CHAPTER

2

ASSERTION METHODOLOGY

The importance of adopting and adhering to an effective design methodology has been a popular topic of discussion within the engineering community and in literature for many years. It is generally accepted that the benefits of a well-defined design methodology far outweigh the cost of implementing it.

In this chapter, we focus primarily on components of an effective *assertion-based methodology.* However, an assertion-based methodology does not stand alone. In order to reap the greatest benefit, it must be tightly integrated with a larger design methodology. So, this chapter briefly discusses some of the broader design methodologies as a foundation for an effective assertion methodology. We then discuss how to apply an assertion-based methodology targeting *new designs*—followed by a discussion on how to apply an assertion-based methodology targeting *existing designs.* Finally, we discuss simulation-based methodology considerations specifically related to assertions.

Incidentally, an effective assertion-based methodology requires the same up front planning that an effective design methodology requires. This chapter guides the reader through steps that provide a blueprint for this type of comprehensive planning.

2.1 Design methodology

A typical design process includes five basic elements, and these drive our assertion methodology discussion:

1 Project planning
2 Design requirements

3 Design documents

4 Design reviews

5 Design validation

By establishing a consistent approach to each element described in this section, a project team solves many problems that can arise during a project's life cycle. Ignoring any one element greatly contributes to a less effective project (for example, slipping schedule or untested functionality). Your project might even fail by ignoring just one element. The magnitude of the failure may vary, but each element of the design process we recommend is in place to enhance the others and make them successful.

This section briefly discusses each of the basic design process elements and introduces the role of an assertion-based methodology within each of these elements. Our goal is to show that an assertion-based methodology is most effective when it is tightly integrated with the existing design process; that is, it should not be seen as an add-on or an afterthought.

2.1.1 Project planning

The first step to a successful project is planning. A project team must define policies and conventions that describe how to implement, verify, and maintain the project. Additionally, the team must define resources required to successfully complete the project, specify schedules, and outline guidelines that the team will uphold during the design process. These items must work together to make all the tasks easier.

Adequate planning ensures an efficient infrastructure, and documenting the process is an essential step in establishing an effective methodology. This allows the project team to concentrate on getting the job done instead of figuring out *how* to get the job done. An important benefit of establishing policies and conventions is that they allow new members of the team to be brought on-line much quicker, since the plans and resources are available for independent review. In addition, the conventions provide commonality across the design for ease of understanding and later debug of the machine. The conventions guide designers away from constructs with known problems (such as in the areas of timing and correctness). Without defining and documenting policies and conventions in the initial stages of the project, many questions arise and are addressed in an ad-hoc mode, which yields inconsistency across the design team. Finally, establishing project conventions simplifies the task of shuffling people resources between blocks in later stages of design.

Planning covers a wide variety of areas, including those presented below. Each of these should be analyzed and defined.

Project documents

This section sets forth guidelines for project policies, conventions, and documentation. This information may include the following resources:

- Requirements document
- Architectural or system specification document
- Micro-architectural or RTL specification document
- Detailed algorithms document
- Design validation document
- Documentation availability standards

An effective design methodology begins with an emphasis on detailing and documenting the project requirements and specifications. *The exact format of the documents is not as important as the content of the documents.* However, for consistency, a project should specify common tools and document file formats, whether they be ASCII text or that of a word processor.

documentation distribution

In addition to actually generating project documents, it is important to define how these documents are disseminated to the team. Many teams use project web pages on company intranets to make project information available. For some projects, it is also necessary to ensure that documents are accessible from all the computer platforms that are in use on the project. For instance, some team members might be using UNIX workstations, while others might be using PCs. Thus, it may be necessary to make choices that are cross-platform compatible.

assertions and project documentation

Project documentation is also an important part of an effective assertion methodology. The project documents are a valuable resource of data that *describes where assertions are needed.* They also provide *details on what the assertions should be validating.* Requirements documents should be used to describe assertions that are placed on externally visible interfaces. Detailed design documents describe assertions needed on internal interfaces and components. Each type of project document provides additional details that can aid the creation of effective assertions.

EDA and internal tools

Tools are an integral part of a design process. EDA vendors continually provide new tools and tool enhancements. Choosing the tools that are used on a project at the *beginning* of the project enables the design team to make specific plans for each task. The tools used on a project may include a combination of the following:

- Source code management tools
- Lint checking tools (design style encouragement)
- RTL simulators
- Synthesis tools
- Back-end tools
- ATPG toolset
- Formal property checking tools
- Assertion libraries
- Productivity scripts
- Databases (test plans, coverage data, metrics)
- Debug tools

Decisions on tools are not limited to commercial tools. Rather, decisions should be made about all aspects of the design and verification process, including how to manage coverage data and organize and track test plans.

assertions and tools

Decisions about tools are also integral to your assertion methodology. Some vendor tools use proprietary assertion conventions, while others support industry standard assertion libraries (such as the OVL). Hence, project teams must carefully consider whether the assertion specification language chosen for the project is supported by the tools used on the project. This becomes an important issue if components of the design are treated as IP and delivered to other organizations or projects using a different set of tools.

RTL styles and conventions

Important, but often overlooked aspects of an effective design methodology, are RTL styles and conventions such as:

- RTL coding styles, coding rules, and coding restrictions
- RTL naming conventions
- IP reuse rules or restrictions
- Functional coverage conventions
- Linting rules

Linting tools continue to offer enhanced features and have become a standard part of design methodologies. However, other conventions can also be used to improve the success of a design. For instance, defining a set of RTL naming conventions, coding styles and rules, and coding restrictions allows designers to produce consistent code that is easily readable by all members of the team. A major reason for coding rules and restrictions is to help steer designers away from RTL code that is not sythesizable or prone to errors. Examples of general RTL coding conventions include the *Reuse Methodology Manual* [Keating and Bricaud 2002] and *Principles of Verifiable RTL Design* [Bening and Foster 2001].

With the increasing size and complexity of today's designs, reuse of IP within these designs has become more important. It is essential that conventions be defined and followed in order to improve the reusability of IP across multiple designs.

Finally, functional coverage models (see Chapter 5, "Functional Coverage"), as well as general coverage processes and coverage goals, must be defined at the beginning of the project. While coverage tools are effective in many ways, there are steps that designers can take that allow the specific coverage tools to recognize design elements (such as state machines) in an easier manner. For example, some tools use pragmas to specify state machines.

assertions and project conventions

An assertion methodology must also include consistent coding styles and conventions. Good coding practices and linting checks are just as applicable for assertions as they are for the synthesizable part of the design.

An assertion methodology convention may specify that the team will avoid using non-clocked procedural assertions due to the issues described in Chapter 4, "PLI-Based Assertions" on page 103. Another convention may be that the team will add interface assertions at the top of each module, versus the bottom of each module (see section 2.2.3, "Assertion density"). As discussed previously, an assertion methodology might specify that all assertions are included in lint checks. Refer to section 2.2.2, "Best practices" for more effective practices for your assertion methodology.

When describing IP reuse conventions, ensuring that assertions are included in the IP to validate the use of all external interfaces is a requirement. With assertions in place, the users of the IP will get immediate feedback if the IP is used incorrectly.

Assertions also provide a form of design coverage themselves and must be considered as part of the overall coverage strategy of a project. Refer to section 5.2.2, "Types of traditional coverage metrics" on page 126 for more details on coverage models.

Support infrastructure

The support infrastructure of a project is utilized throughout the project each and every day. The support component of a project relies on consistent and reliable guidelines and resources such as:

- Naming files
- Structuring design source directories
- Defining available computer resources
- Defining personnel resources

If the infrastructure has not been appropriately defined at the beginning of the project, the team may well be forced to make changes and migrate previous implementations to align with changing definitions. For instance, changing the directory structure of a Verilog model requires designers to become familiar with new file locations and initiate modifications for makefiles, filelists, and scripts.

assertions and infrastructure

An effective assertion methodology requires decisions about resources allocated to the project. In addition to engineering personnel, this includes computer resources to support the tools that use the assertion methodology. Using assertions generally requires more computing time per individual simulation run (in our experience, 10%-30% for multi-million gate designs containing thousands of assertions, yet inefficient PLI-based assertions could require 2-10x more time) than simulation efforts without them. However, people and computer resources are typically planned for on a project time-line basis. Our claim (and experience) is that the savings in debug time when using assertions actually shortens the overall project time-line (by weeks to months on some projects). Hence, identifying complex bugs sooner in the verification process, while ensuring higher quality verification, justifies the resources.

Partner coordination

Most projects draw on multiple teams or organizations to turn out a product. To ensure that all the teams work effectively together, it is important to establish communication guidelines with partners. This includes:

- Coordinating documentation deliverables with partners
- Specifying points of contact for each partner organization
- Creating formal written partner agreements where required

For instance, a company that architects a new complex computer system might require the development of three new ASICs, new circuit board assemblies, and updated software to support the new hardware. The development effort is performed by three ASIC

teams, one circuit board team, and one firmware team. From this example, it should be obvious that communication between teams is important. If the three ASIC teams don't communicate, interfaces between the ASICs might be implemented differently on each chip. Also, if the ASIC teams don't communicate the configuration register settings to the software teams, the initial bring-up in the lab might be marred with incorrect initialization of the ASICs.

Defining contact persons within each team and how communication will occur between teams eliminates costly miscommunications that could possibly delay product shipment. A formal written set of expectations and deliverables may be required for some partners.

assertions and partners

Assertion specification language also plays a role in partner coordination. Models from different teams or projects are often merged into a system-level pre-silicon verification environment. From the example above, system-level verification environments which instantiate all three ASICs gain maximum benefit from assertions if all ASIC models used the same form of assertion specification. Also, the circuit board team creates a model of the circuit board that is combined with the HDL models of the ASICs in a verification environment. A similar situation arises here for any assertions written in the board models. In these cases, it is important to ensure that all models used in the system-level environment utilize compatible and consistent forms of assertion specification.

2.1.2 Design requirements

Every design begins with requirements. Sometimes these are developed in an ad hoc fashion, but ideally they are developed systematically. In essence, these requirements contain specific details on the expected behavior of the system. An example of requirement statements might be:

- "The design must process 3.0 gigabytes/sec of data"
- "The design must execute the x86 instruction set"

Capturing design requirements take several forms. This includes functionality requirements, which are usually specified first (though refined later). Silicon performance metrics are also part of the requirements; for without knowing the frequency, power, and area requirements, it is difficult to produce a good design. A third form includes requirements for delivery, and this form is important both for conveying the time required for completion and the nature of deliverables. Silicon is the common medium, but

intellectual property formats are becoming common. With this soft format, the list of deliverables is much larger than delivering a functioning chip.

assertions and design requirements

Assertions play a key role in this area of a design methodology. As a project team defines requirements at any level, whether system or architectural, the team generates models of the system. Each of these elements has interface specifications describing how they should operate. Assertions should be added to the design model, even early in the process, as a method of documenting the exact intent of the specification. Depending on the method of generating these models, the assertions captured in the earlier models of the system are leveraged by the refined detailed models. For instance, a high-level system model might be generated that captures assertions for the major chip interfaces. Tools using this high-level model make use of the assertions. As more detail is added to the system-level model and the functionality of each chip is added, the original assertions are still in place to continue validating the interfaces between chips in a functional simulation environment as well as formal verification environments.

2.1.3 Design documents

Design creation requires a medium that allows for top down design, successive refinement of details, and redundancy in explanation. RTL and schematic forms do not allow for redundant descriptions (unless you include assertions as redundancy). They also suffer from the rigid requirement of a single form. Project design documents allow a mixture of forms (and detail level) to describe your design. These may include a range of documents (such as; tables, charts, block diagrams, timing diagrams, and prose). All these allow others to understand and analyze your design. In addition, one can include design analysis documentation and data from previous design iterations.

Once the initial set of design documents is complete, you are not done with the documentation effort. Without continuing to document changes as they occur, different project teams can easily get out of sync. For example, if an ASIC team changes the format of a configuration register by adding a new mode bit but fails to update the documentation, it is likely that the firmware team will program the register incorrectly. Design documents force the engineer to consider design details that may have been overlooked by starting the design without a systematic documentation process. These overlooked details can sometimes be costly and require a redesign. Generally, the need for a redesign is not found early, rather after the point at which it could have been corrected with minimal cost. Finally, when the design details

have been thoroughly documented, the design is now transparent and more easily reviewed throughout a project's life.

A common set of design documents will include the following:

1 Analysis
2 Architecture
3 Preliminary design
4 Detailed design
5 Deliverable product specification

assertions and design documents

The design document provides a wealth of information that can be codified into assertions. Designs with assertions as part of the document provide additional insight into how a particular interface will operate. In this way, the assertions themselves provide additional, effective documentation.

2.1.4 Design reviews

Reviewing documentation and implementation code is crucial to the design process. This is where design teams analyze and critique design decisions for overall correctness and ensure that design performance and silicon metrics requirements are met. When the team finds incorrect structures, incomplete specifications, or unacceptable performance, the design *and* document should be updated in the problem areas and re-reviewed. In terms of time and resources, issues that are not addressed at review time become more costly to fix at a later time.

assertions and design reviews

Make assertions an integral part of the design review process. In addition to assessing design implementation, design reviews should analyze where assertions have been added. We recommend that design teams conduct reviews that specifically address adequate assertion density. (Assertion density is the number of assertions per line of code.) By including assertion review in the design review process, teams encourage designers to consciously think about their design and the corner cases and interfaces of their implementation, which are ideal locations for adding assertions.

Our experience at Hewlett-Packard Company followed this methodology, and logic design engineers found bugs in the design just by analyzing the type of assertion needed for a location within the design. The assertion was still added to the design, but this example shows that the design review process can definitely help find design bugs.

2.1.5 Design validation

make everything as simple as possible, but no simpler
-Albert Einstein

Design validation continues to be one of the dominant portions of a project cycle. Designs are more complex and larger than ever. Creative and sound methodologies are required to ensure that a design is effectively validated. EDA vendors are continuing to add features to their tool sets to increase the productivity of today's verification engineers. In addition, project teams must ensure effective verification methodologies to compliment and utilize the tool features.

assertions and design verification

Assertions are an emerging design technique that is enabling verification to be more effective. During the design creation, assertions should be at the forefront of your mind. Each time you add a new feature to the code, assess what you are writing and identify profitable assertions that will enforce correct operation and cover interesting design events. The advantages to adding assertions as you code are seen during the design validation phase of the project. A design created with an effective assertion-based methodology provides:

- a more thoroughly tested design due to an overall increase in the number of assertions (compared to adding them after coding is complete)
- added clarity of the code, which removes confusion when reviewing various code details for interactions with other features
- significantly less time spent debugging. Thinking about assertions up front can frequently clarify issues in the implementation. This removes bugs before a single simulation cycle can begin.

Refer to 1.2, "Verification techniques" on page 2 for more details on the effectiveness and benefits of assertions.

2.2 Assertion methodology for new designs

While using assertions benefits both new and existing designs, the *best results occur when design teams apply the assertion methodology from the beginning of a design process.* An effective assertion methodology plays an integral role in the debug process, and it must begin as early as possible.

For designs that are in the beginning stages of the process, refer to the discussion of the general design methodology and note the role that assertions play in each step, as described previously in section 2.1, "Design methodology". Then, proceed to the discussions that follow for a more in-depth understanding of how an assertion

methodology integrates into a design project. This section includes basic *"Key learnings"* that let you know what you are up against during design verification. It offers "Best practices" to guide your process. And, it ends with ideas on both "Assertion density" and "When not to add assertions".

2.2.1 Key learnings

Buy-in from all members of the project team (from engineers to project and program managers) is essential when attempting to adopt (and fully benefit from) assertion methodologies. For instance, it is important for all members of the design team to adopt the assertion methodology to give good assertion density across the entire design. If some members of the team don't believe that using assertion is worth their time, they are not inclined to add them. Likewise, if management agrees with the importance of the assertion methodology, they will encourage all of their employees to utilize assertions. If they believe it just adds to the overall schedule, they will either not encourage the use of assertions or actually discourage their use. In both cases, the benefits of the assertion methodology will not be realized.

In this section, we draw on our previous experiences to describe design and verification challenges and important points that must be accepted by the entire team. We refer to these points as *key learnings.* As you read this section, you may think that the key learning points are just basic concepts. However, many times these key learnings are forgotten or ignored. It is through experience that these learnings are ingrained within the engineer and the team.

Our intent is to offer a reality-based understanding of what a design team is likely to face and show why it is important for the entire project team to buy into an assertion-based methodology to help solve these verification challenges.

key learning 1 **The design model is not initially correct**

This point should not come as a big revelation to any experienced engineer or manager. However, it is important to consciously acknowledge that the initial RTL model will not work directly after coding. This mind set paves the way for an effective verification process (which includes assertions) to find where the design is not correct.

This idea encourages you to select the proper design elements or architecture with careful thought and not necessarily just the first thing that pops into your mind. Critical thinking about your design points out potential problems in it. This is what a formalized

debug process is about. With this learning and the information in this book, you are taking the first step towards achieving a more reliable design.

key learning 2 **Identifying and debugging design failures is difficult and time consuming**

Consider the following statement from Kernighan [1974].

> *"Debugging is at least twice as hard as writing the program in the first place."*

While many engineers also readily agree with this statement, it is not easy to state the complexity of debugging today's increasingly large designs in a way that allows you to stay on schedule and under budget. Experience has shown that assertions have a substantial positive impact on finding errors while minimizing the debug effort related to design failures.

key learning 3 **For a significantly shorter debug process, teams must accept a slightly longer RTL implementation process**

Most engineers agree that it is important to do everything possible to ensure a high quality, accurate design. However, management often gives into the pressures of schedules and budgets. In doing so, changes in the design process that increase the schedule of any portion normally are not accepted. What must be remembered in this case is that while the RTL implementation process of the project may be increased marginally by adding assertions (our studies indicate between one and three percent), the verification process is substantially reduced (up to fifty percent). (Refer to Chapter 1, "Introduction", for specific industry examples of verification successes that resulted from assertions.)

key learning 4 **Problems creep into the design during creation**

Teams must make a conscious decision to check for incorrectness and actively detect problems. Methodologies are developed to reduce the number of errors created during design capture such as through linting tools. However, it is not realistic to think that all bugs can ever be completely eliminated from the first pass design model. By recognizing this point, teams can take steps to put assertions into the design while the design is being captured. This requires that the designers recognize exactly where they are making design assumptions. These assumptions should be continuously validated during verification efforts through the use of assertions.

key learning 5 **Some portions of the design require additional verification effort**

These locations include both complicated sections of the design and intersections of blocks. Teams must make a conscious decision to seek out and record interesting combinations of events. As discussed in section 2.1.4, "Design reviews", this type of

analysis improves the overall quality of the completed design. By identifying these portions of the design, assertions are fully utilized and their full benefit realized.

2.2.2 Best practices

When the key learnings discussed in the previous section are accepted (that is, buy-in is achieved from the entire project team), the following best practices will help you create your assertion-based methodology and make it effective for your design project.

use your documents

Formalize the natural language specification using assertions

As was discussed in section 2.1.3, "Design documents", the design documents and specifications are an essential resource for knowing where to add assertions. This resource can become even more beneficial if it is captured with assertions in mind. The specification is now truly verifiable, since the assertions automate the process of verifying that the implementation satisfies the specification.

when to capture assertions

Write assertions along with the RTL code

The opposite of this practice is augmenting an existing design with assertions. When assertions are written up front, bugs are often identified prior to any form of verification. However, when assertions are written at the end of design implementation, they are less effective, as many of the bugs have already been found and many of the design assumptions that assertions should be validating have been forgotten. So, you might ask why assertions should be used at all if the bugs are found without them. Assertions are not a silver bullet that will rid your design of all errors, and ensure you are exempt from respin after respin without them. Assertions improve the overall design verification effort by making it easier to find and debug failures that are found with a project's verification environments.

It has been our experience, as well as the experience of teams at many companies, that adding assertions along with the RTL code is the most effective time to add them. At this point in the project, designers are making design implementation decisions, interpreting the design requirements, and implementing the logic necessary to interface with blocks being developed by other designers. With all this information fresh on the designers mind, what better and more effective time could there be to determine where to implement assertions?

when to stop capturing assertions

Consistently implement assertions throughout the design

While it is important to add assertions as the design is beginning to be captured, the process of adding assertions should not be stopped once the initial design work is accomplished. As a general rule, more assertions constantly checking for bugs make it more likely that you will find the bugs. Additionally, as more assertions are added to the design, the assertion density of a design is improved. For these reasons, adding assertions should be a continuous process.

keep adding assertions

Analyze failures not identified by assertions to determine whether you can write new assertions that detect the problem

Adding an assertion to detect a problem that has already been found may seem counter-productive. However, there are reasons to follow this practice. First, this helps get the designer in the practice of adding assertions by identifying locations within the design where assertions are effective. Second, when you add assertions to validate known bugs, you ensure that you are able to detect another occurrence of the same design specification violation. Third, not all errors are seen by debugging a specific failure. It is often said that where you found one bug, others are sure to be close. By adding an assertion in the location of a fixed bug, additional problems may be seen by this new observation point. Adding assertions in this manner also increases your assertion density.

where to put assertions

Co-locate RTL assertions with the design code they are validating

The opposite of this practice is to place the assertions away from the code, possibly at the end of the file or in a separate file. Separating assertions from the design code they are validating removes one of the benefits that assertions provide, that of documenting the code. Additionally, adding assertions in the design code clarifies the design intent.

Including assertions with the design code also simplifies the process of adding them. Since the design file is already in an editor while the code is being created and the designer is actively thinking about the design operation, adding assertions at this point allows the designer to capture the most effective RTL assertions. Finally, it allows reviewers to easily see which portions of the code have assertions and where assertions are lacking.

reuse

Generate IP with assertions

Use assertions that are designed specifically for common design structures. Put in place methods that automatically add assertions as the common design structure is added. You can do this in a variety of ways, including using macros or libraries of modules that implement the common design structures. For instance, if your design uses FIFO structures, consider creating a library

containing FIFO modules that have assertions already embedded within in them to check for underflow and overflow conditions.

assertion names

Name your assertions

AH assertions should be given names or IDs. This eases the effort associated with debugging assertion condition failures. Additionally, the names of the assertions are constant as different verification tools are used (such as simulation and formal tools). Furthermore, incremental releases of the design benefit from consistent assertion names across models.

Note that while some tools can automatically generate names, others require user-specified names. However, the assertion specification form used on a project should not drive whether names are required for assertion instantiations; the assertion methodology should drive this. In all the cases mentioned above, if you use tool-generated names for assertions, the names are not easily readable and can change each time the design is built.

to use or not to use

Provide a consistent method to disable assertions

Use *ifdef* text macro capabilities with assertions to enable easy removal from a model. For SystemVerilog, the assertion constructs are directly part of the language, which means that it is unnecessary to bracket these assertions with an `ifdef. However, for any additional logic created to support the assertion that feeds into an assertion (for example, satellite FSMs to capture a special event), it is best to bracket this logic with an `ifdef construct. Example 2-1 shows an example of this concept in Verilog.

Example 2-1 Compilation control of assertions

```
`ifdef ASSERT_ON
  FIFO_check: assert @ (posedge clk) (reset_n => FIFO_depth < 7);
`endif
```

If you use an assertion library, these commands should be placed within the library to reduce the designer's workload. This is already done in the OVL.

do not synthesize assertions

Provide a consistent method to prevent synthesis errors

The method for accomplishing this is tool-dependent. Many tools use comment meta-commands. These commands can wrap the assertion instantiations, much like the *ifdef* text macros capabilities. Again, depending on the tool, you may not be allowed to nest these commands. Special care should be taken to ensure the commands are used in accordance with the tool's documentation.

If you use an assertion library, these commands should be inserted in the library to reduce the designer's workload.

assertion libraries

Create libraries or templates for common assertions

Multiple designers are implementing similar assertion structures. To reduce the designer's workload, create common template libraries of assertions that provide extra logic, such as state machines that may be needed for some assertions. The OVL are an excellent example of this best practice.

design reviews

Conduct peer reviews of assertions

Peer reviews provide opportunities for designers to explore ideas for new assertions—and opportunities for designers to uncover design problems prior to the verification process. In addition, reviews identify errors within the coding of an assertion. Refer to section 2.1.4, "Design reviews" for more details about assertions and design reviews.

how effective are your assertions?

Create a process that effectively tracks identified problems

When logging bugs, you should document the technique that identified the bug. In doing so, you provide direct feedback for future projects on the effectiveness of your various verification processes. For instance, you should note whether a problem was identified with a directed or random verification environment and whether an assertion detected the bug.

One of the key learnings discussed in section 2.2.1, "Key learnings", is that while adding assertions incrementally increases the RTL development time, assertions greatly reduce the verification debug time, which can significantly improve a project's overall schedule. By capturing data on the effectiveness of assertions along with your bug tracking, you are collecting return-on-investment data that can justify the development of a new assertion-based methodology on a future project.

hook tools to assertions

Provide hooks in the verification environments to "see" assertions

Normally, when the assertion fails, the desired outcome is for the assertion to fire. However, there are some exceptions. For instance, when verifying that the design behaves in a known manner in the presence of *invalid* stimulus (for validating error correction logic), an effective assertion methodology monitors for this violation, but the assertion should not flag an error.

Since the test is known to produce an error condition, seeing the assertion fire will make the test appear as if it failed. In the best scenario, hooks allow the test to specifically "expect" the assertion to be violated. In this manner, the test will fail if the assertion is *not* violated. Alternatively, the model could be compiled with assertions disabled when you execute tests of this nature. However, this method is not highly recommended because it removes all assertions and their benefits are lost.

embedded assertion signals

Provide internal assertion "signals" to aid debugging with waveform viewers

When debugging failed simulations with the use of a waveform viewer, it is convenient to define an internal signal that is equivalent to the assertion's combinatorial expression as shown in Example 2-2. In this case, the signal `assert_valid_pnt` can be shown in the waveform viewer. This signal is inactive except when the assertion fires. This allows an engineer to quickly pinpoint the location in the waveform viewer where the assertion fired.

Example 2-2 Internal assertion signals

```
`ifdef ASSERT_ON
  assign assert_valid_pnt = (4'd2 <= pnt) && (pnt <= 4'd8) &&
                            (pnt != 4'd6);
  assert_always #(0, 0, 0, "illegal pointer value")
                valid_pnt (clk, reset_n, assert_valid_pnt);
`endif
```

2.2.3 Assertion density

An effective assertion methodology ensures sufficient assertion density within the RTL design. Assertion density is a measure of the number of assertions per line of code. Without sufficient assertion density, the full benefits of assertions are not realized. The goal is to have uniform assertion density with minimum holes across the entire design. Listed below are some common locations for assertions. Refer to Chapter 6, "Assertion Patterns" on page 161 for more details with examples of additional areas where assertions are effective.

general guideline

In place of RTL comments

As a general guideline, anywhere you would typically add a comment to document a potential concern, assumption, or restriction in the RTL implementation, this is an ideal location to add an assertion.

block interfaces

Block interfaces assertions

In section 2.1.2, "Design requirements", we described the benefits of adding assertions to block interfaces, particularly those that have different designers. In this case, the multiple designers identify different interpretations of a single interface's specification.

Block interfaces should have their assertions written up front when creating the architecture or specification documents for any

block using the interface. With this method, you are forced to think about specific error corner cases of the interface—and check for these cases using assertions.

Do not underestimate the importance of writing assertions for block-level interfaces. For example, a logic design lead at Hewlett-Packard Company high-end server group once said, *"If a person can't write an assertion for a block or chip interface, he or she probably is not clear about the interface."* The process of specifying assertions on block interfaces helps to clarify its correct behavior while uncovering many misconceptions (that is, bugs).

where to add interface assertions in the RTL modules

We recommend that you add all module interface assertions at the top of the RTL module. This keeps them close to the interface signals' definitions. Alternatively, you can place the interface assertions at the end of the RTL module; however, referencing the interface signal definitions becomes problematic for larger modules. Whichever location you chose for your interface assertions, we recommend that it is consistent across the entire design team. SystemVerilog allows assertions to be added to a new object known as interfaces, which permits you to describe the interface signaling requirements (defined by assertions) in a single place.

overflow and underflow

Queue/FIFO assertions

Specify assertions to check for illegal queue or FIFO overflow and underflow conditions. In addition, assertions should monitor all design-specific corner cases of a FIFO or queue.

state machines states and transitions

State machine assertions

Specify assertions to detect invalid states and invalid transitions in a state machine. For example, in the case of a state machine with a one-hot structure, assertions monitor the state signals to ensure that no two states are ever active simultaneously.

fairness and starvation

Arbiter assertions

Specify assertions to detect fairness problems within arbiters. While fairness and starvation are difficult verification areas, assertions help reduce the associated bugs. You may need to tune the assertion equation to handle the various corner cases involved with fairness on arbiters. Keep in mind that it is better to have a few false firings of an assertion than to let a fairness bug get into silicon.

untested code

Area of code not ready for testing assertions

Often a model is released with code that is not ready for testing. Assertions should be put in these areas to quickly notify testers that features have been enabled in the verification tool that are not ready for testing.

assertion groups

Group common functionality for assertions

It is often useful to group assertions into categories to allow individual control of similar types of assertions. For instance, you may choose to group all fairness assertions together. This becomes useful if you adjust the knobs in a random test environment that stresses the system in non-realistic ways that cause such a flood of transactions that the settings of the fairness assertions fail. In this case, grouping assertions allows you to disable the arbiter class. Refer to 2.4, “Assertions and simulation” for more details on the use of assertion groups in simulations.

2.2.4 Process for adding assertions

In this section, we outline a recommended process for adding assertions to your design. We recommend that you create a set of assertions as you define block interfaces (prior to RTL coding), and then continue adding additional assertions during RTL coding. The process is the following:

- Add assertions between blocks. These assertions help to define how multiple blocks interact.
- Add assertions to each internal interface of a block. These assertions help to define the interface protocol, legal values, required sequencing, and so forth.
- Add assertions as you code specific or unique structures within your RTL (see Chapter 6 "Assertion Patterns" and Chapter 7 "Assertion Cookbook" for examples of common structures).
- Add assertions as you code your control logic.

Following this process ensures good assertion density from the chip boundaries into the core. With this approach, incorrect behavior is isolated closer to the source of the problem. This process is also good when reviewing your RTL to ensure important assertions have not been missed

2.2.5 When not to add assertions

Use this section with caution. As a general rule, adding assertions is always a good idea. However, since there is a cost associated with adding assertions—and a cost associated with using assertions within simulation—a team should perform careful analysis prior to determining exactly where to add the assertion.

common features

Common features that are required for design operation

While this book shows that the debug and isolation benefits of using assertions is greater than the cost of adding them (and running them in simulation), the overall cost must still be recognized. As a result, every detail or aspect of a design will not warrant an assertion. The list of features where assertions should not be added include the monitoring of:

- a free running clock,
- glitch detection, asynchronous timing, or clock edge,
- assertion code that duplicates the RTL code—for example, a simple increment counter should not have an assertion that ensures the value changed by one,
- standard register D input to Q output transfers, and
- known correct components—such as a simple MUX.

duplicate checks

Design features that are validated by other methods

Some design features are checked by alternate methods—such as specific bus functional models in simulation. Also, a PCI interface may use a third party PCI protocol checker. Hence, there might not be a clear return-on-investment to duplicate the checks provided by the protocol checker with a set of assertions.

procedural assertions

Non-clocked procedural assertions

Designers should be careful with the use of non-clocked procedural assertions, as this class of assertions is prone to false firings. This is described in detail in Chapter 4, “PLI-Based Assertions” on page 103. We have found that the use of non-clocked procedural assertions is not required to obtain maximum benefit from an assertion methodology. Therefore, we recommend that you avoid them.

2.3 Assertion methodology for existing designs

An assertion-based methodology offers many of the same benefits for both new designs and existing designs. However, when implementing an assertion methodology for a mature design (for example, one that is well into the verification process), you will find fewer design problems. Hence, developing your assertion methodology at this phase of the design might not provide a dramatically clear return-on-investment.

When you use assertions with a mature design, you lose some of the benefits of capturing early designer assumptions. However,

this should not keep you from using assertions, as many design assumptions can still be documented in existing designs. For example, Krolnik [1999] at Cyrix Corporation documented cases where assertions were added late in the design cycles and many design errors were unexpectedly identified for code that had been exposed to many hours of simulation. In the Cyrix case, bug reports tripled (20 issues per week rose to 60 issues per week) after assertions were added. And the time required to close out problems fell from 7 days to 2 days.

The following are best practices for existing designs that maximum the effectiveness of assertions given the limited time remaining in the project life. However, if time permits, refer to the best practices described in section 2.2, "Assertion methodology for new designs".

clarification

Use assertions to clarify understanding of the design

An existing design without assertions is missing much of the knowledge (that is, design intent) that was developed during the design process. However, important assumptions and restrictions can still be captured as assertions late in the design cycle, which will aid in future understanding (and maintenance) of the design.

use code comments

Write assertions from design code comments that imply intent

Comments such as "this will never occur" or "either of these will cause . . ." are good locations for assertions. It should be recognized, however, that comments may not have been updated when the code was modified. So use good engineering methods to determine the exact design intent.

reused components

Check properties of reused components

Components such as pass-though one-hot muxes and priority encoders that are reused throughout the design are ideal locations for assertions. When you iconsider the number of instantiations of these components throughout the design, you will realize that you are actually adding many assertions. If these common components are contained within a library, a little amount of work adding assertions within the library definition will impact a large portion of the design.

module interfaces

Write assertions for block interfaces

Interface protocols often have well-defined rules. Translate these rules into a set of assertions. This practice is particularly effective for providing a clear return-on-investment when reusing the block on future designs.

2.4 Assertions and simulation

A number of steps and methodology features can make your project's assertion experience much more productive, especially with regard to simulations. This section dives into features that should be a part of an effective assertion methodology. While the specific details of many of these features vary depending on the assertion specification form you use; these areas should be well defined for each project.

global enable or disable

Global enable or disable for assertions

Your assertion methodologies must provide a mechanism for enabling or disabling each assertion. We recommend that assertions be enabled by default and that you use a mechanism to disable them. This is often through the use of a global enable signal—system task (for example, the SystemVerilog **$assertion** described in C.8, "SystemTasks" on page 373)—or through an `ifdef macro pre-processing step. You must add this mechanism to each verification environment and enable or disable it at the appropriate time.

It is also useful to separate assertions into a common group. This could be according to functionality or location. With this approach, each assertion group uses a different enable signal. This allows fine-grain control for various groups of assertions during simulation.

assertion clocks

Global clock versus local clock control

How assertions use clocks is specific to the assertion specification form you use. However, you should define a general strategy for using clocks with assertions. For example, you can use `TOP.assert_clk as the source clock for assertions to use a global clock. By using a global clock, you have control over the sampling of assertions to eliminate races.

assertion error reporting

Assertion error reporting facility

Implement a verification environment that permits easy management of assertion reports produced by simulation. For instance, a script that keeps volume simulations running must be able to manage multiple assertion reports, check for assertion firings when determining if a test failed, and archive the assertion reports along with other logs for failed simulations.

severity levels

Assertion severity levels

The assertion methodology you choose should support a variety of severity levels. This allows flexibility for designers as they add functional coverage. The assertion reporting facility discussed in the previous point should also support multiple severity levels. Different severity levels are used to determine when an assertion

should end the simulation immediately and when it should just report the condition but continue the simulation.

quiescent state assertions

End of simulation assertions

We have found end of simulation checks extremely useful. For example, after a simulation test completes, it is critical to be aware of all outstanding transactions to determine if conditions in the design are preventing forward progress. Similarly, it is useful to know if a queue or FIFO structure contains unread data—or if there were any critical FSMs not returned to their initial state. These problems could indicate a deadlock situation. Hence, a quiescent check on a state variable, counter, or pointer at the end of simulation can uncover many hard-to-find problems.

The OVL **assert_quiescent_state** assertions is useful for performing this type of check. In addition, in Chapter 4, "PLI-Based Assertions" we demonstrate how to create a PLI routine that automatically checks a quiescent condition by performing an automatic callback at the end of simulation.

error thresholds

Alterable assertion error threshold detector

Your simulation environment should make process decisions (that is, take actions) based on the status of an assertion firing. These process decisions include: *stop, finish, print a message, continue, and increment a counter for errors.* Your assertion methodology should provide facilities to control or limit specific action based on a configured threshold. For example, how many assertion violations are required before taking a specific action? What is the limit on the number of times a unique failure should be reported? The OVL provides many examples of methodology facilities automatically built into the library.

error message requirements.

Assertion report messages

Messages reported by assertions should contain the following default information, which is used to locate the failure:

- Time of error
- Location within testbench or design hierarchy of error
- Physical location (file, line number) of error RTL code
- Severity of reported failure - error, warning, info
- Additional user-specified message and details

Composing this information into a standard message format allows for consistent extraction (for example, a *perl* script) and fast location and diagnosis of the failure. Without this complete set of information, it is difficult to isolate the exact location and time for the failure.

Even more important than this default information is the message the user contributes to the specific assertion. The user error

message should contain information about the nature of the failure. This information is important to speed up the debug of the problem. The user message should contain the following information:

- What is the problem
- Where (what structure, interface)
- Optionally—who should investigate the problem

An example of an error message with this information is:

```
"Illegal command on trans_lak interface. See Jeff"
```

2.5 Assertions and formal verification

In this section, we discuss a potential role for formal property checking to play in an assertion-based verification flow. We begin with a discussion of a formal verification framework by detailing the steps required to perform formal property checking. We then outline a methodology for applying formal property checking in an industry setting.

2.5.1 Formal verification framework

In this section, our goal is to introduce the basic elements of formal property checking and in so doing, convey a sense of both its inherent power and limitations. Steps required to perform formal property checking (for example, *model checking*) include:

- compiling a formal model of the design
- creating a precise and unambiguous specification
- applying an automated and efficient proof algorithm

Each of these steps are briefly discussed below.

compile a formal model

In the first step of the formal property checking process, we create a formal model of the design by compiling a synthesizable hardware description (for example, a Verilog RTL model) into a form accepted by the property checker. For the purpose of our discussion, hardware designs are finite state concurrent systems. For example, the value of the *current state* of the system can be determined at a particular point in time by examining all state-

elements of the system. The *next state* of the system can be computed as a function of the system's current state value and design input values[1]. A current state—next state pair describes one particular *transition relation* of the system. For example, (s_i, s_{i+1}) is a transition relation, where s_i represents a current state of the system, and s_{i+1} represents one next state possibility directly reachable from s_i. Usually, a transition relation describes a set of all possible state transitions among states, represented in some data structure like a BDD.

A *path* at state s is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$, which represents a forward progression of time and a succession of states. Note that a simulation trace is one example of a path. A set of paths represents the *behavior* of the system. Hence, a formal model can be created by compiling a synthesizable model of the design into as a state transition graph structure, referred to as a *Kripke structure* [Kripke 1963].

A Kripke structure M is a four tuple M = (S, S_0, R, L), which consist of:

- S a finite set of *states*
- S_0 a set of *initial states,* where $S_0 \subseteq S$
- $R \subseteq S \times S$ a *transition relation,* where for every state $s \in S$, there is a state $s' \in S$, such that the state transition $(s, s') \in R$
- $L: S \rightarrow 2^{AP}$, where L is a function that labels each state with a set of atomic propositions that are true at that particular state

A Kripke structure models the design using a graph, where a node represents a state, and an edge represents transition between states.

creating a formal specification

In the next step of formal property checking, we specify properties as assertions of the design that we wish to verify. In Chapter 1, we informally defined a property as a proposition of expected design behavior (that is, *design intent*). The following is a more formal definition of a property:

Definition 2-1 ***property***: a collection of logical and temporal relationships between and among subordinate Boolean expressions, sequential expressions, and other properties that in aggregate represent a set of behavior (that is, a path). [Accellera PSL-1.1 2004]

1. The next state is derived from the cone-of-logic leading into the input to a state-element. This can also be represented as a transition function $\delta(s, I)$.

We define a safety property as follows:

Definition 2-2 ***safety property***: A property that specifies an invariant over the states in a design. The invariant is not necessarily limited to a single cycle, but it is bounded in time. Loosely speaking, a safety property claims that *something bad* does not happen. More formally, a safety property is a property for which any path violating the property has a finite prefix such that every extension of the prefix violates the property. [Accellera PSL-1.1 2004]

For example, the property, "the signals `wr_en` and `rd_en` are mutually exclusive" and "whenever signal `req` is asserted, signal `ack` is asserted within 3 cycles" are safety properties.

We define a liveness property as follows:

Definition 2-3 ***liveness property***: A property that specifies an eventuality that is unbounded in time. Loosely speaking, a liveness property claims that "something good" eventually happens. More formally, a liveness property is a property for which any finite path can be extended to a path satisfying the property.

For example, the property "whenever signal `req` is asserted, signal `ack` is asserted some time in the future" is a liveness property.

Underlying many property languages are formalism known as *propositional temporal logics,* which allows us to reason about sequences of transitions between states. Two formalisms for describing sequence propositions are *branching-time temporal logic* [Clarke and Emerson 1981][Ben-Ari et al. 1983] and *linear-time temporal* logic [Pnueli 1977]. CTL is an example of branching-time logic. The temporal operators of this formalism allow us to reason about all paths originating from a given state. Whereas in the case of LTL (a linear-time temporal logic), the temporal operators allow us to reason about events along a single computation path. In this book, we introduce the Accellera Property Specification Language (PSL) [Accellera PSL-1.1 2004]. Although PSL supports both branching-time and linear-time temporal logic. However, in Chapter 3 "Specifying RTL Properties", we focus only on the linear-time temporal component (that is, the PSL Foundation Language instead of the PSL Optional Branching Extension) since it is generally easier for the designer to reason about the behavior of hardware design in terms of simulation traces.

applying a proof algorithm

Once we have created a formal model representing the design and a formal specification precisely describing a property that we wish to verify, our next step is to apply an automated proof algorithm. For example, given a formal model of a design described as a Kripke structure M=(S, S_0, R, L), and a temporal logic formula *f* expressing some desired property of the design, the problem of

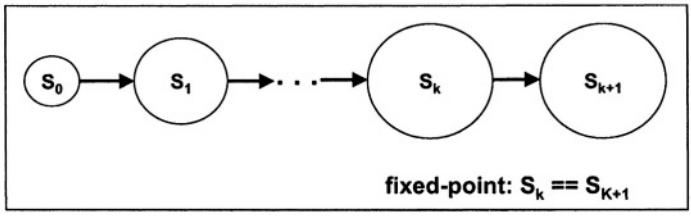
proving correctness involves finding the set of all states in S that satisfy *f*:

$$\{s \in S \mid M, s \models f\}$$

where $s \models f$ means the property represented by temporal formula *f* holds at state s.

Note that the formal model satisfies the specification *if and only if* all initial states (that is, $\forall s_i \in S_0$) are in the set of all states that satisfies *f* (that is, $\{s \in S \mid M, s \models f\}$). Figure 2-1 graphically illustrates (at a high level) one proof algorithm used to find the set of all states in S that satisfy *f*.

Figure 2-1 Fixed-point reachable states



The illustrated proof algorithm we use is known as *reachability analysis* using *image computation.* The algorithm begins with a set of initial states S_0, as shown in Figure 2-1. Using the transition relation R, as previously discussed, we calculate within one step (that is, a tick of the clock) all reachable states from S_0. This calculation process is referred to as *image computation.* The new set of reachable states is S_1 in our example. We iterate on this process, generating a new set of reachable states at each step that grows monotonically, until no new reachable states can be added to the new set (that is, a *fixed-point* occurs when $S_{k+1} == S_k$).

Note that for a safety property described by the temporal formula *f*, we can validate that *f* holds on each new state calculated during the image computation step.

proof results

For this fixed-point proof algorithm, one of three different results occurs:

1. **Pass.** The process reaches a *fixed-point,* and the formula *f* holds on all reachable states. Hence, we are done (that is, the design is valid for this property).
2. **Fail.** The process has yet to reach a fixed-point, and the formula *f* was determined not to hold on a particular state s_i, which was calculated during the search. Hence, a *counter-example* (that is, a path $\pi = s_0 s_1 s_2 ..., s_j$) can be calculated back from the bad state s_j to an initial state $s_0 \in S_0$. This counter-example is then used to debug the problem.

3 **Undecided.** The process aborts prior to reaching a fixed-point due to a condition known as *state-explosion* (that is, there are too many states for the proof engine to represent in memory). In the following section, we discuss a few techniques that might address the state-explosion problem.

Formal property checking tools use a number of different proof algorithms. A detailed discussion of these specific *proof algorithms, creating formal models,* and *temporal logics* is beyond the scope of this book. For in-depth discussions on these subjects, we suggest Clarke et al. [2000] and Kroph [1998].

2.5.2 Formal methodology

As formal research matures and approaches a level of sophistication required by industry (beyond the bounds of research and early adopters), we must take steps to ensure a successful transfer (scaling) to this more demanding level. One step is to fundamentally change design methodologies such that we move from ambiguous natural language forms of specification to forms that are mathematically precise and verifiable. Furthermore, these languages must lend themselves to automation. Formal *property specification* is the key ingredient in this methodological change. The end result is higher design quality through:

- *improved understanding of the design space*—resulting from the engineer's intimate analysis of the requirements, which often uncovers design deficiencies prior to RTL implementation
- *improved communication of design intent* among multiple stakeholders in the design process
- *improved verification quality* through the adoption of assertion-based verification techniques

Although the need for methodological change is clear, transitioning formal verification technology into an industry design environment has been limited by a lack of methodology guidelines for effective use.

Recall that state explosion is one of the difficulties encountered when attempting to apply formal to an industry setting. When attempting to prove correctness of assertions on an RTL implementation, a full proof is not always achievable. However, the value of functional formal verification is not limited by any means to full proofs. In reality, the value lies in finding bugs faster or earlier in the design cycle and finding difficult bugs missed by traditional simulation approaches, which in turn increases

confidence in the correctness of the design while decreasing time to market.

Another difficulty encountered when attempting to apply formal to an industry setting is the methodological requirement for accurately specifying environment constraints. These are used by the formal engine to limit the exhaustive search to a valid set of legal behaviors. Note that the work used to create block-level environmental constraints for a formal engine can often be re-used as block-level interface assertions during full-chip and system simulation. Hence, there is a return on investment for specifying block-level interfaces that include, as previously state, *improved understanding of the design space, improved communication of design intent,* and *improved verification quality.*

2.5.2.1 Handling complexity

In this section, we discuss techniques typically used to handle the state explosion problem when proving properties on industrial RTL models.

Choose appropriate RTL. The first step in handling complexity is to initially choose the right level of RTL to apply formal. For example, RTL contained in control-intensive logic is better suited for formal property checking than RTL modeling datapath involving data computation (like floating point arithmetic). Size of the RTL component (in terms of state directly related to a property) must be considered. Other factors that influence the RTL selection are design-related. For example, not every RTL component (that is, module, block, or unit) is a good candidate for stand-alone verification. Interesting properties may require more logic to be included beyond our selected RTL component. This can be problematic since many internal interfaces are rarely documented. Furthermore, the additional logic not included with our RTL component that we wish to verify may be too complex to model as environment constraints. Nonetheless, if we choose the appropriate RTL wisely, we can have a high degree of success at formally verifying properties on RTL components.

Property decomposition. We recommend that complex sequential assertions be split into simpler assertions. For example, break a req-ack handshake down into its component elements (arcs on a timing diagram). This *think static rather than dynamic* approach works well for formal proofs.

Compositional reasoning. One technique for handling the state explosion problem is to partition a large unverifiable component into a set of smaller, independently verifiable

components. This technique is referred to as *compositional reasoning*. For example, a large super-block component can be partitioned (often quite naturally) into a set of smaller block and sub-block components. When verifying a property of one of these partitioned components, you must specify a set of constraints that model the behavior of the other components (that is, the environment for the component under verification).

We define a constraint as follows:

Definition 2-4 ***constraint***: A condition (usually on the input signals) that limits the set of behavior to be considered by the formal engine. A constraint may represent real requirements on the environment in which the design is used, or it may represent artificial limitations imposed in order to partition the verification task. [Accellera PSL-1.1 2004]

Gradual semi-exhaustive verification. Although in theory, compositional reasoning using constraints sounds attractive, when applying formal property checking within an industrial setting, a more modest approach is generally used. We refer to this approach as *gradual semi-exhaustive formal verification via restrictions.* The advantage of this approach is that it has the potential of flushing out complex bugs as quickly as possible using formal verification to search a large state space.

Essentially, this approach is a gradual development of a formal verification environment around the RTL component you selected using restrictions.

This approach has the following benefits:

- Allows the user to control the state space explored to prevent state explosion using restrictions
- Enables us to initially turn off portions of the design's functionality—and then gradually turn on additional functionality as we validate the design under a set of restrictions
- Allows us to refine the constraint model into more general assumptions without initially encountering state explosion
- Provides an easier method of debugging by selecting, and thus controlling, the functionality in the environment that is enabled

We define a restriction as follows:

Definition 2-5 ***restriction***: A statement that the design is constrained by a given artificial property and a directive to verification tools to consider only paths on which the given property holds. [Accellera PSL-1.1 2004]

A restriction may reduce a set of opcodes to a smaller set of legal values to be explored during the formal search process. Or a restriction may limit the component's mode settings to read only during one phase of a proof, and then re-prove with a write mode restriction. Other examples include restricting the upper eight bits of a 16-bit bus to a constant value while letting the lower eight bits remain unconstrained during the formal search. Then, shifting the restriction to a new set of bits and re-proving with the new bus restrictions. It is important to note that even with the use of restrictions, the number of scenarios that the formal verification engine explores is very large, and complex errors will be detected under these conditions.

We demonstrate a restriction later in Section 2.5.3 "ECC example", and expand on this discussion in Section 2.5.4 "Gradual exhaustive formal verification".

exhaustive proofs

The second technique used in an industrial setting, which is often used *after* the semi-exhaustive bug-hunting approach, is to relax the restrictions into general interface assumptions in an attempt to *prove* properties on the partitioned component. The advantage of performing the semi-exhaustive bug-hunting approach first using restrictions, as opposed to exhaustive proofs, is that if we cannot prove the property under the restriction, then we cannot prove it using general assumptions. Hence, we must employ other techniques (such as abstraction) if a proof is required.

We define an assumption as follows:

Definition 2-6 ***assumption***: A statement that the design is constrained by a given property and a directive to verification tools to consider only paths on which the given property holds. [Accellera PSL-1.1 2004]

Note the subtle distinction between assumptions and restrictions related to our goal of applying formal technology in an industrial setting. For restrictions, our goal is to find bugs and clean up the partitioned components of the design using formal techniques. We are under no obligation to validate restrictions (either in simulation or formal verification). Using assumptions, however, our goal is to prove correctness—which can be a more difficult task. Often, we convert assumptions into assertions, which we then attempt to prove on neighboring components of the design. This strategy is known as *assume-guarantee reasoning* [Grumberg and Long 1994]. If an assumption is too difficult to formally prove, we use simulation to validate these assumptions as interface assertions.

2.5.2.2 Formal property checking role

identify where to apply formal

In this section, we discuss the role formal property checking plays at various phases within a design flow. The first step in the process is to identify good property candidates that provide a clear return on investment (ROI) for the effort involved in the formal verification process and likelihood for success (LFS). Examples include properties related to portions of the design that:

- have historically resulted in respins due to bugs (hence, ROI)
- are estimated to be difficult to verify (or it will be difficult to achieve high coverage) using traditional simulation means (hence, ROI)
- are contained in control-intensive logic versus data path logic (hence, LFS)
- are supported with enough bandwidth from the design team to adequately define required environment constraints when a full proof is required (hence, LFS)

identify when to apply formal

In section 1.4 "Phases of the design process" on page 14, we defined the role of assertion specification at various stages within a design flow. In this section, we discuss the role and goal of applying formal verification at various phases of design. The level of expertise required at each phase varies depending on the verification goals.

Architectural verification. Formal verification has been successfully applied to proving architectural properties on shared memory consistency protocols (for example, cache coherence or sequential consistency protocols) as well as other architectural considerations (for example various arbitration schemes). The goal of this phase of formal verification is to flush out high-level architectural bugs prior to RTL implementation. However, successful architectural formal verification, in general, requires a verification team with a high level of expertise. In part, this expertise requirement comes from the need to create abstract models of the system that are *formal-friendly*.

Concurrent design and verification. Formal verification can be applied early during the RTL development phase in an attempt to flush out bugs prior to module integration into the system verification environment. In general, this is a low-effort task (which could be higher depending on the particular engineer's goals). As the engineer codes assertions into the RTL implementation, formal property checking combined with interface *restrictions* attempt to find bugs.

Block-level regression. Formal verification, when applied to the block-level, offers much more than a low-effort, early bug hunting tool. On the contrary, the strategy offers a means to

deliver high quality blocks to the chip integration environment. Although the initial effort, before chip integration, does allow for early bug hunting, formal property checking's value extends beyond the initial stage. To provide a quick path for finding bugs and saving precious debug time during regression, it can also be performed every time the team modifies the block-level RTL code. This especially makes sense after a team makes the initial constraint investment at the block-level, which allows a formal tool to quickly prove the block-level assertions.

Targeted formal proof. Formal property checking can be applied on the set of properties previously identified as good candidates (that is, clear return on investment and likelihood for success). The effort required to perform formal property checking on an RTL model can obviously range from low (for trivial properties) to very high for a complex RTL implementation or property. Often compositional and assume-guarantee reasoning combined with some degree of abstraction are employed in an attempt to prove properties on complex designs. The effort required is mostly a function of the RTL and the property.

Post-silicon verification. We have successfully applied formal property checking during post-silicon verification. When a bug is identified in the lab, a formal test environment is created around the RTL implementation containing the bug. A property associated with the bug is created, and then the error is demonstrated on the RTL model using formal property checking combined with a formal testbench (that is, environmental properties used as constraints). Once the corrected RTL implementation is available, it is instantiated into the formal testbench and the formal property checker is used again to verify the fix. Note: Like targeted formal proofs, this can take a fair amount of effort to exhaustively prove the property on the corrected RTL.

2.5.3 ECC example

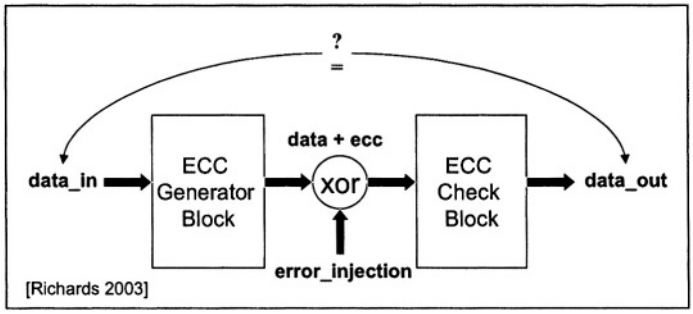
In this section, we use an error correction code (ECC) example based on Richards [2003] to demonstrate how to use *assumptions* and *restrictions* within a proof. In general, ECC algorithms are not sequential in nature. In fact, for most ECCs the proof can be formed in a single (yet complex) combinatorial step. Even though we are not demonstrating the power of a sequential search with this example, the constraint techniques we demonstrate can be applied to other more complex sequential circuits.

Error correction, the process of detecting bit errors during data transfer—and correcting these bits, can be done in either software or hardware. For high data rates, error correction must be done in special-purpose hardware because software is too slow. The ECC bits are generally computed by an algorithm that is implemented as a set of exclusive OR trees in hardware. For our discussion, an *ECC generator block* computes the ECC. Each data bit contributes to more than one ECC bit. Hence, by a careful selection of data bits in the algorithm that directly contribute to the calculation for a specific ECC bit, it is not only possible to detect a single-bit error, but actually identify which bit is in error (including the ECC bits). In fact, the computed ECC is usually designed so that all single-bit errors are corrected, while all double-bit errors are detected (but not corrected).

For our discussion, an *ECC check block,* which consists of a set of exclusive OR trees, recomputes the ECC from the transmitted data bits. The output of the recomputed ECC exclusive OR network is called a *syndrome.* If the syndrome is zero, no error occurred. If the syndrome is non-zero, it can be used to index into a look-up table[2] to determine exactly which bit is in error and then correct it. For a multi-bit error, no match will occur in the lookup table.

Figure 2-2, demonstrates a formal testbench created to exhaustively verify the ECC implementation. The *ECC generator block* reads the *m* bit-wide `data_in` bus and calculates an *n* bit-wide `ecc`, which it outputs as part of an *m+n* bit-wide `data+ecc` bus. The *ECC check block* reads the `data+ecc` bus and recomputes a new ECC syndrome from the data bits, which is used for error detection and correction. For our formal testbench, we create a single level *m+n* bit-wide exclusive OR error injection circuit. This enables us to inject single bit errors during the proof.

Figure 2-2 ECC single-bit error detect and correct proof



There are two techniques used to verify correct behavior of the combined ECC generator and check blocks, using assumptions and restrictions. First, for a single bit error, we can write an

2. The look-up table could be implemented in hardware, firmware, or even software.

assertion that the *ECC check block* will always detect and correct single bit errors as shown in Example 2-3.

Example 2-3 *PSL* assertion that the ECC detects and correct single-bit errors

```
// constrain the error_injection to all zeroes, or a one-hot value
assume always ((error_injection & (error_injection - 1))==1'b0);

// assert that the data_in equals the data_out for single bit errors.
assert always (data_in == data_out);
```

The specific details for the PSL syntax are discussed in Chapter 3 "Specifying RTL Properties". However, for our example, we are using PSL to specify that the `data_in` value will always equal the `data_out` value for a single bit error. We specify a single-bit error possibility as a *zero or one-hot* assumption on the multi-bit variable `error_injection` by writing the following Verilog expression:

```
(error_inection & (error_injection - 1)) == 1'b0
```

A formal engine will use this one-hot assumption when it explores all combinations of single bit errors (that is, each bit for the `error_injection` bus will assume a one during some point in the search, effectively injecting all possible single bit error combinations into the *ECC Check block*).

Using the combined assertion and assumption, the formal proof engine exhaustively explores all possible input values and single-bit error injection combinations.

Note that if the *m*+*n* bit-wide `data+ecc` and `error_injection` bus is too large, the proof could terminate with an *undecided* result. One technique to address this problem would be to use a combination of assumptions and restrictions. For example, we could restrict all bits with the exception of the lower four bits of the `error_injection` bus to a zero (using the PSL `restrict` construct), and then make a *zero or one-hot* assumption on the lower four bits (using the PSL `assume` construct). After proving this simpler model, we would repeat the proof by first shifting the four-bit *zero or one-hot* assumption to a new set of bits and restricting all other `error_injection` bits to zero. This process is repeated until all bits of the `error_injection` variable have had the opportunity to assume one. Ultimately, we will explore all single bit error possibilities as we shift the restriction and assumption across the `error_injection` bits.

A similar proof can be constructed to determine if multi-bit errors are detected correctly by the *ECC check block.*

2.5.4 Gradual exhaustive formal verification

This section presents a technique that restricts the formal proof to (or focuses on) a subset of legal design behaviors (for example, a single mode of operation). This technique is useful in early stages of RTL development to flush out mainstream bugs. After cleaning out mainstream bugs, the engineer then removes all restrictions to identify complex corner-case bugs.

In some cases, engineers prefer to verify various functionality or modes of operation for their design separately. This might be due to the engineer's desire to start an early verification on an incomplete RTL model where some functionality is complete while other functionality remains partially coded. Another reason to perform verification on separate design modes stems from a sense of familiarity. That is, for traditional simulation-based methodologies, the engineer might partition the development of a testbench into separate stimulus generators for various design operating modes. For example, for a design containing a USB interface, individually a host and a device must be able to handle both normal and error conditions. If the engineer starts the verification with both normal and error conditions, then it is likely that too many bugs will be detected for the error condition. This can frustrate the designer who would not have a sense of whether or not the basic functionality for the normal condition is working correctly. Hence, the engineer might take the following course of action in a traditional simulation-based methodology to allow partitioning of the various operating modes during verification:

1. Develop a generator for normal condition transactions
2. Begin verification for this single mode of operation
3. When the testbench is no longer detecting mainstream bugs associated with normal condition transactions, develop a generator for error condition transactions
4. Perform verification on this mode of operation
5. After sufficient verification has occurred on error condition transactions, perform the verification by combining random occurrences of normal and error condition transactions within the testbench with the goal of flushing out corner-case bugs

You can apply a similar methodological approach during functional formal verification. This approach is referred to as *gradual exhaustive formal verification via restrictions,* and it has the potential of flushing out mainstream bugs as quickly as possible while using formal verification to search a large state space.

This approach offers the following benefits:

- Enables you to initially turn off portions of the design's functionality—and then gradually turn on additional functionality as you validate the design under a set of restrictions (note: this is analogous to creating separate testbench generators for simulation-based verification)
- Provides an easier method of debugging by selecting, and thus controlling, the functionality in the environment that is enabled
- Allows you to refine the constraint model (that is, assumptions) into more general assumptions without initially encountering state explosion

Essentially, this approach involves gradually developing a formal verification environment around the RTL component by using restrictions. A restriction is a special type of constraint, in that it constrains the design behavior explored by the formal engine to a given artificial assumption. For example, a restriction may reduce a large set of opcodes to a smaller set of opcodes to be explored during the formal search process. Or, as with a traditional simulation approach, a restriction may limit the input behavior to only READ transactions during one phase of a proof, and then re-prove the design for WRITE transactions. Other examples include restricting the upper eight bits of a 16-bit bus to a constant value while letting the lower eight bits remain unconstrained during the formal search, then shifting the restriction to a new set of bits and re-proving with the new bus restrictions.

Figure 2-3 Restricted state-space

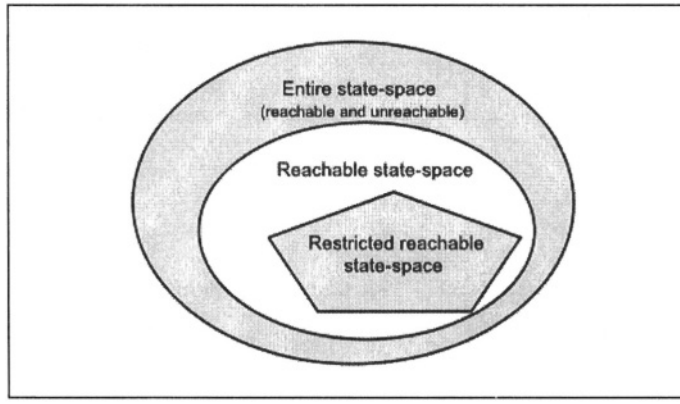


Figure 1 illustrates the restricted state-space concept. The outer circle represents the entire or theoretical state-space, which consists of the reachable as well as unreachable state-space. The entire state-space consists of the following number of theoretical states:

$2^{(R + I)}$

R is the number of state elements found in the design, and *I* is the number of input signals into the design. In general, not all possible combinations of input values are possible or legal. Similarly, not every possible combination of state-element values is possible. Hence, the white region represents the reachable (that is, legal) state-space associated with the design. Note that if we further restrict the input value (for example, limit input values to only READ type transactions), then the behavior of the design considered during verification is simpler and reduced, as illustrated by the inner (non-circular) region in Figure 2-3. The technique we discuss in this application note uses restrictions to reduce the behavior analyzed during formal verification with the goal of targeting simpler mainstream bugs early in the design process.

One characteristic of a restriction is that it cannot be proved on a neighboring block (that is why we referred to the restriction as an artificial assumption). For example, if we restrict all input sequences to READ transactions only, proving this restriction on a neighboring block would fail since WRITE transactions could also be generated (since the neighboring block could produce both READ and WRITE transactions). Nonetheless, restrictions enable the engineer to narrow the verification process and quickly find many mainstream bugs. Even with the use of restrictions, the number of scenarios that the formal engine explores is very large; however, the formal engine will detect complex errors under these conditions as well.

After we have formally verified the design using various restrictions, our next step is to relax the restrictions into more general interface assumptions. By doing this, we are able to prove the general assumptions as properties on neighboring blocks. Note the subtle distinction between assumptions and restrictions. For restrictions, our goal is to quickly find mainstream bugs and clean up the design using formal techniques. We are under no obligation to validate restrictions (neither in simulation nor formal verification). Using assumptions, however, our goal is to prove correctness while finding complex corner-case bugs.

The technique we presented in this section allows the engineer to identify mainstream bugs by focusing on simpler behavior during a formal proof. This is accomplished by restricting the design behavior to simpler modes of operation during a proof. In this respect, the process we presented is similar to the methodology of creating a simulation-based testbench, where separate, simpler input stimulus generators are often created for various modes of operation.

After you become familiar with debugging designs using formal verification, you will probably find that it is more efficient to

avoid using restrictions and identify both mainstream and complex corner-case bugs in parallel. Corner-case bugs can reveal architectural issues requiring complete RTL recoding. Hence, finding corner-case bugs as soon as possible within the design flow minimizes the risk to the project's overall schedule. Nonetheless, the technique we present is useful for focusing the formal proof to a single aspect or mode of operation for your design.

2.6 Summary

In this chapter, we focused primarily on components of an effective *assertion-based methodology* related to some of the broader design methodology considerations We then discussed how to apply an assertion-based methodology targeting *new designs*—followed by a discussion of how to apply an assertion-based methodology targeting *existing designs.* Next, we discussed simulation-based methodology considerations specifically related to assertions. Finally, we presented an overview of formal property checking and methodological considerations for applying formal in an industrial setting.

It is now up to you to choose the elements that best fit your specific project needs. Consider the concepts and guidelines we presented in this chapter when you create your project-specific assertion methodology—and then encourage your entire design team to consistently follow your methodology. By reviewing the *key learnings* with your team, you put them in a better position to fully appreciate the benefits that an assertion-based methodology provides.