1–3.1.4.
40. D. Chinnery et al., Gater expansion with a fixed number of levels to minimize skew, *Proceedings of SNUG Silicon Valley*, Santa Clara, CA, 2012.
41. H. Xiang et al., Row based dual-VDD island generation and placement, *Proceedings of the 51st Design Automation Conference*, San Francisco, CA, 2014, pp. 1–6.
42. S. Natarajan et al., A 14 nm logic technology featuring 2nd-generation FinFET transistors, air-gapped interconnects, self-aligned double patterning and a 0.0588 μm^2 SRAM cell size, *International Electron Devices Meeting*, San Francisco, CA, 2014.
43. W. Joyner et al., The many challenges of triple patterning, *IEEE Design and Test*, 31(4), 52–58, August 2014.
44. M. van der Brink, Many ways to shrink: The right moves to 10 nanometer and beyond, *SPIE Photomask Technology Keynote*, Monterey, CA, 2014.
45. M. Martini, The long and tortuous path of EUV lithography to full production, *Nanowerk Nanotechnology News*, April 2014. http://www.nanowerk.com/spotlight/spotid=35314.php.
46. A. Pirati et al., Performance overview and outlook of EUV lithography systems, *Proceedings of SPIE*, *Extreme Ultraviolet (EUV) Lithography VI*, San Jose, CA, Vol. 9422, March 1–18, 2015.
47. L. Wu, Electron beam testing of integrated circuits, US Patent 3969670, 1976.
48. Cadence, *LEF/DEF Language Reference*, Product Version 5.7, December 2010.
49. D. Pan, B. Yu, and J. Gao, Design for manufacturing with emerging lithography, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(10), 1453–1472, October 2013.
50. C.-K. Wang et al., Closing the gap between global and detailed placement: Techniques for improving routability, *Proceedings of the International Symposium on Physical Design*, Monterey, CA, 2015.
51. A. Kennings, N. Darav, and L. Behjat, Detailed placement accounting for technology constraints, *Proceedings of the International Conference on Very Large Scale Integration*, Playa del Carmen, Mexico 2014.
52. K. Vaidyanathan, Rethinking ASIC design with next generation lithography and process integration, *Proceedings of SPIE, Design for Manufacturability through Design-Process Integration VII*, San Jose, CA, 2013.
53. J. Subramanian, Performance impact from metal fill insertion, *CDNLive!*, 2007.
54. Y. Ma et al., Self-aligned double patterning (SADP) compliant design flow, *Proceedings of SPIE*, *Design for Manufacturability through Design-Process Integration VI*, San Jose, CA, 2012.
55. D. Petranovic, J. Falbo, and N. Kurt-Karsilayan, Double patterning: Solutions in parasitic extraction, *Proceedings of SPIE*, *Design for Manufacturability through Design-Process Integration VII*, San Jose, CA, 2013.
56. D. Chinnery, High performance and low power design techniques for ASIC and custom in nanometer technologies, *Proceedings of the International Symposium on Physical Design*, Stateline, NV, 2013, pp. 25–32.
57. A. Dasdan, and I. Hom, Handling inverted temperature dependence in static timing analysis, *ACM Transactions on Design Automation of Electronic Systems*, 11(2), 306–324, April 2006.
58. C. Constantinescu, Trends and challenges in VLSI circuit reliability, *IEEE Micro*, 23, 14–19, 2003.
59. J. Lienig, Electromigration and its impact on physical design in future technologies, *International Symposium on Physical Design*, Stateline, NV, 2013, pp. 33–40.
60. Y. Zhang, L. Liang, and Y. Liu, Investigation for electromigration-induced hillock in a wafer level interconnect device, *Electronic Components and Technology Conference*, Las Vegas, NV, 2010, pp. 617–624.
61. A. Abbasinasab and M. Marek-Sadowska, Blech effect in interconnects: Applications and design guidelines, *International Symposium on Physical Design*, Monterey, CA, 2015, pp. 111–118.