SECTION I

RTL TO GDS-II, OR SYNTHESIS, PLACE, AND ROUTE

1

Design Flows

Leon Stok
IBM Corporation
TJ Watson Research Center
Yorktown Heights, New York

David Hathaway
IBM Microelectronics
Essex Junction, Vermont

Kurt Keutzer
University of California
Berkeley, California

David Chinnery
University of California
Berkeley, California

1.1 Introduction

Scaling has driven the entire IC implementation RTL to GDSII design flow from one which uses primarily standalone synthesis, placement, and routing algorithms to an integrated construction and analysis flow for design closure. This chapter will address how the challenges of rising interconnect delay led to a new way of thinking about and integrating design closure tools. New scaling challenges such as leakage power, variability, and reliability will keep on challenging the current state of the art in design closure.

The RTL to GDSII flow has undergone significant changes in the last 25 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. Challenges like leakage power, variability, and reliability will continue to require significant changes to the design-closure process in the future. 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 coming to address the latest challenges.

In his keynote at the 40th Design Automation Conference entitled "The Tides of EDA" [1], Alberto Sangiovanni-Vincentelli distinguished three periods of EDA: The Age of the Gods, The Age of the Heroes, and The Age of the Men. These eras were characterized respectively by senses, imagination, and reason.

When we limit ourselves to the RTL to GDSII flow of the CAD area, we can distinguish three main eras in its development: the Age of Invention, the Age of Implementation, and the Age of Integration. During the invention era, routing, placement, static timing analysis and logic synthesis were invented. In the age of implementation they were drastically improved by designing sophisticated data structures and advanced algorithms. This allowed the 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. 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.

1.2 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, 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 routing, partitioning, and placement algorithms were invented. Partitioning is one of the fundamental steps in the physical design flow. Kernighan and Lin [2] pioneered one of the basic partitioning techniques in 1970. Simulated annealing [3] algorithms were pioneered for placement, and allowed for a wide range of optimization criteria to be deployed. Basic algorithms for channel, switchbox, 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 Implementation

With the advent of integrated circuits, 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. The concept of keeping the height of the letters fixed and letting the width of the lead base of each of the letters vary according to the letter's size, enabled significant automation in the development of printing. Similarly, in standard cell Application Specific Integrated Circuits (ASIC) design, one uses standard cells of common height but varying widths depending on the complexity of the single standard cell. These libraries (Chapter 9) 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 ASICs 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). High-level synthesis translated a Verilog or VHDL description into an RTL netlist. Logic synthesis optimized the netlist and mapped it into a netlist with technology gates. This was followed by placement to place the gates, and routing to connect them together. Finally, a timing simulator was used to verify the timing using a limited amount of extracted data.

Design sizes kept on increasing rapidly in this era. While in the discrete space, several tens of components needed to be placed and routed, logic synthesis allowed for rapid creation of netlists with millions of gates, growing from 40,000 gates in 1984 to 40,000,000 gate designs in 2000.

New data structures like Quadtrees [6] allowed very efficient searches in the geometric space. Applications of Boolean Decision Diagrams (BDDs) [7] enabled efficient Boolean reasoning on significantly larger logic partitions.

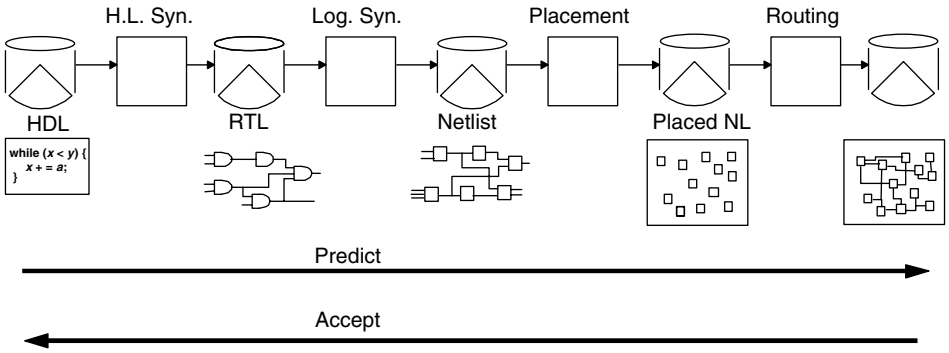

FIGURE 1.1 Sequential design flow.

Much progress was made in implementing partitioning algorithms. A more efficient version of Kernighan and Lin's partitioning algorithm was given by Fidducia and Mattheyses [8]. 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 [9] proved to be very effective for certain problems. Yang [10] demonstrated results that outperformed the two earlier mentioned 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 [11].

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 [12] and redundancy removal [13] 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 [14] based on early work in [15] 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 must have order of complexity less than $O(n^2)$ and preferably not more than $O(n \log n)$. These complexities were met because of the above-mentioned advances in data structures and algorithms. This allowed design tools to be applied to large real problems. However, it became increasingly difficult to find appropriate cost functions for these 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. The optimization algorithms in each of the discrete design steps were guided by objective functions that rely 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 of the prediction functions, the measurement of 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 pretty 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 indicates how the delay of the gate increases with load and the intrinsic delay is the load-independent contribution of the gate delay. The FO4 delay is the delay of one gate of the specified type driving four identical gates of the same size. The fourth column of the table shows that the delay of a more complex NOR4, driving four copies of itself, can be as much as three times the delay of a simple inverter. The logical effort-based calculation to compute this already assumes that the gates have been ideally sized to best match the load they are driving. Simple addition of logic levels is therefore becoming insufficient, and one needs to know what gates the logic is actually mapped to, to predict the delay of a design with reasonable accuracy. 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 assumed by a bad 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 middle of the 1990s, these predictions based on gate delays started to be much more inaccurate. Gate delays became increasingly dependent on the load they were driving, and on the rise and fall rates of the input signals to the gates. At the same time, the fraction of net load due to wires started to increase. Knowledge of the physical design became essential to predict reasonably the delay of a path. Initially, it was mostly just the placement of the cells that was needed. The placement affects the delay, but wiring does so much less, since any route close to the minimum length will 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 for example 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. Buffering of long lines became an integral part of physical design, in addition to placing and routing [16].

Recently, we are seeing that a wire's environment is becoming more significant. The cross-coupling capacitance between wires has increased as the ratio between wire spacing and wire height decreases. The detailed routing is therefore becoming significant in predicting actual delays.

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

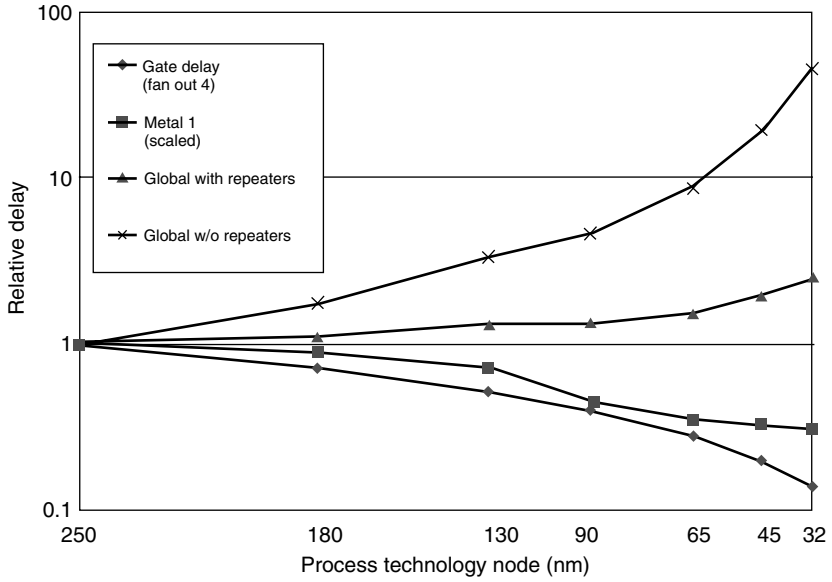

FIGURE 1.2 Gate and interconnect delay.

Net list alterations, 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.

1.4 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 designs 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 also often did not even converge. People have explored several possible solutions to this convergence problem including:

- 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 (side-)effect than desired or anticipated on the final result. The fix-up solution requires an enormous increase in manual design effort. Improving predictions has proven to be very difficult. Gain-based delay models trade off area predictability for significant delay predictability and gave some temporary relief, but in general it has been extremely hard to improve

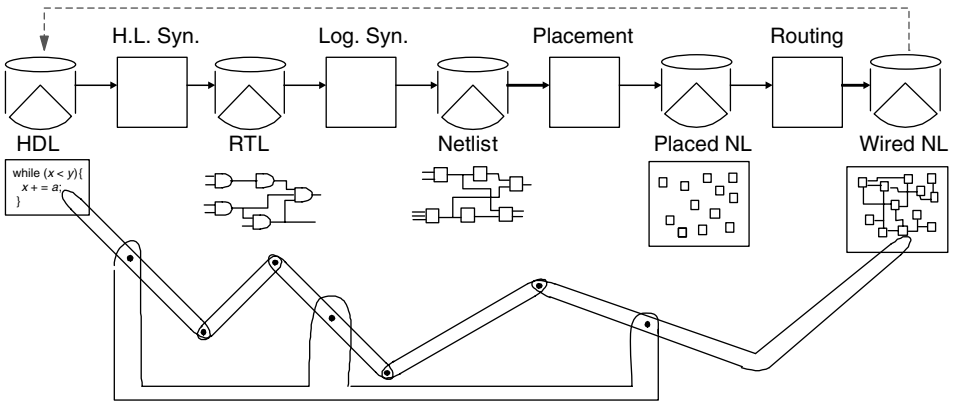

FIGURE 1.3 Controlled design flow.

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 in the design closure flow. 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 is rapidly becoming a very important design metric, and noise issues are starting to affect significantly delays and even make designs function incorrectly. In addition to integrated static timing analysis, power and noise analysis need to be included as well.

All these analysis and optimization algorithms need to work in an incremental fashion, because runtime prevents 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 and can work together 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 the way in which they are developed into EDA tools has changed significantly. In other cases, the required incrementality has led to interesting new algorithms. 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 three basic interrelated principles:

- Tools are integrated
- Tools are modular
- Tools operate incrementally

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

1.4.1 Integrated

Tool integration provides the ability for tools to directly communicate with each other. A superficial form of integration can be provided by initiating the execution of traditional point tools, which continue to operate to and from files, from a common user interface. When we discuss tool integration, however, we will mean a tighter form of integration, which 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 runtime model (see Chapter 12) 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 wiring programs, and still have the placement run to completion before starting wiring.

Careful design of an application can make it easier to integrate with other applications on a common runtime model, even if it was originally written as a standalone application.

Achieving tool integration requires an agreed upon set of semantic elements in the design representation in terms of which the tools can comunicate. In the design closure flow, these elements generally consist of blocks, pins, and nets and their connectivity, block placement locations, and wiring routes. Individual applications will augment this common model data 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 find efficiently the components of the domain-specific data associated with these elements. Although this can be accomplished by name look-up, 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.

1.4.2 Modular

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 more likely to be able to be reused, since they are less likely to have side-effects, which are not wanted in the reuse environment. It simplifies code development and maintenance, since problems are likely to be easier to isolate, and 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 in large part due to a lack of modularity [17]. In the interest of collecting all behavior associated with the common runtime model in one place, they also concentrated control of what would appropriately be application-specific data. This made the runtime model too large and complicated, and inhibited the tuning and reorganization of data structures by individual applications.

1.4.3 Incremental

Incrementally operating tools can update information about a design, or the design itself, without revisiting or reprocessing the entire design. This enables finer-grained interaction between integrated tools. For example, incremental processing capability in static timing analysis makes it possible for logic synthesis to change a design and see the effect of that change on timing, without requiring a complete retiming of the design to be performed.

Incremental processing can reduce the time required to make a "loop" between different analysis and optimization tools. As a result, it can make a higher frequency of tool interaction practical, allowing the complete consequences of each optimization decision to be more accurately understood.

The ordering of actions between a set of incremental applications is important. If a tool like synthesis is to make use of another incremental tool like timing analysis, it needs to be able to immediately see the

effects of any actions it takes in the results reported by the tool being used. This means that the incremental application must behave as if every incremental update occurs immediately after the event which precipitates it.

There are four basic characteristics desirable in an incremental tool:

1.4.3.1 Autonomicity

Autonomicity means that applications initiating events which precipitate changes in the incremental tool 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. The first of these is important because it simplifies the process required for an application to make changes to a design, and eliminates the need to update the application when new incremental tools are added to the design tool environment. It is usually achieved by providing a facility for incremental applications to register callbacks, allowing them to be notified of events which are of interest.

The second is important because it keeps the application using the incremental tool from needing to understand the details of the incremental algorithm. This allows changes to the incremental tool algorithm without requiring changes in the applications that use it, and reduces the likelihood of errors in the control of the incremental processing, since that control is not scattered among many applications that use the incremental tool. It also makes the use of the incremental tool by other applications much simpler.

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, which would otherwise be spent getting results that 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 for many or all of the same points in the design, while only the last values computed are actually used.

Lazy evaluation also simplifies the flow of control when recalculating values. If we have several incremental analysis functions with interdependencies (e.g., timing depends on extraction results), and all try to immediately update results when notified of a netlist change, some callback ordering mechanism would be needed to ensure that the updates are made in the correct order. If in the example given above, timing updates were made before extraction updates, the results would not be correct, as they would be using stale extraction results. However, if each application performs only invalidation when notified of a design change, the updates can be ordered correctly through demand-driven recomputation. In the above example, if the first result requested after a netlist change is from timing, it would in turn request updated extraction results it needed for delay computation. Since all invalidation in all applications would have been completed before any request was made for updated analysis results, the extraction tool would have received its callback and performed any necessary invalidation, so when a value was requested from it by timing, the new value would be computed and returned.

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. Note that 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 subsequent requests may be made for some of this other information. Partial lazy evaluation is employed in some timing analyzers [18] 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.

1.4.3.3 Change Notification (Callbacks and Undirected Queries)

Change notification means having the incremental tool notify 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 it knows where

consequent changes to analysis results (e.g., timing) will result. 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 runtime 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 which 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 which applications can register with it, and which 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)

We need to be able to use incremental applications in an "experimental mode," 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. It is appropriate for each incremental tool to handle the reversing 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. The incremental tool which handles the reversing of changes should be the one actually responsible for storing the data being reversed. Thus, changes to the netlist initiated by logic synthesis should be reversed by the runtime 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, and 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. 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, either to save to a file (e.g., an Engineering Change Order, or 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 redone.

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 undo an undo. Even though the operation of a transformation is deterministic, 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 not be preserved when a change is undone. Also, a considerable amount of analysis may be required by some transformations to determine the exact changes which are to be made, and we would like to avoid having to redo this analysis when we redo the transformation.

Because trial transformations to a network may be nested, we also need to be able to stack multiple checkpoints.

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.

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 runtime 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.

1.5 Future Scaling Challenges

In the previous sections, we mainly focused on how continued scaling changed the way we are able to predict delay throughout to the design flow. This has been one of the main drivers to the design flow changes over the last two decades. However, new challenges are arising that will again require us to rethink 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 power, noise and variability (with required incremental extraction tools) in addition to the well-established incremental timing analysis.

1.5.1 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. In 90 nm, even some high-end microprocessor designers have found power to be a more important constraint than delay.

The price of increasing circuit speed as per 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 $fCV_{dd}^2/2$. Increasing circuit speed increases switching frequency f proportionally, and capacitance per unit area also increases.

Transistor capacitance varies inversely to the transistor gate oxide thickness t_{ox} ($C_{gate} \propto WL/t_{ox}$, where 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 [19]. As device dimensions scale down linearly, the 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 corresponding wire RC delays.

Thus, if the supply voltage is constant, the dynamic power per unit area increases less 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 $I_{D,sat} \propto W(V_{dd} - V_{th})^{\alpha}/Lt_{ox}$, where V_{th} is the transistor threshold voltage, and α is between 1 and 2 [20], which reduces the circuit speed. 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 when the transistor gate-to-source voltage is below V_{th} and the transistor is nominally off, depends exponentially on V_{th} ($I_{subthreshold} \propto e^{-qV_{th}/nkT}$ 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 magnitudes smaller than subthreshold leakage, it has been increasing much faster, and will be significant in future process technologies [21].

Synthesis 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.

Standby power may be reduced by using high threshold voltage sleep transistors to connect to the power rails [22]. In standby, these transistors are switched off to stop the subthreshold leakage path from supply to ground rails. 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 which enter standby at the same time, so that they can share a common sleep transistor to reduce overheads.

Today, leakage can contribute a significant portion of the total power — when the circuit is active, in the range of 5 to 40% — 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. It is essential that tools treat leakage power on equal terms with dynamic power when trying to minimize the total power. Designers are now using standard cell libraries with multiple threshold voltages, using low threshold voltage transistors to maintain speed on the critical path and high threshold voltage transistors to reduce the leakage power of gates that have slack. Synthesis tools must support a choice between cells with smaller and larger leakage as appropriate for delay requirements on the gate within the circuit context.

1.5.2 Variability

To account for process and temperature variation, circuit delay is traditionally estimated from the (worst case) slow process and high temperature corner for the standard cell library. Hold-time violations and worst-case dynamic power may be estimated from the fast process (corresponding to lower threshold

voltages and shorter channel lengths due to variability) and low temperature corner. Worst-case leakage power can be estimated at the fast process and high temperature corner. It is assumed that this simple analysis is sufficient to ensure that timing constraints are met. 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. Secondly, 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 elements of a circuit 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 (whether due to increased gate delay or complete failure such as when a wire is not connected) due to these sorts of problems. Ideally, cell layouts with lower yield would not be used, but these cells may be of higher speed. It may be desirable to accept a small decrease in yield to meet delay requirements. To support yield trade-offs, foundries must give some yield information to 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 the 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.

1.5.3 Reliability

Glitches can be caused by cross-coupling noise, and alpha particle and neutron bombardment. These glitches can cause a temporary error in a circuit's operation.

Cross-coupling noise has increased with higher circuit frequencies and greater coupling capacitance between wires. Wire cross-coupling capacitance has increased with wires being 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, but automated routing to reduce coupling is not supported in tools yet.

With gate capacitance decreasing with device dimensions, and due to reduced supply voltages, smaller amounts of charge are stored, which are more easily disrupted by an alpha particle or neutron strike. Soft error rates due to alpha particles increase by a large factor as device dimensions are scaled down [23]. Soft error rates can be reduced by using silicon on insulator and other manufacturing methods, or by using latches that are more tolerant to transient pulses [23].

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

1.6 Conclusion

In this chapter, we gave a flavor of how the RTL to GDSII design flow has changed over time and will continue to evolve. As depicted in Figure 1.4, we foresee continuing integration of more analysis functions into the integrated environment to cope with design for robustness and design for power.

The other chapters of this book describe each of the algorithms, data structures, and their role in the RTL to GDSII flow in much more detail.

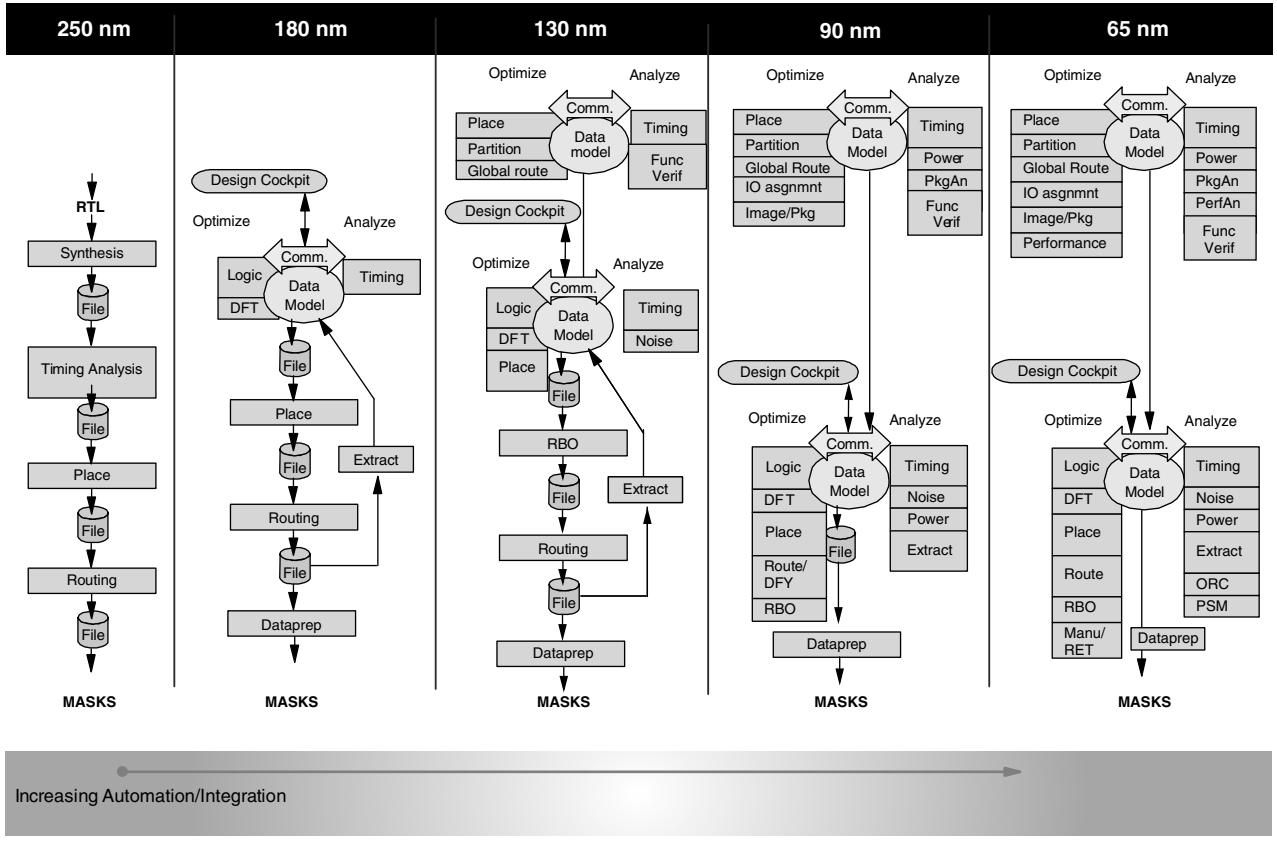

FIGURE 1.4 Integrated design flow.

References

[1] A. Sangiovanni-Vincentelli, The tides of EDA. *Des. Test Comput., IEEE*, 20, 59–75, 2003.

[2] B.W. Kernighan and S. Lin, An efficient heuristic procedure for partitioning graphs, *Bell Syst. Tech. J.*, 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,* 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*, 1980, pp. 543-549.

[6] J.L. Bentley, Multidimensional binary search trees used for associative searching, *Commn. ACM*, 18, 509–517, 1975.

[7] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, *IEEE Trans. Comput.*, C-35, 677-691, 1986.

[8] C.M. Fiduccia and R.M. Mattheyses, A linear time heuristics for improving network partitions, *Proceedings of the 19th Design Automation Conference*, 1982, pp. 175–181.

[9] L. Hagen and A.B. Kahng, Fast spectral methods for ratio cut partitioning and clustering, *Proceedings of the International Conference on Computer Aided Design,* 1991, pp. 10–13.

[10] H. Yang and D.F. Wong, Efficient network flow based min-cut balanced partitioning, *Proceedings of the International Conference on Computer Aided Design,* 1994, pp. 50–55.

[11] J.M. Kleinhaus, 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*, 1999, pp. 506–509.

[12] C.L. Berman, L. Trevillyan, and W.H. Joyner, Global flow analysis in automatic logic design, *IEEE Trans. Comput.*, C-35, 77–81, 1986.

[13] D. Brand, Redundancy and don't cares in logic synthesis, *IEEE Trans. Comput.*, C-32, 947–952, 1983.

[14] R.B. Hitchcock Sr., Timing verification and the timing analysis program, *Proceedings of the 19th Design Automation Conference*, 1982, pp. 594–604.

[15] T.I. Kirkpatrick and N.R. Clark, Pert as an aid to logic design, *IBM JRD*, 10, 135–141, 1966.

[16] L.P.P.P. van Ginneken, Buffer placement in distributed RC-tree networks for minimal Elmore delay, *International Symposium on Circuits and Systems*, 1990, pp. 865–868.

[17] J. Lakos, *Large-Scale C++ Software Design*, Addison-Wesley, Reading, MA, 1996.

[18] R.P. Abato, A.D. Drumm, D.J. Hathaway, and L.P.P.P. van Ginneken, Incremental Timing Analysis, U.S. patent 5 508 937, 1996.

[19] S. Thompson, P. Packan, and M. Bohr, MOS scaling: transistor challenges for the 21st century, *Intel Technol. J.*, Q3, 1998.

[20] T. Sakurai and R. Newton, Delay analysis of series-connected MOSFET circuits, *IEEE J. Solid-State Circuits*, 26, 122–131, 1991.

[21] 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*, 2003, pp. 175–180.

[22] S. Mutoh et al., 1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS, *IEEE J. Solid-State Circuits*, 30, 847–854, 1995.

[23] C. Constantinescu, Trends and challenges in VLSI circuit reliability, *IEEE Micro*, 23, 14–19, 2003.