CHAPTER

1

INTRODUCTION

Ensuring functional correctness on RTL designs continues to pose one of the greatest challenges for today's ASIC and SoC design teams. Rooted in that challenge is the goal to shorten the verification cycle. This requires new design and verification techniques.

In this book, we address the functional correctness challenge within a contemporary verification flow that relies on an *assertion-based methodology* and *property checking* techniques. The methodology we propose enables designers to meet today's aggressive time-to-market goals, while providing higher confidence in functional correctness. It benefits dynamic verification (that is, simulation), while providing a seamless path to static (formal) verification.

This chapter provides a general introduction to property checking and assertion techniques. We present the benefits associated with assertion-based design and address the many fallacies associated with their use. Finally, we discuss the importance of a specification-driven methodology related to design and implementation.

1.1 Property checking

So, what is *property checking*? In general, you can think of a design property as a proposition of expected design behavior (that is, *design intent*). The proposition *only a single tri-state driver on the memory bus is enabled at a time* is an example of a property for a specific tri-state bus within a design. We can then *assert* that the property must hold (that is, evaluate *true*) for our design, and

check our assertion using a dynamic (simulation) or static (formal) verification tool. Other examples of design implementation properties include:

- bus contention
- bus floating
- set/reset conflicts
- RTL (Verilog) “don't care” checks
- full case or parallel case assumptions
- clock-domain crossing checks

Emerging static verification tools automatically extract many design properties through structural analysis of the RTL model. These tools attempt to exhaustively verify these properties using formal techniques (see Section 2.5 Assertions and formal verification on page 44 for a discussion on the *state explosion* problem). When successful, this enables the engineer to verify (or debug) many design properties early in the design cycle—without the need to create testbenches and test vectors. However, designs also include properties that are not as obvious as these examples and that cannot be automatically extracted.

Some properties reflect standard design structures used in standard manners. For example, a queue structure normally operates within its bounds; that is, it neither overfills (overflow) nor removes invalid information (underflow). When a design engineer uses this framework to claim that one of the implemented queue structures can never overflow nor underflow, this is a user-defined property that requires validation. However, until recently, the industry lacked a standard way of specifying RTL user-defined properties that multiple verification tools could recognize and use. In this book, we demonstrate assertion techniques that specify RTL user-defined properties using a subset of the following Accellera standards:

- Open Verification Library (OVL)
- PSL Property Specification Language
- SystemVerilog assertion constructs

1.2 Verification techniques

Traditionally, engineers verify the design's implementation against its requirements using a *black-box* testing approach. In other words, the engineer creates a model of the design written in a hardware description language (for example, Verilog [IEEE 1364-2001] or VHDL [IEEE 1076-1993]). The engineer then creates a testbench, which includes or instantiates a copy of the model or *device under verification* (DUV). Historically,

testbenches would read a vector file as input—and apply the vectors to the DUV cycle-by-cycle. The DUV output results were then compared against a reference model. The ability to directly "observe and validate" came later with the development of *self-checking testbenches*. Recently, testbenches have become complex verification environments often built with a hardware verification language (HVL) that combines:

- automatic vector generation,
- output response validation, and
- coverage analysis.

The specification defines the legal values or sequences of values permitted by the DUV's input and output ports (that is, a black-box view of the design).

One problem encountered when using a black-box testing approach is that the DUV might exhibit improper internal behavior, such as a state machine violating its one-hot property, but still have a proper output response (at a specific, observed point in time). In cases such as this, a design error exists, but it will be missed because it is not directly observable on the output ports. This might be due to the current set of input stimulus, which, when applied to the DUV, impedes the internal problem's value from propagating to an output port. Given a different set of input stimulus (or if the simulation were to run a few clocks longer), the internal error might be observable. However, validating all internal properties of a design using black-box testing techniques is impractical, particularly as design size increases.

Alternatively, *white-box* testing can be implemented to validate properties of a design. This technique adds assertions that monitor internal points within the DUV, and results in an increase in observable behavior during testing. For example, using the DUV described above, we can add an assertion (or monitor) to the design to directly observe and validate whether a state machine is always one-hot. Thus, if the one-hot property is violated, the error is instantly isolated to the faulty internal point. This overcomes the problem associated with black-box testing, which is the possibility of missing an internal error (for a given input stimulus) by observing only the DUV output responses.

1.3 What is an assertion?

In general, an *assertion* is a statement about a design's *intended behavior* (that is, a property), which must be verified. Unlike design code, an assertion statement does not contribute in any

form to the element being designed.[1] Its sole purpose is to ensure consistency between the designer's *intention,* and what is *created.*

Consider the following analogy:

A designer issues the print command for a report, walks to the printer expecting to find 11 pages, but finds only 8. *What happened to the missing pages?*

The printer displays the following message:

> ERROR - PAPER JAM AT SECTIONS 3, 7.

In this case, the printer assertion triggered when it detected a difference between the expectation of the user and the creation from the printer. It notified the user of the error and where the potential problems occurred. This analogy demonstrates three key features of assertions:

- *error detection*
- *error isolation*
- *error notification*

Each of these features is discussed in detail throughout the remainder of this book.

1.3.1 A historical perspective

Design verification is a process used to ensure that a circuit or system model (that is, the implementation) behaves according to a given set of requirements (that is, the specification). Typically, the requirements consist of a natural language list of assertions, each of which must be verified. In fact, over 50 years ago, Alan Turing [1949] made the following observation concerning partitioning a large verification problem into set of assertions:

> *How can one check a large routine in the sense of making sure that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily flows.*

Over 30 years ago, Floyd [1967] and Hoare [1969] proposed the concept of using formalisms (that is, property specification—or assertions) to specify and reason about software programs. Moreover, software engineers have long used assertions within

1. Note that assertions can be used to constrain the synthesis process. For example, the *full_case* and *parallel_case* synthesis directives are really assertions.

their code to check for consistency (for example, checking for null pointers or illegal array index ranges).

design by contract

More recently, new programming languages (such as Eiffel) have emerged that are based on an underlying theory known as *Design by Contract.* Eiffel "views the construction of a software system as the fulfillment of many small and large contracts between clients and suppliers" [Meyer 1992]. Components written in Eiffel specify pre-conditions (a form of assertions) that the user must satisfy to use the component in an acceptable and reliable manner. Then, post-conditions (also assertions) are specified as a definition of what the component will do and the properties the results will satisfy. The post-conditions assume that the component pre-conditions are satisfied. Systems can become very robust when the operating conditions (pre-conditions) are tightly controlled.

RTL assertions

RTL assertions written for hardware interfaces can achieve the same effect that Eiffel components obtain from their assertions: a controlled environment for interface usage and an understood set of expectations. Consider an intellectual component purchased as intellectual component (IP) from another company. It consists of the component in a usable form, and an instruction manual on how to use the component. If the component uses assertions to define its interface, the component will be used more successfully than if assertions are not present. Additionally, the support effort required by the group or company supplying the IP is reduced because the user is told when they are using the IP incorrectly.

In today's design and verification environment, emerging hardware verification languages include various forms of assertion library templates. Furthermore, hardware description languages (HDLs) include constructs that support assertion specification. For instance, VHDL [IEEE 1076-1993] includes a keyword **`assert`**, which permits designers to add embedded checkers to model description code. This language construct, as shown in Example 1-1, ensures that any user-specified condition (that is, a Boolean expression) always evaluates to TRUE.

Example 1-1 VHDL assertion syntax

```
[label] assert VHDL_expression
   [report message]
    [severity level]
```

For the VHDL assertion construct, an error is reported when the VDHL_expression evaluates to FALSE. The assertion's optional report clause specifies a message string that will be included in error messages generated by the assertion. In the absence of a report clause for a given assertion, the string "Assertion violation" is the default value for the message string. The VHDL assertion's

optional severity clause specifies a severity level associated with the assertion. In the absence of a severity clause for a given assertion, the default value of the severity level is ERROR.

Example 1-2 VHDL example to check for inverted signals

```
ASSERT ((a = '1') XOR (b = '1'))
  REPORT "error: A & B must be inverted"
    SEVERITY 0;
```

Unlike VHDL, Verilog [IEEE 1364-2001] does not contain an assertion construct. However, checks can be coded in an explicit fashion as show in Example 1-3.

Example 1-3 Verilog example to check for equality

```
always (a or b) begin
  if (a ^ b) begin // not equal
    $display("error: A&B must be equal: %m");
    $finish;
  end
end
```

Many present-day commercial tools provide their own proprietary HDL assertion solutions. Most recently (and perhaps most importantly) the Accellera standards organization has engaged in efforts to unify the industry with a standard for an HDL assertion specification. [Fitzpatrick et al. 2002].

1.3.2 Do assertions really work?

Assertions have been used by many prominent companies, including:

- Cisco Systems, Inc.
- Digital Equipment Corporation
- Hewlett-Packard Company
- IBM Corporation
- Intel Corporation
- LSI Logic Corporation
- Motorola, Inc.
- Silicon Graphics, Inc.

Designers from these companies describe their success with methodologies that incorporate assertions as follows:

- **34%** of all bugs were found by assertions on DEC Alpha 21164 project [Kantrowitz and Noack 1996]
- **17%** of all bugs were found by assertions on Cyrix M3(p1) project [Krolnik 1998]
- **25%** of all bugs were found by assertions on DEC Alpha 21264 project - The DEC 21264 Microprocessor [Taylor et al. 1998]
- **25%** of all bugs were found by assertions on Cyrix M3(p2) project [Krolnik 1999]
- **85%** of all bugs were found using OVL assertions on HP [Foster and Coelho 2001]

From these papers, a common theme emerges: When designers use assertions as a part of the verification methodology, they are able to detect a significant percentage of design failures. Thus, assertions not only enhance a verification methodology; they are an integral component. Assertions are typically written to describe design assumptions or a potential corner case involving a lower-level implementation detail. This complements traditional verification methods, which typically focus on higher levels of abstraction (for example, bus transactions) and rarely attempt to verify specific implementation details (for example, a specific state machine is one-hot).

On the Cyrix M3(p2) (3rd gen x86 processor) project sited above, 750 bugs were identified prior to adding assertions. However, the week *after* a significant number of assertions were added to the design, the verification team experienced a three-fold increase in its bug reporting rate. In fact, fifty percent of all remaining bugs were identified through assertions. This represented 25% of all bugs found on the project.

1.3.3 What are the benefits of assertions?

This section explores the benefits of using assertions and their tremendous impact on increasing design quality while reducing the time-to-market and verification costs.

Improving observability

Fundamental to understanding the benefits of using assertions is understanding the concept of *observability*. In a traditional verification environment, a testbench is created to generate stimulus, which is applied to the design model (that is, *design under verification* or DUV). In addition to generating input stimulus, the testbench validates proper output behavior by

monitoring (that is, observing) the DUV's output ports. In order to identity a design error using this approach, the following conditions must hold:

1. Proper input stimulus must be generated to activate (that is, sensitize) a bug,
2. Proper input stimulus must be generated to propagate all effects resulting from the bug to an output port.

It is possible, however, to set up a condition where the input stimulus activates a design error that doesn't propagate to an observable output port. In these cases, the first condition cited above applies; however, the second condition is absent.

A benefit of assertions embedded in the code is that they increase the *observability* within the design. In this way, the verification environment no longer depends on the second condition listed above to identify bugs. Thus, any improper or unexpected behavior can be caught closer to the source of the bug.

The experiences of the companies cited above (and others) show that embedded assertions, when added to an existing verification environment, identify problems that previously were not identified. In fact, both the DEC Alpha team [Kantrowitz and Noack 1996] and Cyrix M3 team [Krolnik 1998] demonstrated that when they added assertions after a point when they had assumed simulation was complete, they found additional bugs using the same set of tests that previously passed.

Reducing debug time

isolate bugs

As we previously stated, assertions improve observability, thus enabling us to find bugs exactly when (that is, at a specific time) and where (that is, at a specific location) they occur. Conversely, traditional verification methods do not detect bugs directly in (or close to) the logic that generated them; they typically detect a bug multiple clocks after its occurrence and at some other distant location in the design. The problem with this approach is that it typically requires that multiple designers back-trace multiple paths (or blocks) within the design. This consumes many debug hours before the problematic code is finally identified. When a team is unsure where the bug originated, unpredictable delays ensue as teams pass a failure between several designers for analysis, involve more individuals, and possibly engage in "pointing fingers" and throwing the problem to one another. Isolating bugs closer to the actual source provides a huge time savings and reduces the total resources required to isolate the bug. Thus, it improves projects' time-to-market goals.

The following case illustrates how assertions help isolate bugs by detecting them in the logic that generated them. In this case, the

Cyrix M3 project experienced a test failure that appeared as a time-out to complete an instruction. The flow of failure analysis follows:

- Designer A of the completion logic determined that the memory operation portion of the instruction did not complete.
- Designer B of the data cache controller determined that address translation for the operation was wrong.
- Designer C of the address translation controller indicated the translation was successful and passed it back to designer B.
- Designer B reviewed the simulation, and then realized that he received a wrong translation earlier, and passed the problem back to designer C.
- Designer C reviewed the problem, and then noticed that his code had returned two translations for the preceding translation request from designer B's code.
- Designer B then explained that the extra translation caused the time-out

The actual situation described above occupied most of the day for these three engineers. If designer B had written an assertion to ensure each request was followed by only one completion, the failure would have been identified closer to the problem (the previous instruction) and the actual unit (the translation unit). And, the problem would not have drawn in two of the three engineers. After adding the assertion, the engineers easily fixed the logic and quickly validated the new code.

Improving integration through correct usage checking

check interfaces

Assertions also provide benefits when developing and integrating intellectual property (IP) components. A design team initially validates the IP with a given set of functional constraints and inserts boundary assertions to monitor correct interface communication during integration verification. These boundary assertions form *verifiable contracts* between the IP provider and the IP integrator through *correct usage detection,* a form of self-checking code. When a boundary assertion identifies incorrect usage, it relieves the IP provider of the burden of debugging someone else's design, while enabling the IP user to quickly identify code that doesn't satisfy the IP's specified constraints. In this respect, the IP code is self-checking. That is, it relies on assertions to identify the source of bugs that occur along the input and output boundaries of the IP code.

Improving verification efficiency

find bugs faster

Saving time is perhaps the most significant benefit designers realize with assertions. Experience demonstrates that assertions can save up to 50 percent of debug time, thus reducing overall time-to-market [Abarbanel et al. 2000]. Design teams save debug time when an engineer does not have to backtrack through large simulation trace files and multiple blocks of logic to identify the exact location of the design failure.

work at all times

Unlike conventional debugging processes involving a designer, assertions embedded in the RTL source code are always monitoring for valid (or invalid) behavior. For example, in a conventional debug process, the designer typically goes through the following steps:

- examines a failing testcase,
- identifies the problem,
- fixes the problem, and then
- validates the fix by re-running the testcase that previously failed.

During this process, if designers discover another anomaly associated with the original failing circuit, they will fix it as well. But after the designers find a *fix* for a specific failing testcase (and any other anomalies that emerged in the course of validating the fix), they generally stop looking for other corner cases associated with the bug and move on to the next problem. Unfortunately, there could be a different simulation pattern or sequence, which has not yet been covered, that would identify another corner case associated with the original bug.

Conversely, when designers add assertions to the RTL source, they avoid this situation because assertions never stop monitoring the design for invalid behavior and can help trap many corner case problems during future simulation runs. Hence, a new set of test vectors applied to the design in the future might uncover additional problems associated with a bug they presumed they had fixed.

work with all tools

The assertion-based verification methodology we propose permits the designer to specify assertions in a single form, which then is leveragable across an entire suite of verification tools. In other words, we claim that engineers should only have to specify an assertion once (that is, one way) whether they choose to target the assertions with a custom verification tool, a standard RTL testbench, a commercial simulator, a semi-formal tool, or a formal verification tool.

facilitate formal analysis

Formal specification describing architectural consideration, as well as consistency in protocol design prior to implementation, can be verified using various formal techniques. Once RTL

implementation begins, formal technology can be used to explore the design space around assertions within the implementation, further increasing confidence in the final design's ability to function correctly when built. For example, Bentley [2001] reported that 400 bugs were identified and fixed by formally verifying a large set of assertions on a recent Intel Pentium project prior to silicon.

Improving communication through documentation

specify correct behavior unambiguously

In addition to finding bugs, assertions encapsulated in the RTL provide an excellent form of documentation. For instance, a designer may add an assertion to an interface that states the following expected behavior: *a bus request must be followed by a bus grant within five clock cycles.* By adding an assertion for this property, the engineer documents an aspect of the design in a form that is self-checking and easy to convey to other engineers.

Other engineers can review the assertions to understand the low-level specifics of how to interface with another block. Furthermore, assertions formally document protocols, interfaces, and assumptions in an unambiguous form that clarifies a designer's interpretation of the specification and design intent.

1.3.4 Why are assertions not used?

If assertions are such a great enhancement for verification, why aren't engineers using them more extensively in the design process? This section lists some common arguments that are put forth by teams that are not using assertions and identifies the fallacies in reasoning.

- *Where am I going to find time to write assertions? I don't even have time to write comments in my code!*

  Based on our studies and interviews with multiple projects and engineers, the overhead in writing assertions can amount to anywhere between one and three percent extra time added to the *RTL coding phase* of a design project (note that the RTL coding phase is only one of many phases within a project). In general, the overhead is very minor (that is, closer to one percent). Why? Because, regardless whether you write comments or add assertions prior to or during RTL coding, you are already analyzing and thinking about correct or expected behavior of your design. In other words, you are already considering:

- What is the valid interface behavior?
- What are the legal states for my FSM?
- What are the boundary conditions on my queue controller?

By adding assertions, you are formalizing your thought process in a form that is verifiable. In fact, many design errors are avoided prior to RTL coding and verification through this more systematic approach to design and analysis.

- *I have to get my design working first. If there is time later, then I'll add assertions.*

  This argument translates to: "Assertions will not save me (us) any time". Yet, experience demonstrates that assertions can save up to 50 percent of debug time [Abarbanel et al. 2000]. These people will change their position when they find that the assertions other engineers used in their blocks effectively isolate failures. When they save debug time just by isolating failures that originate outside their block, they will be convinced of the value of including assertions in their own blocks.

- *I will spend more time debugging the assertions than debugging my code; therefore, they are a waste of time.*

  Compare this sentiment to developing testbenches. Does this mean that creating testbenches to verify the design is useless—since the testbenches might also contain bugs? Certainly not, and the same is true with assertions (that is, assertions could contain bugs, but that does not render the practice useless). As with most new experiences, using assertions involves a learning curve. As engineers gain experience with assertions, they recognize what constitutes a correct assertion and spend less time debugging assertions. If engineers are spending time debugging assertions, it is because they did not completely express the assertion, or they did not fully understand some subtle aspect of the design. In fact, the analysis process that takes place while specifying assertions quite often uncovers complex bugs, and this occurs prior to any form of verification.

- *I can't think of any assertions to put in my code. There are no places for them.*

  Designers who say they *can't see any potential errors in their code* will be spending a lot of time rewriting their design code. Experienced designers recognize that typographical, transcription, and design errors all contribute to functional problems that must be addressed. Inexperienced designers must be taught how to see the potential for problems. Then, they will see that they can use assertions to naturally check for correctness.

- *The assertions slow down simulations. They waste time.*

During the post-mortem review for a Convex Computer Corporation project[2], a survey was given to a design team with the following question: "If we had computers that could simulate the design 10x faster than today, would that help you debug faster?" One astute designer responded, "No! We run all the tests overnight and I arrive in the morning with a list of failures I can't work through in one day." This illustrates that the debugging process is where efficiency improvements are required. So, although assertions do have an overhead, that overhead is comparable to monitoring the design for correctness. The Cyrix M3 project and the HP ASIC project found that assertions produced an overhead in the range of 15-30 percent [Krolnik 1998] [Foster and Coelho 2001], depending on the number of assertions and amount of monitoring. For example, Cisco reported a 15 percent increase in simulation runtime on a large ASIC project containing 10,656 OVL assertions [Smith 2002].

- *It is difficult to debug with assertions. It's not possible to fail an assertion and continue debugging. [Borland 2002]*

  This is actually a critique of the assertion methodology. A good assertion methodology has controls that allow simulations to define an error threshold and controls to determine what action (continue simulation, terminate the simulation, stop, debug) must occur when that threshold is reached. (See Chapter 2, "Assertion Methodology" on page 21 for details on an effective assertion-based verification methodology.)

- *Designers shouldn't check their own code. Hence, adding assertions violates this rule.*

  Adding assertions in the RTL design is a way of specifying expected behavior and it is analogous to the following example of asserting that a pointer will never be null in software designs. Software engineers have known for years that it is a good idea to check that a pointer passed into their code is not null prior to use. If a null pointer is encountered during normal execution, the problem can be quickly isolated when an assertion is used. Notice how asserting that a pointer will not be null says nothing about *who* is going to test *what.* In other words, the designer is placing a proposition into the code that states that a particular implementation property will always be TRUE. This is not violating Bergeron's [2000] redundancy verification convergence model (that is, design engineers should not verify their own designs). In fact, the verification engineer will read the design specification and create a set of test scenarios[3] to validate the design. During the course of verification, if a sequence of events emerges in which a calling routine passes a null pointer, then the problem is quickly

2. Based on Foster and Krolnik's experience at Convex Computer Corporation in the early 1990's.

isolated via the implementer's assertion, and this dramatically simplifies debugging. In this respect, RTL design should be no different than software design. When the designer asserts that two signals must always be mutually exclusive, this does not state how the design should be verified. In fact, rarely will the verification engineer focus on implementation-specific testing. Hence, assertions added to the RTL implementation improve the overall quality of the verification process.

1.4 Phases of the design process

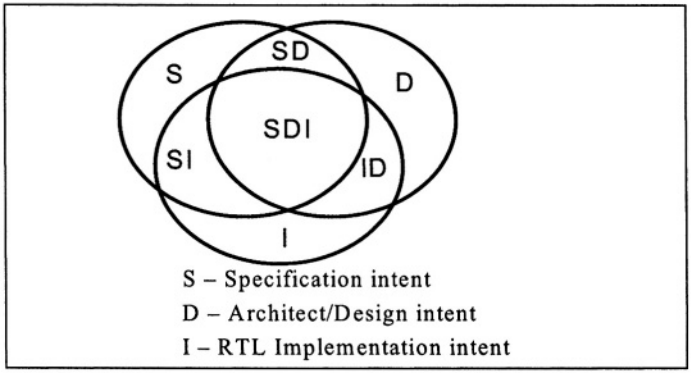
The *waterfall refinement* approach to the design process includes three distinct phases: *specify, architect/design,* and then *implement.* Although theoretically attractive in principle, this approach is seldom practiced in the real world. In other words, often the engineer moves back and forth between specification, architect/design, and RTL implementation as analysis uncovers additional requirements. In this section, we present an abstract view of these three phases within the overall scope of the design process. Our goal is to demonstrate how assertions help validate lower-level details, which are developed during the architect/ design or RTL implementation phase and are rarely described in the higher-level specification. That is, if the verification process focuses only on validating high-level requirements established in the specification, then there are many details in the design that might go unchecked.

Figure 1-1 illustrate three distinct *regions of design intent* related to a product's development process [Piziali and Wilcox 2002]. This corresponds to the following unique phases of development:

- The *specification phase,* which is the initial step in the design process. In this phase, the architect envisions the design intent and then establishes high-level requirements for the product.
- The *architect/design phase,* which is a process of refining the higher-level intent (described in the specification) into a set of detailed requirements, partitioning the high-level architecture into functional blocks, and considering alternative implementations for each block prior to RTL coding.
- The *RTL implementation phase,* which is the process of coding an RTL implementation such that it satisfies both the high-level specification and design requirements.

3. Or better yet, the verification engineer would implement a functional coverage model of the verification space and design and implement functional and verification constraints in their stimulus generator.

Figure 1-1 Regions of intent within the design process



The various regions of intent of the design process, and their overlap in terms of what is actually being specified or designed, are represented as seven domains within the Venn diagram in Figure 1-1:

- S - Specification only. In the worst case, the requirements developed during the specification phase (and contained within this region of the Venn diagram) were for some reason not considered during the architect/design or RTL implementation phases. This generally indicates a serious problem. Hence, ideally this space is very small.
- D - Architect/Design only. This domain consists of design intent details that were not defined in the higher-level specification phase, and for some reason were not implemented in the RTL code. Note that this is either a superfluous part of a design, or missed lower-level functionality during implementation.
- I - RTL implementation only. This domain of intent was not defined by the specification or the architect/design phases. This happens frequently since there are many lower-level implementation details that neither the specification nor the design need to describe.
- SD - Specification and architect/design. This domain was defined by the specification and architect/design phases, but details were either missed during RTL implementation or deemed unnecessary (hence, the requirements were changed but not updated).
- DI - Architect/Design and RTL implementation. The intent details in this domain were defined by the design and RTL implementation phases, but not by the specification phase. This is common, since specifications should not address every detail (allowing enough degrees of freedom during design for optimization).

- SI - Specification and RTL implementation. The intent details in this domain were defined by the specification and RTL implementation phases, but not by architect/design phase. This occurs when the design has errors and is not updated. For example, bug fixes are made that were not covered in the design.
- SDI - Specification, architect/design, and RTL implementation. All requirements, design decisions, and RTL implementation details are covered here. Although the verification engineer's goal is to verify (and ensure consistency and completeness) of all requirements established in the specification phase (represented by the specification domain), it is only the SDI domain that covers all details developed in all three phases of the design process. In other words, any functionality contained within the design or RTL implementation domains that is not contained within the SDI domain would typically not be verified.

In practice, the design process rarely excludes any of these regions of intent. Ideally, some would be minimized while others are maximized. However, the following observations related to Figure 1-1 generally apply to the development process:

1. Designers typically first attempt to merge two of the regions of intent into a single domain (that is, ensure that all higher-level requirements established during the specification phase are satisfied during the architect/design phase) and finally merge the resulting domain with the remaining RTL implementation intent domain (that is, insure that the final RTL implementation satisfies the original requirements of the specification) to achieve higher verification coverage.
2. Each region of intent requires specific techniques for identification and alignment (that is, maximized overlap), which will be discussed in the following sections.

1.4.1 Ensuring requirements are satisfied

the difficult method

Some design methodologies give little attention (or totally ignore) the *architect/design phase,* and instead, immediately begin *RTL implementation* with hopes of meeting all the requirements established by the specification. Thus, in an abstract sense, they attempt to move (that is, merge) the RTL implementation intent with the higher-level specification intent. This is a difficult process to execute. Ignoring the design phase provides little or no benefits typically experienced through a formalized process. This is due to the lack of information sharing between multiple stakeholders in the design process. The designer is essentially working in isolation during the implementation and problem

resolution processes. Using this approach, the RTL implementation may suffer long delays (and ultimately poor silicon). Hence, there are better ways to ensure that the RTL implementation satisfies the specification's requirements (that is, in an abstract sense, merging regions of intent).

the more efficient method

A design methodology that utilizes a formalized process provides time to align much of the detailed architect/design requirements with the specification requirements before RTL implementation begins. It also provides opportunities for review and collaborate to ensure the design has the greatest chance of matching the specifications. Once the design has been completed (and reviews are favorable) an implementation can begin. Assertions can play a major role in ensuring that the requirements specified during one phase of design are satisfied during the next phase (that is, aligning the domains). Furthermore, adding design assertions for lower-level details (beyond the higher-level specification) can help capture design intent in a fashion that is both verifiable in the design phase and dramatically simpler to debug during the RTL implementation phase.

3 phases of RTL implementation functional verification

Many people consider the functional verification of an RTL implementation to be a process with a single goal: *identify (and fix) bugs in the implementation.* However, the functional verification process must actually ensure three distinct goals:

1. All lower-level RTL implementation details (outside the requirements established during specification and architect/design) are bug-free
2. The RTL implementation satisfies all detailed requirements developed during the architect/design phase (which are outside the higher-level specification)
3. The RTL implementation satisfies all properties of its higher-level specification

Meeting these goals is typically performed concurrently during the verification process, although they could be performed in a serial fashion.

An effective way to verify that the RTL implementation matches the design requirements is to write assertions directly in the code during the RTL implementation phase. They immediately identify functional errors in the RTL implementation. As these errors are corrected, the implementation begins to function according to the design intent. In an abstract sense, we are attempting to align the details described by the RTL *implementation* region of intent with the requirements described by the *design/architect* region of intent; that is, ensure that what we have implemented during the RTL coding phase satisfies the detailed requirements we established during the design phase.

Note that for large design errors, it is usually easier to reconsider (that is, return to) the design phase and determine the optimal modification required to address the problem (rather than trying to fix an error directly in the RTL implementation). Ignoring the design phase and considering all problems as implementation errors is tantamount to adjusting the implementation domain to match the specification domain. As previously stated, this is frequently more difficult and (more importantly) requires significantly more time than returning to the design phase for further analysis.

1.4.2 Techniques for ensuring consistency

During the design process, our goal is to ensure that what is implemented in the RTL satisfies the requirements established during the specification phase, as well as the detailed requirements developed during the design phase. This can be viewed as maximizing the overlapping of all regions of intent in Figure 1-1 (or minimizing non-overlapping domains). The following sections list possible techniques for aligning these domains.

specification only

The requirements contained within the specification intent *only* domain (S) point out a serious flaw in our design. The approach to minimize this region is to create a functional coverage model that covers all aspects of the specification. Hence, a good functional coverage model will identify holes in the design, implementation, and verification phases of the design process. That is, a functional coverage model can identify missing functionality during implementation or unverified characteristics of a design.

architect/design only

The architect/design intent only domain (D) might be the result of superfluous architect/design details. However, it may be the result of missing or misunderstood implementation details. Designing functional coverage models (as well as designing reviews against the implementation) is the best way to minimize this domain.

implementation only

The implementation intent only domain (I) occurs frequently due to unspecified RTL implementation details, which were not specified during the specification phase or described during the design phase. Adding assertions to the RTL are necessary during coding to adequately verify the designer's intent. As previously mentioned, this level of detail is rarely known to the verification engineer (and is not contained in the specification), which can result in poor RTL implementation coverage during verification. Code coverage tools are also useful in this region to identify holes in the RTL implementation verification process.

architect/design and implementation

This domain is also common, as details are developed that are not specified during the specification phase. Adding assertions during the design and RTL implementation phase helps verify (and clarify) design intent. Code coverage tools also serve to identify details (outside of the specification) that are not verified within this domain.

specification and architect/ design

This domain represent the case where various higher level requirements (developed during the specification and architect/ design phase) were not implemented. Developing a good functional coverage model helps to identify missing functionality in the RTL implementation (see Chapter 5, "Functional Coverage").

specification and implementation

This domain should be minimal, since it represents details that were not considered during the design phase, yet are necessary for correct functionality. If this domain is large, it may indicate a poor design (that is, little thought was given to design analysis prior to RTL coding).

specification, architect/design and implementation

This is the ideal domain resulting from the three phases of the design process. Assertions derived from the specification, can ensure that the implementation satisfies its requirements. In addition, a good functional coverage model will identify holes in the verification process related to this domain.

1.4.3 Roles and ownership

Specifying intent during the three phases of design generally involves multiple stakeholders, where each stakeholder plays a critical role in the success of the assertion-based methodology. Hence, in this section we outline the various roles and property specification ownership that the architect, verification engineer, and design engineer play in specifying assertions and functional coverage.

- The verification team creates system-level assertions and functional coverage for the *specification intent* (that is, requirements defined in the high-level specification). Assertions, combined with functional coverage, enable the verification team to create a *verifiable testplan.*
- The architect, design leads, or verification team creates block-level assertions, as well as functional coverage, for the *design/ architect intent.* These assertions and functional coverage points are derived from the detailed functional specification, and are used in the testbench or embedded directly in the design model.

- The designer creates assertions and functional coverage for the *RTL implementation intent.* Note that lower-level details of the RTL implementation are rarely the focus of the verification team. Hence, important details could go unverified without the designer's involvement in adding assertions and functional coverage points.

1.5 Summary

An *assertion* is a statement about a design's *intended behavior* (that is, a property), which must be verified. Key verification benefits that assertions provide include:

- *improved error detection*
- *improved error isolation*
- *improved error notification*

These benefits are a result of improved *observability* (when assertions are embedded in the design).

Other benefits include:

- Reduced debug time—up to 50%
- Improved integration through correct usage checking
- Improved verification efficiency through specification
- Improved communication through documentation

In this chapter, we discussed various benefits of adopting an assertion-based design methodology. We also discussed many common arguments that are put forth by teams that are not using assertions and identified the fallacies in their reasoning. Finally, we present an abstract view three phases within the overall scope of the design process *(specify, architect/design, implement).* We then demonstrated how assertions help validate lower-level details, which are developed during the architect/design or RTL implementation phase and are rarely described in the higher-level specification.