CHAPTER 1

An Invitation to Design Verification

Chapter Highlights

- What is design verification?
- The basic verification principle
- Verification methodology
- Simulation-based verification versus formal verification
- Limitations of formal verification
- A quick overview of Verilog scheduling and execution semantics

From the makeup of a project team, we can see the importance of design verification. A typical project team usually has an equal number of design engineers and verification engineers. Sometimes verification engineers outnumber design engineers by as many as two to one. This stems from the fact that to verify a design, one must first understand the specifications as well as the design and, more important, devise a different design approach from the specifications. It should be emphasized that the approach of the verification engineer is different from that of the design engineer. If the verification engineer follows the same design style as the design engineer, both would commit the same errors and not much would be verified.

From the project development cycle, we can understand the difficulty of design verification. Statistical data show that around 70% of the project development cycle is devoted

to design verification. A design engineer usually constructs the design based on cases representative of the specifications. However, a verification engineer must verify the design under all cases, and there are an infinite number of cases (or so it seems). Even with a heavy investment of resources in verification, it is not unusual for a reasonably complex chip to go through multiple tape-outs before it can be released for revenue.

The impact of thorough design verification cannot be overstated. A faulty chip not only drains budget through respin costs, it also delays time-to-market, impacts revenue, shrinks market share, and drags the company into playing the catch-up game. Therefore, until people can design perfect chips, or chip fabrication becomes inexpensive and has a fast turnaround time, design verification is here to stay.

1.1 What Is Design Verification?

A design process transforms a set of specifications into an implementation of the specifications. At the specification level, the specifications state the functionality that the design executes but does not indicate how it executes. An implementation of the specifications spells out the details of how the functionality is provided. Both a specification and an implementation are a form of description of functionality, but they have different levels of concreteness or abstraction. A description of a higher level of abstraction has fewer details; thus, a specification has a higher level of abstraction than an implementation. In an abstraction spectrum of design, we see a decreasing order of abstraction: functional specification, algorithmic description, register–transfer level (RTL), gate netlist, transistor netlist, and layout (Figure 1.1). Along this spectrum a description at any level can give rise to many forms of a description at a lower level. For instance, an infinite number of circuits at the gate level implements the same RTL description. As we move down the ladder, a less abstract description adds more details while preserving the descriptions at higher levels. The process of turning a more abstract description into a more concrete description is called *refinement*. Therefore, a design process refines a set of specifications and produces various levels of concrete implementations.

Design verification is the reverse process of design. Design verification starts with an implementation and confirms that the implementation meets its specifications. Thus, at every step of design, there is a corresponding verification step. For example, a design step that turns a functional specification into an algorithmic implementation requires a verification step to ensure that the algorithm performs the functionality in the specification. Similarly, a physical design that produces a layout from a gate netlist has to be verified to ensure that the layout corresponds to the gate netlist. In general, design verification encompasses many areas, such as functional verification, timing verification, layout verification, and electrical verification, just to name a few. In this book we study only functional verification and refer to it as *design verification*. Figure 1.2 shows the relationship between the design process and the verification process.

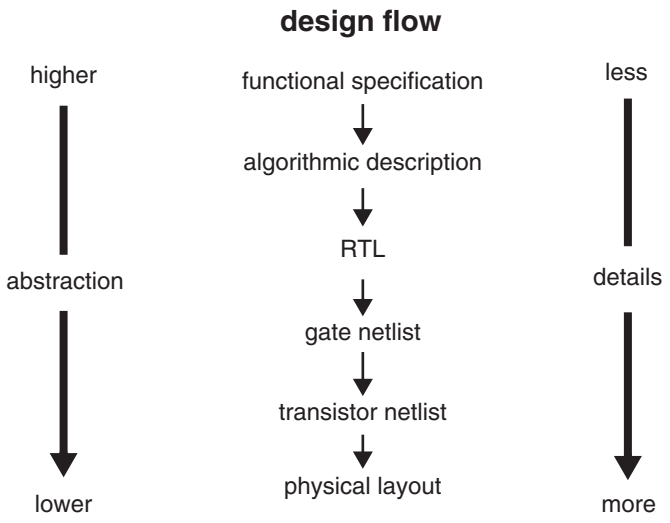

Figure 1.1 A ladder of design abstraction

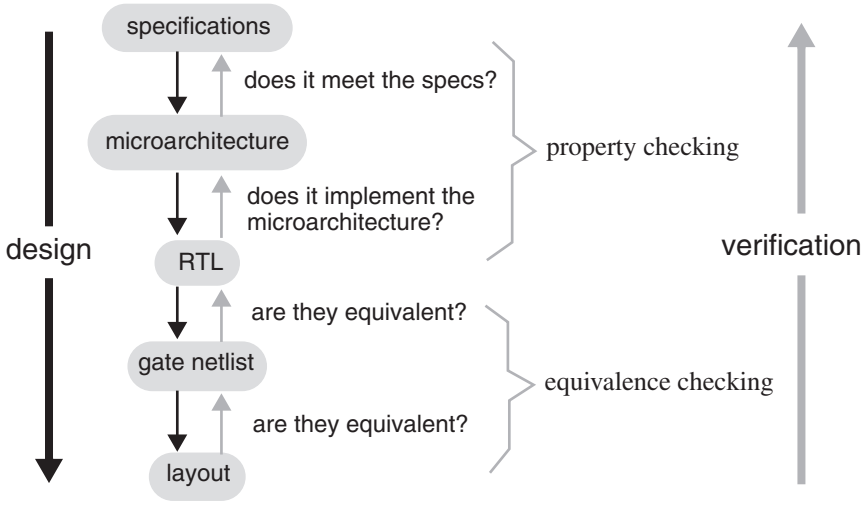

Figure 1.2 The relationship between design and verification

On a finer scope, design verification can be further classified into two types. The first type verifies that two versions of design are functionally equivalent. This type of verification is called *equivalence checking*. One common scenario of equivalence checking is comparing two versions of circuits at the same abstraction level. For instance, compare the gate netlist of a prescan circuit with its postscan version to ensure that the two are equivalent under normal operating mode.

However, the two versions of the design differ with regard to abstraction level. For example, one version of the design is at the level of specification and the other version is at the gate netlist level. When the two versions differ substantially with regard to level of abstraction, they may not be functionally equivalent, because the lower level implementation may contain more details that are allowed, but are unspecified, at the higher level. For example, an implementation may contain timing constraints that are not part of the original specification. In this situation, instead of verifying the functional equivalence of the two versions, we verify instead that the implementation satisfies the specifications. Note that equivalence checking is two-way verification, but this is a one-way verification because a specification may not satisfy an unspecified feature in the implementation. This type of verification is known as *implementation verification, property checking*, or *model checking*. Based on the terminology of property checking, the specifications are properties that the implementation must satisfy. Based on the terminology of model checking, the implementation or design is a model of the circuit and the specifications are properties. Hence, model checking means checking the model against the properties.

1.2 The Basic Verification Principle

There are two types of design error. The first type of error exists not in the specifications but in the implementations, and it is introduced during the implementation process. An example is human error in interpreting design functionality. To prevent this type of error, we can use a software program to synthesize an implementation directly from the specifications. Although this approach eliminates most human errors, errors can still result from bugs in the software program, or usage errors of the software program may be encountered. Furthermore, this synthesis approach is rather limited in practice for two reasons. First, many specifications are in the form of casual conversational language, such as English, as opposed to a form of precise mathematical language, such as Verilog or C++. We know that automatic synthesis from a loose language is infeasible. In fact, as of this writing, there is no high-level formal language that specifies both functional and timing requirements. A reason for this is that a high-level functional requirement does not lend itself to timing requirements, which are more intuitive at the implementation level. Therefore, timing requirements such as delay and power, when combined with high-level functional specifications, are so overtly inaccurate that people relegate timing specifications to

levels of lower abstraction. Second, even if the specifications are written in a precise mathematical language, few synthesis software programs can produce implementations that meet all requirements. Usually, the software program synthesizes from a set of functional specifications but fails to meet timing requirements.

Another method—the more widely used method—to uncover errors of this type is through redundancy. That is, the same specifications are implemented two or more times using different approaches, and the results of the approaches are compared. In theory, the more times and the more different ways the specifications are implemented, the higher the confidence produced by the verification. In practice, more than two approaches is rarely used, because more errors can be introduced in each alternative verification, and costs and time can be insurmountable.

The design process can be regarded as a path that transforms a set of specifications into an implementation. The basic principle behind verification consists of two steps. During the first step, there is a transformation from specifications to an implementation. Let us call this step *verification transformation*. During the second step, the result from the verification is compared with the result from the design to detect any errors. This is illustrated in Figure 1.3 (A). Oftentimes, the result from a verification transformation takes place in the head of a verification engineer, and takes the form of the properties deduced from the specifications. For instance, the expected result for a simulation input vector is calculated by a verification engineer based on the specifications and is an alternative implementation.

Obviously, if verification engineers go through the exact same procedures as the design engineers, both the design and verification engineers are likely to arrive at the same conclusions, avoiding and committing the same errors. Therefore, the more different the design and verification paths, the higher confidence the verification produces. One way to achieve high confidence is for verification engineers to transform specifications into an implementation model in a language different from the design language. This language is called *verification language*, as a counterpart to design language. Examples of verification languages include Vera, C/C++, and *e*. A possible verification strategy is to use C/C++ for the verification model and Verilog/VHSIC Hardware Description Language (VHDL) for the design model.

During the second step of verification, two forms of implementation are compared. This is achieved by expressing the two forms of implementation in a common intermediate form so that equivalency can be checked efficiently. Sometimes, a comparison mechanism can be sophisticated—for example, comparing two networks with arrival packets that may be out of order. In this case, a common form is to sort the arrival packets in a predefined way. Another example of a comparison mechanism is determining the equivalence between a

transistor-level circuit and an RTL implementation. A common intermediate form in this case is a binary decision diagram.

Here we see that the classic simulation-based verification paradigm fits the verification principle. A simulation-based verification paradigm consists of four components: the circuit, test patterns, reference output, and a comparison mechanism. The circuit is simulated on the test patterns and the result is compared with the reference output. The implementation result from the design path is the circuit, and the implementation results from the verification path are the test patterns and the reference output. The reason for considering the test patterns and the reference output as implementation results from the verification path is that, during the process of determining the reference output from the test patterns, the verification engineer transforms the test patterns based on the specifications into the reference output, and this process is an implementation process. Finally, the comparison mechanism samples the simulation results and determines their equality with the reference output. The principle behind simulation-based verification is illustrated in Figure 1.3 (C).

Verification through redundancy is a double-edged sword. On the one hand, it uncovers inconsistencies between the two approaches. On the other hand, it can also introduce incompatible differences between the two approaches and often verification errors. For example, using a C/C++ model to verify against a Verilog design may force the verification engineer to resolve fundamental differences between the two languages that otherwise could be avoided. Because the two languages are different, there are areas where one language models accurately whereas the other cannot. A case in point is modeling timing and parallelism in the C/C++ model, which is deficient. Because design codes are susceptible to errors, verification code is equally prone to errors. Therefore, verification engineers have to debug both design errors as well as verification errors. Thus, if used carelessly, redundancy strategy can end up making engineers debug more errors than those that exist in the design—design errors plus verification errors—resulting in large verification overhead costs.

As discussed earlier, the first type of error is introduced during an implementation process. The second type of error exists in the specifications. It can be unspecified functionality, conflicting requirements, and unrealized features. The only way to detect the type of error is through redundancy, because specification is already at the top of the abstraction hierarchy and thus there is no reference model against which to check. Holding a design review meeting and having a team of engineers go over the design architecture is a form of verification through redundancy at work. Besides checking with redundancy directly, examining the requirements in the application environment in which the design will

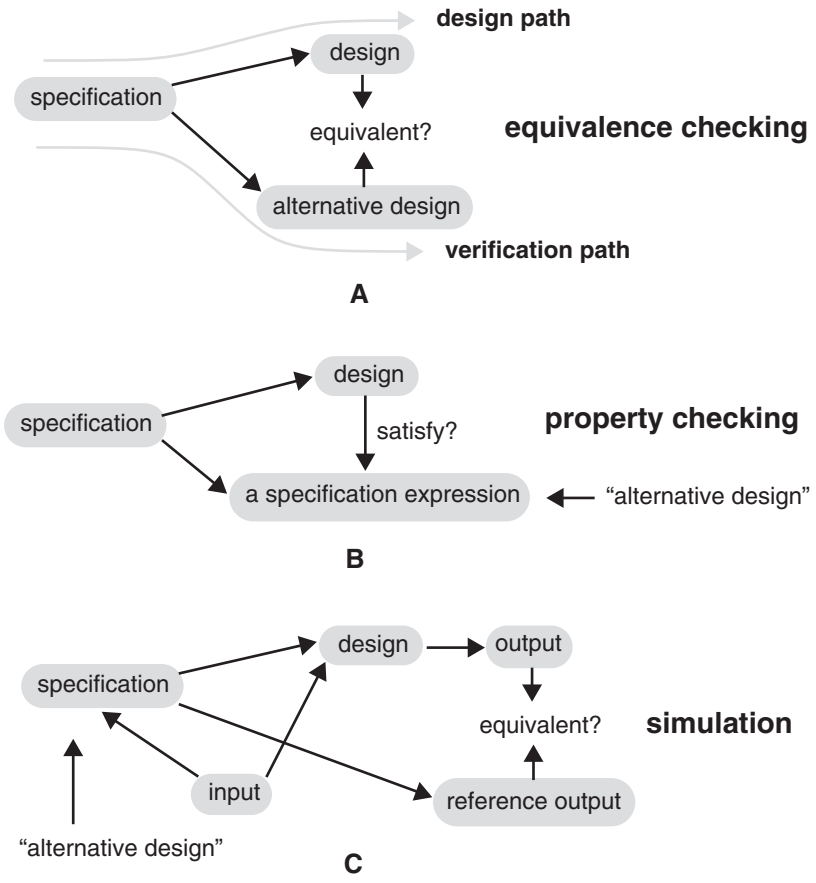

Figure 1.3 The basic principle of design verification. (A) The basic methodology of verification by redundancy. (B) A variant of the basic methodology adapted in model checking. (C) Simulation methodology cast in the form of verification by redundancy.

reside when it has become a product also detects bugs during specification, because the environment dictates how the design should behave and thus serves as a complementary form of design specification. Therefore, verifying the design requirements against the environment is another form of verification through redundancy. Furthermore, some of these types of errors will eventually be uncovered as the design takes a more concrete form. For example, at a later stage of implementation, conflicting requirements will surface as consistencies, and features will emerge as unrealizable given the available technologies and affordable resources.

1.3 Verification Methodology

A sound verification methodology starts with a test plan that details the specific functionality to verify so that the specifications are satisfied. A test plan consists of features, operations, corner cases, transactions, and so forth, such that the completion of verifying them constitutes verifying the specifications. To track progress against a test plan, a "scoreboarding" scheme is used. During scoreboarding, the items in the test plan are checked off when they are completely verified. In reality, it is infeasible to verify a set of specifications completely, in the sense that all bugs are uncovered. Thus, a measure of verification quality is desirable. Some commonly used coverage metrics are functional coverage and code coverage. Functional coverage approximates the percentage of functionality verified, whereas code coverage measures the percentage of code simulated. Neither functional coverage nor code coverage directly corresponds to how thoroughly bugs are eliminated. To compensate this shortcoming, bug rate and simulation cycles are widely used to gauge bug absence.

Besides a test plan, a verification methodology has to decide what language or languages should be used during verification. Verilog or VHDL are common design languages. Verification code often demands a language of its own, because the code is usually of a higher level and does not have to conform to strict coding style guidelines like design code. For example, arithmetic operators occur often in verification code but seldom in design code. On the contrary, design code usually has to be able to be synthesized, but verification code does not. Hence, an ideal verification language resembles a software language more than a hardware language. Popular verification languages include Vera, e, C/C++, and Java.

To study verification methodologies further, let's group verification methodologies into two categories: simulation-based and formal method-based verification. The distinguishing factor between the two categories is the existence or absence of vectors. Simulation-based verification methodologies rely on vectors, whereas formal method-based verification methodologies do not. Another way to distinguish simulation-based and formal method based verification is that simulation is input oriented (for example, the designer supplies input tests) and formal method verification is output oriented (for example, the designer supplies the output properties to be verified). There is a hybrid category called *semiformal verification* that takes in input vectors and verifies formally around the neighborhood of the vectors. Because the semiformal methodology is a combination of simulation-based and formal technology, it is not described separately here.

1.3.1 Simulation-Based Verification

The most commonly used verification approach is simulation-based verification. As mentioned earlier, simulation-based verification is a form of verification by redundancy. The

variant or the alternative design manifests in the reference output. During simulation-based verification, the design is placed under a test bench, input stimuli are applied to the test bench, and output from the design is compared with reference output. A test bench consists of code that supports operations of the design, and sometimes generates input stimuli and compares the output with the reference output as well. The input stimuli can be generated prior to simulation and can be read into the design from a database during simulation, or it can be generated during a simulation run. Similarly, the reference output can be either generated in advance or on the fly. In the latter case, a reference model is simulated in lock step with the design, and results from both models are compared.

Before a design is simulated, it runs through a linter program that checks static errors or potential errors and coding style guideline violations. A linter takes the design as input and finds design errors and coding style violations. It is also used to glean easy-to-find bugs. A linter does not require input vectors; hence, it checks errors that can be found independent of input stimuli. Errors that require input vectors to be stimulated will escape linting. Errors are static if they can be uncovered without input stimuli. Examples of static errors include a bus without a driver, or when the width of a port in a module instantiation does not match the port in the module definition. Results from a linter can be just alerts to potential errors. A potential error, for example, is a dangling input of a gate, which may or may not be what the designer intended. A project has its own coding style guidelines enforced to minimize design errors, improve simulation performance, or for other purposes. A linter checks for violations of these guidelines.

Next, input vectors of the items in the test plan are generated. Input vectors targeting specific functionality and features are called *directed tests*. Because directed tests are biased toward the areas in the input space where the designers are aware, bugs often happen in areas where designers are unaware; therefore, to steer away from these biased regions and to explore other areas, pseudorandom tests are used in conjunction with directed tests. To produce pseudorandom tests, a software program takes in seeds, and creates tests and expected outputs. These pseudorandomly generated tests can be vectors in the neighborhood of the directed tests. So, if directed tests are points in input space, random tests expand around these points.

After the tests are created, simulators are chosen to carry out simulation. A simulator can be an event-driven or cycle-based software or hardware simulator. An event simulator evaluates a gate or a block of statements whenever an input of the gate or the variables to which the block is sensitive change values. A change in value is called an *event*. A cycle-based simulator partitions a circuit according to clock domains and evaluates the subcircuit in a clock domain once at each triggering edge of the clock. Therefore, event count

affects the speed a simulator runs. A circuit with low event counts runs faster on event-driven simulators, whereas a circuit with high event counts runs faster on cycle-based simulators. In practice, most circuits have enough events that cycle-based simulators outperform their event-driven counterparts. However, cycle-based simulators have their own shortcomings. For a circuit to be simulated in a cycle-based simulator, clock domains in the circuit must be well defined. For example, an asynchronous circuit does not have a clear clock domain definition because no clock is involved. Therefore, it cannot be simulated in a cycle-based simulator.

A hardware simulator or emulator models the circuit using hardware components such as processor arrays and field programmable gate arrays (FPGAs). First, the components in a hardware simulator are configured to model the design. In a processor array hardware simulator, the design is compiled into instructions of the processors so that executing the processors is tantamount to simulating the design. In an FPGA-based hardware acceleration, the FPGAs are programmed to mimic the gates in the design. In this way, the results from running the hardware are simulation results of the design. A hardware simulator can be either event driven or cycle based, just like a software simulator. Thus, each type of simulator has its own coding style guidelines, and these guidelines are more strict than those of software simulators. A design can be run on a simulator only if it meets all coding requirements of the simulator. For instance, statements with delays are not permitted on a hardware simulator. Again, checking coding style guidelines is done through a linter.

The quality of simulating a test on a design is measured by the coverage the test provides. The coverage measures how much the design is stimulated and verified. A coverage tool reports code or functional coverage. Code coverage is a percentage of code that has been exercised by the test. It can be the percentage of statements executed or the percentage of branches taken. Functional coverage is a percentage of the exercised functionality. Using a coverage metric, the designer can see the parts of a design that have not been exercised, and can create tests for those parts. On the other hand, the user could trim tests that duplicate covered parts of a circuit.

When an unexpected output is observed, the root cause has to be determined. To determine the root cause, waveforms of circuit nodes are dumped from a simulation and are viewed through a waveform viewer. The waveform viewer displays node and variable values over time, and allows the user to trace the driver or loads of a node to determine the root cause of the anomaly.

When a bug is found, it has to be communicated to the designer and fixed. This is usually done by logging the bug into a bug tracking system, which sends a notification to the owner of the design. When the bug is logged into the system, its progress can be tracked. In

a bug tracking system, the bug goes through several stages: from opened to verified, fixed, and closed. It enters the opened stage when it is filed. When the designer confirms that it is a bug, he moves the bug to the verified stage. After the bug is eradicated, it goes to the fixed stage. Finally, if everything works with the fix, the bug is resolved during the closed stage. A bug tracking system allows the project manager to prioritize bugs and estimate project progress better.

Design codes with newly added features and bug fixes must be made available to the team. Furthermore, when multiple users are accessing the same data, data loss may result (for example, two users trying to write to the same file). Therefore, design codes are maintained using revision control software that arbitrates file access to multiple users and provides a mechanism for making visible the latest design code to all.

The typical flow of simulation-based verification is summarized in Figure 1.4. The components inside the dashed enclosure represent the components specific to the simulation-based methodology. With the formal verification method, these components are replaced by those found in the formal verification counterparts.

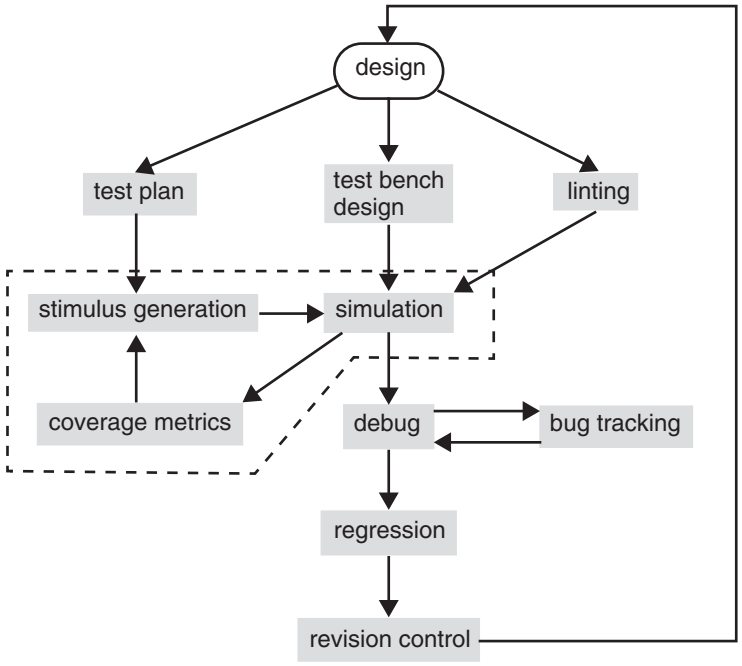

Figure 1.4 Flow of simulation-based verification

1.3.2 Formal Method-Based Verification

The formal method-based verification methodology differs from the simulation-based methodology in that it does not require the generation of test vectors; otherwise, it is similar. Formal verification can be classified further into two categories: equivalence checking and property verification.

Equivalence checking determines whether two implementations are functionally equivalent. Checking equivalence is infeasible using a simulation-based methodology because there are practically an infinite number of vectors in the input space. During formal verification, the decision from a equivalence checker is clear-cut. However, industrial equivalence checkers have not yet reached the stage of turnkey solution and often require the user to identify equivalent nodes in the two circuits to limit the input search space for the checkers. These user-identified equivalent nodes are called *cut points*. At times, a checker can infer equivalent nodes from the node names.

Two basic approaches are behind equivalence checking. The first approach searches the input space in a systematic way for an input vector or vectors that would distinguish the two circuits. This approach, called SAT (satisfiability), is akin to automatic test pattern generation algorithms. The other approach converts the two circuits into canonical representations, and compare them. A canonical representation has the characteristic property that two logical functions are equivalent if and only if their respective representations are isomorphic—in other words, the two representations are identical except for naming. A reduced ordered binary decision diagram is a canonical representation. Binary decision diagrams of two equivalent functions are graphically isomorphic.

Equivalence checking is most widely used in the following circumstances:

1. Compare circuits before and after scan insertion to make sure that adding scan chains does not alter the core functionality of the circuit.
2. Ensure the integrity of a layout versus its RTL version. This is accomplished by first extracting from the layout a transistor netlist and comparing the transistor netlist with the RTL version.
3. Prove that changes in an engineering change order (ECO) check-in are restricted to the scope intended. This is done by identifying the changed regions in the circuits.

If an equivalence check fails, the checker generates an input sequence of vectors that, when simulated, demonstrates the differences between the two circuits. From the waveforms, the user debugs the differences. A failure can result from a true error or an unintended boundary condition. An unintended boundary condition arises when the scan circuitry is not disabled. Then a checker will generate a sequence of vectors that runs the postscan circuit in test mode while running the prescan circuit in normal operating mode. Surely these two

circuits are not equivalent under this input sequence. Therefore, when comparing a prescan circuit and a postscan circuit, the postscan circuit must be configured in normal operating mode.

The other type of formal verification is property checking. Property checking takes in a design and a property which is a partial specification of the design, and proves or disproves that the design has the property. A property is essentially a duplicate description of the design, and it acts to confirm the design through redundancy. A program that checks a property is also called a *model checker*, referring the design as a computational model of a real circuit. Model checking cast in the light of the basic verification principle is visualized in Figure 1.3 (B).

The idea behind property checking is to search the entire state space for points that fail the property. If a such point is found, the property fails and the point is a counterexample. Next, a waveform derived from the counterexample is generated, and the user debugs the failure. Otherwise, the property is satisfied. What makes property checking a success in industry are symbolic traversal algorithms that enumerate the state space implicitly. That is, it visits not one, but a group of points at a time, and thus is highly efficient.

Even though symbolic computation has taken a major stride in conquering design complexity, at the time of this writing only the part of the design relevant to the property should be given to a property verifier, because almost all tools are incapable of automatically carving out the relevant part of the design and thus almost certainly will run into memory and runtime problems if the entire design is processed.

Furthermore, the power and efficiency of a property verifier is highly sensitive to the property being verified. Some properties can be decided readily whereas others will never finish or may not even be accepted by the tool. For example, some property verifiers will not accept properties that are unbound in time (for example, an acknowledgment signal will eventually be active). A bound property is specified within a fixed time interval (for example, an acknowledgment signal will be active within ten cycles).

A failure can result from a true design bug, a bug in the property or unintended input, or state configurations. Because a property is an alternative way of expressing the design, it is as equally prone to errors as the design. A failure can be caused by unintended input or state configurations. If the circuit under verification is carved out from the design, the input waveforms to the circuit must be configured to be identical to them when the circuit is embedded in the design. Failure to do so sends unexpected inputs (for example, inputs that would not occur when the circuit is a part of the design) to the circuit, and the circuit may fail over these unexpected inputs because the circuit has not been designed to handle them. To remove unintended configurations from interfering with property verification,

the inputs or states are constrained to produce only the permissible input waveforms. Underconstraining means that input constraining has not eliminated all unexpected inputs, and this can trigger false failures. Overconstraining, which eliminates some legal inputs, is much more dangerous because it causes false successes and the verifier will not alert the user.

Some practical issues with property verifiers include long iteration times in determining the correct constraining parameters, and debugging failures in the properties and in the design. Because only a portion of the design is given to the property verifier, the environment surrounding that portion has to be modeled correctly. Practical experience shows that a large percentage of time (around 70 percent) in verification is spent getting the correct constraints. Second, debugging a property can be difficult when the property is written in a language other than that of the design (for instance, the properties are in SystemVerilog whereas the design is in Verilog). This is because many property verifiers internally translate the properties into finite-state machines in Verilog, and verify them against the design. Therefore, the properties are shown as the generated machine code, which is extremely hard to relate to signals in the design or is hard to interpret its meaning. Finally, debugging failures in a property has the same difficulty as any other debugging.

As an alternative to a state space search in verifying a property, the theorem-proving approach uses deductive methods. During theorem proving, a property is specified as a mathematical proposition and the design, also expressed as mathematical entities, is treated as a number of axioms. The objective is to determine whether the proposition can be deduced from the axioms. If it can, the property is proved; otherwise, the property fails. A theorem prover is less automatic than a model checker and is more of an assistance to the user. The user drives the tool to arrive at the proposition by assembling relevant information and by setting up intermediate goals (in other words, lemmas), whereas the tool attempts to achieve the intermediate goals based on the input data.

Effective use of a theorem prover requires a solid understanding of the internal operations of the tool and a familiarity with the mathematical proof process. Although less automatic, efficient usage of a theorem prover can handle much larger designs than model checkers and requires less memory. Furthermore, a theorem prover accepts more complex properties. For example, a theorem prover can allow the properties written in higher order logic (HOL) whereas almost all model checkers only accept first-order logic and computation tree logic or their variants. HOL is more expressive than first-order logic and enables a concise description of complex properties.

A typical flow of formal verification is shown in Figure 1.5. Remember that the components in this diagram replace the components inside the dashed box of Figure 1.4. Although

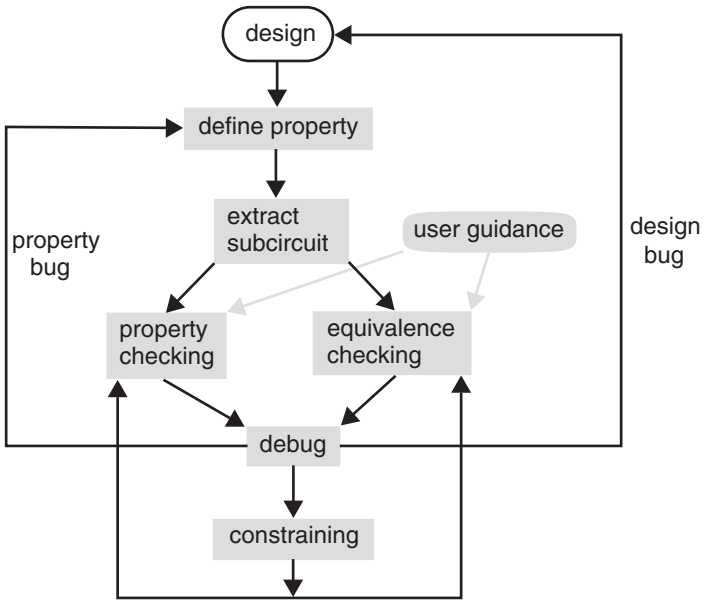

Figure 1.5 A typical flow of formal verification

formal verification does not require test vectors, it does require a test bench that configures the circuit in the correct operational mode, such as input constraining.

1.4 Simulation-Based Verification versus Formal Verification

The most prominent distinction between simulation-based verification and formal verification is that the former requires input vectors and the latter does not. The mind-set in simulation-based verification is first to generate input vectors and then to derive reference outputs. The thinking process is reversed in the formal verification process. The user starts out by stating what output behavior is desirable and then lets the formal checker prove or disprove it. Users do not concern themselves with input stimuli at all. In a way, the simulation-based methodology is input driven and the formal methodology is output driven. It is often more straightforward to think in the input-driven way, and this tendency is reflected in the perceived difficulty in using a formal checker.

Another selling point for formal verification is completeness, in the sense that it does not miss any point in the input space—a problem from which simulation-based verification suffers. However, this strength of formal verification sometimes leads to the misconception

that once a design is verified formally, the design is 100% free of bugs. Let's compare simulation-based verification with formal verification and determine whether formal verification is perceived correctly.

Simulating a vector can be conceptually viewed as verifying a point in the input space. With this view, simulation-based verification can be seen as verification through input space sampling. Unless all points are sampled, there exists a possibility that an error escapes verification. As opposed to working at the point level, formal verification works at the property level. Given a property, formal verification exhaustively searches all possible input and state conditions for failures. If viewed from the perspective of output, simulation-based verification checks one output point at a time; formal verification checks a group of output points at a time (a group of output points make up a property). Figure 1.6 illustrates this comparative view of simulation-based verification and formal verification. With this perspective, the formal verification methodology differs from the simulation-based methodology by verifying *groups* of points in the design space instead of *points*. Therefore, to verify completely that a design meets its specifications using formal methods, it must be further proved that the set of properties formally verified collectively constitutes the specifications. The fact that formal verification checks a group of points at a time makes formal verification software less intuitive and thus harder to use.

A major disadvantage of formal verification software is its extensive use of memory and (sometimes) long runtime before a verification decision is reached. When memory capacity is exceeded, tools often shed little light on what went wrong, or give little guidance to fix

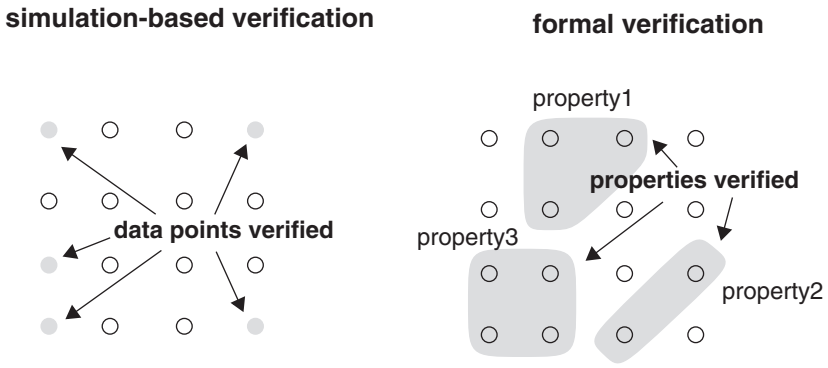

Figure 1.6 An output space perspective of simulation-based verification versus formal verification

the problem. As a result, formal verification software, as of this writing, is applicable only to circuits of moderate size, such as blocks or modules.

1.5 Limitations of Formal Verification

Formal verification, although offering exhaustive checking on properties, by no means guarantees complete functional correctness of design. Many factors can betray the confidence offered by formal verification. These factors include, but are in no way limited to, the following:

1. Errors in specification. An implementation verification requires two versions of a design. Therefore, a bug in the specifications voids the reference model, may escape implementation verification, and may trickle down to implementations. A typical type of specification error is missing specifications.
2. Incomplete functional coverage of specifications. Formal methods verify properties that, of course, can be specifications. In practice, however, it is rare that all specifications are given to a formal checker on the complete design, because of memory and runtime limitations. A partial specification is checked against the relevant portion of the design. Therefore, it leaves the door open to errors resulting from not checking all properties mandated by the specifications and from situations when, for example, a property succeeds on a subcircuit but does not succeed on the whole circuit.
3. User errors. Example user errors are incorrect representation of specifications and overconstraining the design.
4. Formal verification software bugs. Bugs in formal verification software can miss design errors and thus give false confirmation. It is well-known that there is no guarantee for a bug-free software program.

1.6 A Quick Overview of Verilog Scheduling and Execution Semantics

On the assumption that you have a good understanding of Verilog, we will not review the general syntax of Verilog. However, we will review the more advanced features of the language—namely, scheduling and execution semantics—they appear in various places throughout this book. If you are well versed in Verilog, you can skip this section.

1.6.1 Concurrent Processes

Verilog is a parallel hardware descriptive language. It differs from a software language such as C/C++ in that it models concurrency of hardware. Components in a circuit always

execute in parallel, whereas a software program executes one statement at time—in a serial fashion. To model hardware concurrency, Verilog uses two types of data structures: continuous assignment and procedural block. A continuous assignment executes whenever a variable in the right side of the assignment changes. As the name implies, a continuous assignment continuously watches changes of variables on the right side and updates the left-side variables whenever changes occur. This behavior is consistent with that of a gate, which propagates inputs to outputs whenever inputs change. An example of a continuous assignment is shown here. The value of `v` is updated whenever the value of `x`, `y`, or `z` changes:

```
assign v = x + y + z;
```

A procedural block consists of statements called *procedural statements*. Procedural statements are executed sequentially, like statements in a software program, when the block is triggered. A procedure block has a sensitivity list containing a list of signals, along with the signals' change polarity. If a signal on a sensitivity changes according to its prescribed polarity, the procedural block is triggered. On the other hand, if a signal changes that is not on the sensitivity list but is present in the right-hand side of some procedural statements of the block, the procedural statements will not be updated. An example of a procedural block is shown here:

```
always @(posedge c or d) begin
   v = c + d + e;
   w = m - n;
end
```

The signals on the sensitivity list are `c` and `d`. The polarity for `c` is positive edge, and no polarity is specified for `d`, which by default means either positive or negative edge. This block is executed whenever a positive event occurs at `c`. A positive edge is one of the following: a change from 0 to 1, X, or Z; or from X or Z to 1. The block is also triggered when an event occurs at `d`, positive or negative edge. A negative edge is one of the following: a change from 1 to either 0, X, or Z; or from X or Z to 0. Changes on variables `e`, `m`, or `n` will not cause the block to execute because these three variables are not on the sensitivity list.

In a way, a procedural block is a generalization of a continuous assignment: Any variable change on the right side of a continuous assignment triggers evaluation, whereas in a procedural block only changes in the variables on the sensitivity list can trigger evaluation, and the changes must conform to the polarity specification. Any continuous assignment

can be rewritten as a procedural block, but not vice versa. The following procedural block and continuous assignment are equivalent:

```
// continuous assignment
assign v = x + y + z;

// equivalent procedural block
always @(x or y or z)
   v = x + y + z;
```

However, there is no equivalent continuous assignment for the following procedural block, because rising or falling changes are not distinguished in continuous assignment:

```
always @(posedge x)
   y = x;
```

1.6.2 Nondeterminism

When multiple processes execute simultaneously, a natural question to ask is how these processes are scheduled. When multiple processes are triggered at the same time, the order in which the processes are executed is not specified by Institute of Electrical and Electronics Engineers (IEEE) standards. Thus, it is arbitrary and it varies from simulator to simulator. This phenomenon is called *nondeterminism*. Let's look at two common cases of nondeterminism that are caused by the same root cause but that manifest in different ways.

The first case arises when multiple statements execute in zero time, and there are several legitimate orders in which these statements can execute. Furthermore, the order in which they execute affects the results. Therefore, a different order of execution gives different but correct results. Execution in zero time means that the statements are evaluated without advancing simulation time. Consider the following statements:

```
always @(d)
   q = d;

assign q = ~d;
```

These two processes, one procedural block and one continuous assignment, are scheduled to execute at the same time when variable `d` changes. Because the order in which they execute is not specified by the IEEE standard, two possible orders are possible. If the `always` block is evaluated first, variable `q` is assigned the new value of `d` by the `always`

block. Then the continuous assignment is executed. It assigns the complement of the new value of d to variable q. If the continuous assignment is evaluated first, q gets the complement of the new value of d. Then the procedural assignment assigns the new value (uncomplemented) to q. Therefore, the two orders produce opposite results. This is a manifestation of nondeterminism.

Another well-known example of nondeterminism of the first type occurs often in RTL code. The following D-flip–flop (DFF) module uses a blocking assignment and causes nondeterminism when the DFFs are instantiated in a chain:

```
module DFF(clk, q, d);
input clk, d;
output q;
reg q;

always @(posedge clk)
   q = d; // source of the problem

endmodule

module DFF_chain;

DFF dff1(clk, q1, d1); // DFF1
DFF dff2(clk, q2, q1); // DFF2

endmodule
```

When the positive edge of clk arrives, either DFF instance can be evaluated first. If dff1 executes first, q1 is updated with the value of d1. Then dff2 executes and makes q2 the value of q1. Therefore, in this order, q2 gets the latest value of d1. On the other hand, if dff2 is evaluated first, q2 gets the value of q1 before q1 is updated with the value of d1. Either order of execution is correct, and thus both values of q2 are correct. Therefore, q2 may differ from simulator to simulator.

A remedy to this race problem is to change the blocking assignment to a nonblocking assignment. Scheduling of nonblocking assignments samples the values of the variables on the right-hand side at the moment the nonblocking assignment is encountered, and assigns the result to the left-side variable at the end of the current simulation time (for example, after all blocking assignments are evaluated). A more in-depth discussion of scheduling order is presented in the section "Scheduling Semantics." If we change the

blocking assignment to a nonblocking assignment, the output of `dff1` always gets the updated value of `d1`, whereas that of `dff2` always gets the preupdated value of `q1`, regardless of the order in which they are evaluated.

Nonblocking assignment does not eliminate all nondeterminism or race problems. The following simple code contains a race problem even though both assignments are nonblocking. The reason is that both assignments to variable `x` occur at the end of the current simulation time and the order that `x` gets assigned is unspecified in IEEE standards. The two orders produce different values of `x` and both values of `x` are correct:

```
always @(posedge clk)
begin
   x <= 1'b0;
   x <= 1'b1;
end
```

The second case of nondeterminism arises from interleaving procedural statements in blocks executed at the same time. When two procedural blocks are scheduled at the same time, there is no guarantee that all statements in a block finish before the statements in the other block begin. In fact, the statements from the two blocks can execute in any interleaving order. Consider the following two blocks:

```
always @(posedge clk) // always block1
begin
   x = 1'b0;
   y = x;
end

always @(posedge clk) // always block2
begin
   x = 1'b1;
end
```

Both `always` blocks are triggered when a positive edge of `clk` arrives. One interleaving order is

```
x = 1'b0;
y = x;
x = 1'b1;
```

In this case, `y` gets 0.

Another interleaving order is

```
x = 1'b0;
x = 1'b1;
y = x;
```

In this case, `y` gets 1.

There is no simple fix for this nondeterminism. The designer has to reexamine the functionality the code is supposed to implement and, more often than not, nondeterminism is not inherent in the functionality but is introduced during the implementation process.

1.6.3 Scheduling Semantics

Having discussed nondeterminism, it is natural for us to examine the scheduling semantics in Verilog (that is, the order in which events are scheduled to execute). Events at simulation time are stratified into five layers of events in the order of processing:

1. Active
2. Inactive
3. Nonblocking assign update
4. Monitor
5. Future events

Active events at the same simulation time are processed in an arbitrary order. An example of an active event is a blocking assignment. The processing of all the active events is called a *simulation cycle*.

Inactive events are processed only after all active events have been processed. An example of an inactive event is an explicit zero-delay assignment, such as `#0 x = y`, which occurs at the current simulation time but is processed after all active events at the current simulation time have been processed.

A nonblocking assignment executes in two steps. First, it samples the values of the right-side variables. Then it updates the values to the left-side variables. The sampling step of a nonblocking assignment is an active event and thus is executed at the moment the nonblocking statement is encountered. A nonblocking assign update event is the updating step of a nonblocking assignment and is executed only after both active and inactive events at the current simulation time have been processed.

Monitor events are the execution of system tasks `$monitor` and `$strobe`, which are executed as the last events at the current simulation time to capture steady values of variables at the current simulation time. Finally, events that are to occur in the future are the future events.

1.7 Summary

In this chapter we defined design verification as the process of ensuring that a design meets its specifications—the reverse of the design process. We discussed the general verification principle through redundancy, and simulation-based and formal verification methodologies, and then illustrated typical flows of these two methodologies. We then contrasted simulation and formal verification, and emphasized their input- and output-oriented nature. We viewed simulation-based verification as operating on a point basis whereas formal verification operates on a subspace basis. Next, we listed some limitations of formal verification—in particular, the inability to detect errors in the specification, incomplete functional coverage caused by verifying subcircuits, user errors, and software bugs in the verification tool. As a refresher of Verilog, we reviewed scheduling and execution semantics, during which we discussed the concurrent nature of the Verilog language and its inherent nondeterminism.