Ashok B. Mehta

SystemVerilog Assertions and Functional Coverage

Guide to Language, Methodology and Applications

Second Edition

EXTRAS ONLINE

Springer

SystemVerilog Assertions and Functional Coverage

Ashok B. Mehta

SystemVerilog Assertions and Functional Coverage

Guide to Language, Methodology and Applications

Second Edition

Springer

Ashok B. Mehta
Los Gatos, CA
USA

Additional material to this book can be downloaded from http://extras.springer.com.

ISBN 978-3-319-30538-7 ISBN 978-3-319-30539-4 (eBook)
DOI 10.1007/978-3-319-30539-4

Library of Congress Control Number: 2016932750

© Springer International Publishing Switzerland 2014, 2016
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

To

My dear wife Ashraf Zahedi

and

My dear parents Rukshamani and Biren Mehta

Foreword

Louis H. Sullivan, an American architect, considered the father of the modern skyscraper, and mentor to Frank Lloyd Wright, coined the phrase 'form follows function.' The actual quote is 'form ever follows function' which is a bit more poetic and assertive than the version that has found its way into the common vernacular. He wrote those words in an article written for Lippincott's Magazine #57 published in March 1896. Here is the passage in that article that contains the famous quote:

> "Whether it be the sweeping eagle in his light or the open apple-blossom, then toiling work horse, the blithe swan, the branching oak, the winding stream at its base, the drifting clouds—over all the coursing sun, form ever follows function, and this is the law. Where function does not change, form does not change. The granite rocks, the ever brooding hills, remain for ages; the lightning lives, comes into shape, and dies, in a twinkling.
>
> It is the pervading law of all things organic and inorganic, of all things physical and metaphysical, of all things human and all things superhuman—of all true manifestations of the head, of the heart, of the soul—that the life is recognizable in its expression, that form ever follows function. This is the law."

Earlier in the article, Sullivan foreshadows his thought with this passage:

> "All things in nature have a shape, that is to say, a form, an outward semblance, that tells us what they are, that distinguishes them from ourselves and from each other."

The precise meaning of this pithy phrase has been debated in art and architecture circles since Sullivan's article was first published. However, it is widely accepted to mean that the form of something—its shape, color, size, etc.—is related to what it does. Water flows, rocks sit, and birds fly.

In his book 'The Design of Everyday Things,' (Basic Books 1988) Don Norman discusses a similar concept, the notion of affordances. Norman defines the term as '… the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used.' He cites some examples: 'A chair affords ("is for") support and, therefore, affords sitting. A chair

can also be carried. Glass is for seeing through, and for breaking. Wood is normally used for solidity, opacity, support or carving.'

Norman's idea turns Sullivan's upside down. He is saying function follows form. The shape, color, size, etc., of an object affects what it does. Nonetheless, both men would likely agree that form and function, whichever drives the other, are inextricably linked.

Software designers have the luxury of choosing the form to fit the function. They are not as constrained by the laws of physics as say, a cabinetmaker. The cabinetmaker must choose materials that will not only look nice, but will withstand the weight of books or dishes or whatever is to be placed on the shelves. Software designers have some constraints with regard to memory space and processing time, but beyond that they have a lot of freedom to build whatever comes to mind.

Sullivan referred to 'all things physical and metaphysical.' Without much of a stretch, we can interpret that to include software, a most abstract human creation. The form of a piece of software is linked to its function. The complex software that verification engineers build, called a testbench, must be designed before it can be built. The verification engineer, like an architect, must determine the form of his creation.

The architecture space is wide open. Computer code, while much more abstract than say, a staircase or a door handle on a car, has a form and a function. The form of computer code is the set of syntactic elements strung together in a program. The function is what the program does when executed, often referred to as its semantics.

A verification engineer is typically presented a set of requirements, often as a design specification, and asked to build a testbench that meets these requirements. Because of the tremendous flexibility afforded by the software medium, he must choose the form carefully to ensure that not only meets the requirements, but is easy to use, reusable, and robust. He must choose a form that fits the function.

Often an assertion is just the right thing to capture the essence of some part of a design. The *form* of an assertion is short sequence of text that can be inserted easily without disrupting the design. With their compact syntax and concise semantics, assertions can be used to check low-level invariants, protocols, or end-to-end behavior.

The *function* of an assertion, in a simulation context, is to assert that something is always (or never) the case. It ensures that invariants are indeed invariant. Assertions can operate as checkers or as coverpoints. The fact that they can be included in-line in RTL code or in separate checkers and that they can be short or long for simple or complex checking makes them invaluable in any testbench.

The wise verification engineer uses all the tools as his disposal to create an effective and easy-to-use testbench. He will consider the function of the testbench and devise a form that suits the required function. Assertions are an important part of any testbench.

Ashok Mehta has written a book that makes assertions accessible. His approach is very pragmatic, choosing to show you how to build and use assertions rather than engage in a lot of theoretical discussion. Not that theoretical discussion is irrelevant—it is useful to understand the theoretical underpinnings of any technology. However,

there are many other books on that topic. This book fills a gap for practicing engineers where before no text provided the how-tos of building and using assertions in a real-world context.

Ashok opens up the world of assertions to verification engineers who may have thought them too opaque to consider using in a real testbench. He does an especially nice job of deconstructing assertions to show how they work and how to write them. Through detailed examples, he shows all the pieces that go into creating assertions of different kinds, and how they fit together. Ashok completes the picture by demonstrating how assertions and coverage fit together.

Part of the book is devoted to functional coverage. He deconstructs the sometimes awkward SystemVerilog syntax of covergoups and coverpoints. Like he has with assertions, he takes the mystery out of building a high-quality coverage model.

With the mysteries of assertions unmasked, you can now include them in your personal vocabulary of testbench forms. This will enable you to create testbenches with more sophisticated function.

February 2013 Mark Glasser

Preface to the Second Edition

The first edition of this book was well received, and the readers provided many a good suggestion on further elaboration of language semantics. Readers also pointed out some errata on the language syntax. I am greatly indebted to the readers and colleagues for their input and support. In addition, the IEEE 1800-2012 LRM came along. Many features of the 2012 LRM were missing in the first edition, since the LRM was not ready yet. This edition incorporates the errata/suggestions from readers as well as the IEEE 1800-2012 feature set. Among many, features such as 'checkers,' 'let declarations,' past and future global clock sampled value functions, strong and weak properties, abort properties, and '.triggered' end point detection method are included. Furthermore, this edition adds many more examples and adds further clarification of the semantic nuances of the language.

Pleasant reading.

Preface to the First Edition

Having been an end user of EDA tools for over 20 years, I have seen that many new technologies stay on wayside because either the engineers do not have time to learn these new technologies/languages or the available material is too complex to digest. A few years back I decided to tackle this problem by creating a very practical, application-oriented down-to-earth SystemVerilog Assertions (SVA) and functional coverage (FC) class for professional engineers. The class was well received, and I received a lot of feedback on making the class even more useful. That culminated in over 500 slides of class material just on SVA and FC. Many suggested that I had collected enough material for a book. That is how I ended up on this project with the same goal that the reader should understand the concept clearly in an easy and intuitive manner and be able to apply the concepts to real-life applications right away.

The style of the book is such that the concepts are clarified directly in a slide style diagram with talking points. This will hopefully make it easy to use the book as a quick reference as well. Applications immediately following a topic will further clarify the subject matter, and my hope is that once you understand the semantics and applications of a given topic, you are ready to apply that to your daily design work. These applications are modeled such that you should be able to use them in your design with minimal modifications.

This book is meant for both design and verification engineers. As a matter of fact, I have devoted a complete section on the reasons and practicality behind having micro-level assertions written by the design engineers and macro-level assertions written by verification engineers. Gone are the days when designers would write RTL and throw it over the wall for the verification engineer to quality check.

The book covers both IEEE 1800-2005 and IEEE 1800-2009/2012 standard SVA language.

Chapter 1 is introduction to SVA and FC giving a brief history of SVA evolution. It also explains how SVA and FC fall under SystemVerilog umbrella to provide a complete assertions and functional coverage-driven methodology.

Part I: SystemVerilog Assertions (SVA)

Chapter 2 goes in-depth on SVA-based methodology providing detail that you can right away use in your project execution. Questions such as 'How do I know I have added enough assertions?', 'What type of assertions should I add', etc., are explained with clarity.

Chapter 3 describes immediate assertions. These are non-temporal assertions allowed in procedural code.

Chapter 4 goes into the fundamentals of concurrent assertions to set the stage for the rest of the book. How the concurrent multi-threaded semantics work, when and how assertions get evaluated in a simulation time tick, formal arguments, disabling, etc., are described here.

Chapter 5 describes the so-called sampled value functions such as $rose, $fell, $stable, $past, etc.

Chapter 6 is the big one! This chapter describes all the operators offered by the language including clock delay with and without range, consecutive repetition with and without range, non-consecutive repetition with and without range, 'through-out,' 'within,' 'and,' 'or,' 'intersect,' 'first_match,' and 'if…else,'. Each of the operator descriptions is immediately followed by examples and applications to solidify the concept.

Chapter 7 describes the system functions and tasks such as $isunknown and $onehot.

Chapter 8 discusses a very important aspect of the language that being properties with multiple clocks. There is not a single design nowadays that uses only a single clock. A simple asynchronous FIFO will have a read clock and a write clock which are asynchronous. Properties need to be written such that check in one clock domain triggers a check in another clock domain. The chapter goes in plenty detail to demystify semantics to write assertions that cross clock domains. The so-called CDC (Clock Domain Crossing) assertions are explained in this chapter.

Chapter 9 is probably the most useful one describing local variables. Without this multi-threaded feature, many of the assertions would be impossible to write. There are plenty of examples to help you weed through the semantics.

Chapter 10 is on recursive properties. These are rarely used but are very handy when you want to know that a property holds until another becomes true or false.

Chapters 11–13 describe other useful features such as 'expect,' 'assume,' and detecting end point of a sequence. The .triggered and .matched end points of sequences are indeed very practical features. Note that .ended (of LRM 2005) is now deprecated and replaced with .triggered.

Chapter 14 is entirely devoted to very powerful and practical features that do not quite fit elsewhere. Of main interest, here is the example/testbench for asynchronous FIFO checks, concurrent assertions in procedural code, sequence in Verilog 'always' block sensitivity list, and the phenomenon of a 'vacuous pass' !

Chapter 15 is solely devoted to asynchronous assertions. The example in this chapter shows why you need to be extremely careful in using such assertions.

Chapter 16 is entirely devoted to IEEE 1800 2009-2012 features. There are many useful features added by the language designers.

Chapter 17 describes 6 LABs for you to try out. The LABs start with simple example moving gradually onto complex ones.

Note: The LABs are available on Springer download site. All required Verilog files, testbenches, and run scripts are included for both PC and Linux OS.

Chapter 18 provides answers to the LABs of Chap. 17

Part II: SystemVerilog Functional Coverage (FC)

Chapter 19 provides introduction to functional coverage and explains differences with code coverage.

Chapter 20 is fully devoted to functional coverage including in-depth detail on covergroups, coverpoints, and bins including transition and cross coverage.

Chapter 21 provides practical hints to performance implications of coverage methodology. Do not try to cover everything all the time.

Chapter 22 describes coverage options, which you may keep in your back pocket as reference material for a rainy day!

Acknowledgements

I am very grateful to many who helped with review and editing of the book, in particular, Mark Glaser for his excellent foreword and in-depth review of the book, Vijay Akkati for detailed review of the chapters, Dr. Sandeep Goel for motivation and editing of the book, and Bob Slee for his sustained support throughout the endeavor and for facilitating close cooperation with EDA vendors. I would also like to thank Tom Slee, Kea Hunt, Norbert Eng, Joe Chang, and Frank Lee for all things verification.

And last but certainly not the least, I would like to thank my wife Ashraf Zahedi for her enthusiasm and encouragement throughout the writing of this book and putting up with long nights and weekends required to finish the book. She is the cornerstone of my life always with a positive attitude to carry the day through up and down of life.

Contents

About the Author

Ashok B. Mehta has been working in the ASIC/SoC design and verification field for over 20 years. He started his career at Digital Equipment Corporation (DEC) working first as a CPU design engineer, moving on to hardware design verification of the VAX11-785 CPU design. He then worked at Data General, Intel (first Pentium design team) and, after a route of a couple of startups, worked at Applied Micro and TSMC. He was a very early adopter of Verilog and participated in Verilog, VHDL, iHDL (Intel HDL), and SDF (standard delay format) technical subcommittees. He has also been a proponent of ESL (Electronic System Level) designs, and at TSMC, he released two industry-standard reference flows that take designs from ESL to RTL while preserving the verification environment for reuse from ESL to RTL. Lately, he has been involved with 3DIC design verification challenges at TSMC which is where SystemVerilog Assertions played an instrumental role in stacked die SoC design verification.

Ashok earned an MSEE from University of Missouri. He holds 13 US Patents in the field of SoC and 3DIC design verification. In his spare time, he is an amateur photographer and likes to play drums on 1970s rock music driving his neighbors up the wall.

List of Figures

List of Tables

Chapter 1
Introduction

As is well known in the industry, the design complexity at 16 nm node and below is exploding. Small form factor requirements and conflicting demands of high performance and low power and small area result in ever so complex design architecture. Multi-core, multi-threading and Power, Performance and Area (PPA) demands exacerbate the design complexity and functional verification thereof.

The burden lies on functional and temporal domain verification to make sure that the design adheres to the specification. Not only is RTL (and Virtual Platform level) functional verification important but so is silicon validation. Days when engineering teams would take months to validate the silicon in the lab are over. What can you do during pre-silicon verification to guarantee post-silicon validation a first pass success.

The biggest challenge that the companies face is short time-to-market to deliver first pass working silicon of increasing complexity. Functional design verification is the long poll to design tape-out. Here are two key problem statements.

1. Design Verification Productivity: 40–50 % of project resources go to functional design verification. The chart in Fig. 1.1 shows design cost for different parts of a design cycle. As is evident, the design verification cost component is about 40+ % of the total design cost. In other words, this problem states that we must increase the productivity of functional design verification and shorten the design ⇔ simulate ⇔ debug ⇔ cover loop. This is a productivity issue, which needs to be addressed.
   Continuing with the productivity issue, the chart in Fig. 1.2 shows that the compounded complexity growth rate per year is 58 % while the compounded productivity growth rate is only 21 %. There is a huge gap between what *needs* to get done and what *is* getting done. This is another example of why the productivity of design cycle components such as functional design verification must be improved.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_1

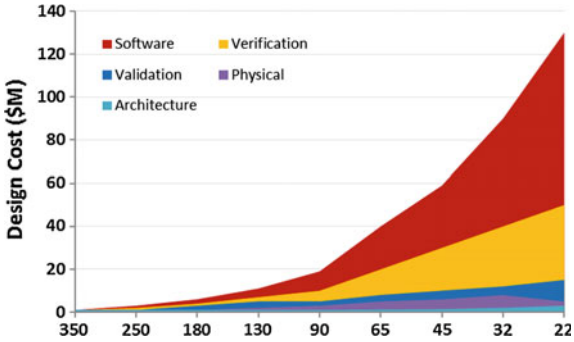


Fig. 1.1 Verification cost increases as the technology node shrinks

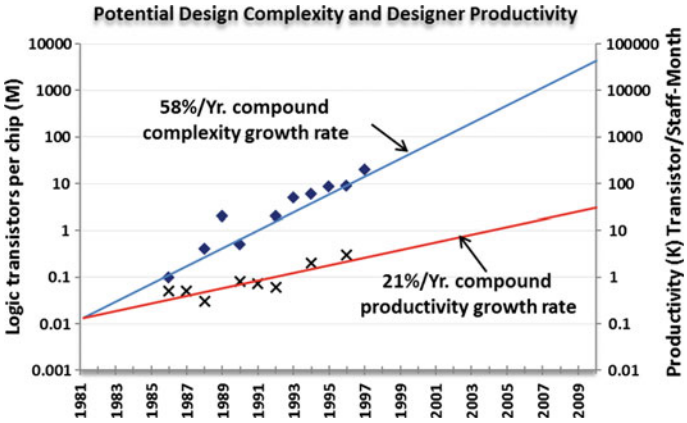


Fig. 1.2 Design productivity and design complexity

2. Design Coverage: The second problem statement states that more than 50 % of designs require re-spin due to functional bugs. One of the factors that contribute to this is the fact that we did not objectively determine *before* tape-out that we had really *covered* the entire design space with our testbench. The motto "If it's not verified, it will not work" seems to have taken hold in design cycle. Not knowing if you have indeed covered the entire design space is the real culprit towards escaped bugs and functional silicon failures.

So, what's the solution to each problem statement?

1. Increase Design Verification Productivity
   a. *Reduce Time to Develop*
      i. Raise abstraction level of tests. Use TLM (Transaction Level Modeling) methodologies such as UVM, SystemVerilog/C++/DPI, etc. The higher

the abstraction level, easier it is to model and maintain verification logic. Modification and debug of transaction level logic is much easier, further reducing time to develop testbench, reference models (scoreboard), peripheral models and other such verification logic.

ii. Use constrained random verification (CRV) methodologies to reach exhaustive coverage with fewer tests. Fewer tests mean less time to develop and debug.

iii. Develop Verification Components (UVM agents, for example that are reusable). Make them parameterized for adoptability in future projects.

iv. Use SystemVerilog Assertions to reduce time to develop complex temporal domain and combinatorial checks. As we will see, assertions are intuitive and much simpler to model, especially for complex temporal domain checks. Verilog code for a given assertion will be much lengthier, hard to model and hard to debug. SVA indeed reduces time to develop and debug.

b. *Reduce Time to Simulate*

i. Again, higher level of abstraction simulates much faster than pure RTL testbench which is modeled at signal level. Use transaction level test bench.

ii. Use SystemVerilog Assertions to directly point to the root cause of a bug. This reduces the simulate ⇔ debug ⇔ verify loop time. Debugging the design is time consuming as is, but not knowing where the bug is and trial and error simulations further exacerbate the already lengthy simulation time.

c. *Reduce Time to Debug*

i. Use SystemVerilog Assertion Based Verification (ABV) methodology to quickly reach to the source of the bug. As we will see, assertions are placed at various places in design to catch bugs where they occur. Traditional way of debug is at IO level. You see the effect of a bug at primary output. You then trace back from primary output until you find the cause of the bug resulting in lengthy debug time. In contrast, an SVA assertion points directly at the source of the failure (for example, a FIFO assertion will point directly to the FIFO condition that failed and right away help with debug of the failure) drastically reducing the debug effort.

ii. Use Transaction level methodologies to reduce debugging effort (and not get bogged down into signal level granularity).

iii. Again, Constraint Random Verification allows for fewer tests. They also narrow down the cone of logic to debug. CRV indeed reduces time to debug.

2. *Reduce Time to Cover* and build confidence in taping out a fully verified design.

   (i) Use '*cover*' feature of SystemVerilog Assertions to cover complex *temporal* domain specification of your design. As we will see further in the book, 'cover' helps with making sure that you have exercised low level temporal domain conditions with your testbench. *If an assertion does not fire, that does not necessarily mean that there is no bug*. One of the reasons for an assertion to not fire is that you probably never really stimulated the required condition (antecedent) in the first place. If you do not stimulate a condition, how would you know if there is indeed a bug in the design logic under simulation? 'cover' helps you determine if you have indeed exercised the required temporal domain condition. More on this in later chapters.

   (ii) Use SystemVerilog *Functional Coverage* language to measure the '*intent*' of the design. How well have your testbench verified the 'intent' of the design. For example, have you verified all transition of Write/Read/Snoop on the bus? Have you verified that a CPU1-snoop occurs to the same line at the same time that a CPU2-write invalid occurs to the same line? Code Coverage will not help with this. We will cover Functional Coverage in plenty detail in the book.

   (iii) Use Code Coverage to cover *structural* coverage (yes, code coverage is still important as the first line of defense even though it simply provides structural coverage). As we will see in detail in the section on SV Functional Coverage, structural coverage does not verify the intent of the design, it simply sees that the code that you have written has been exercised (e.g. if you have verified all 'case' items of a 'case' statement, or toggled all possible assigns, expressions, states, etc.). Nonetheless, code coverage is still important as a starting point to measure coverage of the design.

As you notice from above analysis, SystemVerilog Assertions and Functional Coverage play a key role in about every aspect of Functional Verification. Note that in this book, I use Functional Verification to include both the 'function' domain functional coverage as well as the 'temporal' domain functional coverage.

1.1 How Will This Book Help You?

This book will go systematically through each of SystemVerilog Assertions (SVA) and Functional Coverage (FC) language features and methodology components with practical applications at each step. These applications are modeled such that you should be able to use them in your design with minimal modifications. The book is organized using power point style slides and description to make it very easy to grasp the key fundamentals. Advanced applications are given for

those users who are familiar with the basics. For most part, the book concentrates on the in-depth discussion of the features of the languages and shows examples that make the feature easily understandable and applicable. Simulation logs are frequently used to make it easier to understand the underlying concepts of a feature or method.

The book is written by a design engineer for (mainly) hardware design engineers with the intent to make the languages easy to grasp avoiding decipher of lengthy verbose descriptions. The author has been in System and Chip design field for over 20 years and knows the importance of learning new languages and methodologies in shortest possible time to be productive.

The book concentrates on SVA features of the IEEE 1800-2005 standard. Author believes that the features of this standard are plenty to designing practical assertions for the reader's project(s). However, the author has indeed covered the entire IEEE 1800-2009/2012 feature set in a standalone Chap. 16 to give an in-depth look at the new standard. Note that some of the 2009/2012 features were not supported by popular simulators as of this writing and the examples provided were not simulated. Please do send your suggestions/corrections to the author (ashok_mehta@yahoo.com).

1.2 SystemVerilog Assertions and Functional Coverage Under IEEE 1800 SystemVerilog Umbrella

SystemVerilog assertions (SVA) and Functional Coverage (FC) are part of IEEE 1800 SystemVerilog standard. In other words, SVA and FC are two of the four distinct language subsets that fall under the SystemVerilog umbrella.

1. SystemVerilog Object Oriented language for functional verification (using UVM style libraries)
2. SystemVerilog language for Design
3. SystemVerilog Assertions (SVA) language and
4. SystemVerilog Functional Coverage (FC) language to see that the verification environment/testbench have fully verified your design

As shown in Fig. 1.3, SVA and FC are two of the important language subsets of SystemVerilog.

In any design, there are 3 main components of verification. (1) Stimulus Generators to stimulate the design (2) Response Checkers to see that the device adheres to the device specifications (3) Coverage components to see that we have indeed structurally and functionally covered everything in the DUT according to the device specifications.

1. *Stimulus Generation*. This entails creating different ways in which a DUT needs to be exercised. For example, a peripheral (e.g. USB) maybe modeled as a Bus Functional Mode (or a UVM (Universal Verification Methodology) agent) to

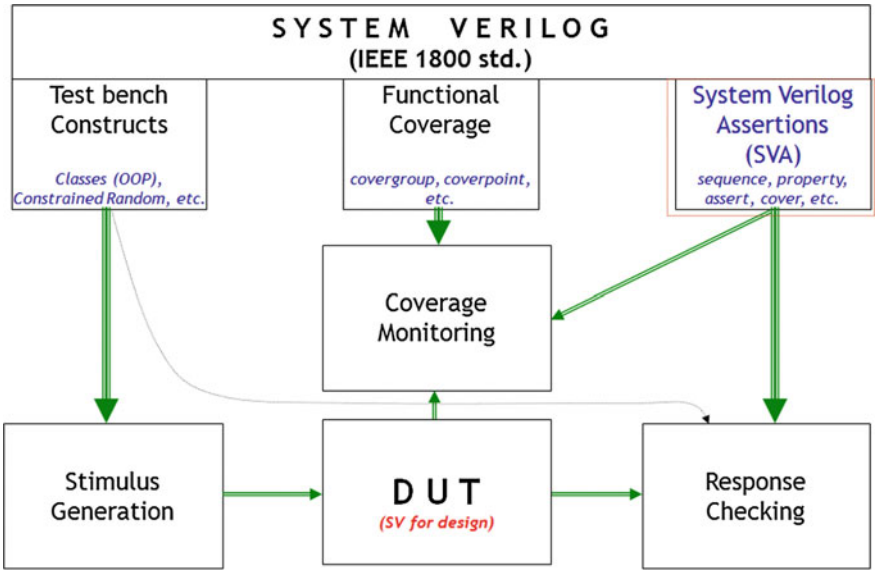


Fig. 1.3 SystemVerilog assertions and functional coverage components under SystemVerilog IEEE 1800-2009 umbrella

drive traffic through SystemVerilog transactions to the DUT. Different techniques are deployed to achieve exhaustive coverage of the design. For example, constrained random, transaction based, UVM based, memory based, etc. These topics are beyond the scope of this book.

2. *Response checking*. Now that you have stimulated the DUT, you need to make sure that the device has responded to that stimulus according to the device specs. Here is where SVA comes into picture along with UVM monitors, scoreboards and other such techniques. SVA will check to see that the design not only meets high level specifications but also low level combinatorial and temporal design rules.
3. *Functional Coverage*. How do we know that we have exercised everything that the device specification dictates? Code Coverage is one measure. But code coverage is only structural. For example, it will point out if a conditional has been exercised. But code coverage has no idea if the conditional itself is correct, which is where Functional Coverage comes into picture (more on this later when we discuss Functional Coverage—See Chap. 19. Functional coverage gives an objective measure of the design coverage (e.g. have we verified all different cache access transitions (for example, write followed by read from the same address) to L2 from CPU? Code Coverage will not give such measure). We will discuss entire coverage methodology in detail in Chap. 19.

1.3 SystemVerilog Assertions Evolution

To set the stage, here is a brief history of Verilog to SystemVerilog evolution (Figs. 1.4 and 1.5). Starting with Verilog 95, we reached Verilog 2001 with Multi-dimensional arrays and auto variables, among other useful features. Meanwhile, functional verification was eating up ever more resources of a given

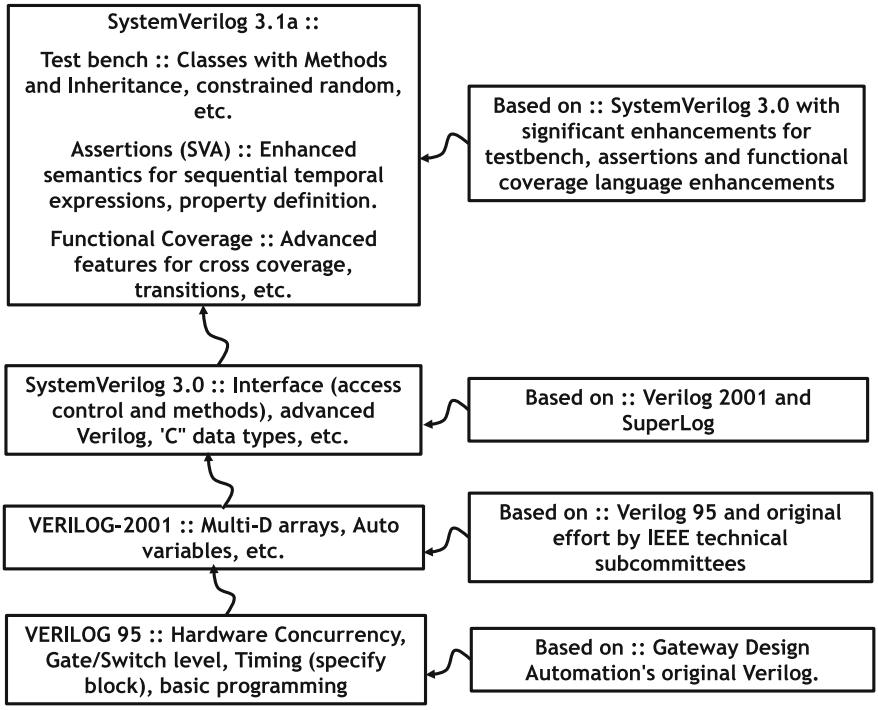


Fig. 1.4 SystemVerilog evolution

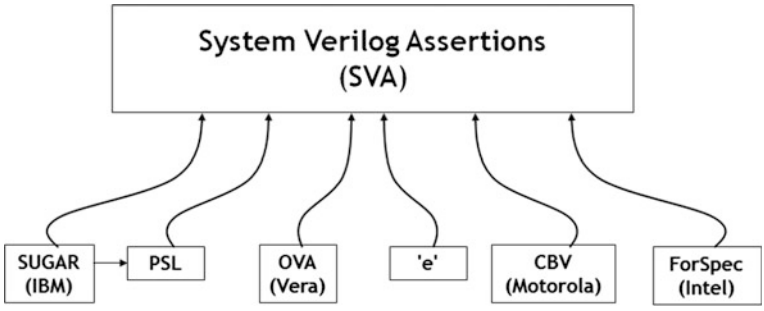


Fig. 1.5 SystemVerilog assertion evolution

project. Everyone had disparate functional verification environments and methodologies around Verilog. This was no longer feasible.

Industry recognized the need for a standard language that allowed the design *and* verification of a device and a methodology around which reusable components can be built avoiding multi-language cumbersome environments. Enter Superlog, which was a language with high level constructs required for functional verification. Superlog was donated (along with other language subset donations) to create SystemVerilog 3.0 from which evolved SystemVerilog 3.1, which added new features for design but over 80 % of the new language subset was dedicated to functional verification. We can only thank the Superlog inventor (the same inventor as that for Verilog—namely, Phil Moorby) and the Accellera technical subcommittees for having a long term vision to design such a robust all-encompassing language. No multi-language solutions were required any more. No more reinventing of the wheel with each project was required anymore.

As shown in Fig. 1.5, SystemVerilog Assertion language is derived from many different languages. Features from these languages either influenced the language or were directly used as part of the language syntax/semantic.

Sugar from IBM led to PSL. Both contributed to SVA. The other languages that contributed are Vera, 'e', CBV from Motorola and ForSpec from Intel.

In short, when we use SystemVerilog Assertions language, we have the benefit of using the latest evolution of an assertions language that benefited from many other robust assertions languages.

Chapter 2
SystemVerilog Assertions

2.1 What Is an Assertion?

Introduction: This chapter will start with definition of an assertion with simple examples, moving on to its advantages as applied to real life projects, who and what types of assertions need to be added for a given SoC project and the methodology components to successfully adopt assertions in your project.

An assertion is simply a check against the specification of your design that you want to make sure never violates. If the specs are violated, you want to see a failure.

A simple example is given below. Whenever FRAME_ is de-asserted (i.e. goes High), that the Last Data Phase (LDP_) must be asserted (i.e. goes Low). Such a check is imperative to correct functioning of the given interface. SVA language is precisely designed to tackle such temporal domain scenarios. As we will see in Sect. 2.2.1, modeling such a check is far easier in SVA than in Verilog. Note also that assertions work in temporal domain (and we will cover a lot more on this later); and are concurrent as well as multi-threaded. These attributes are what makes SVA language so suitable for writing temporal domain checks.

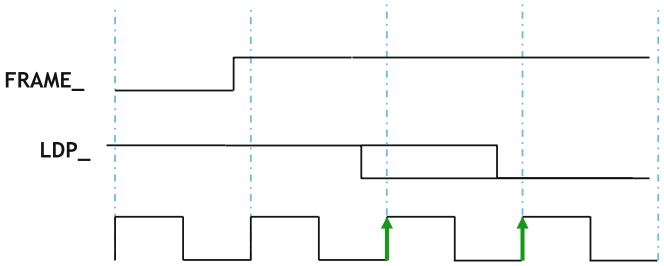
Figure 2.1 shows the assertion for this simple bus protocol. We will discuss how to read this code and how this code compares with Verilog in the immediately following Sect. 2.2.1.

2.2 Why Assertions? What Are the Advantages?

As we discussed in the introductory section, we need to increase productivity of the design/debug/simulate/cover loop. Assertions help exactly in these areas. As we will see, they are easier to write than standard Verilog or SystemVerilog (thereby increasing design productivity), easier to debug (thereby increasing debug productivity), provide functional coverage and simulate faster compared to the same assertion written in Verilog or SystemVerilog. Let us see these advantages one by one.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_2



When FRAME_ is de-asserted (high), LDP_ (last data phase) (low) must be asserted within the next 2 clocks

```
property ldpcheck;
  @(posedge clk) $rose (FRAME_) |-> ##[1:2] $fell (LDP_);
endproperty
aP: assert property (ldpcheck) else $display("ldpcheck FAIL");
cP: cover property (ldpcheck) $display("ldpcheck PASS");
```

Fig. 2.1 A simple bus protocol design and its SVA property

2.2.1 Assertions Shorten Time to Develop

Referring to the timing diagram in Fig. 2.1, let us see how SVA shortens time to develop. The SVA code is very self-explanatory. There is the property 'ldpcheck' that says "at posedge clock, if FRAME_ rises, it implies that within the next 2 clocks LDP_ falls". This is almost like writing the checker in English. We then 'assert' this property, which will check for the required condition to meet at every posedge clk. We also 'cover' this property to see that we have indeed exercised the required condition. But we are getting ahead of ourselves. All this will be explained in detail in coming chapters. For now, simply understand that the SV assertion is easy to write, easy to read and easy to debug.

Now examine the Verilog code for the same check (Fig. 2.2). There are many ways to write this code. One of the ways at behavioral level is shown. Here you 'fork' out two procedural blocks; one that monitors LDP and another that waits for 2 clocks. You then disable the entire block ('ldpcheck') when either of the two procedural blocks complete. As you can see that not only is the checker very hard to read/interpret but also very prone to errors. You may end up spending more time debugging your checker than the logic under verification.

Verilog Code

```
always @(posedge FRAME_)
begin:ldpcheck
  @(posedge clk);
  fork
    begin
      @(negedge LDP_) disable ldpcheck;
    end
    begin
      repeat (2) @(posedge clk); $display("ldpcheck FAIL");
      disable ldpcheck;
    end
  join
end
```

Fig. 2.2 Verilog code for the simple bus protocol

2.2.2 *Assertions Improve Observability*

One of the most important advantage of assertions is that they fire at the source of the problem. As we will see in the coming chapters, assertions are located local to temporal conditions in your design. In other words, you don't have to back trace a bug all the way from primary output to somewhere internal to the design where the bug originates. Assertions are written such that they are close to logic (e.g. @ (posedge clk) state0 |-> Read); Such an assertion is sitting close to the state machine and if the assertion fails, we know that when the state machine was in state0 that Read did not take place. Some of the most useful places to place assertions are FIFOs, Counters, block to block interface, block to IO interface, State Machines, etc. These constructs are where many of the bugs originate. Placing an assertion that check for local condition will fire when that local condition fails, thereby directly pointing to the source of the bug (Fig. 2.3).

Traditional verification can be called Black Box verification with Black Box observability, meaning, you apply vectors/transactions at the primary input of the 'block' without caring for what's in the block (blackbox verification) and you observe the behavior of the block only at the primary outputs (blackbox observability). Assertions on the other hand allow you to do black box verification with white box (internal to the block) observability.

2.2.3 *Assertions Provide Temporal Domain Functional Coverage*

Assertions not only help you find bugs but also help you determine if you have covered (i.e. exercised) design logic, mainly temporal domain conditions. They are very useful in finding temporal domain coverage of your testbench. Here is the reason why this is so important (Fig. 2.4).

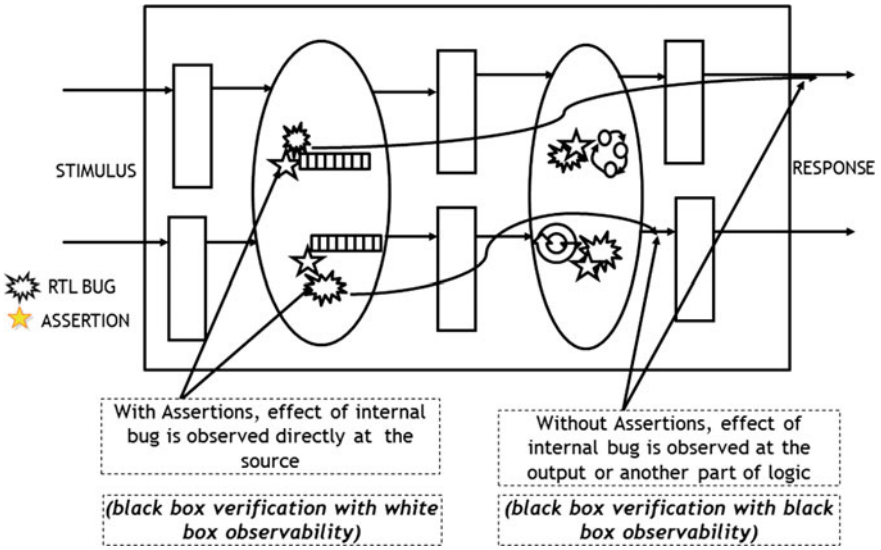


Fig. 2.3 Assertions improve observability

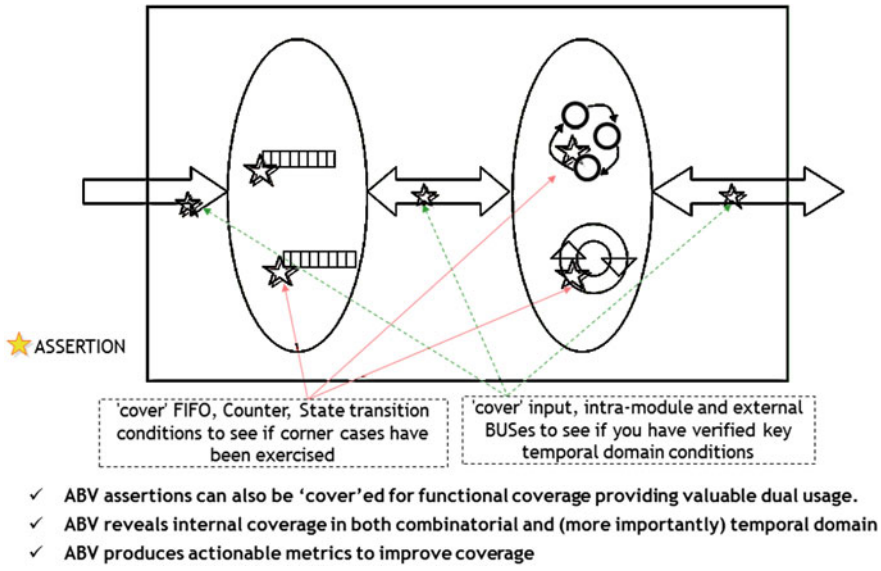


Fig. 2.4 SystemVerilog assertions provide temporal domain functional coverage

Let us say, you have been running regressions 24 * 7 and have stopped finding bugs in your design. Does that mean you are done with verification? No. Not finding a bug could mean one of two things. (1) There is indeed no bug left in the

design or (2) you have not exercised (or covered) the conditions that exercise the bugs. You could be continually hitting the same piece of logic in which no further bugs remain. In other words, you could be reaching a wrong conclusion that all the bugs have been found.

In brief, coverage includes three components (we will discuss this in detail in the chapter on Functional Coverage). (1) Code Coverage (which is structural) which needs to be 100 % (2) Functional Coverage that need to be designed to cover *functionality* (i.e. intent) of the entire design and must completely cover the design specification (3) temporal domain coverage (using SVA 'cover' feature) which need to be carefully designed to fully cover all required temporal domain conditions of the design.

Ok, let us go back to the simple bus protocol assertion that we saw in the previous section. Let us see how the 'cover' statement in that SVA assertion works. The code is repeated here for readability.

```
property ldpcheck;
  @(posedge clk) $rose (FRAME_) |-> ##[1:2] $fell (LDP_);
endproperty
aP: assert property (ldpcheck) else $display("ldpcheck FAIL");
cP: cover property (ldpcheck) $display("ldpcheck PASS");
```

In this code, you see that there is a 'cover' statement. What it tells you is "did you exercise this condition" or "did you cover this property". In other words, and as discussed above, if the assertion never fails, that could be because of two reasons. (1) you don't have a bug or (2) you never exercised the condition to start with! With the cover statement, if the condition gets exercised but does not fail you get that indication through the 'pass' action block associated with the 'cover' statement. Since we haven't yet discussed the assertions in any detail, you may not completely understand this concept but *determination of temporal domain coverage of your design is an extremely important aspect of verification and must be made part of your verification plan*.

To reiterate, SVA supports the 'cover' construct that tells you if the assertion has been exercised (covered). Without this indication and in the absence of a failure, you have no idea if you indeed exercised the required condition. In our example, if FRAME_ never rises, the assertion won't fire and obviously there won't be any bug reported. So, at the end of simulation if you do not see a bug or you do not even see the "ldpcheck PASS" display, you know that the assertion never fired. In other words, you must see the 'cover property' statement executed in order to know that the condition did get exercised. We will discuss this further in coming chapters. Use 'cover' to full extent as part of your verification methodology.

2.2.4 Assertion Based Methodology Allows for Full Random Verification

Huh! What does that mean? This example, I learnt from real life experience. In our projects, we always do full random concurrent verification (i.e. all initiators of the design fire at the same time to all targets of the design) after we are done with directed and constrained random verification. The idea behind this is to find any deadlocks (or livelock for that matter) in the design. Most of the initiator tests are well crafted (i.e. they won't clobber each other's address space) but with such massive randomness, your target model may not be able to predict response to randomly fired transactions. In all such cases, it is best to disable scoreboards in your target models (unless the scoreboards are full proof in that they can survive total randomness of transactions) BUT keep assertions alive. Now, fire concurrent random transactions, the target models will respond the best they can but assertions will pin point to a problem if it exists (such as simulation hang (deadlock) or simply keep bouncing between two state machines without advancing functionality (livelock)).

In other words, assertions are always alive and regardless of transaction stream (random or directed), they will fire as soon as there is the detection of an incorrect condition.

- Example Problem Definition:
  - Your design has Ethernet Receive and Video as Inputs and is also a PCI target.
  - It also has internal initiators outputting transactions to PCI targets, SDRAM, Ethernet Transmit and Video outputs.
  - After you have exhausted constrained random verification, you now want to simulate a final massive random verification, blasting transactions from all input interfaces and firing transactions from internal masters (DMA, Video Engine, Embedded processors) to all the output interfaces of the design.
  - BUT there's a good chance your reference models, self-checking tests, scoreboards may not be able to predict the correct behavior of the design under such massive randomness.
- Solution:
  - Turn off all your checking (reference models, scoreboards, etc.) unless they are full proof to massive random transaction streams.
  - BUT keep Assertions alive.
  - Blast the design with massive randomness (keep address space clean for each initiator).
  - If any of the assertions fire, you have found that corner case bug.

2.2.5 Assertions Help Detect Bugs not Easily Observed at Primary Outputs

This is a classic case that we encountered in a design and luckily found before tape-out. Without the help of an assertion, we would not have found the bug and there would have to be a complex software workaround. I will let the following example explain the situation.

- The Specification:
  - On a store address Error, the address in Next Address Register (NAR) should be frozen the same cycle that the Error is detected.
- The Bug:
  - On a store address error, the state machine that controls the NAR register actually froze the next address the *next* clock (instead of freezing the current address the same clock when store address error occurred). In other words, an incorrect address was being stored in NAR.
- So, why were the tests passing with this bug?
  - The tests that were triggering this bug used the same address back to back. In other words, even though the incorrect 'next' address was being captured in NAR, since the 'next' and the 'previous' addresses were the same, the logic would *seem* to behave correct.
  - The Assertion: An assertion was added to check that when a store address error was asserted the state machine should not move to point to the next address in pipeline. Because of the bug, the state machine actually did move to the next stage in pipeline. The assertion fired and the bug was caught.

2.2.6 Other Major Benefits

- SVA language supports Multi-Clock Domain Crossing (CDC) logic
  - SVA properties can be written that cross from one clock domain to another. Great for data integrity checks while crossing clock domains.
- Assertions are Readable: Great for documenting and communicating design intent
  - Great for creating executable specs.
  - Process of writing assertions to specify design requirements and conducting cross design reviews identify

    Errors, Inconsistencies, omissions, vagueness
  - Use it for design verification (test plan) review.

- Reusability for future designs
  - parameterized assertions (e.g. for a 16-bit bus interface) are easier to deploy with the future designs (with a 32-bit bus interface).
  - Assertions can be modeled outside of RTL and easily bound (using 'bind') to RTL keeping design and DV logic separate and easy to maintain.
- Assertions are always ON
  - Assertions never go to sleep (until you specifically turn them off).
  - In other words, active assertions take full advantage of every new test/stimulus configuration added by monitoring designs behavior against the new stimulus.
- Acceleration/Emulation with Assertions
  - Long latency and massive random tests need acceleration/emulation tools. These tools are beginning to support synthesizable assertions. Assertions are of great help in quick debug of long/random tests. We will discuss this further in coming sections.
- Global Severity Levels ($Error, $Fatal, etc.)
  - Helps maintain a uniform Error reporting structure in simulation.
- Global turning on/off of assertions (as in $dumpon/$dumpoff)
  - Easier code management (no need to wrap each assertion with an on/off condition).
- Formal Verification depends on Assertions
  - The same 'assert'ions used for design are also used directly by formal verification tools. Static formal applies its algorithms to make sure that the 'assert'ion never fails.
  - 'assume' allows for correct design constraint important to formal.
- One language, multiple usage
  - 'assert' for design check and for formal verification
  - 'cover' for temporal domain coverage check
  - 'assume' for design constraint for formal verification

2.3 How Do Assertions Work with an Emulator?

This section is to point out that assertions are not only useful in software based simulation but also hardware based emulation. The reason you can use assertions to fire directly in hardware is because assertions are synthesizable (well, at least the

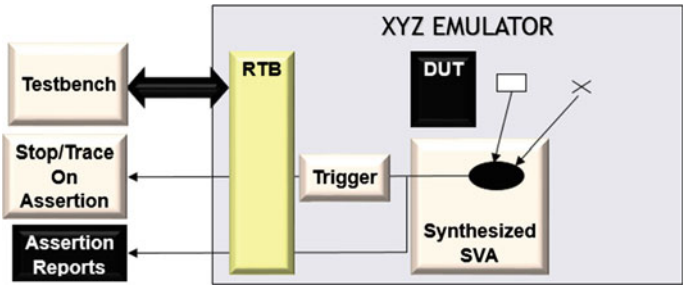


Fig. 2.5 Assertions for hardware emulation

simpler ones). Even though assertion synthesis has ways to go, there is enough of a subset covered by synthesis and that is sufficient to deploy assertions in hardware.

In Fig. 2.5 a generic emulation system is shown. Synthesizable assertions are part of the design that get synthesized and get partitioned to the emulation hardware. During emulation, if the design logic has a bug, the synthesized assertion will fire and trigger a stop/trace register to stop emulation and directly point to the cause of failure.

Anyone who has used emulation as part of their verification strategy, very well know that even though emulator may take seconds to 'simulate' the design, it takes hours thereafter to debug failures. Assertions will be a great boon to the debug effort. Many commercial vendors now support synthesizable assertions.

On the same line of thought, assertions can be synthesized in silicon as well. During post-silicon validation, a functional bug can fire and a hardware register can record the failure. This register can be reflected on GPIO of the chip or the register can be scanned out using JTAG boundary scan. The output can be decoded to determine which assertion fired and which part of silicon caused the failure. Without such facility, it takes hours of debug time to pin point the cause of silicon failure. This technique is now being used widely. The 'area' overhead of synthesized assertion logic is negligible compared to the overall area of the chip but the debug facilitation is of immense value. Note that such assertions can make it easier to debug silicon failures in field as well.

2.4 Assertions in Static Formal

The same assertions that you write for design verification can be used with static functional verification or the so-called hybrid of static functional plus simulation algorithms. Figure 2.6 shows (on LHS) SVA *Assumptions* and (on RHS/Center) SVA *Assertions*. As you see the assumptions are most useful to *Static Functional Verification* (*aka Formal*) (even though assumptions can indeed be used in

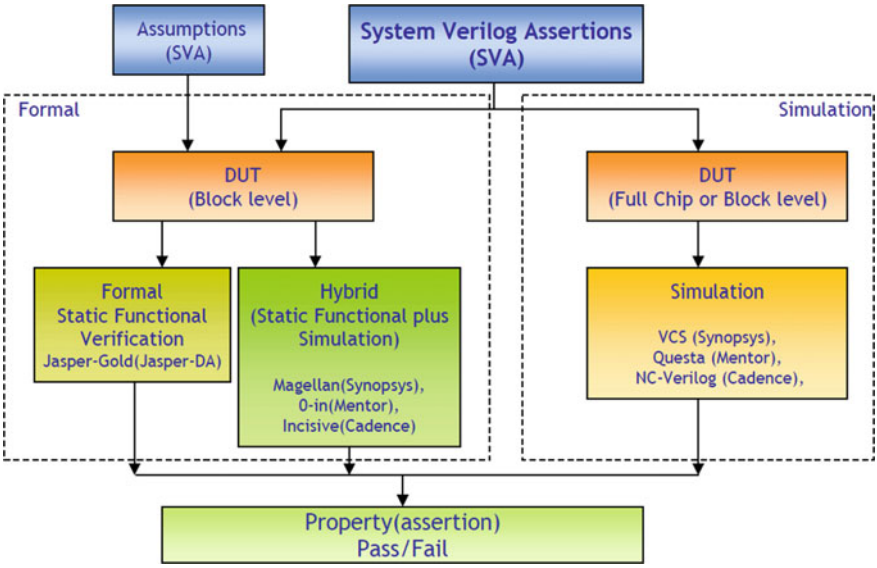


Fig. 2.6 Assertions and assumptions in formal (static functional) and simulation

Simulation as well, as we will see in later sections) while SVA assertions are useful in both Formal and Simulation.

So, what is Static Functional Verification (also called Static Formal Functional or simply Formal)? In plain English, static formal is a method whereby the static formal algorithm applies all possible combinational and temporal domain stimulus possibilities to exercise all possible 'logic cones' of a given logic block and see that the assertion(s) are not violated. This eliminates the need for a testbench and also makes sure that the logic never fails under *any* circumstance. This provides 100 % comprehensiveness to the logic under verification. So as a side note, why do we ever need to write a testbench? The static formal (as of this writing) is limited by the size of the logic block (i.e. gate equivalent RTL) especially if the temporal domain of inputs to exercise is large. The reason for this limitation is that the algorithm has to create different logic cones to try and prove that the property holds. With larger logic blocks, these so-called logic cones explode. This is also known as 'state space explosion' problem. To counter this problem, simulation experts came up with the *Hybrid Simulation* technique. In this technique, simulation is deployed to 'reach' closer to the assertion logic and then employ the static functional verification algorithms to the logic under assertion checking. This reduces the scope of the # of logic cones and their size and you may be successful in seeing that the property holds. Since static functional or hybrid is beyond the scope of this book, we'll leave it at that.

2.5 One-Time Effort, Many Benefits

Figure 2.7 shows the advantage of assertions. Write them once and use them with may tools.

We have discussed at high level the use of assertions in Simulation, Formal, Coverage and Emulation. But how do you use them for Testbench Generation/Checker and what is OVL assertions library?

Testbench Generation/Checker: With ever-increasing complexity of logic design, the testbenches are getting ever so complex as well. How can assertions help in designing testbench logic? Let us assume that you need to drive certain traffic to a DUT input under certain condition. You can design an assertion to check for that condition and upon its detection the FAIL action block triggers, which can be used to drive traffic to the DUT. Checking for a condition is far easier with assertions language than with SystemVerilog alone. Second benefit is to place assertions on verification logic itself. Since verification logic (in some cases) is even more complex than the design logic, it makes sense to use assertions to check testbench logic also.

OVL Library: Open Verification Library. This library of predefined checkers was written in Verilog before PSL and SVA became mainstream. Currently the library includes SVA (and PSL) based assertions as well. The OVL library of assertion checkers is intended for use by design, integration and verification engineers to check for good/bad behavior in simulation, emulation and formal verification. OVL contains popular assertions such as FIFO assertions, among

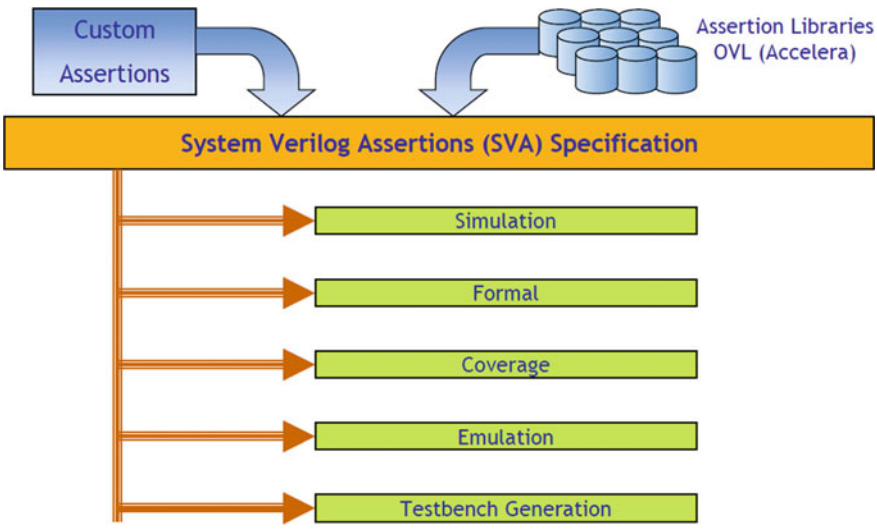


Fig. 2.7 Assertions and OVL for different uses

other. OVL is still in use and you can download the entire standard library from Accellera website. http://www.accellera.org/downloads/standards/ovl.

We will not go into the detail of OVL since there is plenty of information available on OVL on net. OVL code itself is quite clear to understand. It is also a good place to see how assertions can be written for 'popular' checks (e.g. FIFO) once you have better understood assertion semantics.

2.6 Assertions Whining

Maybe the paradigm has now shifted but as of this writing there is still a lot of hesitation on adopting SVA in the overall verification methodology. Here are some popular objections.

- *I don't have time to add assertions. I don't even have time to complete my design. Where am I going to find time to add assertions?*
  - That depends on your definition of "completing my design". If the definition is to simply add all the RTL code without—any—verification/debug features in the design and then throw the design over the wall for verification, it will take significantly longer to debug your design for it to work as specified.
  - During design you are already contemplating and assuming many conditions (state transition assumptions, inter-block protocol assumptions, etc.). *Simply convert your assumptions into assertions as you design*. They will go a long way in finding those corner case bugs even with your simple sanity testbenches.
- *I don't have time to add assertions. I am in the middle of debugging the bugs already filed against my design*.
  - Well, actually you will be able to debug your design in shorter time, if you *did* add assertions as you were designing (or at least add them now) so that if a failing test fires an assertion, your debug time will be drastically short.
  - Assertions point to the source of the bug and significantly reduce time to debug as you verify your block level, chip level design.
  - In other words, this is a bit of chicken and egg problem. You don't have time to write assertions but without these assertions you will spend a lot more time debugging your design!
- *Isn't writing assertions the job of a verification engineer?*
  - Not quite. Design Verification (DV) engineers do not have insight into the micro-architectural level RTL detail. But the real answer is that BOTH Design and DV engineers need to add assertions. We will discuss that in detail in upcoming section.

- *DV engineer says: I am new to assertions and will spend more time debugging my assertions than debugging the design*
  - Well, don't you spend time in debugging your testbench logic? Your reference models? What's the difference in debugging assertions? If anything, assertions have proven to be very effective in finding bugs and cutting down on debug time.
  - In my personal experience (over the last many SoC and Processor projects), approximately 25 % of the total bugs reported for a project were DV/Testbench bugs. There are significant benefits to adding assertions to your testbench that outweigh the time to debug them.
- *The designer cannot be the verifier also. Doesn't asking a designer to add assertions violate this rule?*
  - As we will discuss in the following sections, assertions are added to check the 'intent' of the design and validate your own assumptions. *You are not writing assertions to duplicate your RTL*. Following example makes it clear that the designer does need to add assertions.
  - For example,

    For every 'req' issued to the next block, I will indeed get an 'ack' and that I will get only 1 'ack' for every 'req'. This is a cross module assumption which has nothing to do with how you have designed your RTL. You are not duplicating your RTL in assertions.
    My state machine should never get stuck in any 'state' (except 'idle') for more than 10 clocks.

2.6.1 Who Will Add Assertions? War Within!

Both Design and Verification engineers need to add assertions...

- Design Engineers:
  - Micro architectural level decisions/assumptions are not visible to DV engineers. So, designers are best suited to guarantee uArch level logic correctness.
  - Every assumption is an assertion. If you assume that the 'request' you send to the other block will always get an 'ack' in 2 clocks; that's an assumption. So, design an assertion for it.
  - Add assertions as you design your logic, not as an afterthought.

- DV Engineers:
  - Add assertions to check macro functions and Chip/SoC level functionality.

    Once the packet has been processed for L4 layer, it will indeed show up in the DMA queue.
    A machine check exception indeed sets PC to the exception handler address.
  - Add assertions to check Interface IO logic

    After Reset is de-asserted none of the signals ever go 'X'.
    If the processor is in Wait Mode and no instructions are pending that it must assert a SleepReq to memory subsystem within 100 clocks.
    On Critical Interrupt, the external clock/control logic block must assert CPU_wakeup within 10 clocks.

2.7 A Simple PCI Read Example—Creating an Assertions Test Plan

Let us consider a simple example of PCI Read. Given the specification in Fig. 2.8, what type of assertions would the design team add and what type would the verification team add? The tables below describe the difference. I have only given few of the assertions that could be written. There are many more assertions that

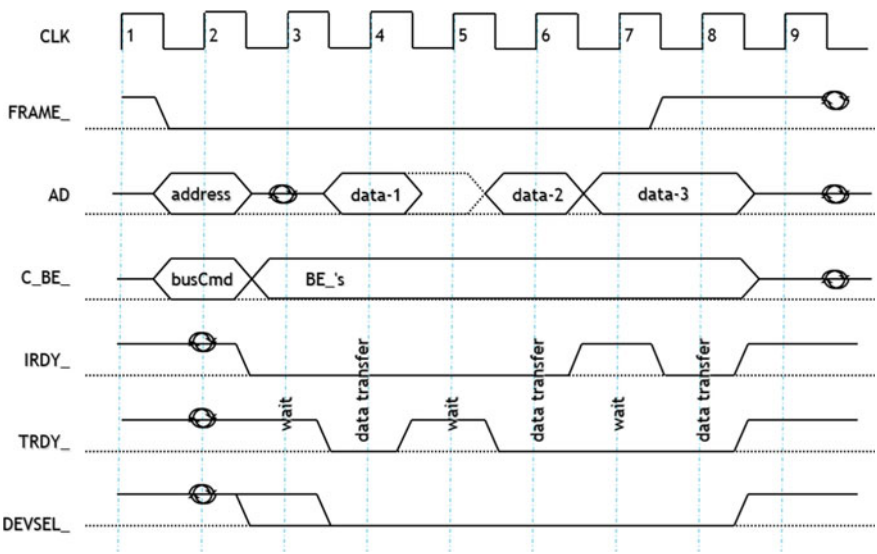


Fig. 2.8 A simple PCI read protocol

need to be written by verification and design engineers. However, this example will act as a basis for differentiation.

Designers add assertions at micro-architecture level while verification engineers concentrate at system level, specifically the interface level in this example.

We will model the assertions for this PCI protocol later in the book under LAB6 exercise. It is too early to jump into writing assertions without knowing the basics at this stage.

The PCI protocol is for a simple READ. With FRAME_ assertion, AD address and C_BE_ have valid values. Then IRDY_ is asserted to indicate that the master is ready to receive data. Target transfers data with intermittent wait states. Last data transfer takes place a clock after FRAME_ is de-asserted.

Let us see what type of assertions need to be written by design and verification engineers (Tables 2.1 and 2.2).

Note that in the table there are two columns. (1) Did the property FAIL? (2) Did the property get covered? There is no column for the property PASS. That is because, 'cover' in an assertion triggers only when a property is exercised but does not fail; in other words, it passes. Hence, there is no need for a PASS column. This 'cover' column tells you that you indeed covered (exercised) the assertion and that it did not fail. When the assertion FAILs, it tells you that the assertion was exercised (covered) and that it Failed during the exercise.

Table 2.1 PCI read protocol test plan by functional verification team

PCI: basic read protocol test plan: verification team			
Property name	Description	Property FAIL?	Property covered?
Protocol interface assertions			
checkPCI_AD_CBE (check1)	On falling edge of FRAME_, AD and C_BE_ bus cannot be unknown		
checkPCI_DataPhase (check2)	When both IRDY_ and TRDY_ are asserted, AD or C_BE_ bus cannot be unknown		
checkPCI_Frame_Irdy (check3)	FRAME can be de-asserted only if IRDY_ is asserted		
checkPCI_trdyDevsel (check4)	TRDY_ can be asserted only if DEVSEL_ is asserted		
checkPCI_CBE_during_trx (check5)	Once the cycle starts (i.e. at FRAME_ assertion) C_BE_ cannot float until FRAME_ is de-asserted		

Table 2.2 PCI read protocol test plan by design team

PCI: basic read protocol test plan: design team			
Property name	Description	Property FAIL?	Property covered?
Microarchitectural assertions			
check_pci_adrcbe_St	PCI state machine is in 'adr_cbe' state the first clock edge when FRAME_ is found asserted		
check_pci_data_St	PCI state machine is in 'data_transfer' state when both IRDY_ and TRDY_ are asserted		
check_pci_idle_St	PCI state machine is in 'idle' state when both FRAME_ and IRDY_ are de-asserted		
check_pci_wait_St	PCI state machine is in 'wait' state if either IRDY_ or TRDY_ is de-asserted		

2.8 What Type of Assertions Should I Add?

It is important to understand and plan for the types of assertions one needs to add. Make this part of your verification plan. It will also help you partition work among your team.

Note the 'performance implication' assertions. Many miss on this point. Coming from processor background, I have seen that these assertions turn out to be some of the most useful assertions. These assertions would let us know of the (e.g.) cache read latency upfront and would allow us enough time to make architectural changes.

- RTL Assertions (design intent)
  - Intra Module

    Illegal state transitions; deadlocks; livelocks;
    FIFOs, onehot, etc.
- Module interface Assertions (design interface intent)
  - Inter-module protocol verification; illegal combinations (ack cannot be '1' if req is '0'); steady state requirements (when slave asserts write_queue_full, master cannot assert write_req);
- Chip functionality Assertions (chip/SoC functional intent)
  - A PCI transaction that results in Target Retry will indeed end up in the Retry Queue.

- Chip interface Assertions (chip interface intent)
  - Commercially available standard bus assertion VIPs can be useful in comprehensive check of your design's adherence to std. protocol such as PCIe, AXI, etc.
  - Every design assumption on IO functionality is an assertion.
- Performance Implication assertions (performance intent)
  - Cache latency for read; packet processing latency; etc. to catch performance issues before it's too late. This assertion works like any other. For example, if the 'Read Cache Latency' is greater than 2 clocks, fire the assertion. This is an easy to write assertion with very useful return.

2.9 Protocol for Adding Assertions

- Do not duplicate RTL
  - White box observability does not mean adding an assertion for each line of RTL code. This is a very important point, in that if RTL says 'req' means 'grant', don't write an assertion that says the same thing!! Read on.
  - Capture the intent

    For example, a Write that follows a Read to the same address in the request pipe will always be allowed to finish before the Read. This is the intent of the design. How the designer implements reordering logic is not of much interest. So, from verification point of view, you need to write assertions that verify the chip specifications.
    A note here that the above does not mean you do not add low-level assertions. Classic example here is FIFO assertions. Write FIFO assertions for all FIFOs in your design. FIFO is low-level logic, but many of the critical bugs hang around FIFO logic and adding these assertions will provide maximum bang for your buck.
- Add assertions throughout the RTL design process
  - They are hard to add as an after-thought.
  - Will help you catch bugs even with your simple block level testbench.
- If an assertion did not catch a failure…
  - If the test failed and none of the assertions fired, see if there are assertions that need to be added which would fire for the failing case.
  - The newly added assertion is now active for any other test that may trigger it. Note: This point is very important towards making a decision if you have added enough assertions. In other words, if the test failed and none of the assertions fired, there is a good chance you still have more assertions to add.

- Reuse
  - Create libraries of common 'generic' properties with formal arguments that can be instantiated (reused) with 'actual' arguments. We will cover this further in the book.
  - Reuse for the next project.

2.10 How Do I Know I Have Enough Assertions?

- It's the "Test plan, test plan, test plan…"
  - Review and re-review your test plan against the design specs.
  - Make sure you have added assertions for every 'critical' function that you must guarantee works.
- If tests keep failing but assertions do not fire, you do not have enough assertions.
  - In other words, if you had to trace a bug from primary outputs (of a block or SoC) without any assertions firing that means that you did not put enough assertions to cover that path.
- 'formal' (aka static formal aka static functional verification) tool's ability to handle assertions
  - What this means is that if you don't have enough 'assertion density' (meaning if a register value does not propagate to an assertion within 3–5 clocks—resulting in assertions sparsely populated within design), the formal analysis tool may give up on the state/space explosion problem. In other words, a static functional formal tool may not be able to handle a large temporal domain. If the assertion density is high, the tool has to deal with smaller cone of logic. If the assertion density is sparse, the tool has to deal with larger cone of logic in both temporal and combinatorial space and it may run into trouble.

2.11 Use Assertions for Specification and Review

- Use assertions (properties/sequences) for specification
  - DV (Design Verification) Team:

    Document as much of the 'response checking' part of your test plan as practical directly into executable properties.
    Use it for verification plan review and update.

  - Design Team:

    Document micro-arch. level assertions directly into executable properties. Use it for design reviews.

- Assertions Cross-Review
  - Review:

    DV team reviews macro, chip, interface level assertions with the design team.

  - Cross Review

    Block A designer reviews 'module B' interface assertions
    Block B designer reviews 'module A' interface assertions

  - Mis-assumptions, incorrect communication are detected early on.

2.12 Assertion Types

There are three kinds of assertions supported by SVA. In brief, here's their description. We will discuss them in plenty detail throughout this book.

- Immediate Assertion
- Concurrent Assertion
- Deferred Assertion (aka Deferred Immediate Assertion) (introduced in IEEE 1800-2009)

Immediate Assertions

- Simple *non-temporal domain assertions* that are executed like statements in a procedural block,
- Interpreted the same way as an expression in the conditional of a procedural 'if' statement,
- Can be specified only where a procedural statement is specified.

Concurrent Assertions

- These are *temporal domain assertions* that allow creation of complex sequences using clock (sampling edge) based semantics.
- They are edge sensitive and not level sensitive. In other words, they must have a 'sampling edge' on which it can sample the values of variables used in a sequence or a property.

Deferred Assertions (introduced in IEEE 1800-2009)

- Deferred assertions are a type of Immediate assertions. Note that immediate assertions evaluate immediately without waiting for variables in its combinatorial expression to *settle* down. This also means that the immediate assertions

are very prone to glitches as the combinatorial expression settles down and may fire multiple times. On the other hand, deferred assertions do not evaluate their sequence expression until the end of time stamp when all values have settled down (or in the reactive region of the time stamp). Detailed explanation is in Sect. 16.2.

If some of this does not quite make sense, that's OK. That is what the rest of the book will explain. Let us start with Immediate assertions and understand its semantics. We then move on to Concurrent assertions and lastly Deferred assertions. The book focuses on concurrent assertions because that is really the main gist of SystemVerilog Assertion Language.

2.13 Conventions Used in the Book

Note that the level sensitive attribute of a signal is shown as a 'fat' High and Low symbol. I could have drawn regular timing diagrams but saw that they look very cumbersome and does not easily convey the point. Hence, I chose the fat arrow to convey that ***when the fat arrow is high, the signal was high before the clock; at the clock and after the clock. The same applies for the fat low arrow*** (Table 2.3).

Table 2.3 Conventions used in this book

	**LEVEL SENSITIVE HIGH:** This symbol means that the signal is detected HIGH (level sensitive) at the clock edge noted in a timing diagram. It could have been high or low the previous clock and may remain high or low after the clock edge. *It does NOT however mean that a 'posedge' is expected on this signal at the noted clock edge.*
	**LEVEL SENSITIVE LOW:** This symbol means that the signal is detected LOW (level sensitive) at the clock edge noted in a timing diagram. It could have been high or low the previous clock and may remain high or low after the clock edge. *It does NOT however mean that a 'negedge' is expected on this signal at the noted clock edge.*
	**EDGE SENSITIVE HIGH:** This symbol means that a posedge is expected on this signal.
	**EDGE SENSITIVE LOW:** This symbol means that a negedge is expected on this signal.
	**PROPERTY PASSes:** This symbol means that a sequence/property match is detected here (i.e. the sequence/property PASSes).
	**PROPERTY FAILs:** This symbol means that a sequence/property did not match here (i.e. the sequence/property FAILs).

For edge sensitive assertions, I chose the regular timing diagram to distinguish them from the level sensitive symbol.

A high (thin) arrow is for PASS and a low (thin) arrow is for FAIL.

Chapter 3
Immediate Assertions

Introduction: This chapter will introduce the 'Immediate' assertions (immediate 'assert', 'cover', 'assume') starting with a definition and leading to detailed nuances of its semantics and syntax.

Immediate assertions are simple non-temporal domain assertions that are executed like statements in a procedural block. Interpret them as an expression in the condition of a procedural 'if' statement. Immediate assertions can be specified only where a procedural statement is specified. The evaluation is performed immediately with the values taken at that moment for the assertion condition variables. The assertion condition is non-temporal, which means its execution computes and reports the assertion results at the *same* time.

Figure 3.1 describes the basics of an immediate assertion. It is so called because it executes immediately at the time it is encountered in the procedural code. It does not wait for any temporal time (e.g. 'next clock edge') to fire itself. The assertion can be preceded by a level sensitive or an edge sensitive statement. As we will see that concurrent assertions can only work on a 'sampling/clock edge' sensitive logic and not level sensitive logic.

We see in Fig. 3.1 that there is an immediate assertion embedded in the procedural block that is triggered by @ (posedge clk). The immediate assertion is triggered after @ (posedge d) and checks to see that (b || c) is true.

We need to note a couple of points here. First, the very preceding statement in this example is @ (posedge d), an edge sensitive statement. However, it does not have to be. It can be a level sensitive statement also or any other procedural statement. The reason I am pointing this out is that concurrent assertions can work only off of a sampling 'edge' and not off of a level sensitive control. Keep this in your back pocket because it will be very useful to distinguish immediate assertions from concurrent assertions when we cover the latter. Second, the assertion itself cannot have temporal domain sequences. In other words, an immediate assertion cannot consume 'time'. It can only be combinatorial which can be executed in zero time. In other words, the assertion will be computed and results will be available at

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_3

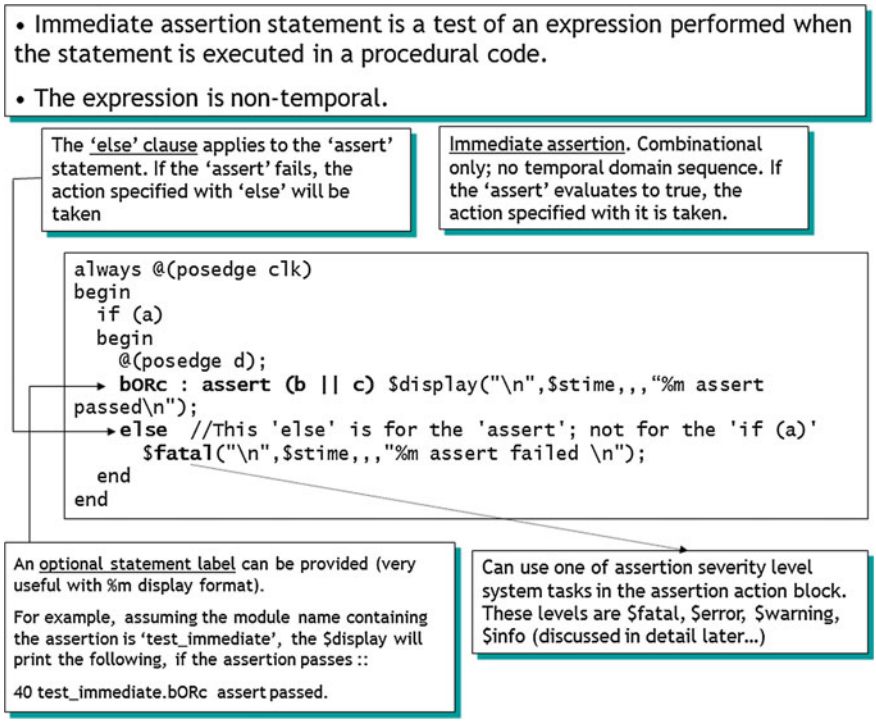


Fig. 3.1 Immediate assertion—basics

the *same* time that the assertion was fired. If the 'assert' statement evaluates to 0, X, Z then the assertion will be considered to FAIL else it will be considered to PASS.

We also see in the figure that there is (what is known as) an Action Block associated with FAIL or PASS of the assertion. This is no different than the PASS/FAIL logic we design for an 'if…else' statement.

From syntax point of view, an immediate assertion uses only "assert" as the keyword in contrast to a concurrent assertion that requires "assert property".

One key difference between immediate and concurrent assertions is that concurrent assertions always work off of the sampled value in preponed region (see Sect. 4.3) of a simulation tick while immediate assertions work immediately when they are executed (as any combinatorial expression in a procedural block) and do not evaluate its expression in the preponed region. Keep this thought in your back pocket for now since we haven't yet discussed concurrent assertions and how assertions get evaluated in a simulation time tick. But this key difference will become important to note as you learn more about concurrent assertions.

Finally, as we discussed above, the immediate assertion works on a combinatorial expression whose variables are evaluated 'immediately' at the time the expression is evaluated. These variables may transition from one logic value to another (e.g. 1 to 0 to 1) within a given simulation time tick and the immediate

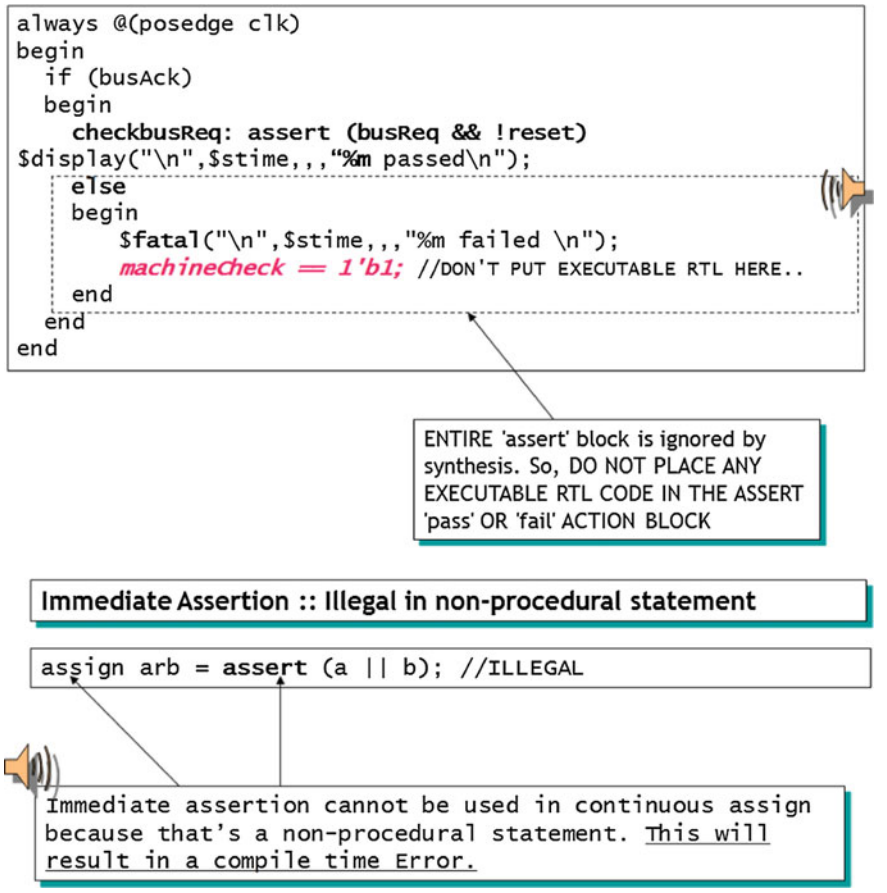


Fig. 3.2 Immediate assertions: finer points

assertion may get evaluated multiple times before the expression variable values 'settle' down. This is why immediate assertions are also known to be 'glitch' prone. This is where the 'deferred immediate' assertions come into picture. We will discuss those in Sect. 16.2.

To complete the story, there are three types of immediate assertions.

immediate **assert**
immediate **assume**
immediate **cover**

Note also that the book contains a lot more information on other types assertions that can be called from a procedural block (just as you call immediate assertions). For example, you can call a 'property' or a 'sequence' (or 'restrict' for formal verification—see Sect. 16.15) from a procedural block.

'assume', 'cover' are too early to discuss. And also I haven't discussed 'property' and 'sequence' yet (see Chap. 4).

Moving on,

Figure 3.2 points out a couple of finer points. First, do not put anything in the so-called action block (PASS or FAIL) of the immediate assertion. Most synthesis tools will simply ignore the entire immediate assertion with its action blocks (which makes sense) and with it will go your logic that (if) you were planning on putting in your design. This is rather obvious but easy to miss.

Note that an immediate assertion cannot be used in a continuous assignment statement because continuous assign is not a procedural block.

Lastly, please note that IEEE-1800 2009-2012 LRM also defines another type of immediate assertion, which is called Deferred Assertion. We will discuss deferred assertion under Sect. 16.2.

Chapter 4
Concurrent Assertions—Basics (Sequence, Property, Assert)

Introduction: This chapter introduces basics of concurrent assertions namely 'sequence', 'property', 'assert', 'cover' and 'assume'. It discusses fine grained nuances of Clocking of concurrent assertions and also implication operators, multi-threaded semantics, formal arguments, 'bind'ing of assertions, severity levels, among other topics.

Concurrent assertions are temporal domain assertions that allow creation of complex sequences which are based on *clock* (*sampling*) *edge* semantics. This is in contrast to the immediate assertions that are purely combinatorial and do not allow temporal domain sequences.

Concurrent assertions are the gist of SVA language. They are called concurrent because they execute in parallel with the rest of the design logic and are multi-threaded. Let us start with basics and move onto the complex concepts of concurrent assertions.

Let us first learn the basic syntax of a concurrent assertion and then study its semantics.

In Fig. 4.1 we have declared a property 'pr1' and asserted it with a label 'reqGnt' (label is optional but highly recommended). The figure explains various parts of a concurrent assertion including a property; a sequence and assertion of the property.

The 'assert property (pr1)' statement triggers property 'pr1'. 'pr1' in turn waits for the antecedent 'cStart' to be true at a (posedge clk) and on it being true implies (fires) a sequence called 'sr1'. 'sr1' checks to see that 'req' is high when it is fired and that 2 'clocks' later 'gnt' is true. If this temporal domain condition is satisfied, then the sequence 'sr1' will PASS and so will property 'pr1' and the 'assert property' will be a PASS as well. Let us continue with this example and study other key semantics.

As explained in Fig. 4.2, following are the basic and mandatory parts of an assertion. Each of these features will be further explored as we move along.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_4

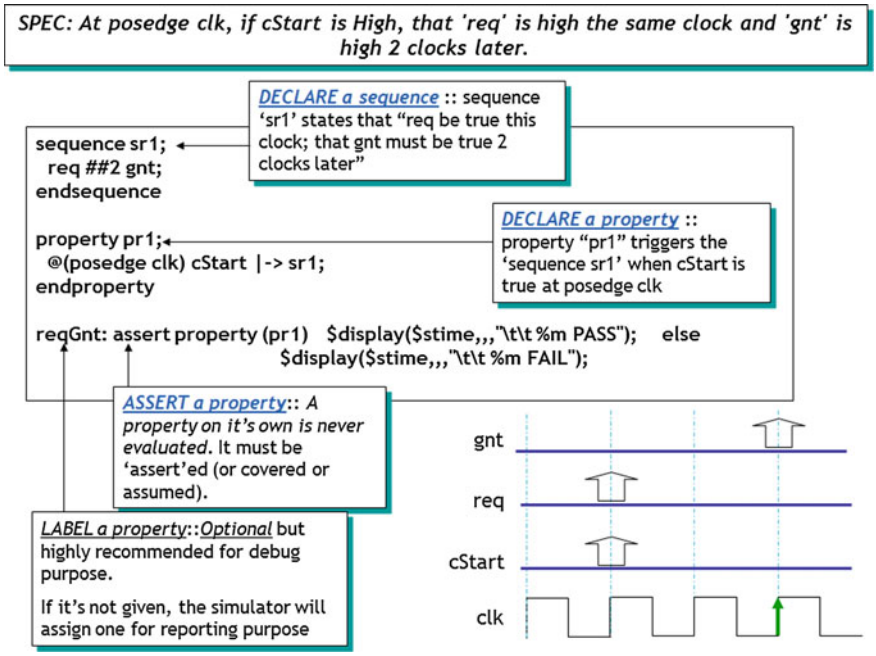


Fig. 4.1 Concurrent assertion—basics

1. 'assert'—you have to assert a property; i.e. invoke or trigger it.
2. There is an action block associated with either the pass or fail of the assertion.
3. 'property pr1' is edge triggered on posedge of clk (more on the fact that you *must* have a sampling edge for trigger is explained further on).
4. 'property pr1' has an *antecedent* which is a signal called cStart, which if sampled high (in the preponed region) on the posedge clk, will imply that the *consequent* (sequence sr1) be executed.
5. Sequence sr1 samples 'req' to see if it is sampled high the same posedge of clk when the sequence was triggered because of the *overlapping implication* operator and then waits for 2 clocks and sees if 'gnt' is high.
6. Note that each of 'cStart', 'req', 'gnt' are *sampled* at the edge specified in the property which is the posedge of 'clk'. In other words, even though there is no edge specified in the sequence, the edge is inherited from property pr1.
   Note also that we are using the notion of sampling the values at posedge clk which means that the 'posedge clk' is the '*sampling edge*'. In other words, the sampling edge can be anything (as long as it's an edge and is not level sensitive), meaning it does not necessarily have to be a synchronous edge such as a clock. It can be an asynchronous edge as well. However, *be very careful about using an asynchronous edge* unless you are sure what you want to achieve. I have devoted a complete example (see Chap. 15) on the pitfalls of using an asynchronous edge as the sampling edge. It's too soon to get into that. This is a

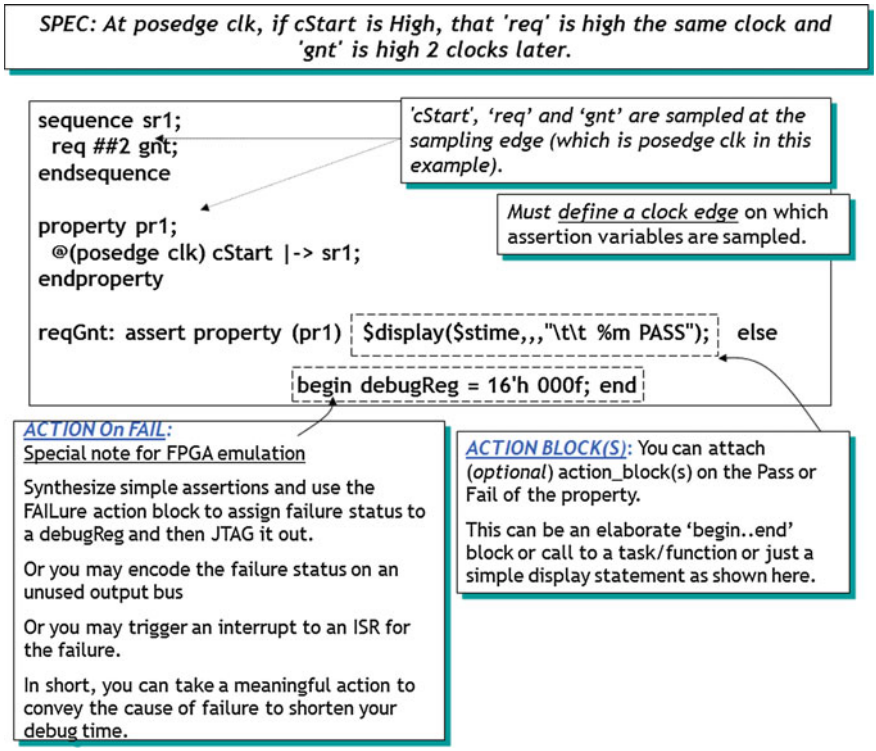


Fig. 4.2 Concurrent assertion—sampling edge and action blocks

very important concept in concurrent assertions and should be well understood. However, do not worry, you will get much more insight as we move further.

Now, let us slightly modify the sequence 'sr1' to highlight Boolean expression in a sequence or a property and study some more key elements of a concurrent assertion.

As shown in Fig. 4.3 there are three main parts of the expression that determines when an assertion will fire, what it will do once fired and time duration between the firing event and execution event.

The condition under which an assertion will be fired is called an '*antecedent*'. This is the LHS of the implication operator.

RHS of the assertion that executes once the antecedent matches is called the *'consequent*'.

The way to 'read' the implication operator is "if there is a match on the antecedent that the consequent will be executed". If there is no match, consequent will not fire and the assertion will continue to wait for a match on the antecedent. The '*implication*' operator also determines the time duration that will lapse between the antecedent match and the consequent execution. In other words, the implication operator ties the antecedent and the consequent in one of two ways. It ties them with

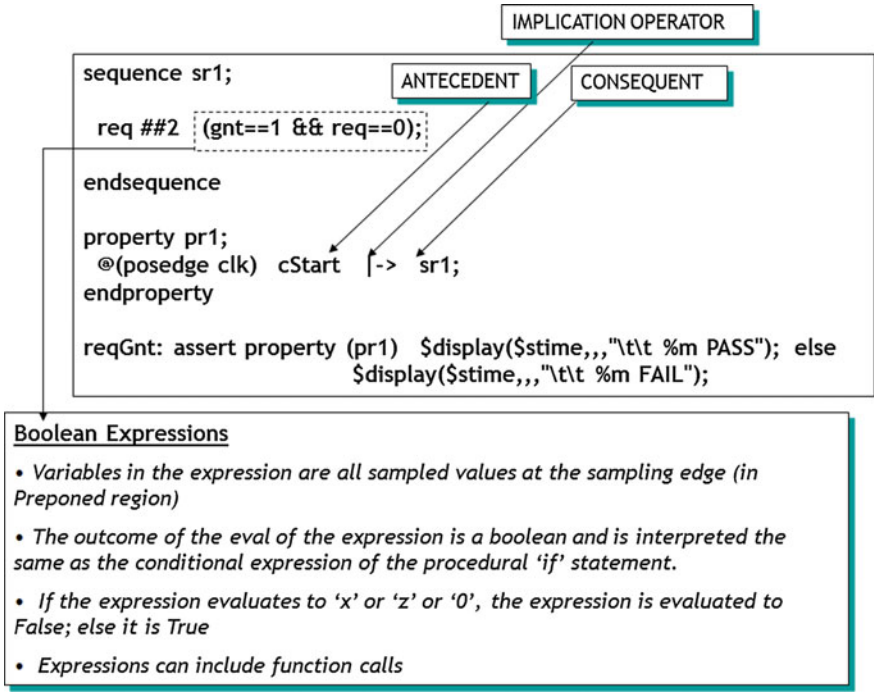


Fig. 4.3 Concurrent assertion—implication, antecedent and consequent

an '*overlapping*' implication operator or a '*non-overlapping*' implication operator. More on this coming up…

One additional note on Boolean Expressions before we move on. Following types are not allowed for the variables used in a Boolean Expression.

- dynamic Arrays
- class
- string
- event
- real, shortreal, realtime,
- associative Arrays
- chandle

Here are explicit rules that govern Boolean Expressions

– An expression must result in a type that is cast compatible with an integral type. Subexpressions need not meet this requirement as long as the overall expression is cast compatible with an integral type.
– Elements of dynamic arrays, queues, and associative arrays that are sampled for assertion expression evaluation may get removed from the array or the array may get resized before the assertion expression is evaluated. These specific array

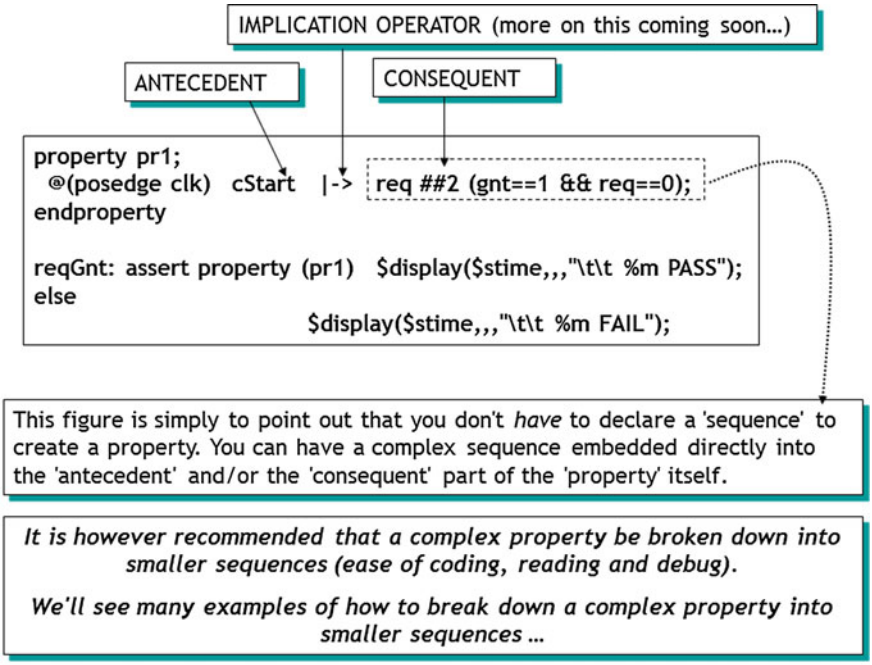


Fig. 4.4 Property with an embedded sequence

elements sampled for assertion expression evaluation must continue to exist within the scope of the assertion until the assertion expression evaluation completes.
- Expressions that appear in *procedural* concurrent assertions may reference automatic variables. Otherwise, expressions in concurrent assertions shall not reference automatic variables. Procedural concurrent assertions are discussed in Sect. 14.2.
- Expressions must not reference non-static class properties or methods.
- Expressions must not reference variables of the **chandle** data type.
- Functions that appear in expressions must not contain output or ref arguments (const ref is allowed).

Figure 4.4 further explains the antecedent and consequent. As shown, you don't have to have a sequence in order to model a property. If the logic to execute in consequent is simple enough, then it can be declared directly in consequent as shown. But please note that *it is always best to break down a property into smaller sequences to model complex properties/sequences.* Hence, consider this example only as describing the semantics of the language. Practice should be to divide and conquer. You will see many examples, which seem very complex to start with, but once you break them down into smaller chunks of logic and model them with smaller sequences, tying all those together will be much easier then writing one long complex assertion sequence.

4.1 Implication Operator, Antecedent and Consequent

Implication operator ties the antecedent and consequent. If antecedent holds true it implies that the consequent should hold true.

There are two types of implication operators as shown in Fig. 4.5

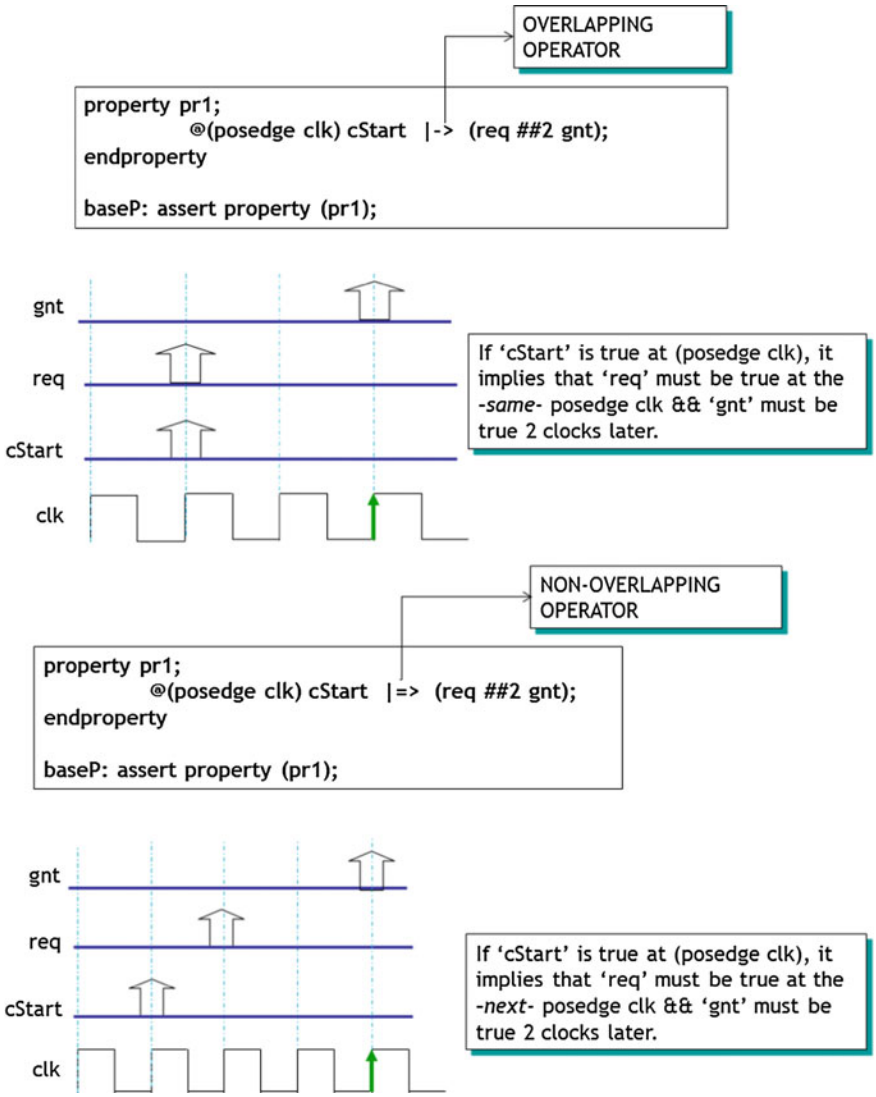


Fig. 4.5 Implication operator—overlapping and non-overlapping

1. Overlapping Implication Operator: Referring to Fig. 4.5, the top most property shows the use of an overlapping operator. Note its symbol (|->), which differs from that of the non-overlapping operator (|=>). Overlapping means that when the antecedent is found to be true, that the consequent will start its execution (evaluation) the '*same*' clk. As shown in the figure, when cStart is sampled High at posedge of clk that the req is required to be High at the '*same*' posedge clk. This is shown in the timing diagram associated with the property.
   a. So, what happens if 'req' is sampled true at the next posedge clk after the antecedent (and false before that)? Will the overlapping property pass?
2. Non-overlapping Implication Operator: In contrast, non-overlapping means that when the antecedent is found to be true that the consequent should start its execution, one clk *later*. This is shown in the timing diagram associated with the property.
   a. So, what happens if 'req' is sampled true at the same posedge clk as the antecedent (and False after that)? Will the non-overlapping property pass?

Answer to both 1.an and 2.a is NO.

Figure 4.6 further shows the equivalence between overlapping and non-overlapping operators. '|=>' is equivalent to '|-> ##1'. Note that ##1 is not the same as Verilog's #1 delay. ##1 means one clock edge (sampling edge). Hence '|-> ##1' means the same as '|=>'.

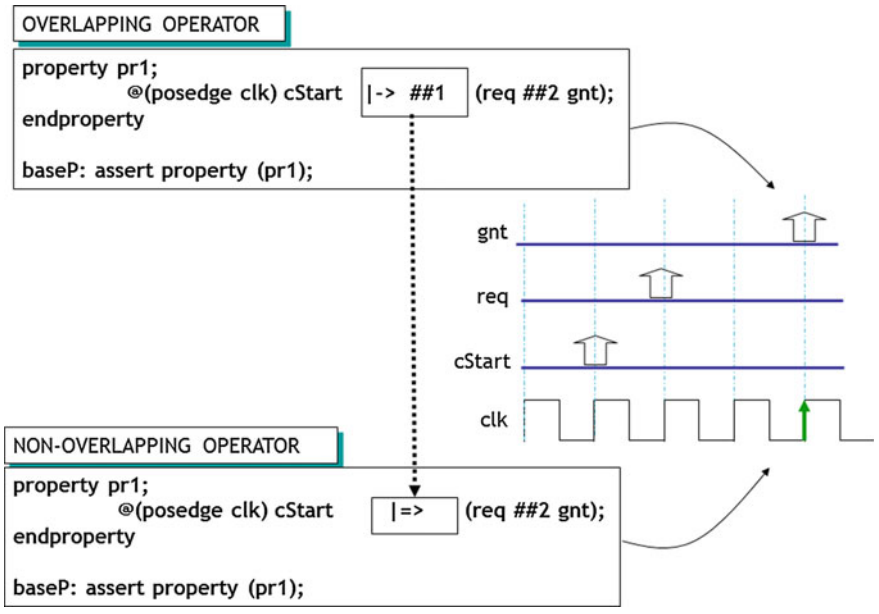


Fig. 4.6 Equivalence between overlapping and non-overlapping implication operators

Suggestion: To make debugging easier and have project wide uniformity, use the overlapping operator in your assertions. Reason? Overlapping is the common denominator of the two types of operator. You can always model non-overlapping from overlapping and but you cannot do vice versa. What this means is that during debug everyone would know that all the properties are modeled using overlapping and that the # of clocks are exactly the same as specified in the property. You do not have to add or subtract from the # of clocks specified in the chip specification. More important, if everyone uses his or her favorite operator, debugging would be very messy not knowing which property uses which operator.

Finally, do note that concurrent assertions can be placed:

1. in 'always' procedural block
2. in 'initial' procedural block
3. standalone (static)—outside of the procedural block—which is what we have seen so far. 1 and 2 are described in Sect. 14.2.

4.2 Clocking Basics

As mentioned before, a concurrent assertion is evaluated only on the occurrence of an 'edge', known as the 'sampling edge'. The reason for continually mentioning this 'edge' as 'clk' is because it is best to have this 'edge' synchronous to either posedge or negedge for a signal. You can indeed have an asynchronous edge as well. BUT be very careful. I have devoted a complete example precisely to explain how an assertion with an asynchronous edge works. In Fig. 4.7, we are using a non-overlapping implication operator, which means that at a posedge of clk if cStart is high, then one clock later sr1 should be executed.

Let us revisit 'sampling' of variables. The expression variables cStart, req and gnt are all sampled in the *preponed region* (see Sect. 4.3) of posedge clk. In other words, if (e.g.) cStart = 1 and posedge clk changed at the same time, the sampled value of cStart in the 'preponed region' will be equal to 'zero' and not 'one'. We will soon discuss what 'preponed region' really means in a simulation time tick and how it affects the evaluation of an assertion, especially when the sampling edge and the sampled variable change at the same time.

Note again that 'sequence sr1' does not have a clock in its expression. The clock for 'sequence sr1' is inherited from the 'property pr1'. This is explained next using Fig. 4.8.

As explained in Fig. 4.8, the 'clk' as an edge can be specified either directly in the assert statement or in the property or in the sequence. Regardless of where it is declared, it will be inherited by the entire assertion (i.e. the assert, property and sequence blocks).

Suggestion: As noted in Fig. 4.8, my recommendation is to specify the 'clk' in a property. Reason being you can keep sequences void of sampling edge (i.e. 'clk') and thus make them reusable. The sampling edge can change in the property but

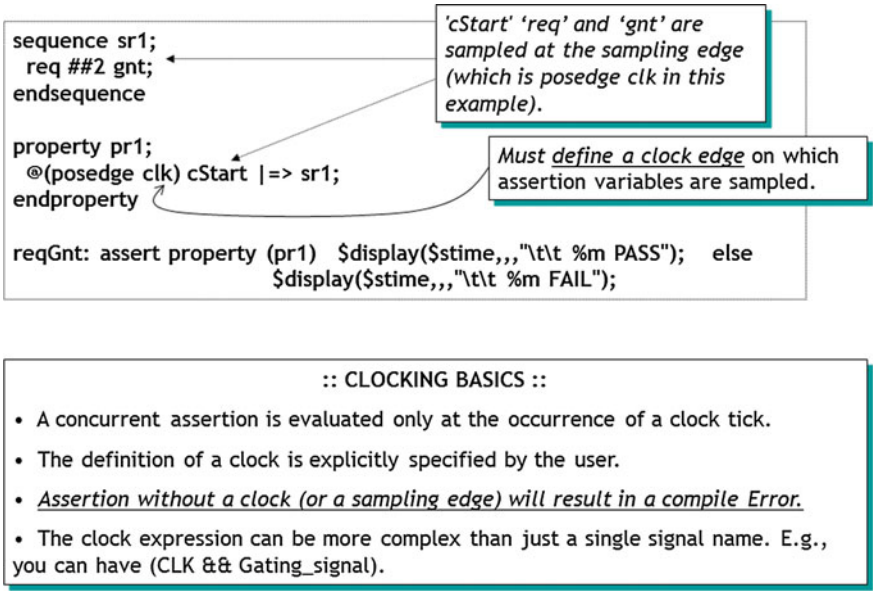


Fig. 4.7 Clocking basics

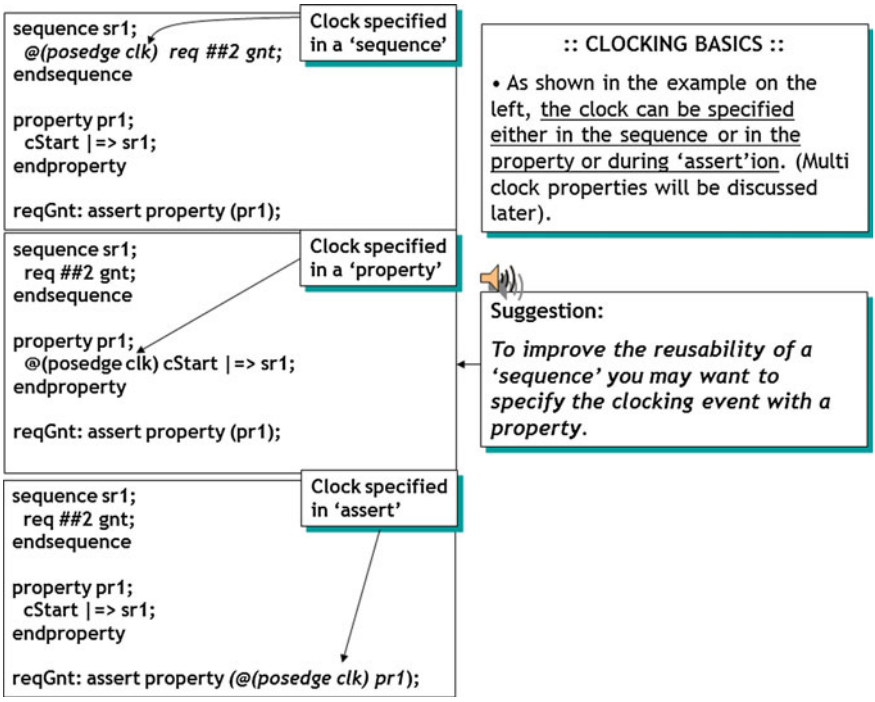


Fig. 4.8 Clocking basics—clock in 'assert', 'property' and 'sequence'

sequence (or cascaded sequences) remain untouched and can change their logic without worrying about the sampling edge. Note that it is also more readable when the sampling edge 'clk' is declared just before the antecedent in a property. "At posedge of clk, if cStart is high, trigger sr1".

Note that (as described in Sect. 14.2), a clock can be contextually inferred from a procedural block for a concurrent assertion. For example,

```
always @(posedge clk) assert property (not (FRAME_ ##2 IRDY));
```

Here the concurrent assertion is fired from a procedural block. The clock is @(posedge clk), inferred from the 'always @(posedge clk)' statement.

There is also an entire Sect. 8.1 devoted to multi-clock properties. It is too early to delve into its detail.

4.3 Sampling Edge (Clock Edge) Value: How Are Assertions Evaluated in a Simulation Time Tick?

How does the so-called sampling edge sample the variables in a property or a sequence is one of the most important concept you need to understand when designing assertions. As shown in Fig. 4.9 the important thing to note is that the variables used in assertions (property/sequence/expression) are sampled in the *preponed* region. What does that mean? It means (for example) if a sampled

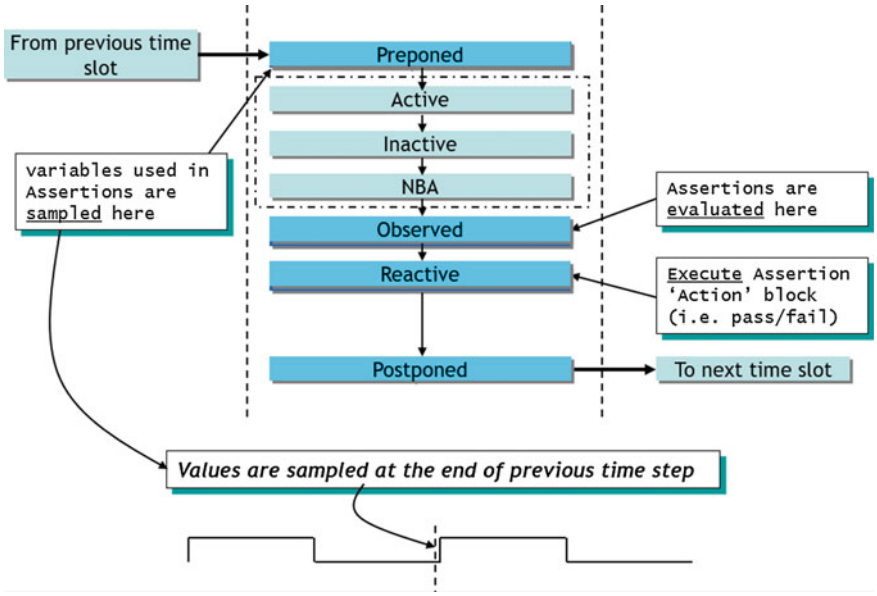


Fig. 4.9 Assertions variable sampling and evaluation/execution in a simulation time tick

variable changes the same time as the sampling edge (e.g. clk) that the value of the variable will be the value it held—*before*—the clock edge.

```
@ (posedge clk) a |=> !a;
```

In the above sequence, let us say that variable 'a' changes to '1' the same time that the sampling edge clock goes posedge clk (and assume 'a' was '0' before it went to a '1'). Will there be a match of the antecedent 'a'? No! Since 'a' went from '0' to '1' the same time that clock went posedge clk, the sampled value of 'a' at posedge clk will be '0' (in the preponed region) and not '1'. This will not cause the property to trigger because the antecedent is not evaluated to be true. This will confuse you during debug. You would expect '1' to be sampled and the property triggered thereof. However, you will get just the opposite result.

This is a very important point to understand because in a simulation waveform (or for that matter with Verilog $monitor or $strobe) you will see a '1' on 'a' with posedge clk and would not understand why the property did not fire or why it failed (or passed for that matter). Always remember that at the sampling edge, the 'previous' value (i.e. a delta before the sampling edge in the preponed region) of the sampled variable is used. To reiterate, preponed region is a precursor to the time slot, where only sampling of the data values take place. No value changes or events occur in this region. Effectively, sampled values of signals do not change through the time slot.

Note that a named event can also act as a clock. For example:

```
module eventtrig;

event e;

        always @(posedge clk) -> e;

a1: assert property (@e a |=> b);

endmodule
```

Following is to establish what is *not* sampled. As you read the book further, you will better understand what this means. For now, keep it in your back pocket.

Following are NOT sampled in the pre-poned region of the time tick.

- Assertion *local* variables
- Assertion action blocks (pass and fail)
- Clocking event expression (e.g. @(posedge clk); here clk is not sampled—but the *current* value of clock is used)
- Disable condition (variables of conditional expression) of 'disable iff' (more on this in Sect. 4.6—this concept is important to understand)
- Actual arguments passed to 'ref' or 'const ref' arguments of subroutines attached to sequences.
- The sampled value of a const cast expression is defined as the current value of its argument. For example, if 'a' is a variable, then the sampled value of const'(a) is the current value of a. When a past or a future value of a const cast

expression is referenced by a sampled value function, the current value of this expression is taken instead.

- Sampled value of the .triggered event property and the sequence methods .triggered and .matched (see Chap. 11) is defined as the current value returned by the event property or sequence method. For example, if 'a' is a static module variable, 's' is a sequence, and 'f' is a function, the sampled value of f (a, s.triggered) is the result of the application of 'f' to the sampled values of 'a' and s.triggered, i.e., to the value of 'a' taken from the Preponed region and to the *current* value of s.triggered.

Here is a complete example including the testbench and comments that explain how sampling of variables in the preponed region affect assertion results.

```
module assert1;
reg A, B, C, D, clk;
        property ab;
          @ (posedge clk) !A |-> B;
        endproperty
aba: assert property (ab) else $display ($stime,,, "ab FAIL");
abc: cover property (ab) $display($stime,,, "ab PASS");
initial begin
     clk=0; A=0; B=0;  //Note: A and B are equal to '0' at time 0.
     forever #10 clk=! clk;
end
initial begin
`ifdef PASS
 /* Following sequence of events will cause property 'ab' to PASS because even though A=0 and B=1 change simultaneously they had settled down because of #1 before posedge clk. Hence when @ (posedge clk) samples A, B; A=0 and B=1 are sampled. The property antecedent '!A' is evaluated to be true and at that same time (overlapping operator) B==1. Hence the property passes */
 A=0;
 B=1;
 #1;
```

```
  @ (posedge clk)
`else
  /* Following sequence of events will cause property 'ab' to FAIL. Here's the story. A=0 and B=1 change at the
same time as posedge clk. This causes the sampled value of B to be equal to '0' and not '1' because the sampling
edge (posedge clk) samples the variable values in the preponed region and B was equal to '0' in the preponed
region. Note that A was equal to '0' in the preponed region because of its initialization in the 'initial' block above.
So, now you have both 'A' and 'B' == 0. Since A is 0, !A is true and the property evaluation takes place. Property
expects B==1 the same time (overlapping operator) that !A is true. However, 'B's sampled value is '0' and the
property fails. */
  @ (posedge clk)
     A=0;
     B=1;
`endif
  @ (negedge clk)
  $finish(2);
end
endmodule
```

Here are some detailed rules on how variables and expressions are sampled (at a clock edge). Keep this in your back pocket. At this time, you may not understand them well. But once you better familiarize yourself with how sampled values of variables (local, automatic, normal) take place, this will be good reference.

The definition of a sampled value of an expression is based on the definition of a sampled value of a variable. The general rule for variable sampling is as follows:

- The sampled value of a variable in a time slot corresponding to time greater than 0 is the value of this variable in the *Preponed* region of this time slot.
- The sampled value of a variable in a time slot corresponding to time 0 is its default sampled value.

This rule has the following exceptions:

- Sampled values of automatic variables, local variables and active free checker variables (see Sect. 16.19 for checker variables) are their current values. However,
- When a past or a future value of an active free checker variable is referenced by a *sampled value function* (e.g. $past), this value is sampled in the *Postponed* region of the corresponding past or future clock tick;
- When a past or a future value of an automatic variable (automatic variable is a SystemVerilog construct and not discussed in this book) is referenced by a sampled value function, the current value of the automatic variable is taken instead.

The sampled value of an expression is defined as follows:

- The sampled value of an expression consisting of a single variable is the sampled value of this variable.
- The sampled value of the triggered event property and the sequence methods .triggered and .matched (see Sect. 11.2) is defined as the current value returned by the event property or sequence method.
- The sampled value of any other expression is defined recursively using the values of its arguments. For example, the sampled value of an expression e1 & e2, where e1 and e2 are expressions, is the bitwise AND of the *sampled* values of e1 and e2. In particular, if an expression contains a function call, to evaluate the sampled value of this expression, the function is called on the sampled values of its arguments at the time of the expression evaluation.

4.3.1 Default Clocking Block

For a long chain of properties and sequences in the file, you can also use the default clocking block as explained in Fig. 4.10. The figure explains the different ways in which clocking block can be declared and the scope in which it is effective. The top block of the figure shows declaration of 'default clocking cb1' which is then inherited by the properties 'checkReqGnt' and 'checkBusGrant' that follow. This default clocking block will be in effect until another default clocking block is defined. The bottom part of the figure is interesting. Here the properties are directly embedded in the default clocking block. I don't recommend doing that though. The clocking block should only contain clock specifications, which will keep it modular and reusable. Use your judgment call wisely on such issues.

Figure 4.11 declares two clocking blocks, namely 'cb1' and 'cb2' in a standalone Verilog module called 'design_clocks'. This is a great way to organize your clocking strategy in one module. Once defined, you can use any of the clocking block that is required simply by referring to it by its hierarchical instance name as shown in the figure.

Here's some food for thought. I have outlined a couple of pros and cons of using a default-clocking block. It is mostly advantageous but there are some caveats.

Pros: The argument towards default block is reusability. You may change the clocking relation in the default block and it will be applicable to all the following blocks. You do not have to individually change clocking scheme in each property. This is indeed a true advantage and if you plan to change the clocking scheme in the default block without affecting the properties that follow, do use the default block by all means.

Cons: Readability/debuggability: When you see a property without any sampling edge, you have to scroll back to 'someplace' above the property to see what sampling edge is being used. You have to find the very preceding clocking block

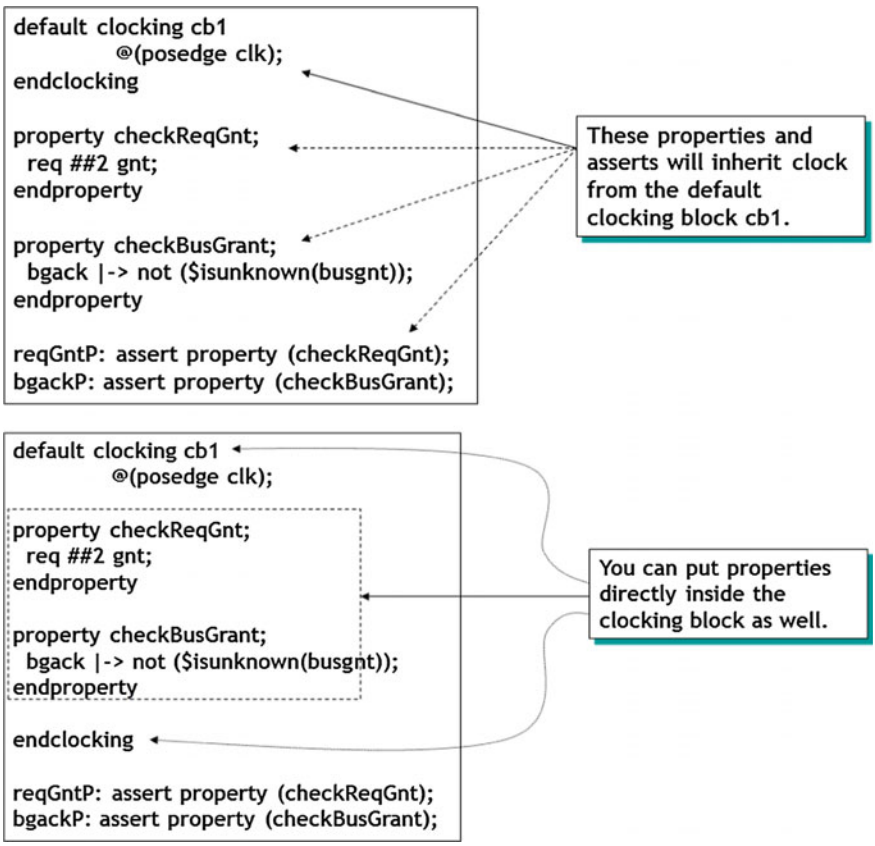


Fig. 4.10 Default Clocking block

and can't just go to the top of the file. I like properties that are mostly self-contained with the sampling edge. Sure, it's a bit more typing but a lot more readable.

Here's an example of how you can use the 'default' clocking block but also override it with explicit (non-default) clocking.

```
module default_explicit_clocking;

default clocking negedgeClock @(negedge clk1); endclocking
clocking posedgeClock @(posedge clk2); endclocking
d2: assert property (x |=> y); //will inherit default clock–negedgeClock
d3: assert property (z [=2] |-> a); //will inherit default clock–negedgeClock
nd1: assert property (@posedgeClock b |=> c); //will use non-default clocking posedgeClock

endmodule
```

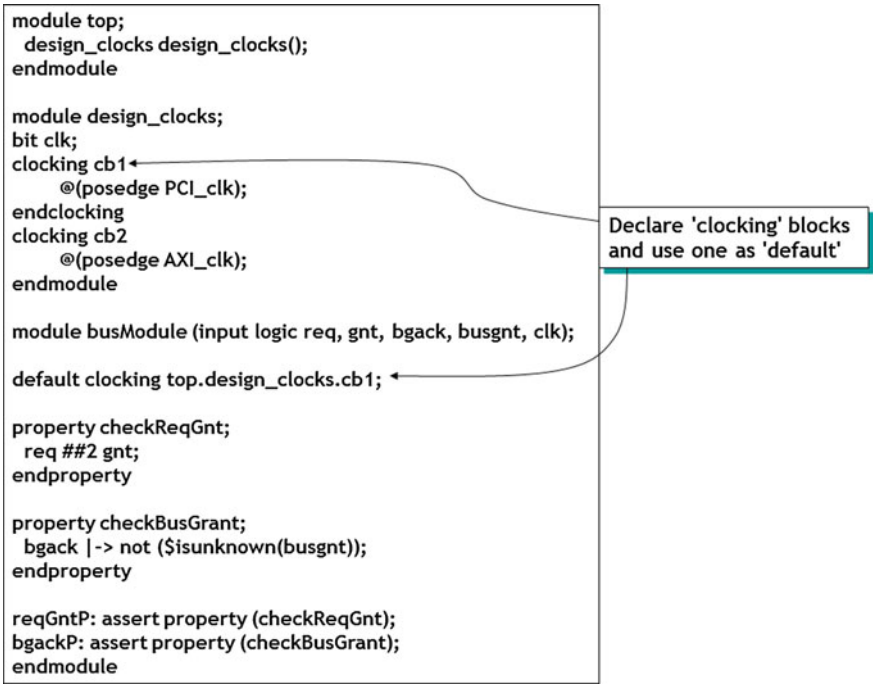


Fig. 4.11 'clocking' and 'default clocking'

Obviously, you don't *have* to declare the 'clocking' block as shown above. You can simply use @ (posedge clk2) directly in the property assertion, as shown below.

module default_explicit_clocking;

default clocking negedgeClock @(negedge clk1); **endclocking**
d2: assert property (x |=> y); //will inherit default clock–negedgeClock
d3: assert property (z [=2] |-> a); //will inherit default clock–negedgeClock
nd1: assert property (@(posedge clk2) b |=> c); //explicit declaration of clock–clk2

endmodule

Or you can model the same clocking structure as follows, using a property with its own explicit clock.
module default_explicit_clocking;

default clocking negedgeClock @(negedge clk1); **endclocking**

```
property nClk; @(posedge clk2) b |=> c; endproperty
d2: assert property (x |=> y); //will inherit default clock–negedgeClock
d3: assert property (z[= 2] |-> a); //will inherit default clock–negedgeClock
nd1: assert property (nClk); //explicit declaration of clock–clk2
endmodule
```

Note the following rules that apply to a clocking block:

1. Multiclocked sequences and properties (Chap. 8) are not allowed within the clocking block.
2. If a named sequence or property that is declared outside the clocking block is instantiated within the clocking block, the instance is singly clocked and its clocking event is identical to that of the clocking block.
3. An explicitly specified leading clocking event in a concurrent assertion statement supersedes a default clocking event.

Note the following example that shows the application of above rules and points to Legal and Illegal cases (courtesy LRM)

```
property q1;
        $rose(a) |-> ##[1:5] b;
endproperty

property q2;
        @(posedge clk) q1;
endproperty


default clocking posedge_clk @(posedge clk);

        property q3;
                $fell(c) |=> q1;  // legal: q1 has no clocking event
        endproperty

        property q4;
                $fell(c) |=> q2;   // legal: q2 has clocking event identical to that of the clocking block
        endproperty

        sequence s1;
                @(posedge clk) b[*3]; // illegal: explicit clocking event in clocking block
        endsequence


endclocking
```

Following may not be too intuitive at this stage since we haven't seen concurrent assertions in practice. But these are important legal and illegal conditions which will help avoid unnecessary debugging. Examples are for properties that does not have a default clocking block.

```
module examples_NO_default (input logic a, b, c, clk);
property q1;
        $rose(a) |-> ##[1:5] b;
endproperty

property q5;
        @(negedge clk) b[*3] |=> !b;
endproperty

property q6;
        q1 and q5;
endproperty

a5: assert property (q6); // illegal: no leading clocking event
a6: assert property ($fell(c) |=> q6); // illegal: no leading clocking event

sequence s2;
        $rose(a) ##[1:5] b;
endsequence

c1: cover property (s2); // illegal: no leading clocking event
c2: cover property (@(negedge clk) s2); // legal: explicit leading clocking event, @(negedge clk)

sequence s3;
        @(negedge clk) s2;
endsequence

c3: cover property (s3); // legal: leading clocking event, @(negedge clk), determined from declaration of s3

endmodule
```

4.3.2 Gated Clk

Figure 4.12 shows an interesting modeling application of using a gated clk as the sampling edge for a property. Note that assign is out of the scope of assertion. But it's assigned value 'clkstart' can indeed be used in the property. In general, any variable declared in a given scope in which the property/sequence is defined is available to the assertion. If the assertions are declared out of the module but bound to the module using 'bind' method, the same rule applies. More on 'bind' statement coming up soon.

In this example, the sampling edge will be the posedge of (clk && cGate). In the preponed region of this sampling edge, the variables in property and sequence will be sampled.

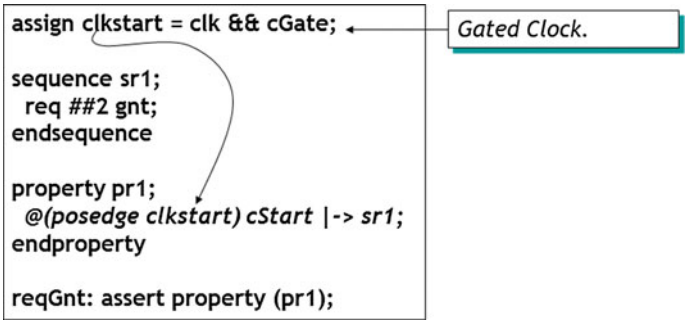


Fig. 4.12 Gated clock

4.4 Concurrent Assertions Are Multi-threaded

This is about the most important concept you need to grasp when it comes to concurrent assertions. We all know SystemVerilog is a concurrent language but is it multi-threaded (except when automatic variables are used)? SVA by default is concurrent and multi-threaded.

In Fig. 4.13, we have declared the same assertion that you have seen before, namely at posedge clk, if cStart is sampled high that sr1 will be triggered at the same posedge clk which will then look for 'req' to be high at that clock and 'gnt' to be sampled high two clocks later.

Now, let us say that cStart is *sampled* high (S1) at a posedge of clk and that 'req' is also sampled high at that same edge. After this posedge clk, the sequence will wait for 2 clocks to see if 'gnt' is high.

But before the two clocks are over, clk cStart goes low and then goes high (S2) exactly two clocks after it was sampled high. This is also the same edge when our first trigger of assertion will look for gnt to be high (S1). So, what will the assertion do? Will it re-fire itself because it meets its antecedent condition (S2) and ignore 'gnt' that it's been waiting for from the first trigger (S1)? No, it will not ignore 'gnt'. It will sample 'gnt' to be high (S1) and consider the first trigger (cStart (S1)) to PASS. So, what happens to the second trigger (cStart (S2))? It will start *another* thread. It will again wait for 2 clocks to check for 'gnt'. So far so good. We see one instance of SVA being threaded.

But life just got more interesting.

After S2, the very next clock cStart is sampled high again (S3). And 'req' is high as well (req(S3)). Now what will the assertion do? Well, S3 will thread itself with S2. In other words, there are now two distinct threads of the same assertions waiting to sample 'gnt' two clocks after their trigger. The figure perfectly (!) lines up 'gnt' to be high two clocks after both S2 as well as after S3 and all 3 triggers of the same assertions will PASS.

This has many implications in terms of design of assertions and performance thereof. We will discuss this further when we discuss edge triggered antecedent.

```
sequence sr1;
  req ##2 gnt;
endsequence

property pr1;
  @(posedge clk) cStart |-> sr1;
endproperty

reqGnt: assert property (pr1)  $display($stime,,,"\t\t %m PASS");
else $display($stime,,,"\t\t %m FAIL");
```

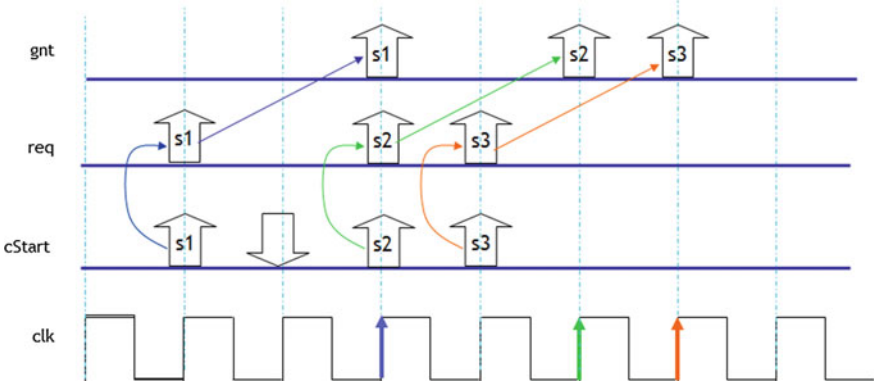


Fig. 4.13 Multi-threaded concurrent assertions

In other words, the way the property in our example is coded, it will drag your simulation performance because every time the property sees cStart to be high at posedge of clk, it will start a new thread. But if you want to evaluate the property only at the first rise of cStart and then ignore it if it stays high (unless it goes low and goes high again) then you have to use edge sensitive antecedent. More on this in Chap. 5. In addition, the concept of multi-threaded language gets much more interesting as you will see in Sect. 6.2.1.

Here's an explanation of further nuances of concurrent assertions.

It is important that the defined clock behavior be glitch free. Otherwise, wrong values can be sampled.

If a variable that appears in the expression for clock also appears in an expression with an assertion, the values of the two usages of the variable can be different. *The current value of the variable is used in the clock expression, while the sampled value of the variable (in preponed region) is used within the assertion*. This concept is especially important to understand if your 'sampling edge' is not a synchronous clock but an 'asynchronous edge'. We will cover this concept via an explicit example in Chap. 15.

4.5 Formal Arguments

One of the key features of assertions is that they can be parameterized. In other words, assertions can be designed with formal arguments to keep them generic enough for use with different actual arguments.

Figure 4.14 is self-explanatory. Notice that the formal arguments can be specified in a sequence and in a property.

The application shows the advantage of formal arguments in reusability. Property 'noChangeSig' has 3 formal arguments, namely pclk, refSig and Sig. The property checks to see that if refSig is sampled low at posedge pclk, that the Sig is 'stable'. Once such a generic property is written you can invoke it with different clk, different refSig and Sig. CheckRd is a property that uses sysClk and OE_ and RdData to check for 'stable' condition while CheckWr uses WE_ and WrData to check for WrData to be 'stable'.

In any project, there are generic properties that can be reused multiple times by passing different actual arguments. This is reusable not only in the same project but also among projects.

Companies have created libraries of such pool of properties that projects look up and reuse according to their needs.

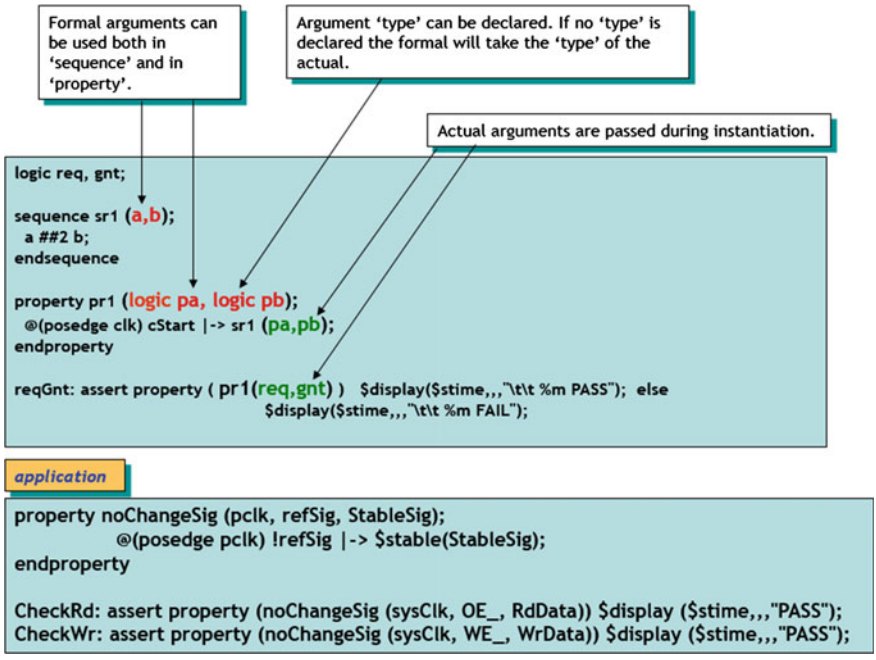


Fig. 4.14 Formal and actual arguments

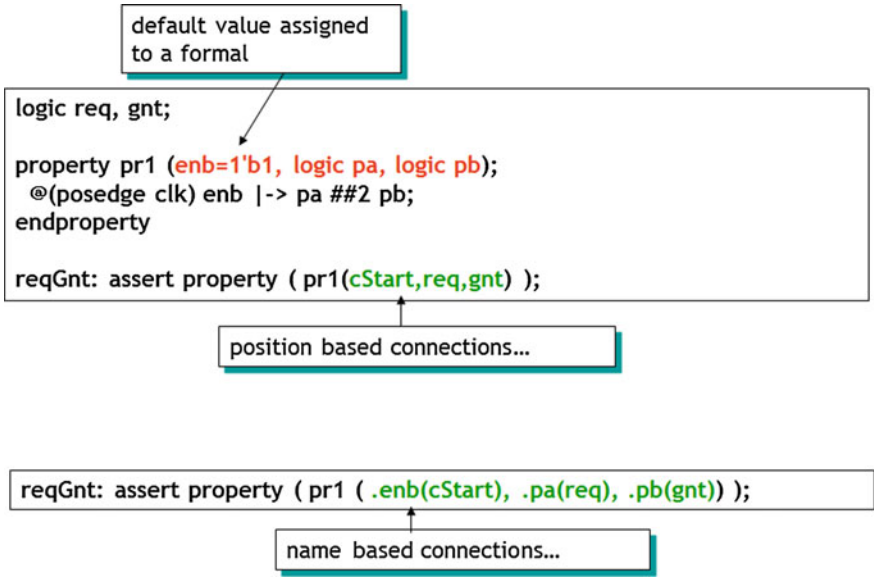


Fig. 4.15 Formal and Actual arguments—default value and name based connection

As shown in Fig. 4.15, properties can be both position based as well as name based. I highly recommend name based to make sure that actuals are connected to correct formals without ambiguity. This rule is the same as that we have been using for Verilog port connections.

Figure 4.16 describes the following points

1. Default values can be assigned to the formal arguments.
   a. If actual and formal both specify a 'default' value, the actual will overwrite the formal default value.
   b. You may leave passing an actual to a formal if the formal has a default value. Please refer to Fig. 4.16.

This is a very interesting feature and very useful at that for reusability. A formal can be used for event control as well. A sampling edge can be passed as an actual to a formal and the actual can be used as a sampling edge in the property. We are passing 'posedge clk' as an actual to the formal 'csig'. The property uses @ (csig) as it's sampling edge. '@ (csig)' will change to '@ (posedge clk)' when the property 'pr1' is called with 'posedge clk' as the actual argument. Please refer to Fig. 4.17 for clarity on this point. Such properties can indeed be part of a common pool of properties that individual projects can reuse with their own sampling edge specification.

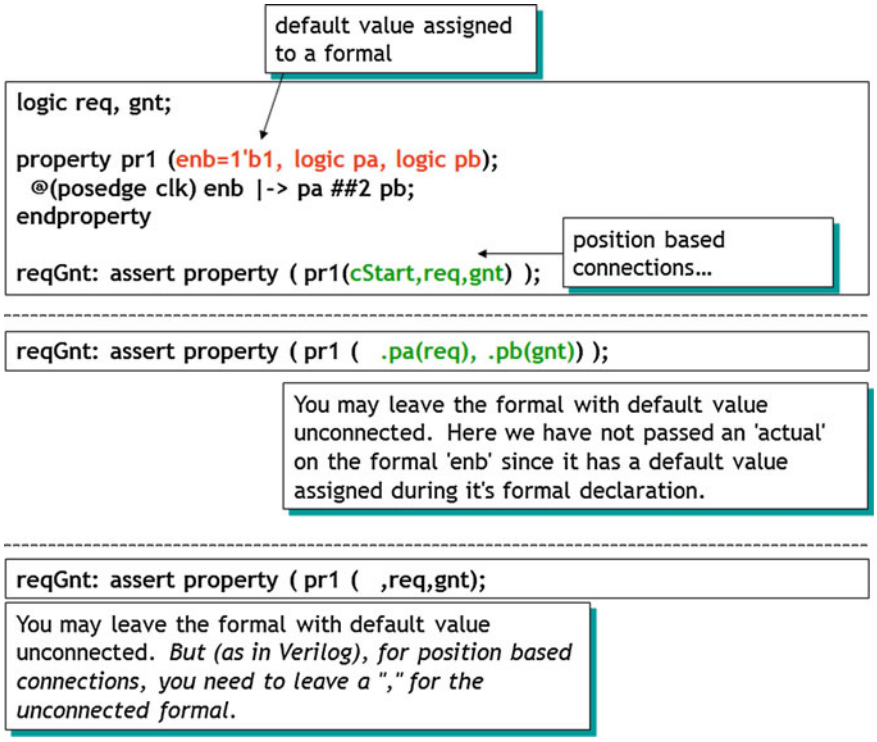


Fig. 4.16 Formal and actual arguments—default value and position based connection

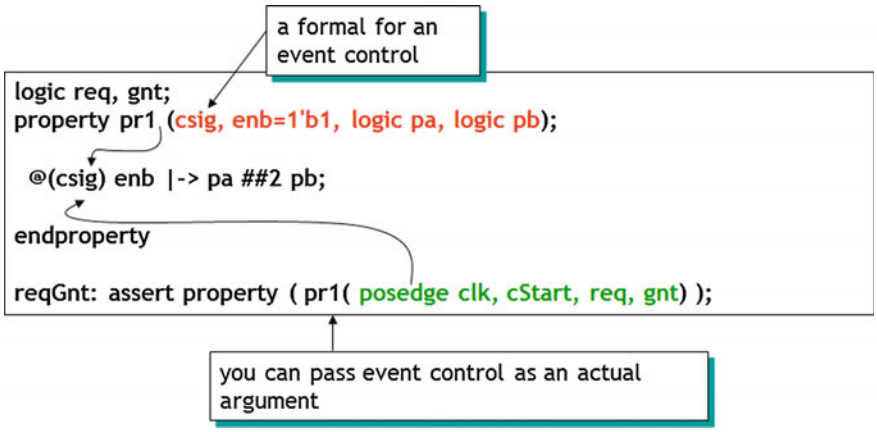
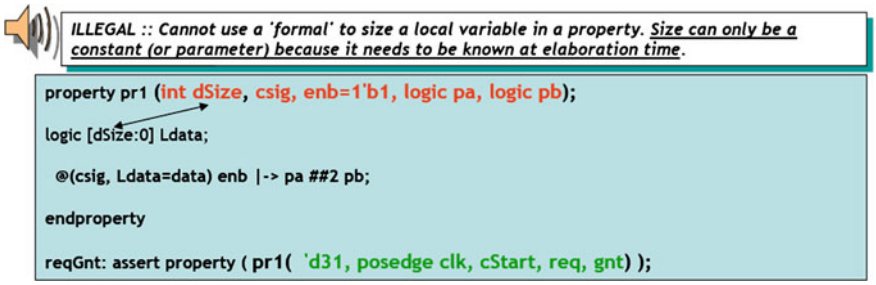


Fig. 4.17 Passing event control to a formal



Note also that you cannot pass a variable as an actual to size a local variable (see Chap. 9) in the property pr1. The size parameter needs to be a constant for sizing a local parameter.

Here are a some more rules governing binding between formal and actual.

A formal argument is said to be untyped if there is no type specified prior to its declaration in the port list. There is no default type for a formal argument.

The supported data types for sequence formal arguments are the types that are allowed for operands in assertion expressions and the keyword **untyped**.

Note that you can also use '$' as an actual argument. The terminal '$' may be an actual argument in an instance of a named sequence, either declared as a default actual argument or passed in the list of arguments of the instance. If '$' is an actual argument, then the corresponding formal argument must be untyped and each of its references will be an upper bound in a cycle_delay_const_range_expression.

4.6 Disable (Property) Operator—'Disable Iff'

Of course, you need a way to disable a property under conditions when the circuit is not stable (think Reset). That's exactly what 'disable iff' operator does. It allows you to explicitly disable the property under a given condition. Note that 'disable iff' reads as 'disable if and only if'. The example in Fig. 4.18 shows how you can disable an assertion during an active Reset. There is a good chance you will use this Reset based disable method in all your properties throughout the project.

Ok, so what happens if a property has started executing and the 'disable iff' condition occurs in the middle of its execution?

The property in Fig. 4.18 checks to see that sdack_ falls (i.e. contained) within soe_ (don't worry, we'll see how such properties work in later chapters—see Sect. 6.10). It also has the 'disable iff (! reset)' condition. Disable this property if reset is asserted (active low).

Let us examine the simulations logs.

In the LHS simulation log, reset is never asserted and the assertion completes (and passes in this case).

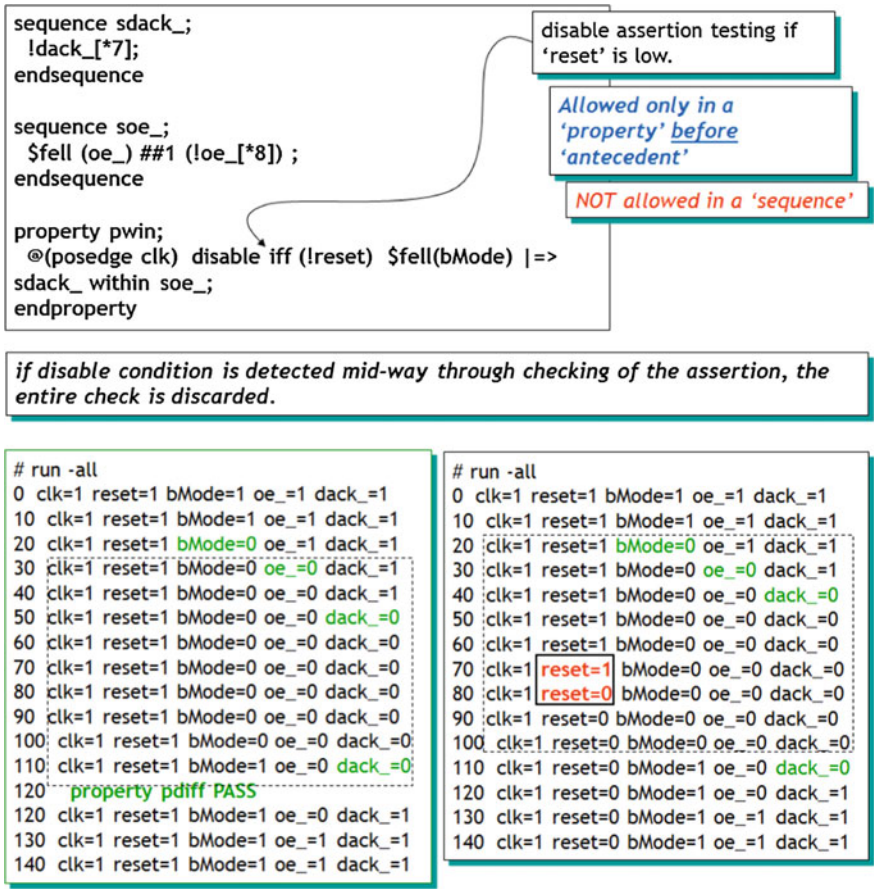


Fig. 4.18 'disable iff' operator

In the RHS simulation block, reset is asserted in the middle of check "sdack_ within soe" and the entire assertion is discarded. You will not see pass/fail for this assertion because it has been discarded. Entire assertion is disabled if the disable iff condition occurs in the middle of an executing assertion. Some folks mistake such discard as a failure, which is incorrect.

Once an assertion has been disabled with 'disable iff' construct, it will re-start only after the 'disable iff' condition is not true anymore.

Note below the rules governing 'disable iff'

1. 'disable iff' can be used only in a property—not in a sequence
2. 'disable iff' can only be used before the declaration of the antecedent condition.
3. 'disable iff' expression is *not* sampled at a clock edge as with other expressions in the concurrent assertion. The expressions in a disable condition are evaluated using the *current* values of variables (not sampled). 'disable iff' expression can

be thought of as asynchronous, it can trigger in between clock events or at clock event. This is an important point because we will be discussing a lot about sampled values and this here is an exception. In our example, we have disable iff (!reset). Here the 'reset' signal is not sampled. The disable iff condition will trigger as soon as 'reset' goes low.

4. Nesting of 'disable iff' clauses, explicitly or through property instantiations, is not allowed.
5. We haven't discussed .triggered or .matched methods but here's the rule for your reference later. 'disable iff' may contain the sequence Boolean method .triggered. But it cannot not contain any reference to local variables or to the sequence method .matched.

As we will discuss in Sect. 7.4, there are system tasks that provide global control over execution of assertions.

4.7 Severity Levels (for Both Concurrent and Immediate Assertions)

Assertions also allow error reporting with different severity levels. $fatal, $error (default), $warning and $info. Figure 4.19 explains meaning of each.

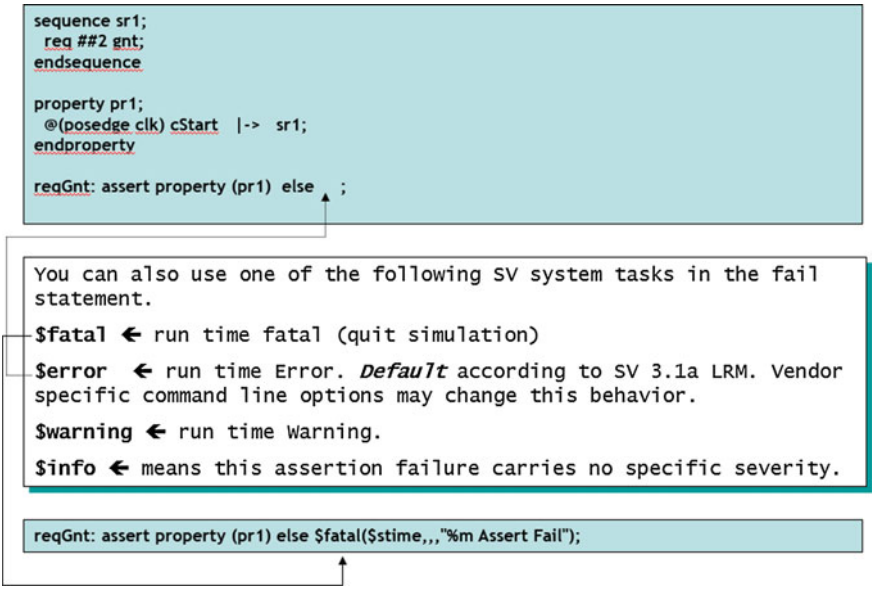


Fig. 4.19 Severity levels for concurrent and immediate assertions

$error is default, meaning if no failure clause is specified in the assert statement, $error will kick in and provide a simulator generated error message. If you have specified a label (and you should have) to the assertion, that will be (most likely) displayed in the $error message. I say most likely because the SystemVerilog LRM does not specify exact format of $error. It is simulator vendor specific. $warning and $info are self-explanatory as described in Fig. 4.19.

4.8 Binding Properties

'bind' allows us to keep design logic separate from the assertion logic. Design managers do not like to see anything in RTL that is not going to be synthesized. 'bind' helps in that direction.

There are three modules in Fig. 4.20. The 'designModule' contains the design. The 'propertyModule' contains the assertions/properties that operate on the logic in 'designModule'. And the 'test_bindProperty' module binds the propertyModule to the designModule. By doing so, we have kept the properties of the 'propertyModule' separate from the 'designModule'. That is the idea behind 'bind'. You do not have to place properties in the same module as the design module. As mentioned before, you should keep your design void of all constructs that are non-synthesizable. In addition, keeping assertions and design in separate modules allow both the design and the DV engineers work in parallel without restrictions of a database management system where a file cannot be modified by two engineers at the same time.

In order for 'bind' to work, you have to declare either the instance name or the module name of the designModule in the 'bind' statement. You need the design module/instance name, property module name and the 'bind' instance name for 'bind' to work. In our case the design module name is designModule, its instance name is 'dM' and the property module name is propertyModule.

The (uncommented) 'bind' statement uses the module instance 'dM' and binds it to the property module 'propertyModule' and gives this 'bind' an instance name 'dpM'. It connects the ports of propertyModule with those of the designModule. With this the 'property rc1' in propertyModule will act on designModule ports as connected.

The commented 'bind' statement uses the module name 'designModule' to bind to the 'propertyModule' whereby all instances of the 'designModule' will be bound to the 'propertyModule'.

In essence, we have kept the properties/assertions of the design and the logic of the design separate. This is the recommended methodology. You could achieve the same results by putting properties in the same module as the design module but that is highly non-modular and intrusive methodology. In addition, as noted above, keeping them separate allows both the DV and the Design engineer to work in parallel.

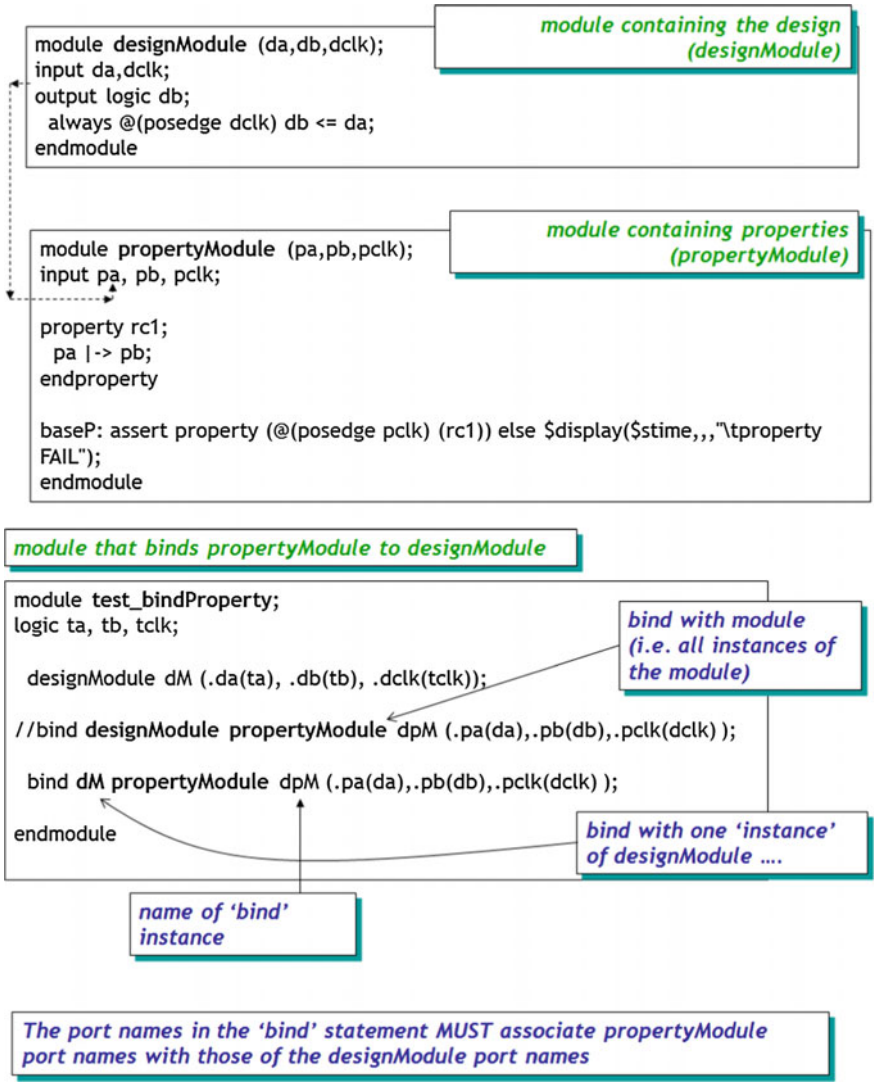


Fig. 4.20 Binding properties

4.8.1 Binding Properties (Scope Visibility)

But what if you want to bind the assertions of the propertyModule to internal signals of the designModule? That is quite doable.

As shown in Fig. 4.21, 'rda' and 'rdb' are signals internal to designModule. These are the signals that you want to use in your assertions in the 'propertyModule'. Hence, you need to make 'rda' and 'rdb' visible to the

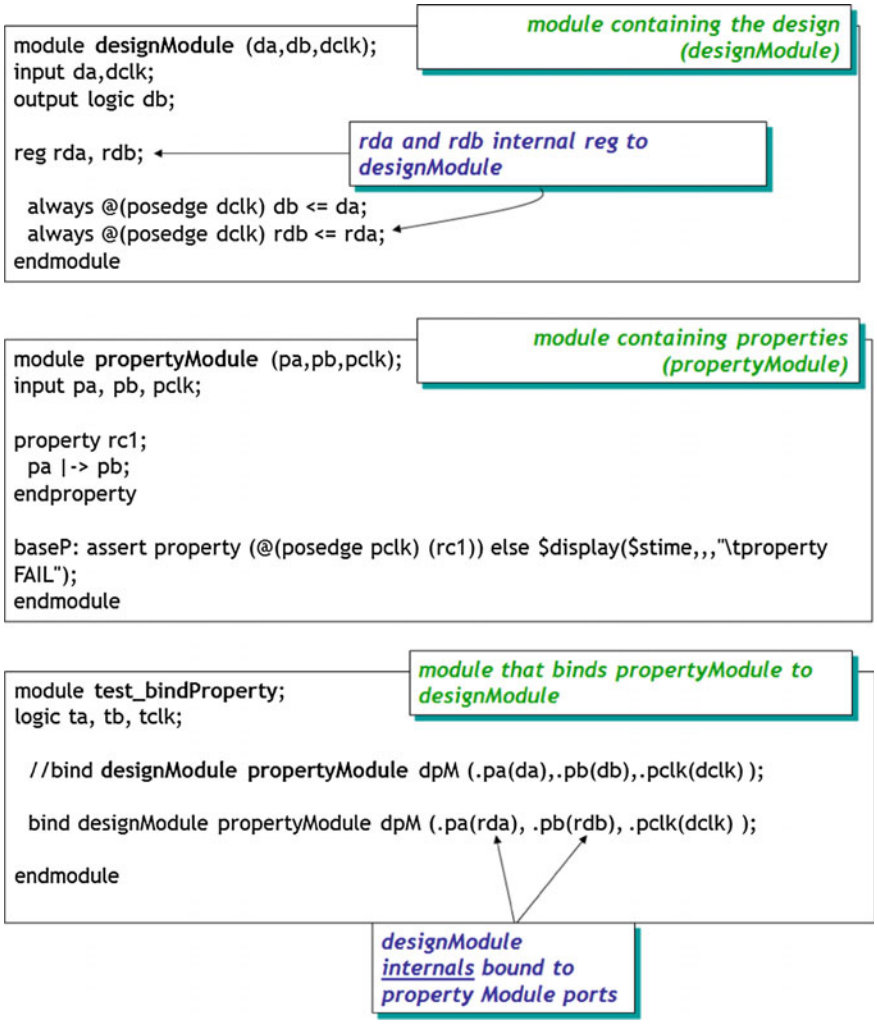


Fig. 4.21 Binding properties to design 'module' internal signals (scope visibility)

'propertyModule'. However, you do not want to bring 'designModule' internal variables to external ports in order to make them visible to the 'propertyModule'. You want to keep the 'designModule' completely untouched. To do that, you need to add input ports to the 'propertyModule' and bind those to the internal signals of the 'designModule' as shown in Fig. 4.21. Note that in our example we bind the propertyModule ports 'pa' and 'pb' to the designModule internal registers 'rda' and 'rdb'. In other words, you can directly refer to the internal signals of designModule during 'bind'. 'bind' has complete scope visibility into the bound module

'designModule'. Note that with this method you do not have to provide the entire hierarchical instance name when binding to 'propertyModule' input ports.

4.8.2 Assertion Adoption in Existing Design

Figure 4.22 shows that if you have an existing design, you can effectively use the 'bind' construct to write assertions outside of the design scope and bind them. This can be very useful, if you are bringing in legacy blocks in your new SoC and want to make sure that the legacy blocks work well in your new design. This figure is a methodology component. Upfront in your project, determine your 'bind' methodology. See that all the assertions are outside RTL and not a messy mix of some in RTL and some bound with external properties file.

Other advantage of keeping assertions in a separate file is that they can be independently verified without the need to have control of RTL files. A big advantage when you want to make sure that both the design and verification progress in parallel.

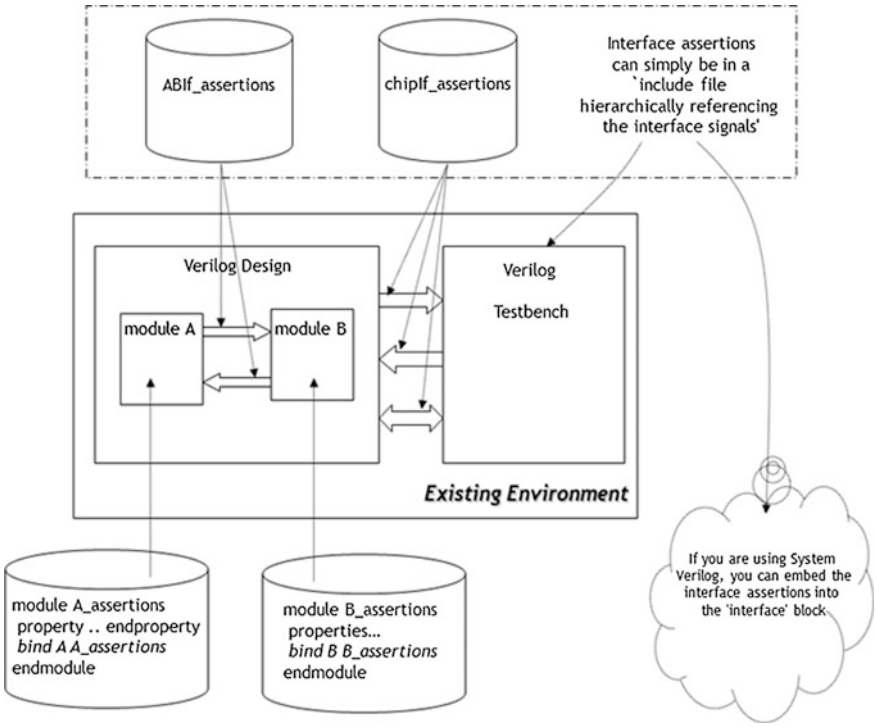


Fig. 4.22 Binding properties to an existing design. Assertions adoption in existing design

4.9 Difference Between 'Sequence' and 'Property'

Now that we have seen assertions using sequences and properties, it is good to recap and clearly understand the differences between the two.

- **'sequence'**
  - A sequence is a building block. Think of it as a macro or a subroutine where you can define a specific relationship for a given set of signals.
  - A sequence on its own does not trigger. It must be 'assert'ed.
  - A named sequence may be instantiated by referencing its name. The reference may be a hierarchical name.
  - A sequence does not allow implication operator. Simply allows temporal (or combinatorial) domain relationship between signals.
  - A sequence can have optional formal arguments
  - A clocking event can be used in a sequence
  - A sequence can be declared in a module, an interface, a program, a clocking block, a package, a compilation-unit scope, a checker and a generate block (***but—not—in a 'class'***).
- **'property'**
  - A property also does not trigger by itself until 'assert'ed (or 'cover'ed or 'assume'd).
  - Properties have implication operator that imply the relationship between an antecedent and a consequent.
  - Sequences can be used as building blocks of complex properties.
  - Clocking event can be applied in a property, in a sequence, or in both.
  - The formal and actual arguments can also be 'property expressions'—meaning you can pass a property as an actual to a formal of type 'property'.
  - Local variable arguments (see Chap. 9) can only be 'local input'.
  - A property can be declared in a module, an interface, a program, a clocking block, a package (***but—not—in a 'class***').

Chapter 5
Sampled Value Functions $rose, $fell, $stable, $past

Introduction: This chapter introduces and provides applications for Sampled Value Functions $rose, $fell, $past, $stable. Note that there are also quite a few new sampled value functions introduced in 2009/2012 LRM (e.g. $changed, $rose_gclk, $sampled, etc.). These are covered in Chap. 16 which is solely devoted to the entire 2009/2012 LRM feature set.

These sampled value functions allow for antecedent and/or the consequent to be edge triggered. $rose means that the least significant bit of the expression (in $rose (expression)) was sampled to be '0' (or 'x' or 'z') at the previous clk edge (previous meaning the immediately preceding clk from current clk) and that it is sampled '1' at this clk edge. For $fell, just the opposite need to take place. Preceding value should be sampled '1' (or 'x' or 'z') and current sampled value '0'. As explained with examples below, one needs to understand the difference between level sensitive sample versus edge sensitive sample.

But why do we call these functions 'sampled value'? That's because they are triggered only when the sampled value of the expression in the preponed region differ at two successive clock edges as described above. In other words, $rose(abc) does *not* mean 'posedge abc' as in Verilog. $rose(abc) does not evaluate to true as soon as abc goes from 0 to 1. $rose(abc) simply means that abc was sampled '1' at the current clock edge (in preponed region) and that it was *not* sampled a '1' at the immediately preceding clock edge.

Note also that both $rose and $fell work only on the Least Significant Bit of the expression. You will soon see what happens if you use a bus (vector) in these two sampled value functions.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_5

5.1 $rose—Edge Detection in Property/Sequence

property 'checkiack' in the top logic/timing diagram will (Fig. 5.2) PASS because both the 'inter' and 'iack' signals meet the required behavior of $rose (value at two successive clks are different and are '0' followed by '1'). However, the logic in the bottom diagram fails because while $rose(intr) meets the requirement of $rose, 'iack' does not. 'iack' does not change from '0' to '1' between the two clk edges (Fig. 5.1).

Important Note: To reiterate the points made above. $rose does *not* mean posedge and $fell does *not* mean negedge. In other words, the assertion won't consider $rose(intr) to be true as soon as a posedge on 'intr' is detected. The $rose ()/$fell() behavior is derived by 'sampling' the expression at two successive clk edges and see if the values are opposite and in compliance with $rose() or $fell().

In other words, the fundamental of concurrent assertions specifies that everything must be sampled at the sampling edge. Behavior is based on sampled value.

5.1.1 *Edge Detection Is Useful Because …*

This is a very imporant example. It explains the difference between use of level sensitive sampled values versus edge sensitive. Both are correct to use, except that

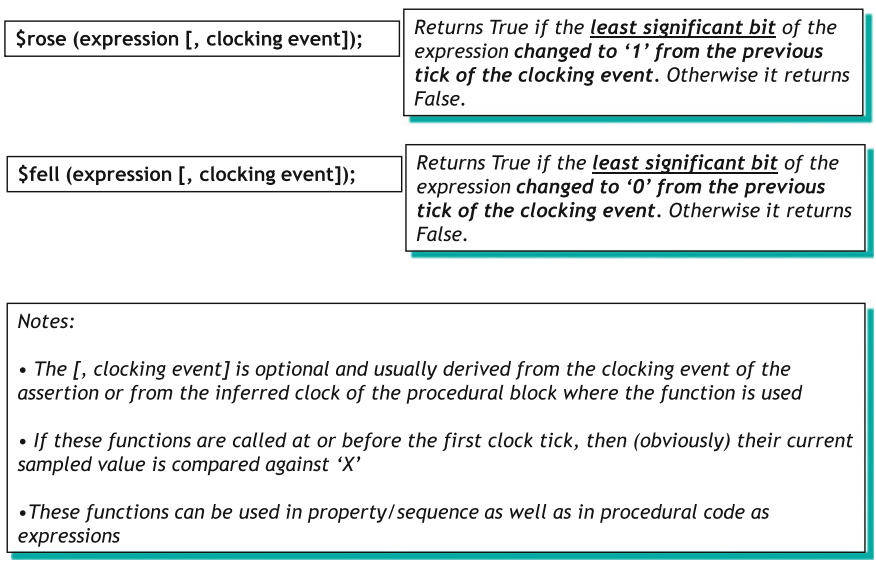


Fig. 5.1 Sampled value functions $rose, $fell—basics

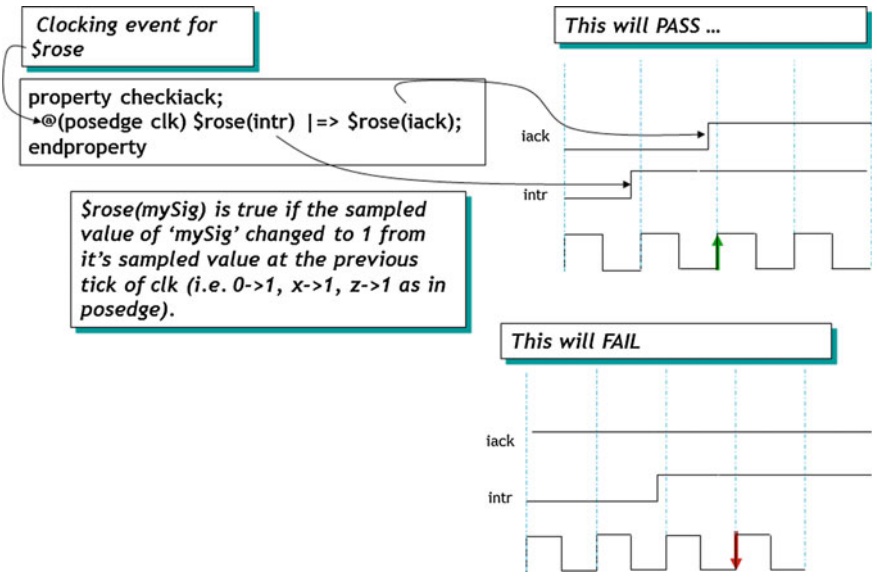


Fig. 5.2 $rose—basics

you need to know which to use when. As shown in Fig. 5.3, level sensitive evaluation is a superset of edge sensitive evaluation. But when you use level sensitive sample, you will degrade simulation performance, if in fact you meant for an edge sensitive evaluation.

In the above property, the following would be more appropriate if all you wanted to do was to check for iack to go from inactive '0' state to active state '1' once edge-sensitve intr is asserted. After that, the state of intr does not matter. Following is a better way to write the property if your intention was to check for riging edge of iack one clock after rising edge of intr.

```
property checkiack;
@ (posedge clk) $rose(intr) |=> $rose(iack);
endproperty
```

But what if you decide to do the following (as shown in Fig. 5.4)? Now you are courting real trouble. As Fig. 5.4 explains, since 'intr' is level sensitive sample, when it is sampled high it will look for the edge sensitive 'iack'. BUT since 'intr' is level sensitive and high the very next clock, it will start a new thread and check for $rose(iack) every clock. Since iack did not go from '0' to '1'on this second thread, the property fail. There is very good chance you did not want to see this failure.

In short, as simple as these functions look, you have to be careful in their usage and also keep in mind performance implications.

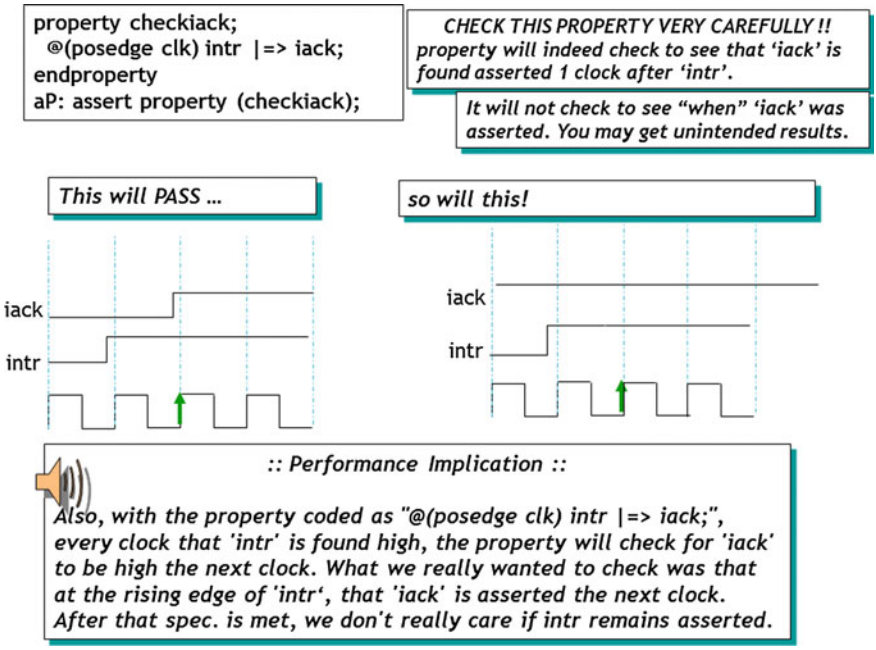


Fig. 5.3 usefulness of 'edge' detection and performance implication

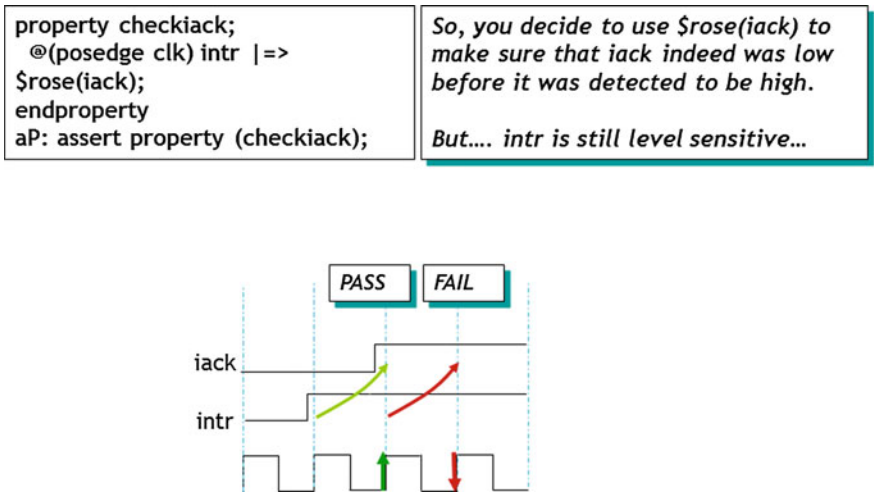


Fig. 5.4 ; $rose—finer points

5.1.2 $fell—Edge Detection in Property/Sequence

See Fig. 5.5.

5.1.3 $rose, $fell—in Procedural

$rose and $fell are very useful not only in concurrent assertions but also in sequential procedural blocks. They work exactly the same way as in concurrent assertions. Please see the examples in Fig. 5.6.

Since every assertion requires a clocking event (i.e. sampling edge), when you use a concurrent assertion in a procedural code without an explicit clocking event associated with them, the simulator looks for a clocking event in the code that precedes the concurrent assertion. We will discuss more on use of concurrent assertions in procedural code, under the Advanced Topics (Chap. 16).

In Fig. 5.6, $(posedge clk) is the preceding clocking event and acts as the sampling edge for $rose(iStreamDone) (example at the top of the figure).

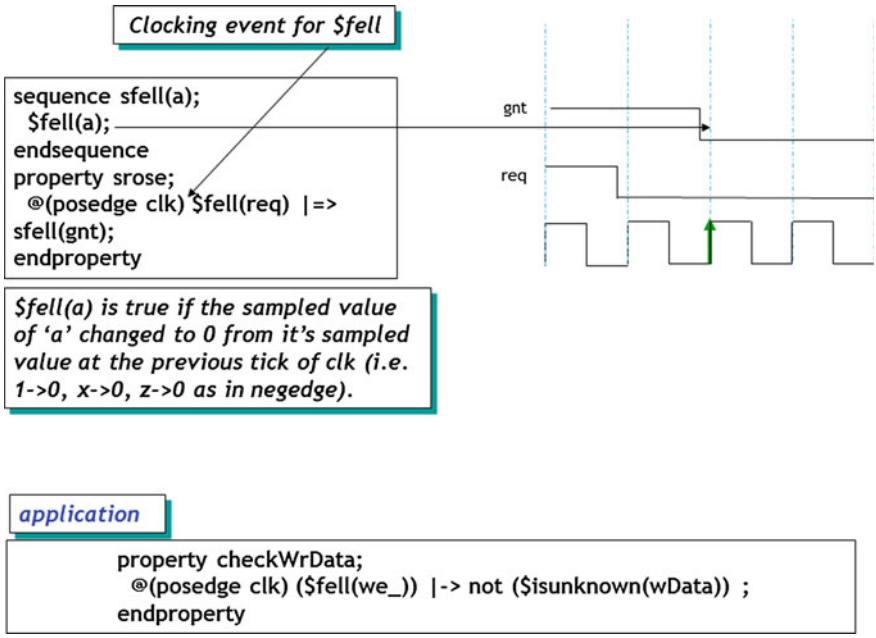


Fig. 5.5 $fell—basics

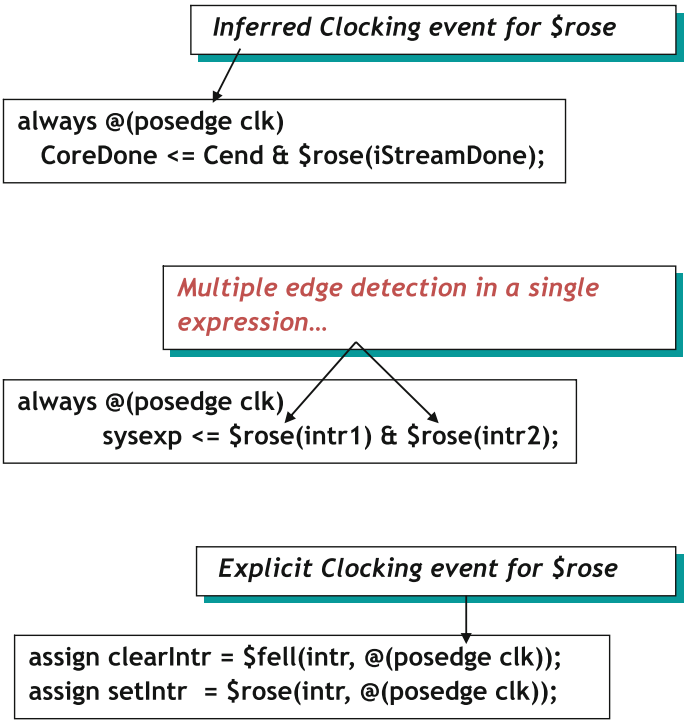


Fig. 5.6 $rose and $fell in procedural block and continuous assignment

Note the use of $rose and $fell in continuous assignment statement. Since continuous assign cannot have an edge behavior, you have to explicitly embed a clocking event with $rose() and/or $fell(). This is the same rule that applies to other sampled value functions.

5.2 $stable

$stable(), as the name implies, looks for its expression to be stable between two clock edges (i.e. two sampling edges). It evaluates the expression at current clock edge and compares it with the value sampled at the immediately preceding clock edge. If both values are same, the check passes (Fig. 5.7).

Note the use of $stable in continuous assignment statement. Since continuous assign cannot have an edge behavior, you have to explicitly embed a clocking event with $stable. This is the same rule that applies to other sampled value functions.

But what if you want to check if the signal has been stable for more than 1 clock? Read on… $past () will solve that problem.

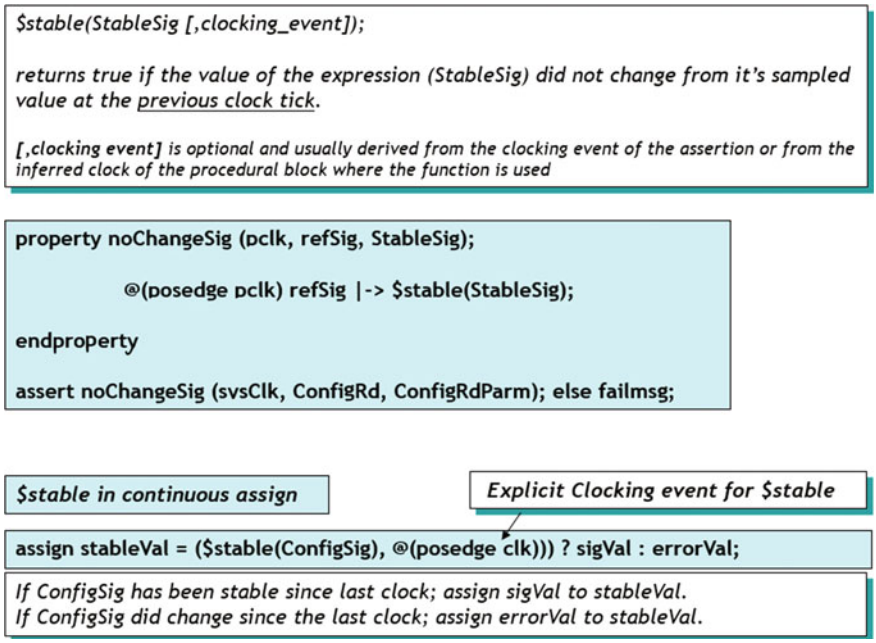


Fig. 5.7 $stable—basics

5.2.1 $stable in Procedural Block

As in $rose() and $fell(), $stable can also be embedded in the procedural code and works the same way as in a property or a sequence. As shown in the example above, $stable samples the value of expression (‘a’ and ‘b’ in this example) at the current and the immediately preceding edge (posedge clk in this example) to see if the value of the expression did not change. At time 15, ‘b’ has been stable, at time 25 ‘a’ has been stable and so on (Fig. 5.8).

5.3 $past

$past () is an interesting function. It allows you to go into past as many clocks as you wish to. You can check for an ‘expression1’ to have a certain value, number of clocks (strictly prior time ticks) in the past. Note that number of ticks is optional. If you do not specify it, the default will be to look for ‘expression1’ one clock in the past.

Another caveat that you need to be aware of is when you call $past in the initial time ticks of simulation and there are not enough clocks to go in the ‘past’. For example, you specify “a |-> $past (b)” and the antecedent ‘a’ is true at time ‘0’.

```
always @(posedge clk)
begin
  if ($stable(a)) $display ($stime,,,"\t 'a' stable from previous clock");
  if ($stable(b)) $display ($stime,,,"\t 'b' stable from previous clock");

  if ($stable(a) && $stable(b))
    $display ($stime,,,"\t 'a' AND 'b' Stable this clock");
end
```

```
# run -all
#        5  clk=1 a=1 b=0
#       15  clk=1 a=0 b=0
#       15    'b' stable from previous clock

#       25  clk=1 a=0 b=1
#       25    'a' stable from previous clock

#       35  clk=1 a=1 b=0
#       45  clk=1 a=1 b=1
#       45    'a' stable from previous clock

#       55  clk=1 a=1 b=1
#       55    'a' stable from previous clock
#       55    'b' stable from previous clock
#       55    'a' AND 'b' Stable this clock
```

Fig. 5.8 $stable in procedural block

There isn't a clock tick to go in the past. In that case, the assertion will use the 'initial' value of 'b'. What is the initial value of 'b'? It's not the one in the 'initial' block, it's the value with which the variable 'b' was declared (as in "logic b = 1'b1;"). In our case, if 'b' was not initialized in its declaration, the assertion will fail. If it was declared with an initial value of 1'b1, the assertion will pass.

You can also 'gate' this check with expression2. The example in Fig. 5.10 shows how $past works. We are using a gating expression in this example. It is not a requirement as noted in the Fig. 5.9, but it will most likely be required in your application. If expression2 is not specified, no clock gating is assumed.

If you understand the use of $past with a gating expression, then its use without one will be straightforward to understand.

Figure 5.10 asserts the following property

```
assert property (@ posedge clk) done |-> lV (mySig,2, enb, lastVal) $display(….);
```

And the property lV models the following

```
property lV(Sig, numClocks, enb, lastV);
(lastV == $past (Sig, numClocks, enb));
endproperty
```

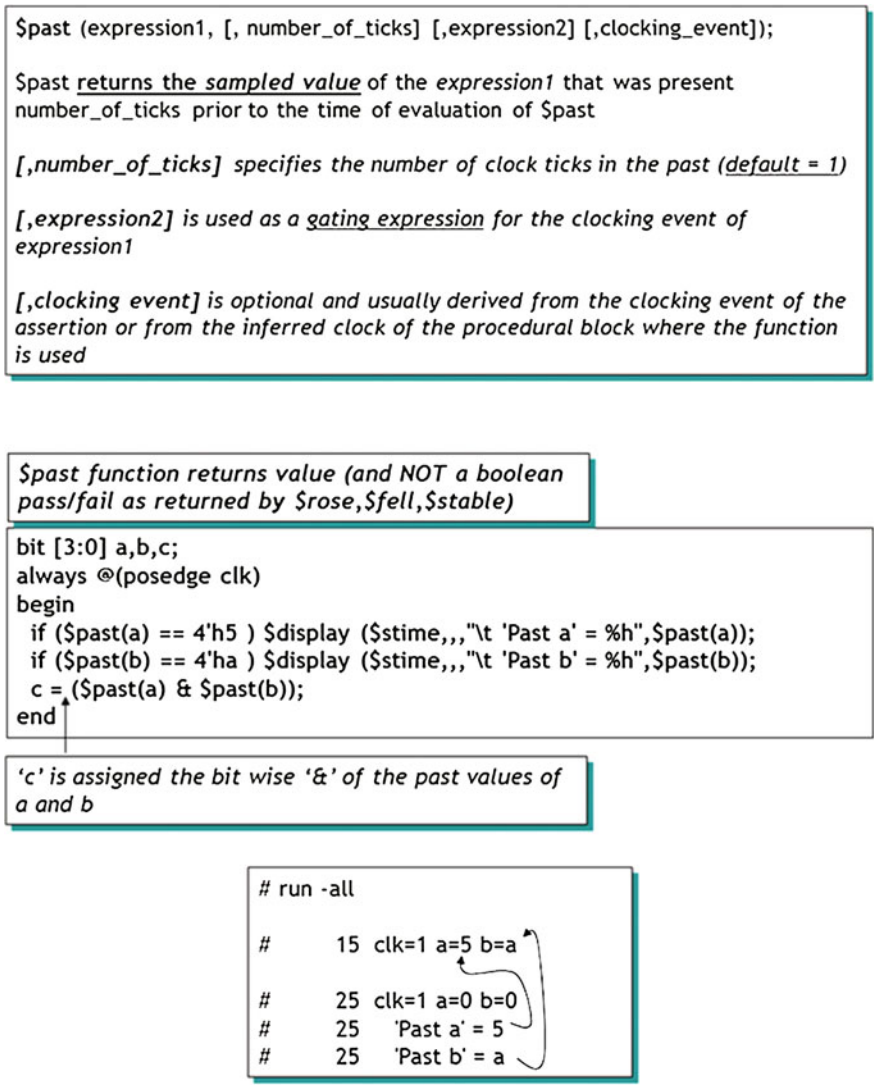


Fig. 5.9 $past—basics

The property says check on Sig, numClocks in the past and see if it has the value 'lastV' and do this check *if and only if* 'enb' (the gating expression) is high when you *start* the check (i.e. when antecedent done=1 in the assert statement). *Let me re-emphasize that the gating expression is checked when the 'antecedent' is true*. When you *start* the check, the gating expression need to be true. Many seem to miss this point. Let us look at the simulation log which will explain this concept.

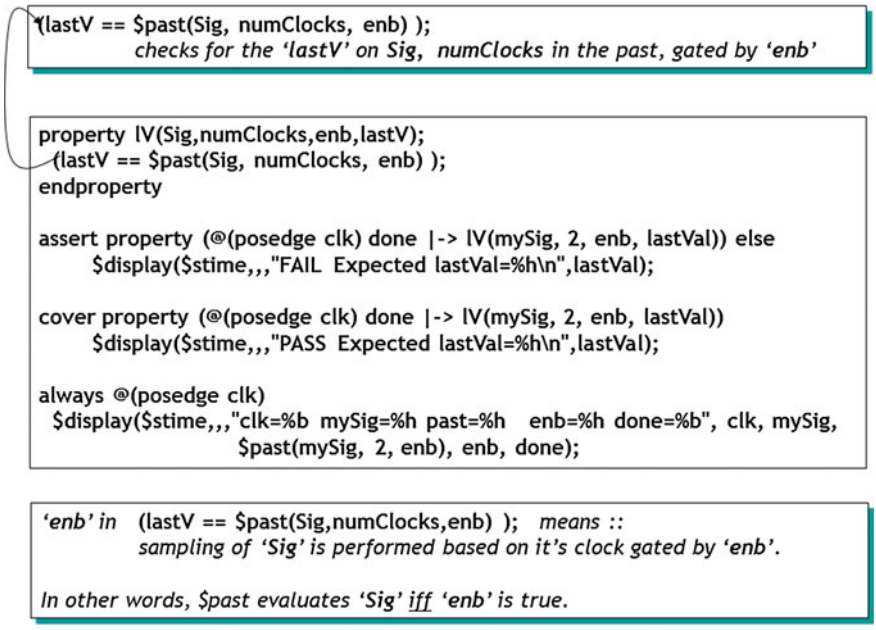


Fig. 5.10 $past—gating expression

The example from previous page is repeated here with lastV='ha when we assert/cover the property 'lV' (stands for Last Value) for easy reference to the simulation log (Fig. 5.11).

Let us examine the simulation log carefully to see how $past works. At time 30, done=1 for the first time which means that the antecedent of the property is true implying that the property lV be executed. lV has formal arguments which are replaced by the actual arguments from the assert statement.

So, at time 30, the property first checks to see if the gating signal ('enb') is true. Since it is indeed true, the property now evaluates the value of mySig 2 clocks in the past. It sees that it is indeed h'a (at time 10 in the simulation log). The property passes.

At time 50, done=1 and enb=1 but 2 clocks in the past (at time 30), mySig was not equal to h'a and the property fails.

At time 80, done=1, but the gating signal 'enb' is a '0'. The interesting thing to note here is that the property FAILs even though the value of mySig is indeed h'a two clocks in the past at time 60. That's because the gating expression enb=0 at the current clock tick (when antecedent 'done' is true) and the $past () does *not* evaluate itself. In other words, $past() did not look for mySig 2 clocks in the past, instead returned its *previously* evaluated value h'5. The property compares this value of h'5 with expected value of h'a and fails.

```
property lV(Sig, numClocks, enb, lastV);
  (lastV == $past(Sig, numClocks, enb) );
endproperty

assert property (@(posedge clk) done |-> lV(mySig, 2, enb, 'ha)) else
      $display($stime,,,"FAIL  Expected lastVal=%h\n",lastVal);

cover property (@(posedge clk) done |-> lV(mySig, 2, enb, 'ha))
      $display($stime,,,"PASS  Expected lastVal=%h\n",lastVal);

always @(posedge clk)
  $display($stime,,,"clk=%b  mySig=%h past=%h   enb=%h done=%b", clk, mySig,

$past(mySig, 2, enb), enb, done);
```

```
# run -all
#       10  clk=1 mySig=a past=0   enb=1 done=0
#       20  clk=1 mySig=5 past=0   enb=1 done=0
#       30  clk=1 mySig=5 past=a   enb=1 done=1
#       30  PASS Expected lastVal=a
#
#       40  clk=1 mySig=5 past=5   enb=1 done=0
#       50  clk=1 mySig=a past=5   enb=1 done=1
#       50  FAIL Expected lastVal=a
#
#       60  clk=1 mySig=a past=5   enb=1 done=0
#       70  clk=1 mySig=5 past=5   enb=0 done=0
#       80  clk=1 mySig=5 past=5   enb=0 done=1
#       80  FAIL Expected lastVal=a
#
```

A time 80 :

Even though mySig's $past(2) value (at time 60) is "a", && enb=1, the property fails at 80 when evaluated (with done=1) because enb=0 at the current clock tick and the lastV retains the previously sampled value of "5" and the comparison with "a" fails.

Fig. 5.11 $past—gating expression—simulation log

Without a gating signal, the property will always evaluate whenever the antecedent is true and look for the required expression value N number of clocks in the past.

Note also the use of $past in the $display statement which is a procedural statement. This is an excellent debug feature. You can always display what happened in the past to debug the current state of design.

5.3.1 Application: $past ()

Figure 5.12 is self-explaining. Note that $past is used in consequent in the application at the top of the figure and used as an antecedent in the bottom application.

5.3.2 $past Rescues $fell!

Figure 5.13 shows the difference between $rose/$fell with $past. Recall that $fell (or $rose) samples only the LSB of the expression/signal whose value we are evaluating. In contrast, $past evaluates the entire expression. Hence, if you want to check (for example) the value of an entire 'bus', you have to use $past. As shown in the figure, $fell will give an incorrect evaluation of the 32-bit bus 'dBus' if in fact

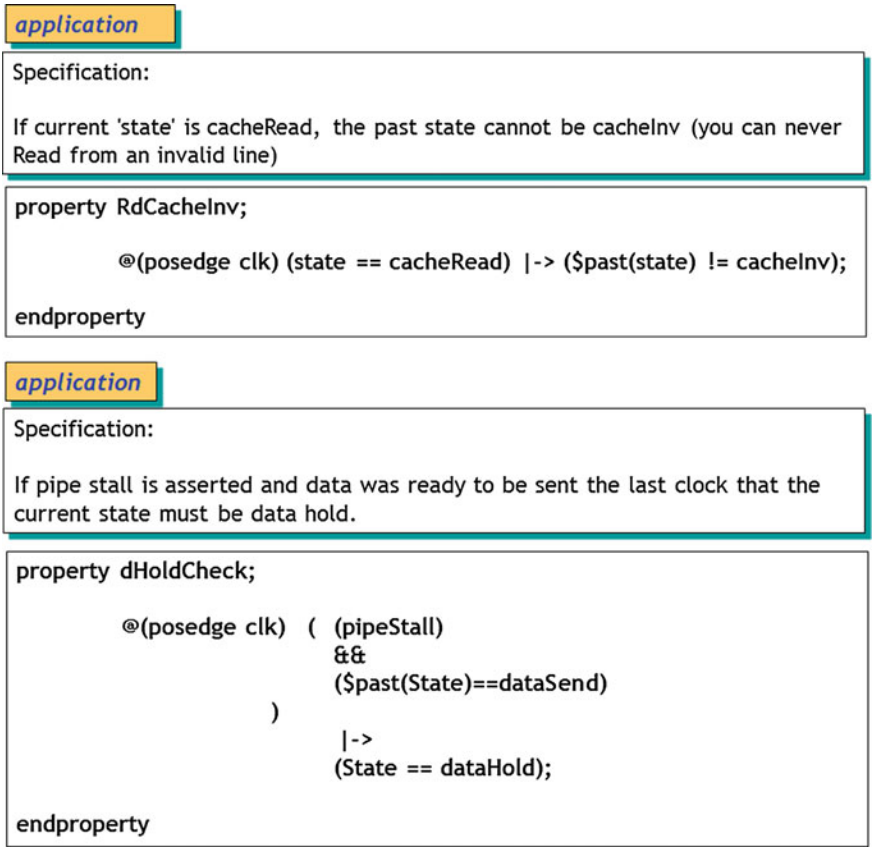


Fig. 5.12 $past application

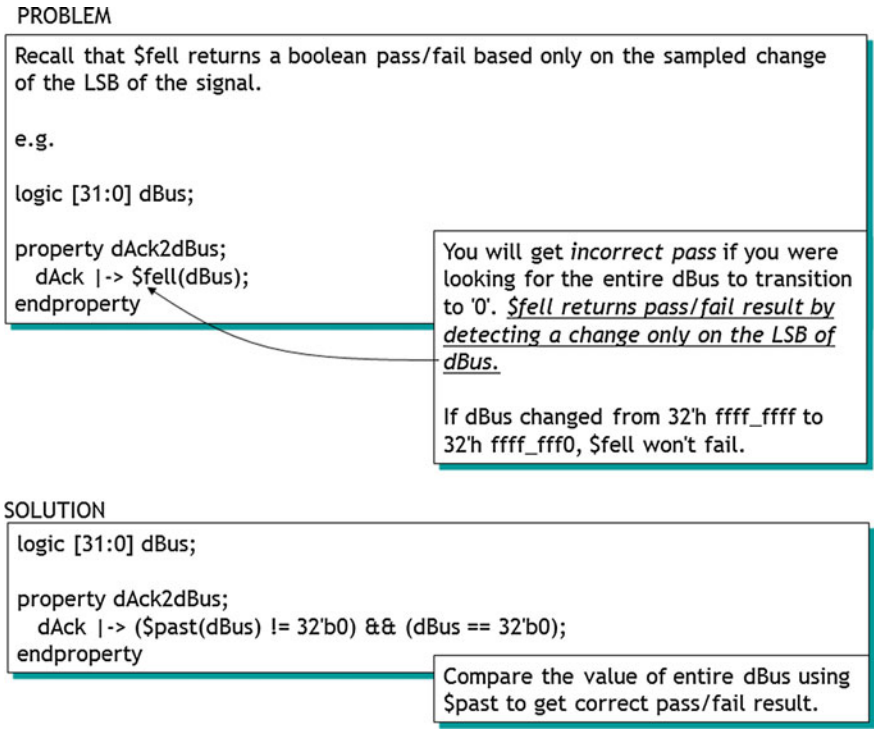


Fig. 5.13 $past rescues $fell

you want to check how the entire bus evaluated at some number of clocks in the past. The figure explains how you can use $past to solve this problem which $fell could not.

Chapter 6
Operators

Introduction: This chapter is the big one! This chapter describes all the operators offered by the language (both for a 'sequence' and a 'property') including Clock Delay with and without range, Consecutive repetition with and without range, non-consecutive repetition with and without range, 'throughout', 'within', 'and', 'or', 'intersect', 'first_match', 'if…else', etc. Each of the operator description is immediately followed by examples and applications to solidify the concept.

Following lists all the operators offered by the language (IEEE-1800, 2005). We will discuss features of1800-2009/2012 LRM in a separate chapter (see Chap. 16). We will examine each operator in detail since these operators are the stronghold of the language (Table 6.1).

6.1 ##m—Clock Delay

Clock delay is about the most basic of all the operators and probably the one you will use the most! First of all, note that ##m means a delay of 'm' number of sampling edges. In this example, the sampling edge is a 'posedge clk', hence ##m means m number of posedge clks (Fig. 6.1).

The property evaluates antecedent 'z' to be true at posedge clk and implies the sequence 'Sab'. 'Sab' looks for 'a' to be true at that same clock edge (because of the overlapping operator used in the property) and if that is true, waits for two posedge clks and then looks for 'b' to be true.

In the simulation log, we see that at time 10, posedge of clk, z=1 and a=1. Hence, the sequence evaluation continues. Two clks later (at time 30), it checks to see if b=1, which it finds to be true and the property passes.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_6

Table 6.1 Concurrent assertion operators

Operator	Description
##m ##[m:n]	Clock delay
[*m] [*m:n]	Repetition—consecutive
[=m] [=m:n]	Repetition—non consecutive
[->m] [->m:n]	GoTo repetition—non consecutive
Sig1 throughout seq1	Signal sig1 must be true throughout sequence seq1
Seq1 within seq2	Sequence seq1 must be contained within sequence seq2
Seq1 intersect seq2	'intersect' of two sequences; same as 'and' but both sequences must also 'end' at the same time
Seq1 and seq2	'and' of two sequences. Both sequences must start at the same time but may end at different times
Seq1 or seq2	'or' of two sequences. It succeeds if either sequence succeeds.
first_match complex_seq1	Matches only the first of possibly multiple matches
not <property_expr>	If <property_expr> evaluates to true, then not <property_expr> evaluates to false; and vice versa
if (expression) property_expr1 else property_expr2	If…else within a property
|->	Overlapping implication operator
|=>	Non-overlapping implication operator

Similar scenario unfolds starting time 40. But this time, b is not equal to 1 at time 60 (two clks after time 40) and the property fails.

We can see that ##m is absolute delay. Can you have 'm' to be a variable? Short answer is No. But there is an interesting way to make it variable using a 'counter' technique. Please see Sect. 14.8.

Now, let us look at what happens if m=0. That would mean ##m is equal to ##0… hmmm, no delay!

6.1.1 Clock Delay Operator: ##m Where m=0

We examine the property as before but with m=0. As expected, the sequence 'Sab' looks for 'a' to be true and then at the same clock looks for 'b' to be true. In addition, in this property we are using overlapping implication operator, which means when 'z' is true, 'a' should be true and so should be 'b'—all at the same time. This is one of the ways you can check for multiple expressions to be true at the same sampling edge (or 'clk' edge) (Fig. 6.2).

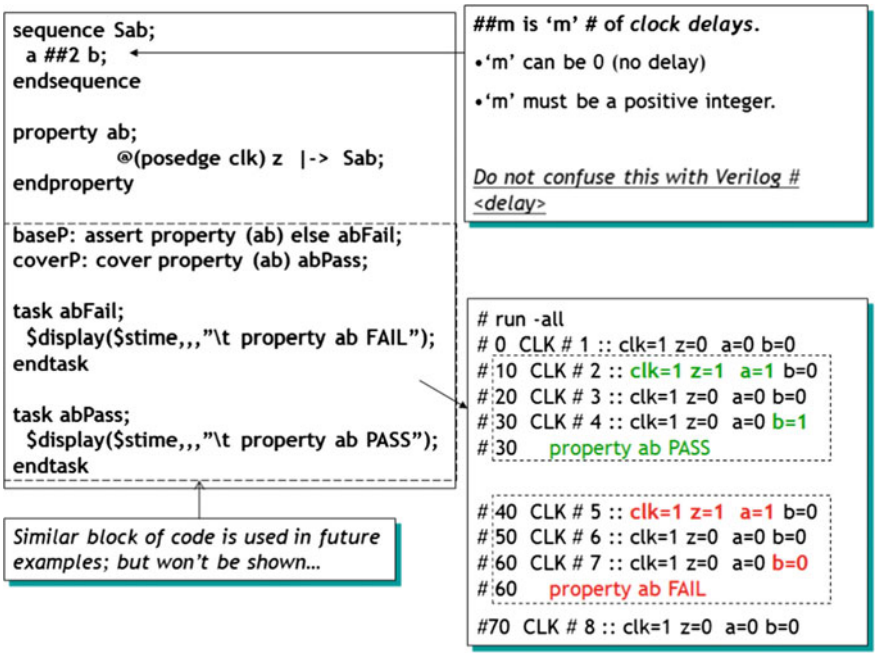


Fig. 6.1 ##m clock delay—basics

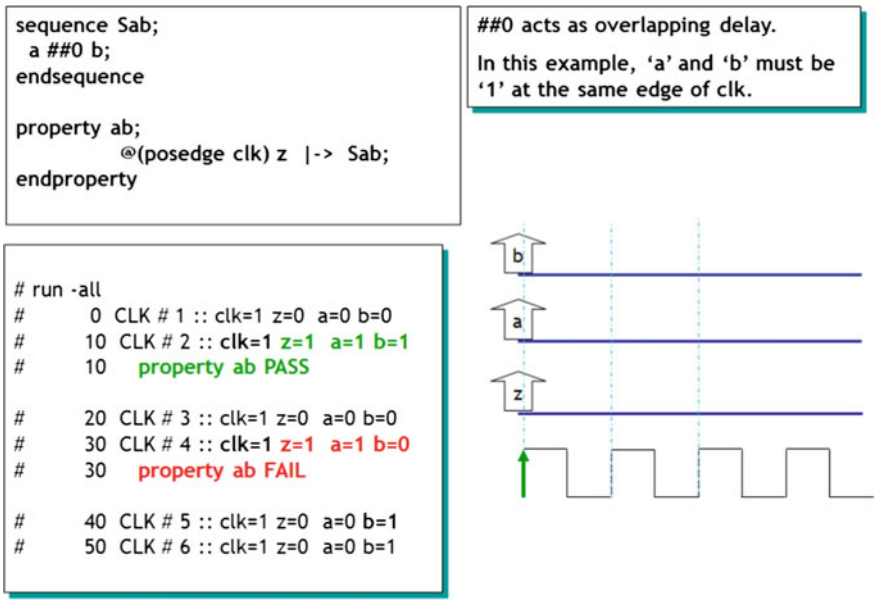


Fig. 6.2 ##m clock delay with m=0

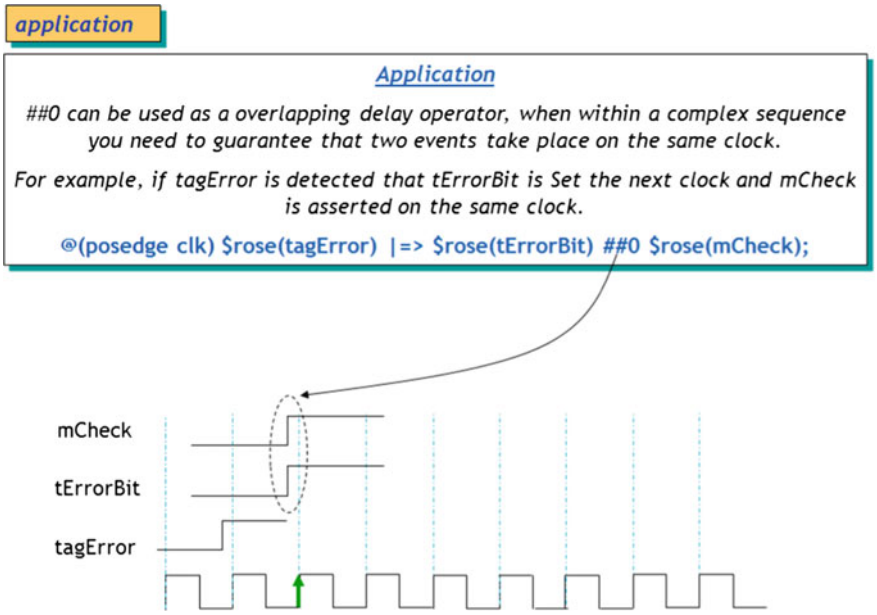


Fig. 6.3 ##0—application

Here's a good application where, in a complex sequence, you can effectively use ##0. *Note that you could have also used '&&' in place of ##0, obviously.*

6.1.1.1 Application: Clock Delay Operator :: ##m (m=0)

See Fig. 6.3.

6.2 ##[m:n]—Clock Delay Range

Since it is quite necessary for a signal or expression to be true in a given *range* of clocks (as opposed to fix number of clocks), we need an operator that does just the same.

##[m:n] allows a range of sampling edges (clock edges) in which to check for the expression that follows it. Figure 6.4 explains the rules governing the operator. Note that here also, m and n need to be constants. They cannot be variables.

The property 'ab' in the figure says that if at the first posedge of clk that 'z' is true that sequence 'Sab' be triggered. Sequence 'Sab' evaluates 'a' to be true the

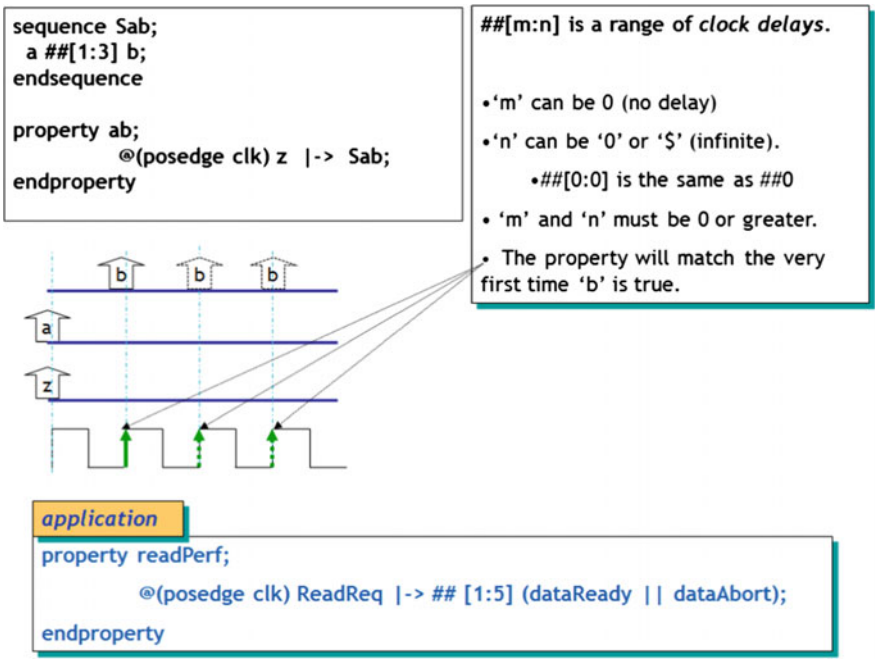


Fig. 6.4 ##[m:n] clock delay range

same clock that 'z' is true and then looks for 'b' to be true delayed by either 1 clk or 2 clks or 3 clks. The very –first- instance that 'b' is found to be true within the 3 clocks, the property will pass. If 'b' is not asserted within 3 clks, the property will fail.

Note that in the figure you see 3 passes. That simply means that whenever 'b' is true the first time within 3 clks that the property will pass. It does not mean that the property will be evaluated and pass 3 times. To reiterate, the property will pass as soon as (i.e. the first time) that 'b' is true.

6.2.1 Clock Delay Range Operator: ##[m:n]: Multiple Threads

Back to multiple threads! But this is a very interesting behavior of multi-threaded assertions. This is something you really need to understand.

At s1, 'rdy' is high and the antecedent is true. That implies that 'rdyAck' be true within the next 5 clks. s1 thread starts looking for 'rdyAck' to be true. The very next clock, rdyAck is not yet true but luck has it that 'rdy' is indeed true at this next clk (s2). This will fork off another thread that will also wait for 'rdyAck' to be true within the next 5 clks. The 'rdyAck' comes along within 5 clks from s1 and that thread is satisfied and will pass.

But the second thread will also pass at the same time, because it also got its 'rdyAck' within the 5 clks that it was waiting for.

This is a—very-important point to understand. *The range operator can cause multiple threads to complete at the same time.* This is in contrast to what we saw earlier with ##m constant delay where each thread will always complete only after the fixed ##m clock delays. There is a separate end to each separate thread. With the range delay operator multiple threads can end at the same time.

IMPORTANT: Let us further explore this concept since *it can indeed lead to false positive*. How would you know if rdyAck that satisfied both 'rdy's is for which 'rdy'? Also, if you did not receive 'rdyAck' for the second 'rdy' you will indeed get a false positive.

One hint is to keep the antecedent an edge sensitive function. For example, in the above example, instead of "@ (posedge clk) rdy" we could have used "@ (posedge clk) $rose(rdy)" which would have triggered the antecedent only once and there won't be any confusion of multiple threads ending at the same time. This is a performance hint as well. Use edge sensitive sampled value functions whenever possible. Level sensitive antecedent can fork off unintended multiple threads affecting simulation performance.

But a better solution is to use Local Variables to ID each 'rdy' and 'rdyAck'. This will indeed make sure that you received a 'rdyAck' for each 'rdy' and also that each 'rdyAck' is associated with the correct 'rdy.

Now, I haven't explained Local Variables yet! Please refer to the chapter on Local Variables (see Chap. 9) to get familiar with it. That chapter is in-depth study of Local Variables. But you don't need to understand that entire chapter to understand the following example. Just scan through the initial pages and you will understand that Local Variables are *dynamic* variables with each instance of the sequence forking off another independent thread. Very powerful feature. You don't need to keep track of the pipelined behavior. The local variable does it for you. Having understood that, you should be able to follow the following example.

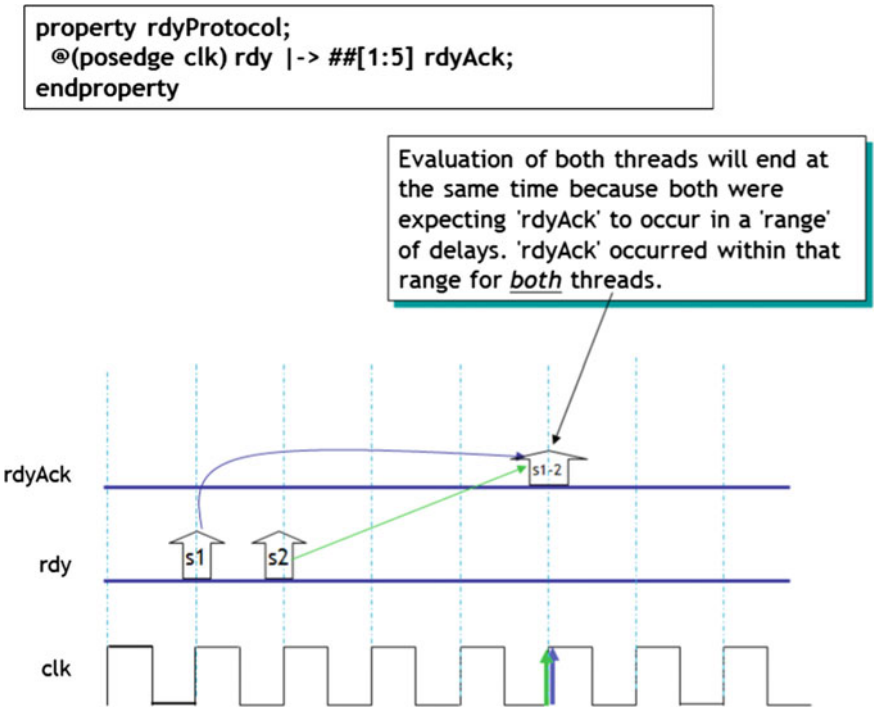


Fig. 6.5 ##[m:n]—multiple threads

Problem Statement:

Please refer to Fig. 6.5. Two 'rdy' signals are asserted on consecutive clocks with a range of clocks during which a 'rdyack' must arrive. 'rdyack' arrives that satisfies the time range requirements for *both* 'rdy's and the property passes. We have no idea whether a 'rdyack' arrived for each 'rdy'. The PASS of the assertion does not guarantee that. In this example, I am also using the concept of attaching subroutines, which is not covered so far. Please see Sect. 14.3 to get a high level glimpse of it; but it is quite intuitive and you should be able to understand attachment of $display statements in the example.

This example simulates the property and shows that both instances of 'rdy' will PASS with a single 'rdyAck'.

```
module range_problem;

logic clk, rdy, rdyAck;

initial
begin

 clk=1'b0; rdy=0; rdyAck=0;

 #500 $finish(2);

end

always begin

        #10 clk=!clk;

end


initial

begin

          repeat (5) begin @(posedge clk) rdy=~rdy; end

end


initial $monitor($stime,,,"clk=",clk,,,"rdy=",rdy,,,"rdyAck=",rdyAck);

initial

begin

          repeat (4) begin @(posedge clk); end

          rdyAck=1;

end


sequence rdyAckCheck;

        (1'b1,$display($stime,,,"ENTER SEQUENCE rdy ARRIVES")) ##[1:5]
($rose(rdyAck),$display($stime,,,"rdyAck ARRIVES"));

endsequence
```

```
gcheck: assert property (@(posedge clk) $rose (rdy) |-> rdyAckCheck) begin $display($stime,,,"PASS");
end

        else begin $display($stime,,,"FAIL"); end


endmodule


/* Simulation log

      0  clk=0  rdy=0  rdyAck=0

#     10  clk=1  rdy=1  rdyAck=0

#     20  clk=0  rdy=1  rdyAck=0

#     30  ENTER SEQUENCE rdy ARRIVES ← First 'rdy' is detected

#     30  clk=1  rdy=0  rdyAck=0

#     40  clk=0  rdy=0  rdyAck=0

#     50  clk=1  rdy=1  rdyAck=0

#     60  clk=0  rdy=1  rdyAck=0

#     70  ENTER SEQUENCE rdy ARRIVES ← Second 'rdy' is detected

#     70  clk=1  rdy=0  rdyAck=1

#     80  clk=0  rdy=0  rdyAck=1

#     90  clk=1  rdy=1  rdyAck=1

#     90  rdyAck ARRIVES ← 'rdyack' detected for both 'rdy's and the property PASSes.

#     90  PASS

*/

Solution:

module range_solution;

logic clk, rdy, rdyAck;

byte rdyNum, rdyAckNum;

initial

begin

 clk=1'b0; rdy=0; rdyNum=0; rdyAck=0; rdyAckNum=0;

 #500 $finish(2);

end
```

```
always
begin
        #10 clk=!clk;
end
initial
begin
      repeat (4)
      begin
        @(posedge clk) rdy=0;
        @(posedge clk) rdy=1; rdyNum=rdyNum+1;
      end
end


initial
$monitor($stime,,,"clk=",clk,,,"rdy=",rdy,,,"rdyNum=",rdyNum,,,"rdyAckNum",rdyAckNum,,,"rdyAck=",rd
yAck);
always
begin
          repeat (4)
            begin
                @(posedge clk); @(posedge clk); @(posedge clk);
            rdyAck=1; rdyAckNum=rdyAckNum+1;
            @(posedge clk) rdyAck=0;
          end
end
```

```
sequence rdyAckCheck;

byte localData; //local variable 'localData' declaration. Note this is a dynamic variable. For every entry
into the

                //sequence it will create a new instance of localData and follow an independent thread.

    (1'b1,localData=rdyNum,

        $display($stime,,,"ENTER SEQUENCE",,,"LOCAL rdyNum=",localData))

        ##[1:5]

        ((rdyAck && rdyAckNum==localData),

            $display($stime,,,"rdyAck ARRIVES ",,,"LOCAL",,,"rdyNum=",localData,,,
        "rdyAck=",rdyAckNum));

endsequence

gcheck: assert property (@(posedge clk) $rose (rdy) |-> rdyAckCheck) begin $display($stime,,,"PASS");
end

    else begin $display($stime,,,"FAIL",,,"rdyNum=",rdyNum,,,"rdyAckNum=",rdyAckNum); end

endmodule

/*

#       0  clk=0  rdy=0  rdyNum=   0  rdyAckNum   0  rdyAck=0

#      10  clk=1  rdy=0  rdyNum=   0  rdyAckNum   0  rdyAck=0

#      20  clk=0  rdy=0  rdyNum=   0  rdyAckNum   0  rdyAck=0

#      30  clk=1  rdy=1  rdyNum=   1  rdyAckNum   0  rdyAck=0

#      40  clk=0  rdy=1  rdyNum=   1  rdyAckNum   0  rdyAck=0

#      50  ENTER SEQUENCE  LOCAL rdyNum=   1 ← First 'rdy' arrives. A 'rdyNum' (generated in your
testbench

            as shown above) is assigned to 'localData'. This 'rdyNum' is a unique number for each
            invocation of the sequence and arrival of 'rdy'.

#      50  clk=1  rdy=0  rdyNum=   1  rdyAckNum   1  rdyAck=1

#      60  clk=0  rdy=0  rdyNum=   1  rdyAckNum   1  rdyAck=1

#      70  rdyAck ARRIVES   LOCAL  rdyNum=   1  rdyAck=   1

#      70  PASS ← When 'rdyAck' arrives, the sequence checks to see that its 'rdyAckNum' (again,
assigned in

            the testbench) corresponds to the first rdyAck. If the numbers do not match the
            property fails. Here they are indeed the same and the property PASSes.
```

70 clk=1 rdy=1 rdyNum= 2 rdyAckNum 1 rdyAck=0

80 clk=0 rdy=1 rdyNum= 2 rdyAckNum 1 rdyAck=0

90 ENTER SEQUENCE LOCAL rdyNum= 2 ← Second 'rdy' arrives. localData is assigned the second

'rdyNum'. This redNum will not overwrite the first rdyNum. Instead a second thread is forked off and 'localData' will maintain (store) the second 'rdyNum'.

90 clk=1 rdy=0 rdyNum= 2 rdyAckNum 1 rdyAck=0

100 clk=0 rdy=0 rdyNum= 2 rdyAckNum 1 rdyAck=0

110 clk=1 rdy=1 rdyNum= 3 rdyAckNum 1 rdyAck=0

120 clk=0 rdy=1 rdyNum= 3 rdyAckNum 1 rdyAck=0

130 ENTER SEQUENCE LOCAL rdyNum= 3

130 clk=1 rdy=0 rdyNum= 3 rdyAckNum 2 rdyAck=1

140 clk=0 rdy=0 rdyNum= 3 rdyAckNum 2 rdyAck=1

150 rdyAck ARRIVES LOCAL rdyNum= 2 rdyAck= 2

150 PASS ← When 'rdyAck' arrives, the sequence checks to see that its 'rdyAckNum' (again, assigned in

the testbench) corresponds to the second rdyAck. If the numbers do not match the property fails. Here they are indeed the same and the property PASSes. This is what we mean by pipelined behavior, in that, the second invocation of the sequence maintains its own copy of 'localData' and compares with the second 'rdyAck'. This way there is no question of which 'rdy' was followed by which 'rdyAck'. No false positive. Rest of the simulation log follows the same chain of thought.

⇨ Can you figure out why the property fails at #270? Hint: Start counting clocks at time #170 when fourth 'rdy' arrives. Did a 'rdyAck' arrive for that 'rdy'?

```
#     150 clk=1 rdy=1 rdyNum=  4 rdyAckNum  2 rdyAck=0
#     160 clk=0 rdy=1 rdyNum=  4 rdyAckNum  2 rdyAck=0
#     170 ENTER SEQUENCE  LOCAL rdyNum=  4
#     170 clk=1 rdy=1 rdyNum=  4 rdyAckNum  2 rdyAck=0
#     180 clk=0 rdy=1 rdyNum=  4 rdyAckNum  2 rdyAck=0
#     190 clk=1 rdy=1 rdyNum=  4 rdyAckNum  2 rdyAck=0
#     200 clk=0 rdy=1 rdyNum=  4 rdyAckNum  2 rdyAck=0
#     210 clk=1 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=1
#     220 clk=0 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=1
#     230 rdyAck ARRIVES  LOCAL rdyNum=  3 rdyAck=  3
#     230 PASS


#     230 clk=1 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=0
#     240 clk=0 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=0
#     250 clk=1 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=0
#     260 clk=0 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=0
#     270 FAIL rdyNum=  4 rdyAckNum=  3
#     270 clk=1 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=0
#     280 clk=0 rdy=1 rdyNum=  4 rdyAckNum  3 rdyAck=0
#     290 clk=1 rdy=1 rdyNum=  4 rdyAckNum  4 rdyAck=1
*/
```

Sequence ‘rdyAckCheck’ is explained as follows.

Upon entry in the sequence, a copy of localData is created and a rdyNum is stored into it. While the sequence is waiting for #[1:5] for the rdyAck to arrive another ‘rdy’ comes in and sequence ‘rdyAckCheck’ is invoked. Again, the localData is assigned the next rdyNum and stored. This is where dynamic variable concept comes into picture. The second store of rdyNum into localData does not clobber the first store. A second copy of the localData is created and its thread will also now wait for #[1:5]. This way we make sure that for each ‘rdy’ we will indeed a unique ‘rdyAck’. Please carefully examine the simulation log to see how this works. I’ve placed comments in the simulation log to explain the operation.

6.2.2 *Clock Delay Range Operator :: ##[m:n] (m=0; n=$)*

In Fig. 6.6 we are going extreme at both ends of the range, from 0 to infinity ('$' means infinite delay). As explained above, the sequence 'Sab' will look for 'b' to be true the same time as 'a' or expect it to be true any time until the simulator ends. It is fine and good if it finds 'b' to be true before simulation ends. If not, the simulator will (should) give a Warning that this assertion remains incomplete (Fig. 6.7).

This example is similar to what we saw earlier. But in this example, we expect 'tErrorBit' to rise in a certain range of clock delays. The figure explains how the assertion works. Note also that you could use '&&' in place of ##0 to achieve the same results. Since assertions are mainly temporal domain, I prefer to tie in everything with temporal domain constructs. But that's a matter of preference.

Note also the following two semantically equal statement but with different syntax. These are short forms.

- ##[*] is used as an equivalent representation of ##[0:$].
- ##[+] is used as an equivalent representation of ##[1:$].

6.3 [*m]—Consecutive Repetition Operator

As depicted in Fig. 6.8 the consecutive repetition operator [*m] sees that the signal/expression associated with the operator stays true for 'm' consecutive clocks. Note that 'm' *cannot* be $ (infinite # of consecutive repetition).

The important thing to note for this operator is that it will match at the *end* of the last iterative match of the signal or expression.

The example in Fig. 6.8 shows that when 'z' is true that at the next clock, sequence' Sc1' should start its evaluation. 'Sc1' looks for 'a' to be true and then waits for 1 clock before looking for 2 consecutive matches on 'b'. This is depicted in the simulation log. At time 10 'z' is high; at 20 'a' is high as expected (because of non-overlapping operator in property); at time 30 and 40, 'b' remains high matching the requirement b[*2]. At the end of the second high on 'b' the property meets all its requirements and passes.

The very next part of the log shows that the property fails because 'b' does not remain high for 2 consecutive clocks. Again, the comparison ends at the *last* clock where the consecutive repetition is supposed to end and then the property fails.

Figure 6.9 shows an interesting application where we effectively use the 'not' of the repetition operator. At posedge busClk, if ADS is high that starting the same busClk (overlapping operator), ADS is checked to see if it remains high consecutively for 2 busClk(s). If it does, then we take a 'not' of it to declare that the

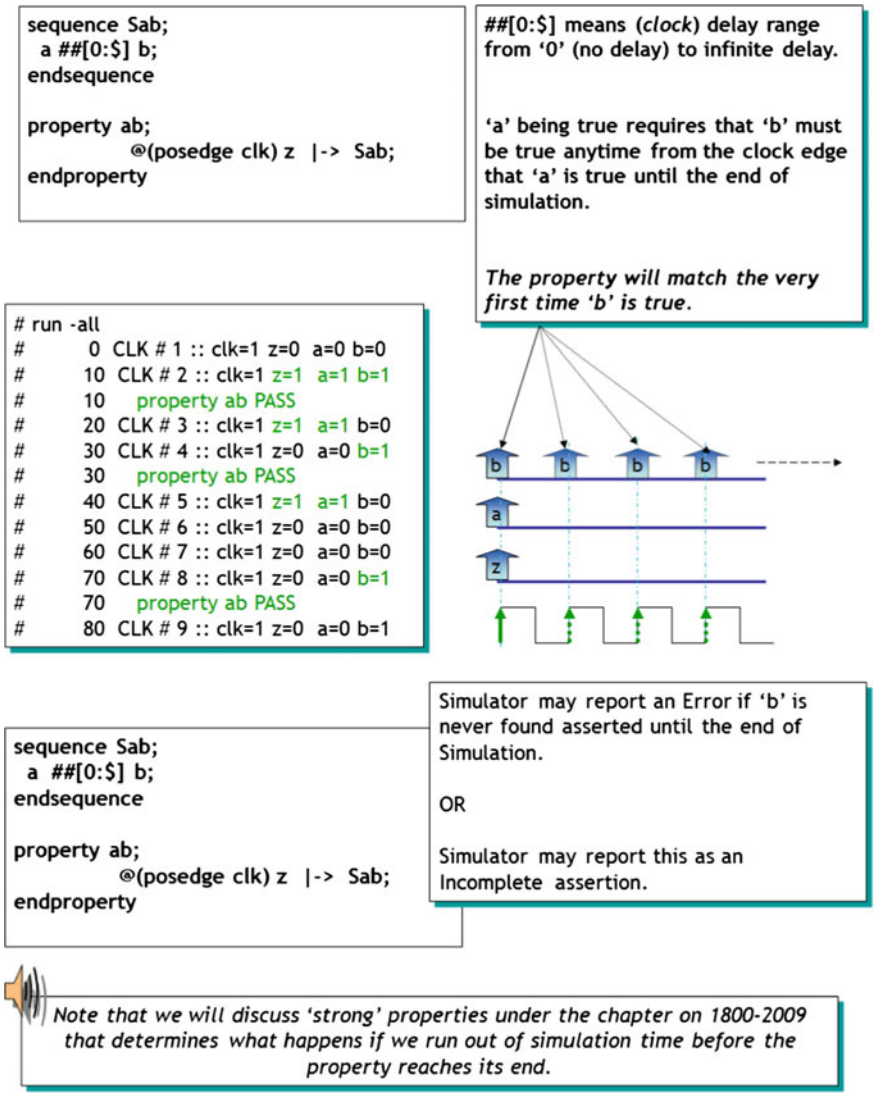


Fig. 6.6 ##[m:n] clock delay range with m=0 and n=$

property has failed. This is a very useful property, as simple as it looks. In many protocols one needs to make sure that certain signals follow very strict protocol. This application models just such a protocol. Note also the use of parameterized property.

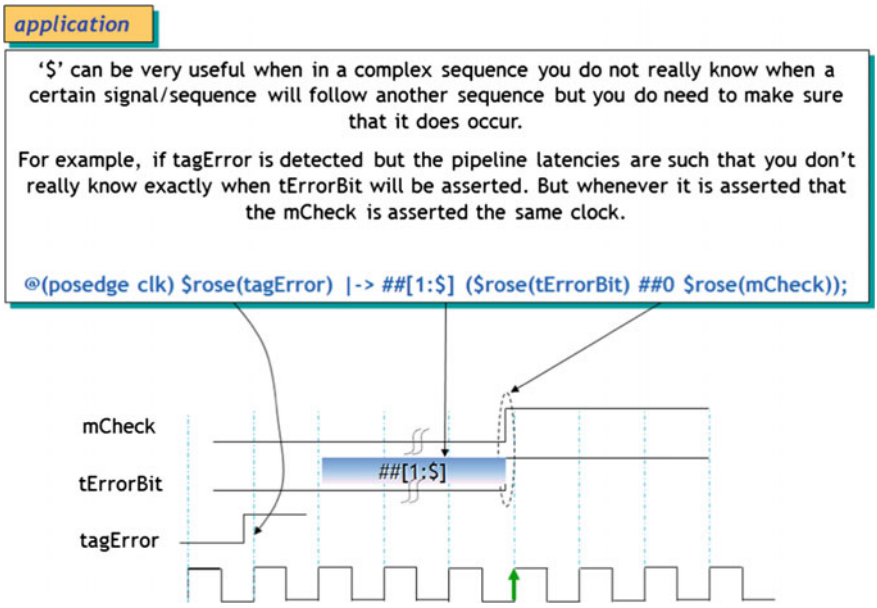


Fig. 6.7 ##[1:$] delay range application

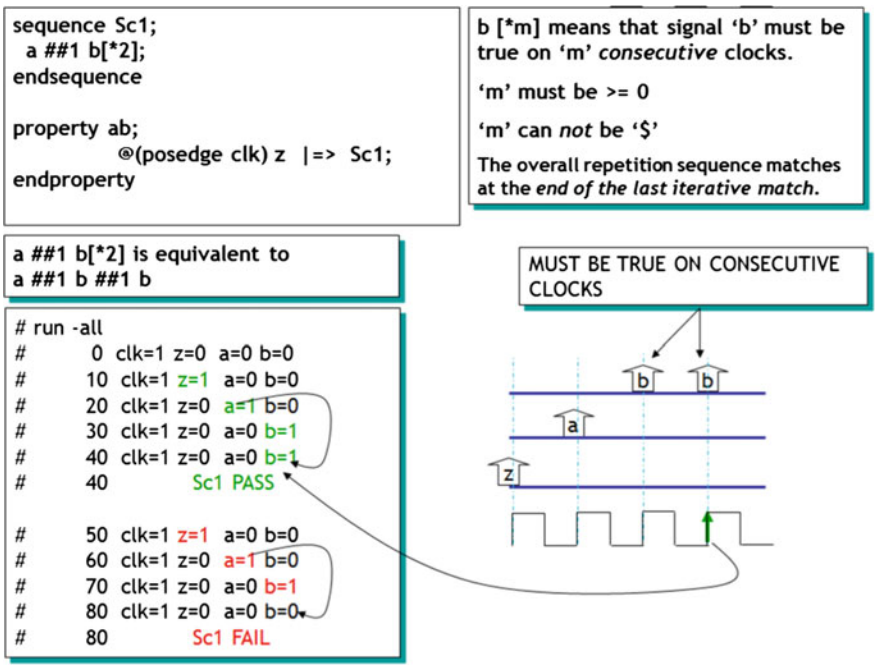


Fig. 6.8 [*m]—consecutive repetition operator—basics

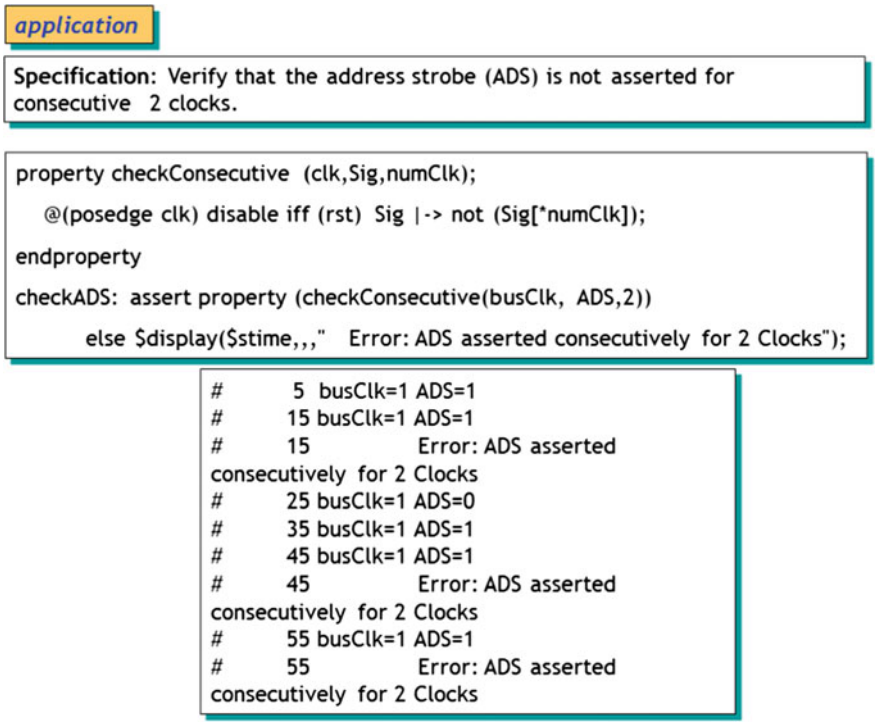


Fig. 6.9 [*m] consecutive repetition operator—application

Interesting note is that if ADS is high for 3 consecutive clocks, the property will fail twice during those 3 clocks. Please see if you can figure out why. Hint, 'Sig' is level sensitive.

What if you use [*m] on the antecedent side? Here's an example that explains that.

```
property cons_on_antecedent;
        @(posedge clk) a[*2] |-> ((##[1:3] c) or (d |=> e));
endproperty
```

Property 'cons_on_antecedent' says that if 'a' holds and 'a' also held *last cycle*, then either 'c' must hold at some point one to three cycles after the current cycle or, if 'd' holds in the current cycle, then 'e' must hold one cycle later. Note that here a[*2] means that the 'a' should have been consecutively held asserted for 2 clocks *at* the posedge clk.

6.4 [*m:n]—Consecutive Repetition Range

This is another important operator that you need to understand carefully how it works, as benign as it appears to be.

Let us start with the basics. sig[*m:n] means that sig should remain true for minimum 'm' number of consecutive clocks but no more than maximum 'n' number of consecutive clocks. That is simple enough. But here's the first thing that differs from the sig[*m] operator we just learnt. The consecutive range operator match ends at the *first* match of the sequence that meets the required condition. Note this point carefully. It ends at the *first* match of the range operator (in contrast the non-range operator [*m] which ends at the *last* match of the 'm').

Figure 6.10 outlines the fact that b[*2:5] is essentially an OR of four different matches. When any one of these four sequences matches that the property is considered to match and pass. In other words, the property first waits looking for

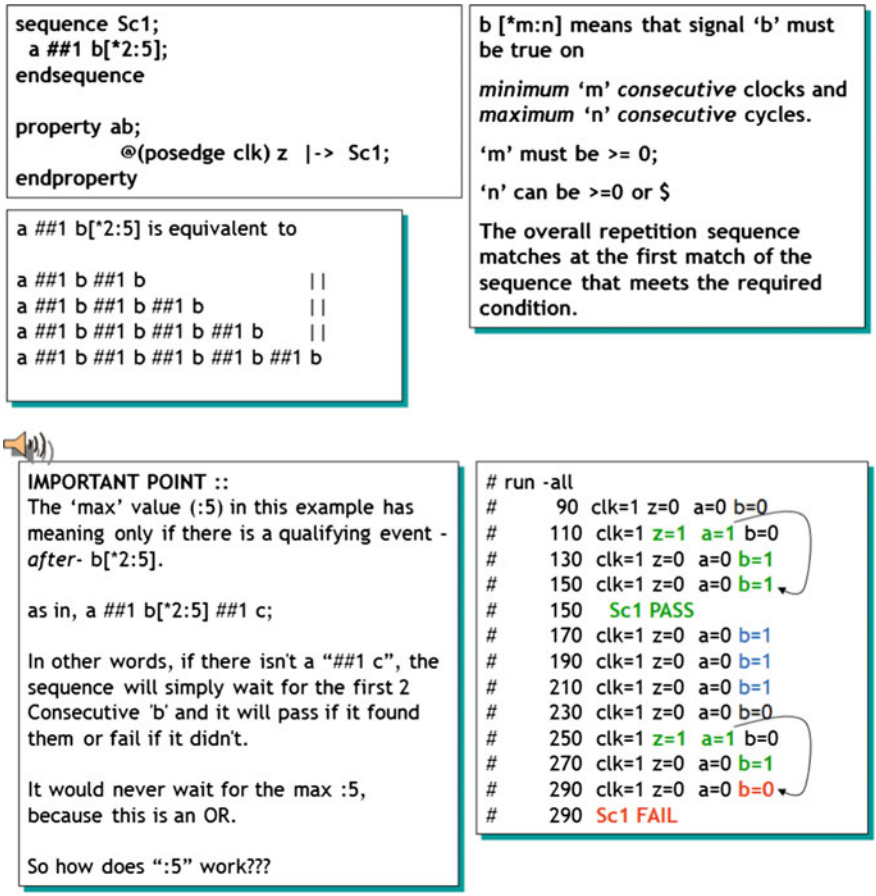


Fig. 6.10 [*m:n] consecutive repetition range—basics

two consecutive high on 'b'. If it finds that sequence, the property ends and passes. If it does not find the second 'b' to be true, the property will fail. It does *not* wait for the third consecutive high on 'b' because there isn't a third consecutive 'b' if it wasn't consecutively high in the second clock. The chain was already broken. So, if 'b' arrives in the second clock, the property will pass. If 'b' was not true in the second clock, the property would fail. It will not wait for the max range.

Back to the range b[*2:5]. If you think about it, :5 will *never* get executed!! If 'b' is true for 2 consecutive clocks, the property matches and ends (because the property ends at first match). And if 'b' wasn't true for 2 consecutive edges, the property will fail. Please study simulation log in Fig. 6.10 carefully.

So, why do we need the max range? When does the maximum range :5 come into picture? What does :5 really mean? See Fig. 6.11, it will explain what max range :5 means and how it gets used.

Note that we added '##1 c' in sequence Sc1. It means that there must be a 'c' (high) 1 clock after the consecutive operator match is complete. Ok, simple enough.

Now let's look at the simulation log. Time 30–90 is straightforward. At time 30, z=1 and a=1, the next clock 'b'=1 and remains '1' for two consecutive clocks and then 1 clock later c=1 as required and the property passes. But what if 'c' was not equal to '1' at time 90? That is what the second set of events show.

Z=1 and a=1 at time 110 and the sequence Sc1 continues. OK. b=1 the next 2 clocks. Correct. But why doesn't the property end here? Isn't it supposed to end at the first match? Well, the reason the property does not end at 150 is because it needs to wait for c=1 the next clock. OK, so it waits for C=1 at 170. But it does not see a c=1. Shouldn't the property now fail? NO. This is where the max range :5 comes into picture. Since there is a range [*2:5], if the property does not see a c=1 after the first two consecutive repetitions of 'b', it waits for the next consecutive 'b' (total 3 now) and then looks for 'c=1'. If it does not see c=1 it waits for the next consecutive b=1 (total 4 now) and then looks for c=1. Still no 'c'? It finally waits for max range 5th b=1 and then the next clock looks for c=1. If it finds one, the property ends and passes. If not, the property fails.

Continuing with the simulation log, the last part shows how the property would fail. One way it would fail is what I have described above. The other way is shown in the log file. I have repeated the log file here to help us concentrate only on that part of the log file.

```
#      250  clk=1 z=1  a=1 b=0 c=0

#      270  clk=1 z=0  a=0 b=1 c=0

#      290  clk=1 z=0  a=0 b=1 c=1

#      310  clk=1 z=0  a=0 b=0 c=0

#      310  Sc1 FAIL
```

At time 250, z=1 and a=1 so the sequence evaluation continues to consecutive operator. 'b' is equal to 1 for the next two consecutive clocks. Good. But at time

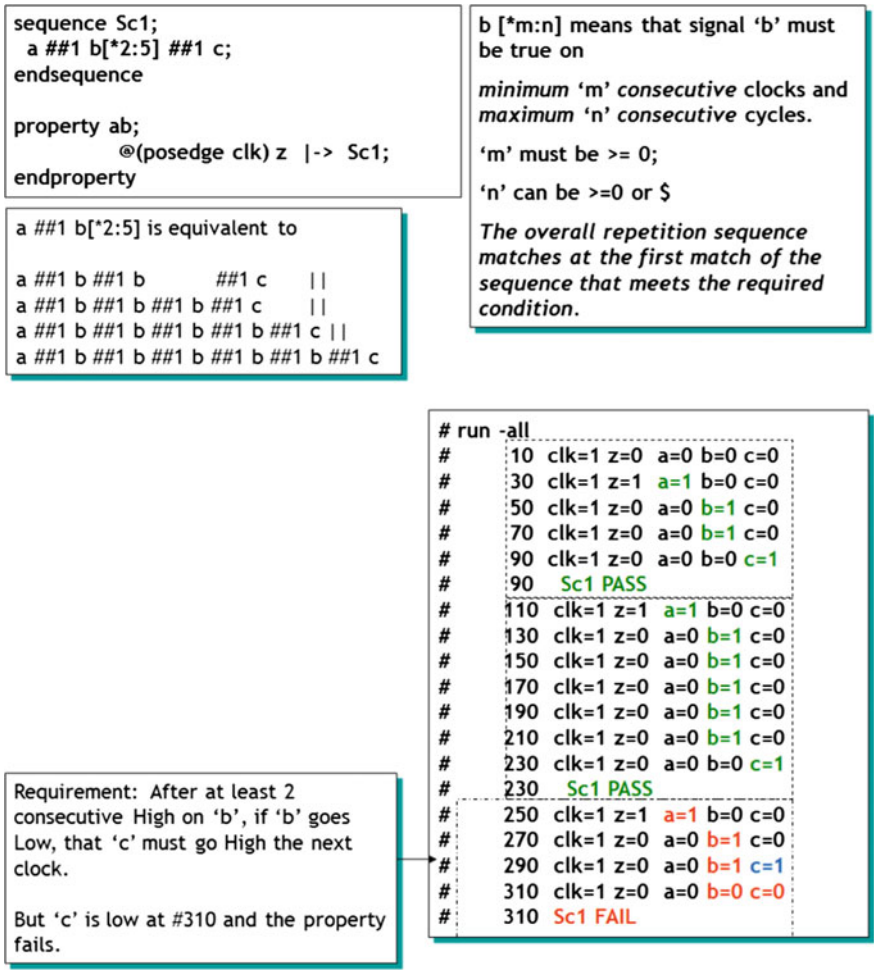


Fig. 6.11 [*m:n] consecutive repetition range—example

310, b=0 *and*—also—c=0. Hence the property fails. After two consecutive 'b', there should be either a third 'b' or a 'c=1'. Neither of them is present and the property fails. If C=1 at time 310, the property would pass. If b=1 and c=0 at time 310, the property would continue to evaluate until it sees 5 consecutive 'b' or a c=1

before 5 consecutive 'b' are encountered. Or after 5 consecutive 'b' that there is a c=1 as shown in the previous part of the simulation log file.

Confusing? That could be the case at first. However, please see the next few applications and this concept will be clear. *This is one of the most useful operators in the language* and the better you understand it, the more productive you will be.

Note also following new shortcuts introduced in LRM 2009.

##[*] is an equivalent representation of ##[0:$]
##[+] is an equivalent representation of ##[1:$]
[*] is an equivalent representation of [*0:$]
[+] is an equivalent representation of [*1:$]

6.4.1 Application: Consecutive Repetition Range Operator

This application is again on the same line that we have been following. Reason to carry on with the same example is to show how specification can change around seemingly similar logic.

Property in Fig. 6.12 says that at $rose(tagError), check for tErrorBit to remain asserted until mCheck is asserted. If tErrorBit does not remain asserted until mCheck gets asserted, the property should fail.

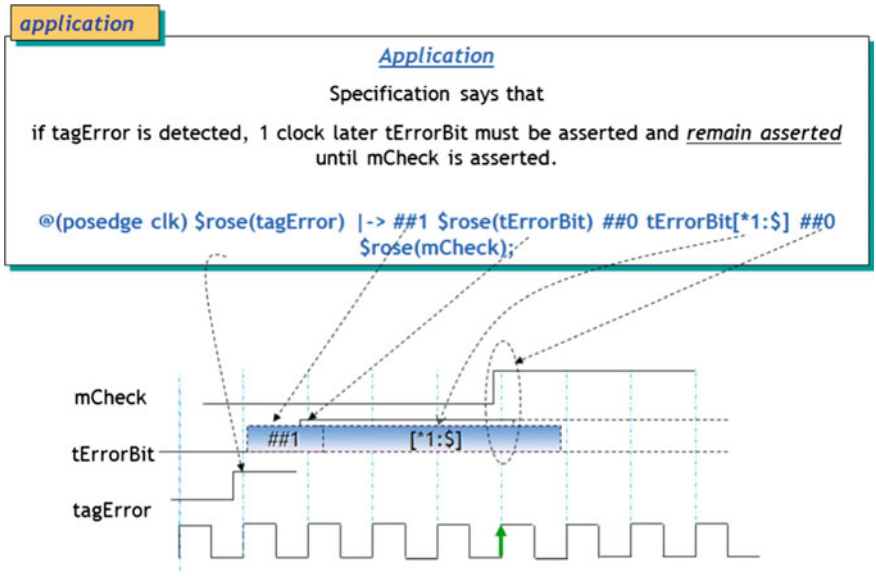


Fig. 6.12 [*m:n] consecutive repetition range—application

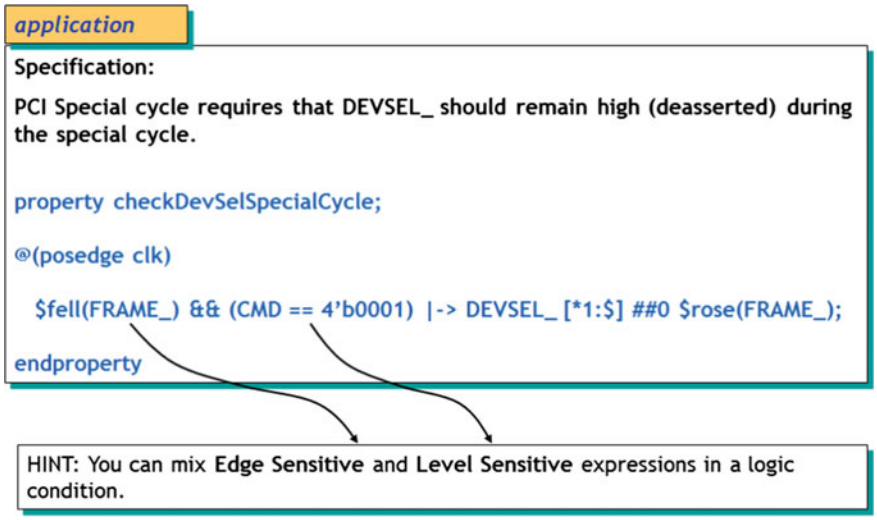


Fig. 6.13 [*m:n] consecutive repetition range—application

So, at $rose(tagError) and one clock later we check to see that $rose(tErrorBit) occurs. If it does, then we move forward at the same time (##0) with tErrorBit[*1:$]. This says that we check to see that tErrorBit remains asserted consecutively (i.e. at every posedge clk) until the *qualifying event* $rose(mCheck) arrives. In other words, the qualifying event is what makes consecutive range operator very meaningful as well as useful. Think of the qualifying event as the one that ends the property. This way, you can check for some expression to be true until the qualifying event occurs (Fig. 6.13).

A PCI cycle starts when FRAME_ is asserted (goes Low) and the CMD is valid. A CMD == 4'b0001 specifies the start of a PCI Special cycle. On the start of such a cycle (i.e. the antecedent being true), the consequent looks for DEVSEL_ to be high forever consecutively at every posedge clk until FRAME_ is de-asserted (goes High). This is by far the easiest way to check for an event /expression to remain true (and we do not know for how long) until another condition/expression is true (i.e. until what I call the qualifying event, is true).

Note also that you can mix edge sensitive and level sensitive expressions in a single logic expression. That is indeed impressive and useful.

Property in Fig. 6.14 states that if the currentState of the state machine is not IDLE and if the currentState remains stable for 32 clocks that the property should fail.

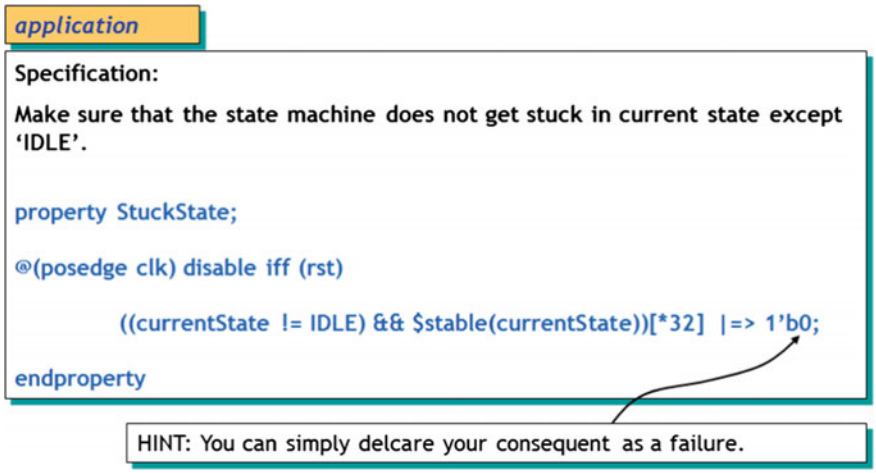


Fig. 6.14 [*m:n] consecutive repetition range—application

There are a couple of points to observe.

Note that the entire expression **((currentState != IDLE) && $stable (currentState))** is checked for consecutive repetition of 32 times because we need to check at every clock for 32 clocks that the currentState is not IDLE and whatever state that existed in previous clock still remains the same (i.e. $stable). In other words, you have to make sure that within these 32 clocks, the current State does not go back to IDLE. If it does, then the antecedent does not match and it will start all over again to check for this condition to be true (i.e. the antecedent to be true).

Note that if the antecedent is indeed true it would mean that the state machine is indeed stuck into the same state for 32 clocks. In such a case, we want to assertion to fire. That is taken care of by a hard failure in the consequent. We simply program consequent to fail without any pre-requisite.

As you notice, this property is unique in that the condition is checked for in the antecedent. The consequent is simply used to declare a failure (Fig. 6.15).

This application states that the state machine matches the state transition specification. If we are in `readStart state that after 1 clock, the state machine should be

application

```
Specification:
Make sure that the state machine follows the specified transitions

`define readStart (read_enb ##1 readStartState)
`define readID (readStartState ##1 readIDState)
`define readData (readIDState ##1 readDataState)
`define readEnd (readDataState ##1 readEndState)

sequence checkReadStates;

  @(posedge clk)
   `readStart              ##1
   `readID        [*1:$]   ##1
   `readData      [*1:$]   ##1
   `readEnd       ;

endsequence
```

Fig. 6.15 [*m:n] consecutive repetition range—application

in `readID state and stays in that state until the state machine reaches `readData state. It then is expected to stay in `readData state until `readEnd arrives. In short, we have made sure that the state machine does not stray and do an illegal transition until it reaches `readEnd.

Here's complete SystemVerilog code with built-in testbench and simulation log that exemplifies above property. Code is simple and self-explanatory. I tactfully (!) crafted the testbench such that the property passes.

Exercise: See if you can tweak the testbench to make the property fail. That will further solidify your concepts.

Note the use of `define to establish temporal relationship between signals and states. This makes the code very readable.

```
module state_transition;
int readStartState, readIDState, readDataState, readEndState;
logic clk, read_enb;

`define readStart (read_enb ##1 readStartState)
`define readID (readStartState ##1 readIDState)
`define readData (readIDState ##1 readDataState)
`define readEnd (readDataState ##1 readEndState)

property checkReadStates;

 @(posedge clk)
        `readStart        ##1
        `readID [*1:$]    ##1
        `readData[*1:$] ##1
        `readEnd          ;
endproperty

sCheck: assert property (checkReadStates) else $display ($stime,,,"FAIL");
cCheck: cover property (checkReadStates) $display ($stime,,,"PASS");

initial
begin
        read_enb=1; clk=0;
```

```
        @(posedge clk) readStartState=1;
        @(posedge clk) @(posedge clk); readIDState=1;
        @(posedge clk) @(posedge clk); readDataState=1;
        @(posedge clk) @(posedge clk); readEndState=1;

end

initial $monitor($stime,,,"clk=",clk,
                "read_enb=%0b",read_enb,,,
                "readStartState=%0b",readStartState,,
                "readIDState=%0b",readIDState,,
                "readDataState=%0b",readDataState,,
                "readEndState=%0b",readEndState);

always #10 clk=!clk;

endmodule

/*
#        0  clk=0read_enb=1  readStartState=0 readIDState=0 readDataState=0 readEndState=0
#       10  clk=1read_enb=1  readStartState=1 readIDState=0 readDataState=0 readEndState=0
#       20  clk=0read_enb=1  readStartState=1 readIDState=0 readDataState=0 readEndState=0
#       30  clk=1read_enb=1  readStartState=1 readIDState=0 readDataState=0 readEndState=0
#       40  clk=0read_enb=1  readStartState=1 readIDState=0 readDataState=0 readEndState=0
#       50  clk=1read_enb=1  readStartState=1 readIDState=1 readDataState=0 readEndState=0
```

```
#      60 clk=0read_enb=1 readStartState=1 readIDState=1 readDataState=0 readEndState=0
#      70 clk=1read_enb=1 readStartState=1 readIDState=1 readDataState=0 readEndState=0
#      80 clk=0read_enb=1 readStartState=1 readIDState=1 readDataState=0 readEndState=0
#      90 clk=1read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=0
#     100 clk=0read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=0
#     110 clk=1read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=0
#     120 clk=0read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=0
#     130 clk=1read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=1
#     140 clk=0read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=1
#     150 PASS
#     150 clk=1read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=1
#     160 clk=0read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=1
#     170 PASS
#     170 clk=1read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=1
#     180 clk=0read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=1
#     190 PASS
#     190 clk=1read_enb=1 readStartState=1 readIDState=1 readDataState=1 readEndState=1
*/
```

Let us examine one more application as follows (Figs. 6.16 and 6.17).

6.5 [=m]—Repetition Non-consecutive

Non-consecutive repetition is another useful operator (just as the consecutive operator) and used very frequently. In many applications, we want to check that a signal remains asserted or de-asserted a number of times and that we need *not* know when exactly these transitions take place. For example, (as we will see in Fig. 6.21), if there is a non-burst READ of length 8, that you expect 8 RDACK. These RDACK may come in a consecutive sequence *or not* (based on read latency). But you must have 8 RDACK before read is done.

Specification:

- When request is asserted that grant is asserted the very next clock.
  - *grant must have been de-asserted prior to it's assertion.*
- grant must remain asserted as long as request is asserted.
- grant must de-assert the very next clock after request is de-asserted.

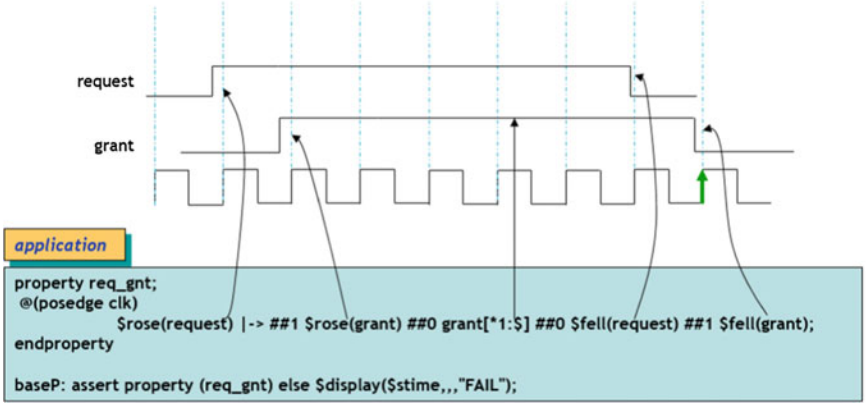


Fig. 6.16 Design application

```
# run -all
#        5  clk=1 request=0 grant=0
#       15  clk=1 request=0 grant=0
#       25  clk=1 request=1 grant=0
#       35  clk=1 request=1 grant=1
#       45  clk=1 request=1 grant=1
#       55  clk=1 request=1 grant=1
#       65  clk=1 request=0 grant=1
#       75  clk=1 request=0 grant=0
#       75  PASS
#       85  clk=1 request=0 grant=0
#       95  clk=1 request=1 grant=0
#      105  clk=1 request=1 grant=1
#      115  clk=1 request=1 grant=1
#      125  clk=1 request=1 grant=1
#      135  clk=1 request=1 grant=1
#      145  clk=1 request=1 grant=0
#      145  FAIL
```

grant falls before request. Hence the property fails.

Fig. 6.17 Design application—simulation log

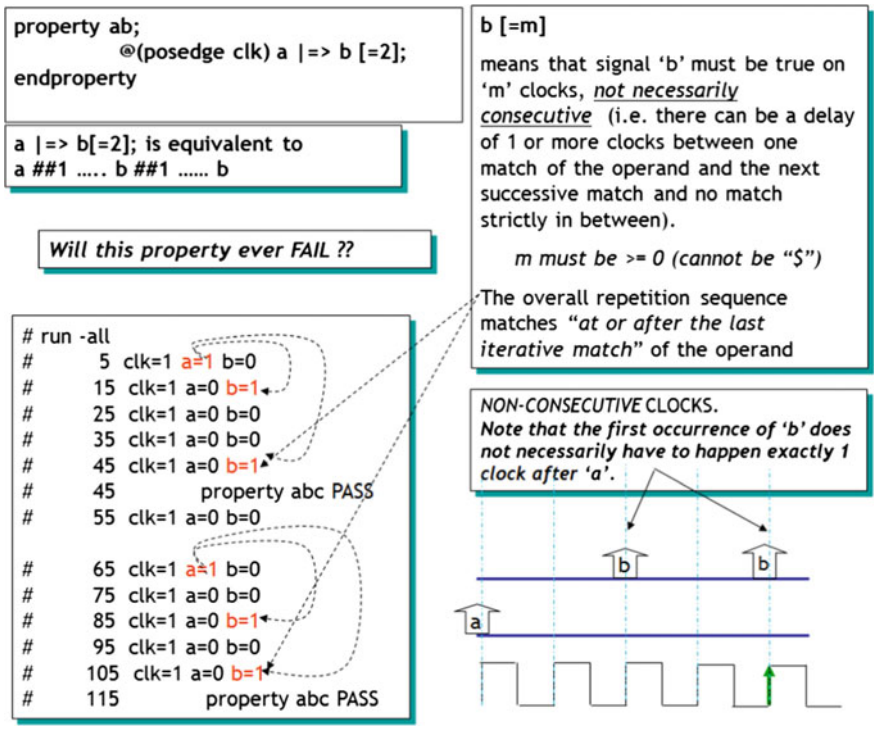


Fig. 6.18 Repetition non-consecutive operator—basics

In Fig. 6.18, property 'ab' says that if 'a' is sampled high at the posedge of clock that starting next clock, 'b' should occur twice not necessarily consecutively. They can occur any time after the assertion of 'a'. The interesting (and important) thing to note here is that even though the property uses non-overlapping implication operator (i.e. the first 'b' should occur 1 clock after 'a'=1), the first 'b' can occur *any time after* 1 clock after 'a' is found high. Not necessarily exactly 1 clock after 'a' !!!

In the simulation log, the first part shows that 'b' does occur 1 clock after 'a' and then is asserted again a few clocks later. This meets the property requirements and the assertion passes.

But note that second part of the log. 'b' does—not—occur 1 clock after 'a', rather 2 clocks later. And then it occurs again a few clocks later. Even this behavior is considered to meet the property requirements and the assertion passes.

Based on the description above, do you think this property will ever fail? Please experiment and see if you can come up with the answer. It will also further confirm your understanding. Hint: There is no qualifying event after 'b[=2]'.

Continuing with the same analogy, refer to the example below. Here again, just like in the consecutive operator, the qualifying event (##1 C in the example below) plays a significant role.

The example in Fig. 6.19 is identical to the previous except for the '##1 C' at the end of the sequence. The behavior of 'a |=>b[=2]' is identical to what we have seen above. '##1 c' tells the property that after the last 'b', 'c' must occur once and then it can occur any time after one clock after the last 'b'. Note again that even though we have '##1 c', 'c' does *not* necessarily need to occur 1 clock after the last 'b'. It can occur after any # of clks after 1 clock after the last 'b'—as long as—no other 'b' occurs while we are waiting for 'c'. Confusing! Not really. Let us look at the simulation log in Fig. 6.19. That will clarify things.

In the log, a=1 at time 5; b=1 at time 25 and then at 45. So far so good. We are marching along just as the property expects. Then comes in c=1 at time 75. That also meets the property requirement that 'c' occurs any time after last b=1. BUT note that *before* c=1 arrived at time 75, 'b' did not go to a '1' after its last occurrence at time 45. The property passes. Let us leave this at that for the moment. Now let us look at the second part of the log.

a=1 at time 95; then b=1 at 105 and 125; we are doing great. Now we wait for c=1 to occur any time after last 'b'. C=1 occurs at time 175. But the property fails before that!! What is going on? Note b=1 at time 145. That is not allowed in this property. The property expects a c=1 after the *last* occurrence of 'b' but *before* any other b=1 occurs. If another b=1 occurs *before* c=1 (as at time 145), then all bets are off.

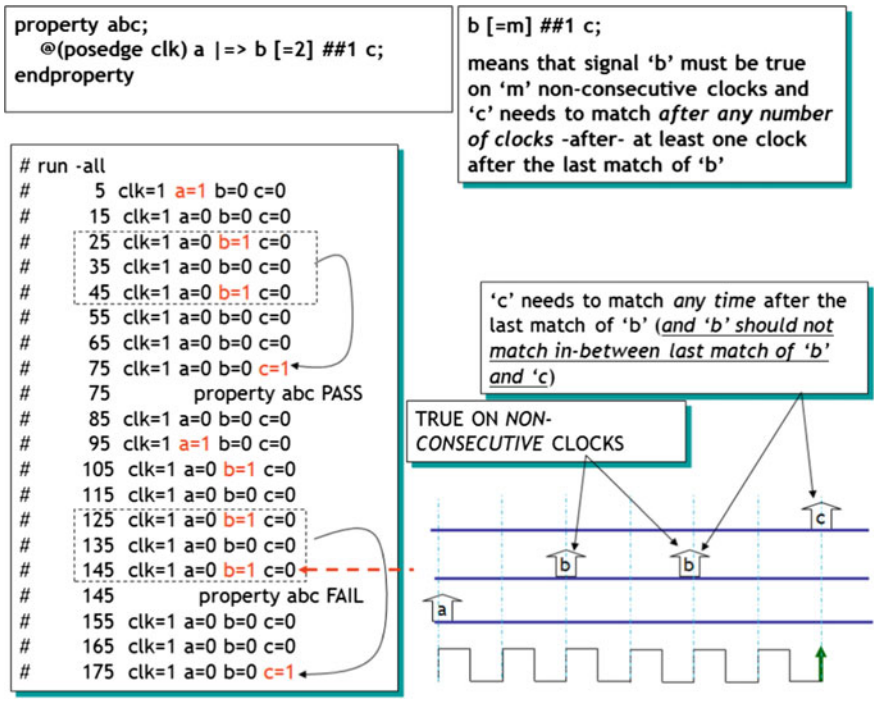


Fig. 6.19 Non-consecutive repetition operator—example

Property does not wait for the occurrence of c=1 and fails as soon as it sees this extra b=1. In other words, (what I call) the qualifying event "##1 c" encapsulates the property and strictly checks that b[=2] allows only 2 occurrences of 'b' before 'c' arrives.

6.6 [=m:n]—Repetition Non-consecutive Range

Property in Fig. 6.20 is analogs to the non-consecutive (non-range) property, except that this has a range. The range says (in the example above) that 'b' must occur minimum 2 times or maximum 5 times after which 'c' can occur one clock later any

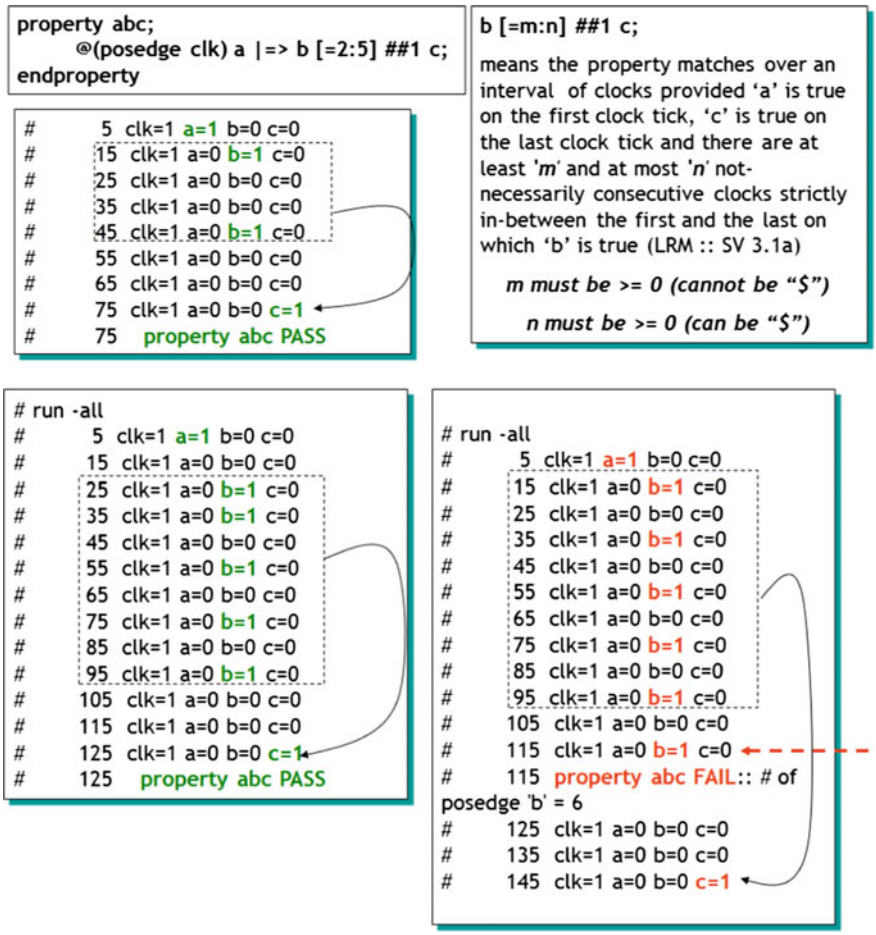


Fig. 6.20 Repetition non-consecutive range—basics

time and that no more than maximum of 5 occurrences of 'b' occur between the last occurrence of b=1 and c=1.

First simulation log (Top left) shows that after a=1 at time 5, b occurs twice (the minimum # of times) at time 15 and 45 and then c=1 at time 75. Why didn't the property wait for 5 occurrences of b=1? That is because after the second b=1 at time 45, c=1 arrives at time 75 and this c=1 satisfies the property requirement of minimum of two b=1 followed by a c=1. The property passes and does not need to wait for any further b=1. In other words, the property starts looking for 'c=1' after the minimum required (2) 'b==1'. Since it did find a 'c=1' after two 'b=1', the property ends there and passes.

Similarly, the simulation log on bottom left shows that 'b' occurs 5 (max) times and then 'c' occurs without any occurrence of b. The property passes. This is how that works. As explained above, after two 'b=1', the property started looking for 'c==1'. But before the property detects 'c==1', it sees another 'b==1'. That's OK because 'b' can occur maximum of five times. So, after the third 'b==1', the property continues to look for either 'c==1' or 'b==1' until it has reached maximum of five 'b==1'. This entire process continues until five 'b's are encountered. Then the property simply waits for a 'c'. While waiting for a 'c' at this stage, if a 6th 'b' occurs, the property fails. This failure behavior is shown in simulation log in the bottom right corner of Fig. 6.20.

6.6.1 Application: Repetition Non-consecutive Operator

Here is a practical example of using non-consecutive operator. The specs are provided in Fig. 6.21.

The property RdAckCheck will wait for nBurstRead to be high at the posedge clk. Once that happens, it will start looking for 8 RdAck before ReadDone comes along. If ReadDone comes in after 8 RdAck (and not 9) the property will pass. If ReadDone comes in before 8 RdAck come in the property will fail. Note that this will guarantee that the non-burst protocol is completely adhered to.

Following is an interesting example, just for fun… (Fig. 6.22).

The first case is interesting. At time 5 'a' is 1 (antecedent is true) which triggers the consequent. At time 15, c=1 and the property passes. But there was no occurrence of 'b'. b[=0] is an *empty* sequence which states that 'b' should *not* occur. We'll discuss empty sequences later in the book (see Sect. 14.16). The log from time 35–65 is quite straightforward. 'a==1' at time 35; 'b==1' at 55 and 'c==1' at time 65. Since the property states b[=0:$] and since 'b' did occur once followed by a 'c', the property passes.

But let us examine the log from time 75. At time 75, a=1 so the consequent fires. At time 85, both b=1 and c=1 and property passes! How? Note again that b[=0] part of the b[=0:$] range states that 'b' may never occur. That property is satisfied at time 75 (i.e. 'b' does not occur) and 1 clock later 'c' arrives, hence the property passes. Once you learn a bit more about the empty sequences, you will better

application

Specification:

- If nonBurst Read of length 8 is asserted that the RdAck must be asserted 8 times and ReadDone must be asserted anytime after the last Read and that there are no more RdAck's between the last RdAck and ReadDone.

```
property RdAckCheck (int length);

        @(posedge clk) nBurstRead  |=> RdAck [=length] ##1 ReadDone;

endproperty

aP: assert property (RdAckCheck (8));
```

Fig. 6.21 Repetition non-consecutive range—application

```
property abc;
        @(posedge clk) a |=> b [=0:$]
##1 c;
endproperty
```

b [=0:$] ##1 c;

means that the signal 'b' should be true either at no time 1 clock (after 'a' is true in this example) or it can be true infinite times until 'c' is asserted.

Will this property ever fail??

```
# run -all
#        5  clk=1 a=1 b=0 c=0
#       15  clk=1 a=0 b=0 c=1
#       15    property abc PASS

#       35  clk=1 a=1 b=0 c=0
#       45  clk=1 a=0 b=0 c=0
#       55  clk=1 a=0 b=1 c=0
#       65  clk=1 a=0 b=0 c=1
#       65    property abc PASS

#       75  clk=1 a=1 b=0 c=0
#       85  clk=1 a=0 b=1 c=1
#       85    property abc PASS
#       95  clk=1 a=0 b=1 c=1
```

Fig. 6.22 Repetition non-consecutive range—[=0:$]

understand this example. But the point here is that you need to be careful using the minimum and maximum range in an operator. The results may not be that apparent.

Empty sequences are discussed in Sect. 14.16

6.7 [->m] Non-consecutive GoTo Repetition Operator

This is the so-called non-consecutive goto operator! Very similar to [= m] non-consecutive operator. Note the symbol difference. The goto operator is [->2].

In Fig. 6.23, b[->2] acts exactly the same as b[=2]. So, why bother with another operator with the same behavior? It is the *qualifying event* that makes the difference. Recall that the qualifying event is the one that comes *after* the non-consecutive or the 'goto' non-consecutive operator. I call it qualifying because it is *the end event* that qualifies the sequence that precedes for final sequence matching.

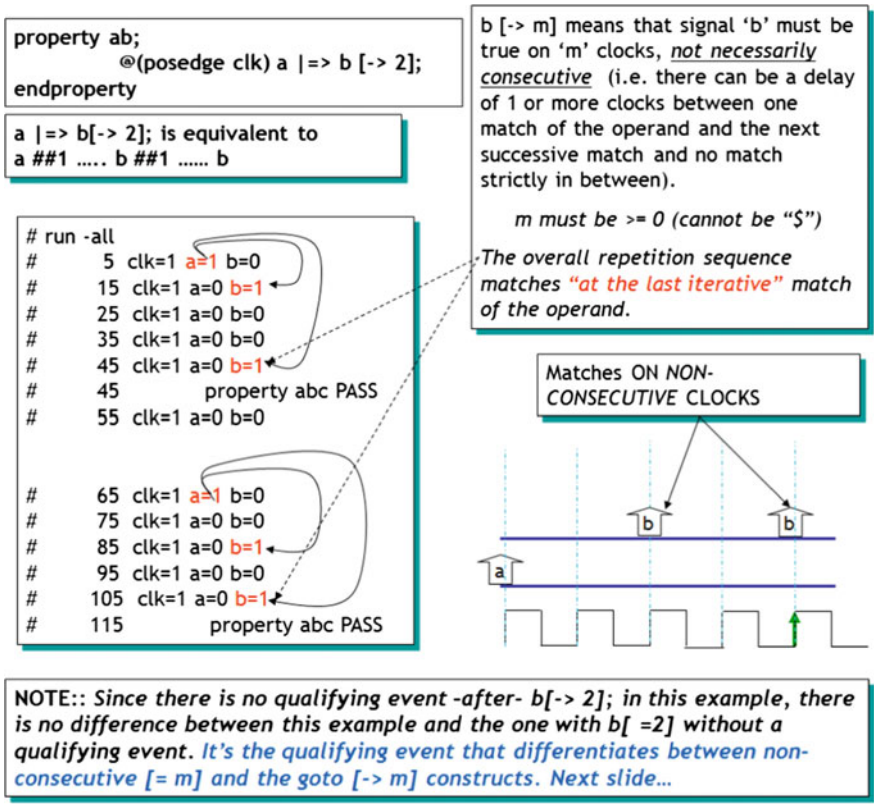


Fig. 6.23 GoTo non-consecutive repetition—basics

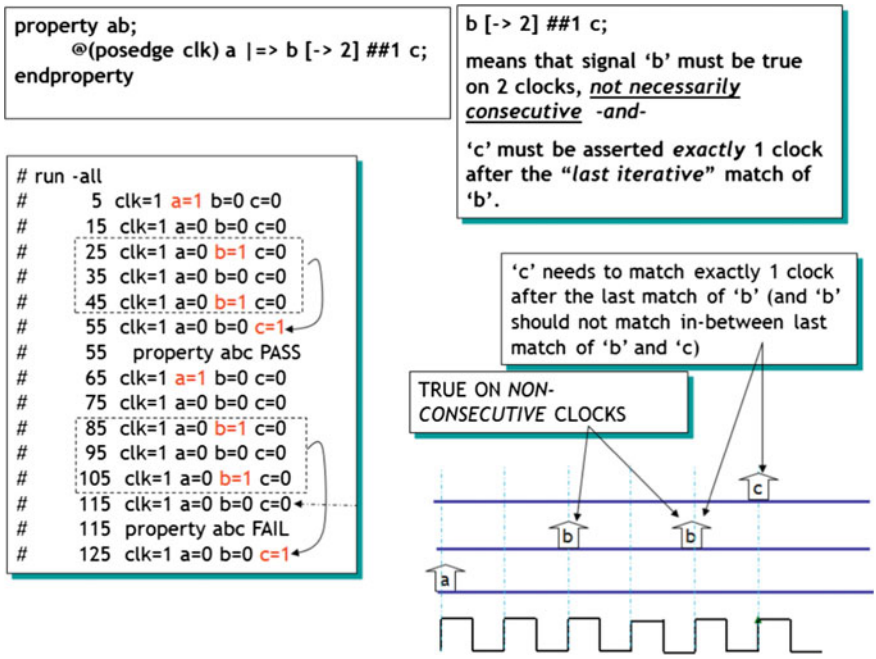


Fig. 6.24 Non-consecutive repetition—example

In Fig. 6.24, we have the so-called qualifying event '##1 c'. The property says that on finding a=1 at posedge clk, b must be true 2 times (1 clock after a=1) non-consecutively and 'c' must occur *exactly* 1 clock after the last occurrence of 'b'. In contrast, with "b[=2] ##1 c", 'c' could occur *any time* after 1 clock after the last occurrence of 'c'. That is the difference between [=m] and [->m].

The simulation log in Fig. 6.24 shows a PASS and a FAIL scenario. PASS scenario is quite clear. At time 5, a==1, then two non-consecutive 'b' occur and then *exactly* 1 clock after the last 'b=1', 'c=1' occurs. Hence, the property passes. The FAIL scenario shows that after 2 occurrences of b==1, c==1 does *not* arrive *exactly* 1 clock after the last occurrence of b=1. That is the reason the b[->2] ##1 c check fails.

6.8 Difference Between [=m:n] and [->m:n]

The simulation log in Fig. 6.25 is quite self-explaining. The left and the right side properties are identical except that the LHS uses [=2:5] and RHS uses [->2:5]. The LHS log, i.e. the one for b[=2:5] PASSes while the one for b[->2:5] fails because according to the semantics of "b[->2:5] ##1 c', 'c' must arrive exactly 1 clock after the last occurrence of 'b'.

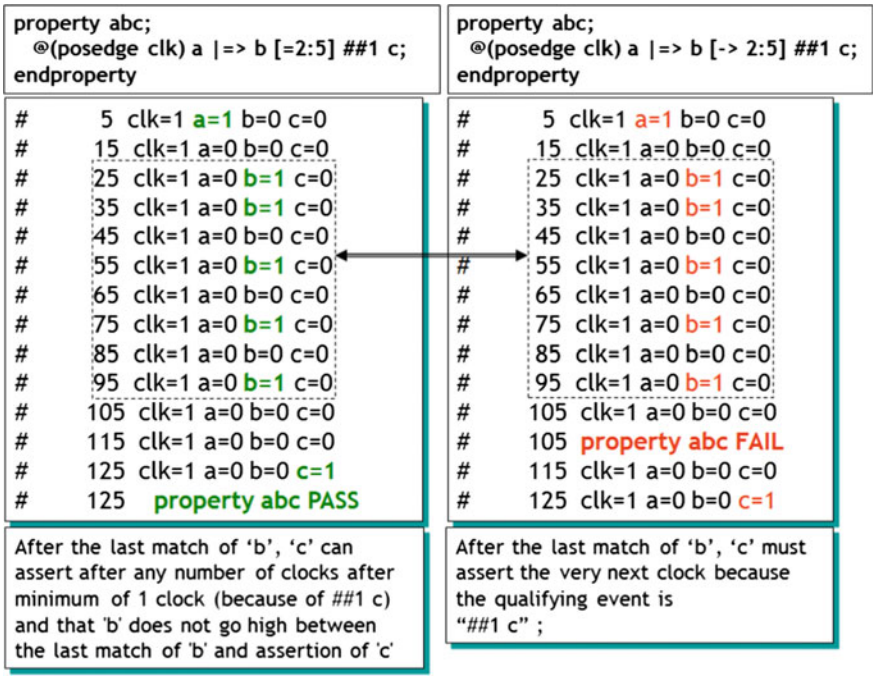


Fig. 6.25 Difference between [=m:n] and [->m:n]

Now here is a very important point. Note that 'c' is expected to come in 1 clk after the last occurrence of b = 1 because of '##1 c'. But what if you have '##2 c' in the property?

b[=m] ##2 C : This means that after 'm' non-consecutive occurrence of 'b', 'c' can occur any time *after* 2 clocks. If 'c=1' arrives before 2 clocks, the property will fail. b[->m]##2 c: This means that after 'm' non-consecutive occurrence of 'b', 'c' must occur *exactly* after 2 clocks. No more no less.

6.8.1 Application: GoTo Repetition—Non-consecutive Operator

The application says that at the rising edge of 'req', after 1 clock (because of non-overlapping operator) 'ack' must occur once and that it must de-assert (go low) exactly 1 clock after its occurrence. If 'ack' is not found de-asserted (low) exactly 1 clock after the assertion of 'ack', the property will fail (Fig. 6.26).

Example: Once an Ethernet Frame starts transmitting, that exactly 16 packets are sent between 'frame_start' and 'frame_end'. 'packet_sent' is asserted every time a packet is sent.

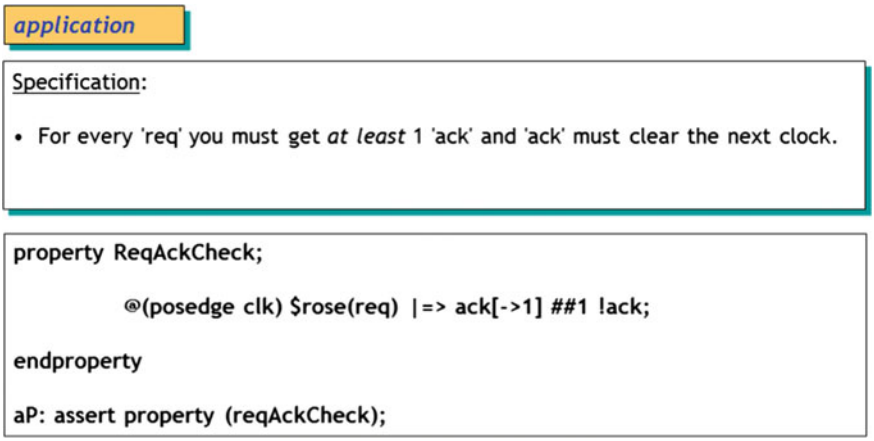


Fig. 6.26 GoTo repetition—non-consecutive operator—application

Solution:
Frame_check:

```
assert property (frame_start) |=>
(!frame_end throughout packet_sent [->16] ##1 !packet_sent) ##1 frame_end;
```

'throughout' operator is explained in Sect. 6.9. The property reads as: frame_end must remain deasserted 'throughout' the sequence where non-consecutive 16 packets are sent and packet_sent is deasserted. Once frame_end remains deasserted 'throughout' this sequence that it is found asserted 1 clock after the sequence is over.

6.9 Sig1 *throughout* Seq1

The 'throughout' operator makes it that much easier to test for condition (signal or expression) to be true throughout a sequence. Note that the LHS of 'throughout' operator can only be a signal or an expression, but it cannot be a sequence (or subsequence). The RHS of 'throughout' operator can be a sequence. So, what if you want a sequence on the LHS as well? That is accomplished with the 'within' operator, discussed right after 'throughout' operator (Fig. 6.27).

Let us examine the application in Fig. 6.28 which will help us understand the throughout operator.

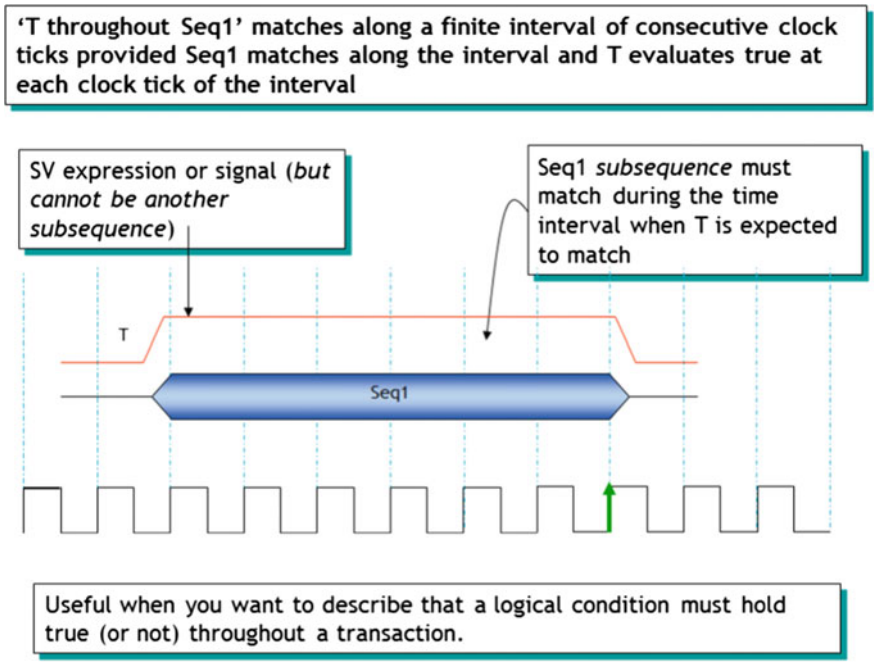


Fig. 6.27 Sig1 *throughout* seq1

6.9.1 Application: Sig1 throughout Seq1

In Fig. 6.28 the antecedent in property pbrule1 requires bMode (burst Mode) to fall. Once that is true, it requires checkbMode to execute.

checkbMode makes sure that the bMode stays low 'throughout' the data_transfer sequence. Note here that we are, in a sense, making sure that the antecedent remains true through the checkbMode sequence. If bMode goes high before data_transfer is over, the assertion will fail. The data_transfer sequence requires both dack_ and oe_ to be asserted (active low) and to remain asserted for 4 consecutive cycles. Throughout the data_transfer, burst mode (bMode) should remain low.

There are two simulation logs presented in Fig. 6.29. Both are for FAIL cases! FAIL cases are more interesting than the PASS cases, in this example! The first simulation log (left hand side) shows $fell(bMode) at time 20. Two clocks later at time 40, oe_=0 and dack_=0 are detected. So far so good. oe_ and dack_ retain their state for 3 clocks That's good too. But in the 4th cycle (time 70), bMode goes high. That's a violation because bMode is supposed to stay low throughout the data-transfer sequence, which is 4 clocks long.

The second simulation log (Right hand side) also follows the same sequence as above but after 3 consecutive clocks that the oe_ and dack_ remain low, dack_ goes

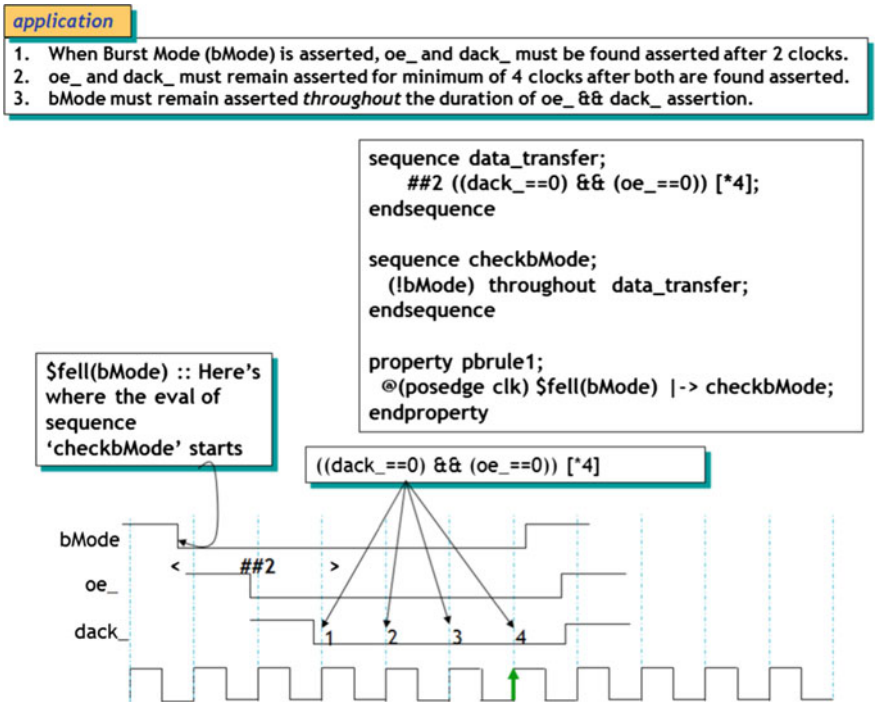


Fig. 6.28 Sig1 *throughout* Seq1—application

high at time 160. That is a violation because data_transfer (oe_=0 and dack_=0) is supposed to stay low for 4 consecutive cycles.

This also points to a couple of other important points

1. Both sides of the *throughout* operator must meet their requirements. In other words, if either the LHS or the RHS of the throughout sequence fails, the assertion will fail. Many folks assume that since bMode is being checked to see that it stays low (in this case), only if bMode fails that the assertion will fail. Not true as we see from the two failure logs.
2. Important Point: In order to make it easier for the reader to understand this burst mode application, I broke it down into 2 distinct subsequences. But what if someone just gave you the timing diagram and asked you to write assertions for it?

Break down any complex assertion requirement into smaller chunks. This is probably the most important advice I can part to the reader. If you look at the entire AC protocol (the timing diagram) as one monolithic sequence, you will indeed make mistakes and spend more time debugging your own assertion then debugging the design under test.

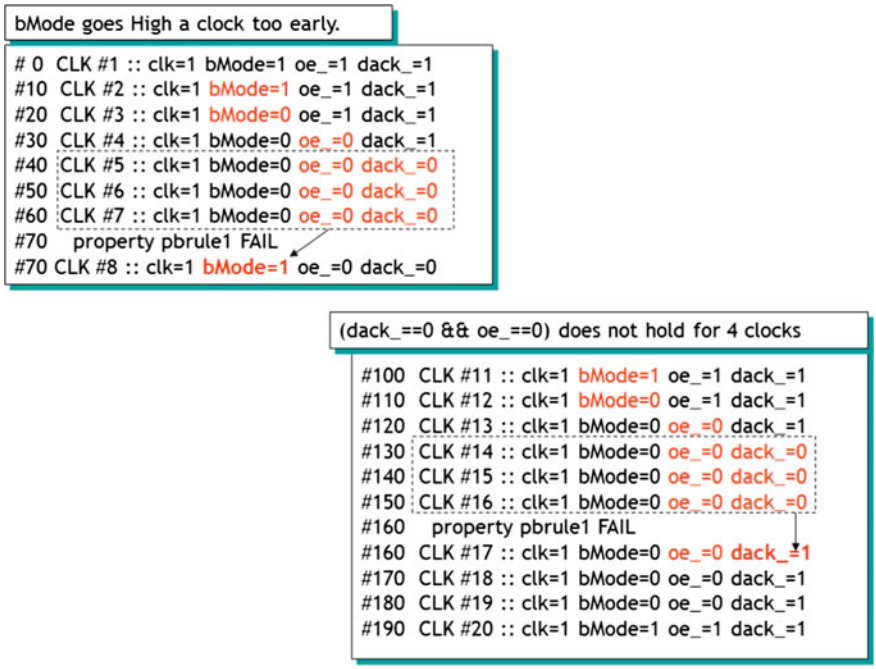


Fig. 6.29 Sig1 throughout seq1—application simulation log

Exercise: How would you model this application using only the consecutive repetition [*m] operator? Please experiment to solidify your concepts of both the *throughout* and the [*m] operators.

One more application, derived from PCI protocol.

Specification:

1. If Frame_ is high then Frame_ must be low the very next cycle (1 clock pulse) and remain low until the next strictly subsequent cycle in which IRDY_ is high.
2. IRDY_ can be high only if Frame_ was high and that IRDY_ was not high in any of the intervening cycles.

Solution:

1. Frame_to_IRDY: **assert property** (

```
Frame_ |=>!Frame_ throughout IRDY_ [->1]
);
```

2. IRDY_to_Frame_: **assert property** (

```
IRDY_ |=>!IRDY_ throughout Frame_ [->1]
);
```

Please study the solution and you will see the symmetry between Frame_ and IRDY_ functionality.

6.10 Seq1 *within* Seq2

Analogous to 'throughout', the 'within' operator sees if one sequence is contained within or of the same length as another sequence. Note that the 'throughout' operator allowed only a signal or an expression on the LHS of the operator. 'within' operator allows a sequence on both the LHS and RHS of the operator.

The property 'within' ends when the larger of the two sequences end, as shown in Fig. 6.30.

Let us understand 'within' operator with the application in Fig. 6.31.

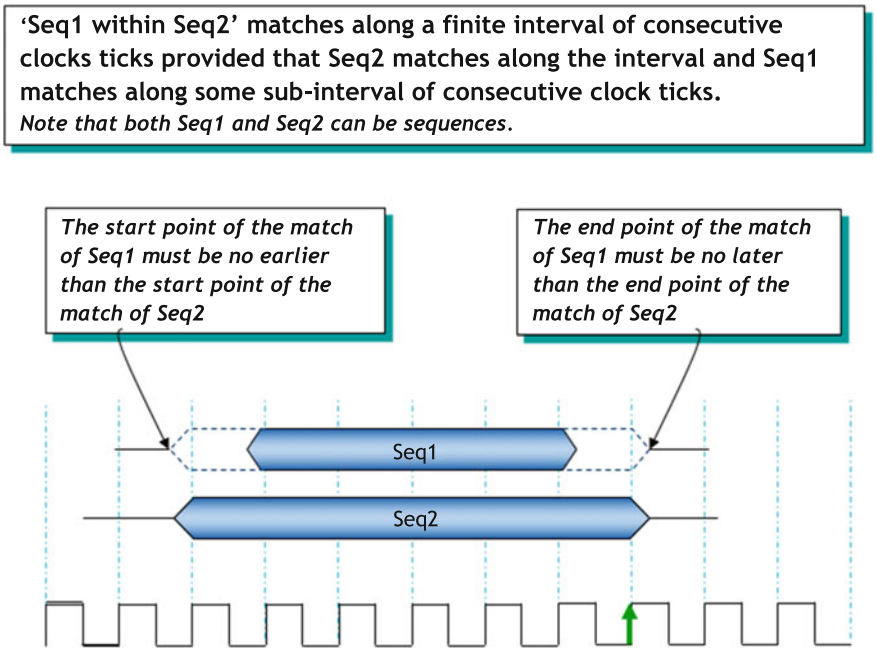


Fig. 6.30 Seq1 within seq2

6.10.1 *Application: Seq1* within *Seq2*

In Fig. 6.31, we again tackle the nasty protocol of burst mode! When burst mode is asserted, the master transmits (smtrx) must remain asserted for 9 clocks and the Target Ack (tack) must remain asserted for 7 clocks and that the 'tack' sequence occurs within the 'smtrx' sequence. This makes sense because from the protocol point of view, the target responds only after the master starts the request and the master completes the transaction after target is done.

In Fig. 6.31, the assertion of bMode ($fell(bMode)) implies that 'stack' is valid 'within' 'smtrx'. Now, carefully see the implication property "@ (posedge clk) $fell(bMode) |=> stack **within** smtrx;"

LHS and RHS sequences start executing once the consequence fires. 'stack' will evaluate to see if it's condition remains true and 'smtrx' starts its own evaluation. At the same time, the 'within' operator continually makes sure that 'stack' is

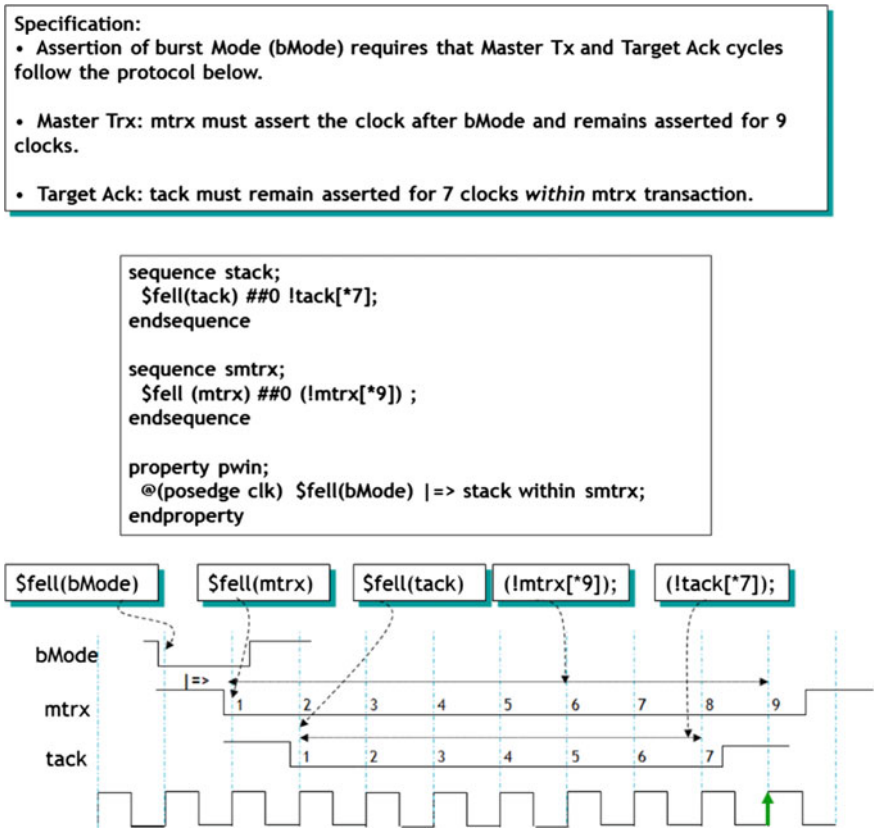


Fig. 6.31 Seq1 *within* seq2—application

contained with 'smtrx'. The annotations in Fig. 6.31 show how the property handles different parts of the protocol. Simulation logs are presented in Fig. 6.32.

6.10.2 'within' Operator PASS CASES

On the left hand side of Fig. 6.32, bMode is '1' at time 0 (not shown) and at time 10, it goes to '0'. That satisfies $fell(bMode). After that the consequent starts execution. Both 'stack' and 'smtrx' sequences start executing. As shown in the left side simulation log, 'mtrx' falls and stays low for 9 clocks, as required. 'tack' falls after 'mtrx' falls, stays low for 7 clocks and goes high the same time when 'mtrx' goes high (i.e. both sequences end at the same time). In other words, 'tack' is contained within 'mtrx'. This satisfies the 'within' operator requirement and the property passes. Note that the operator 'within' can have either sequences start or end at the same time. Similarly, the right side log shows that both sequences start at the same time and 'tack' is contained within 'mtrx' and the property passes. Now let us look at fail cases.

Fig. 6.32 *within* operator—simulation log example—PASS cases

6.10.3 'within' Operator: FAIL CASES

The simulation logs show different cases of failure. In the top log of Fig. 6.33, 'tack' is indeed contained within 'mtrx', but 'tack' does not remain asserted for required 7 clocks and the property fails.

In the bottom log, again 'tack' is contained within 'mtrx' but this time around, 'mtrx' is asserted 1 clock too less.

The last log shows both 'tack' and 'mtrx' asserted for their required clks but 'tack' starts one clk before the falling edge of 'mtrx', thus violating the 'within' semantics. Sequences on either side of 'within' can start at the same time or end at the same time but the sequence that is to be contained within the larger sequence cannot start earlier or end later than the larger sequence.

Another important point to note from these simulation logs is that the *property ends when the larger of the two sequences end*. In our case, the property does not end as soon as there is a violation on 'stack'. It waits for 'smtrx' to end to make a judgment call on pass/fail of the property 'pwin'.

6.11 Seq1 *and* Seq2

As the name suggests, 'and' operator expects both the LHS and RHS side of the operator 'and' to evaluate to true. It does not matter which sequence ends first as long as both sequences meet their requirements. The property ends when the longer of the two sequences ends. *But note that both the sequences must start at the same time.*

The 'and' operator is very useful, when you want to make sure that certain concurrent operations in your design, start at the same time and that they both complete/match satisfactorily. As an example, in the processor world, when a Read is issued to L2 cache, L2 will start a tag match and issue a DRAM Read at the same time, in anticipation that the tag may not match. If there is a match, it will abort the DRAM Read. So, one sequence is to start tag compare while other is to start a DRAM Read (ending in DRAM Read Complete or Abort). The DRAM Read sequence is designed such that it will abort as soon as there is a tag match. This way we have made sure that both sequences start at the same time and that both end. Let us look at the following cases to clearly understand 'and' semantics. The figures are self-explaining with annotation within the figures (Figs. 6.34, 6.35 and 6.36).

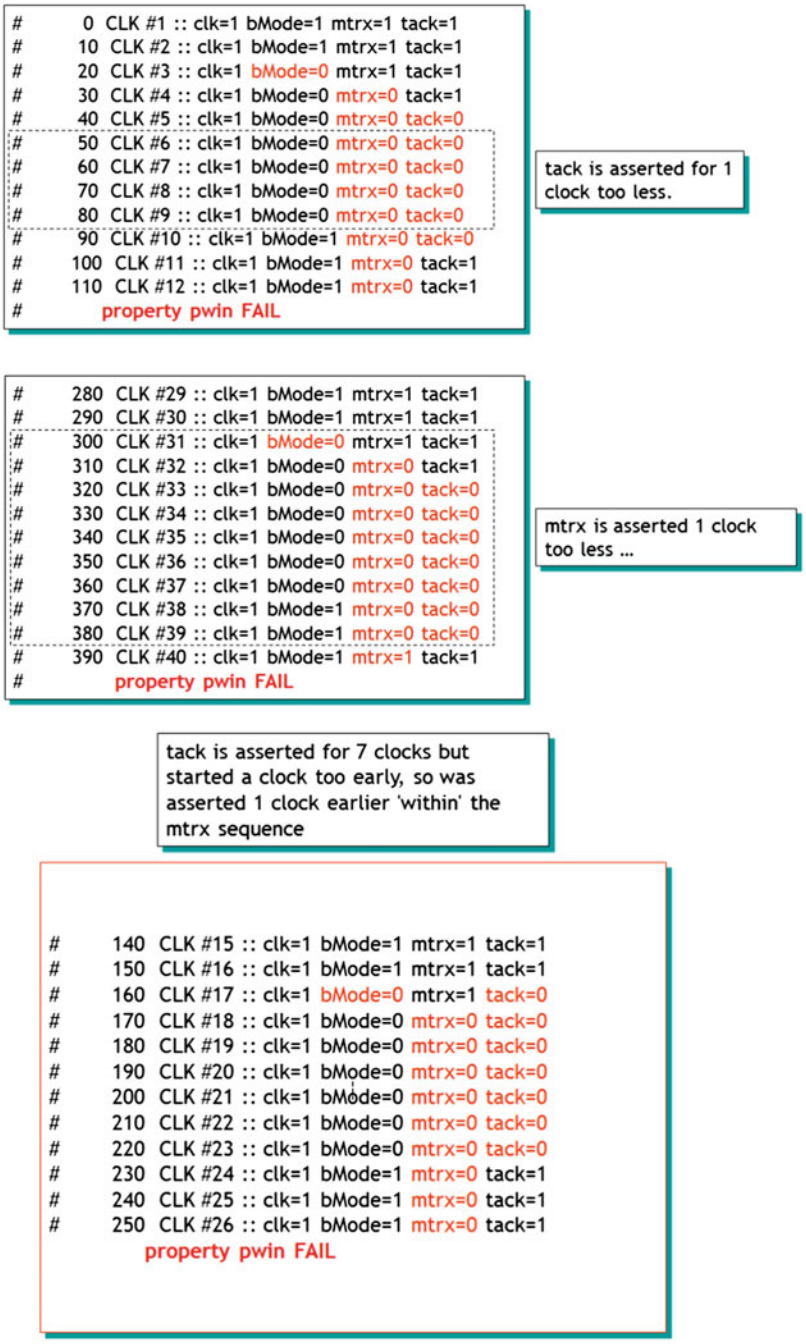


Fig. 6.33 *within* operator—simulation log example—FAIL cases

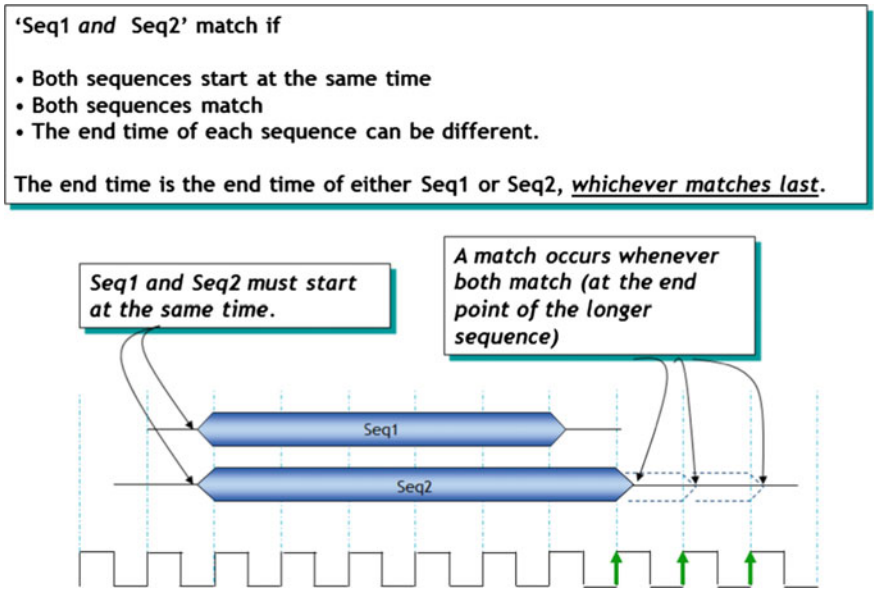


Fig. 6.34 Seq1 *and* seq2—basics

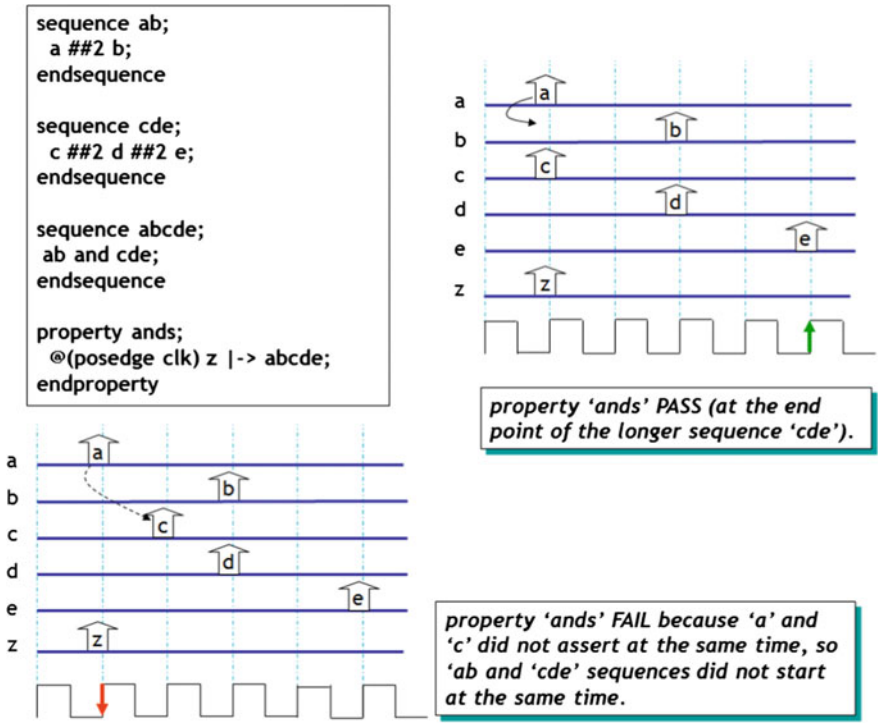


Fig. 6.35 *and* operator—application

6.11.1 Application: 'and' Operator

In Fig. 6.37, we 'and' two expressions in a property. In other words, as noted before, an 'and' operator allows a signal, an expression or a sequence on both the LHS and RHS of the operator. The simulation log is annotated with pass/fail indication (Fig. 6.37).

6.12 Seq1 '*or*' Seq2

'Or' of two sequences means that when either of the two sequences match its requirements that the property will pass. Please refer to Fig. 6.38 and examples that follow to get a better understanding.

The feature to note with 'or' is that as soon as either of the LHS or RHS sequence meets its requirements that the property will end. This is in contrast to 'and' where only after the longest sequence ends that the property is evaluated.

Note also that if the shorter of the two sides fails, the sequence will continue to look for a match on the longer sequence. Following examples make this clear.

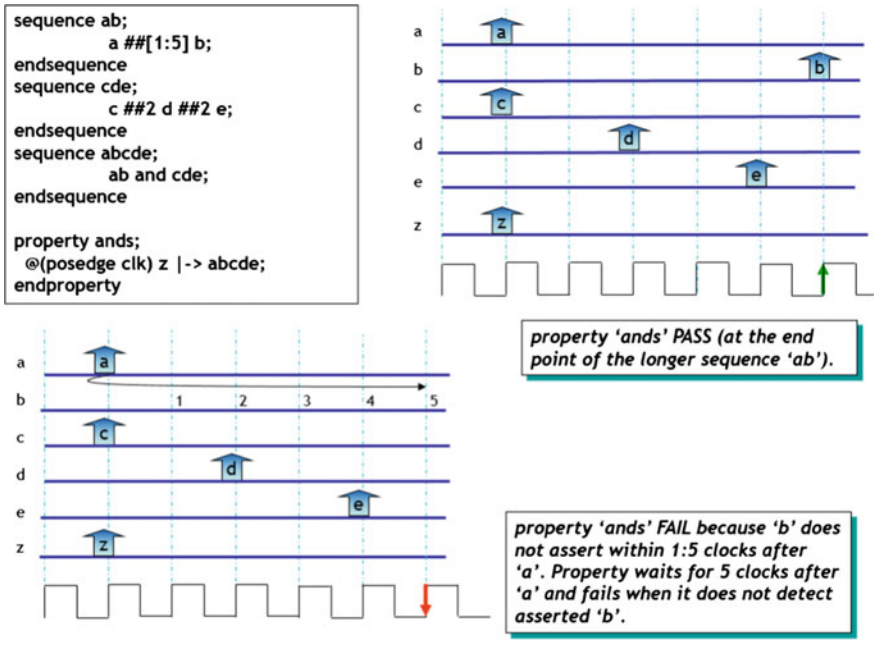


Fig. 6.36 *and* operator—application-II

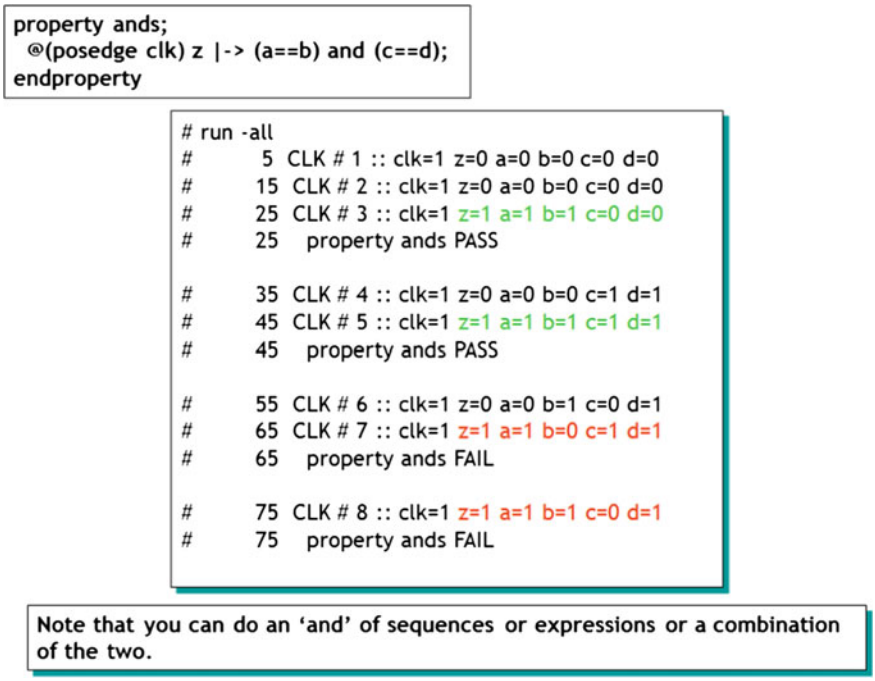


Fig. 6.37 *and* of expressions

6.12.1 Application: or Operator

A simple property is presented in Fig. 6.39. Different cases of passing of the property are shown. On the top right of the figure, both 'ab' and 'cde' sequences start at the same time. Since this is an 'or', as soon as 'ab' completes, the property completes and passes. In other words, the property does not wait for 'cde' to complete anymore.

On the bottom left corner, we see that 'ab' sequence fails. However, since this is an 'or' the property continues to look for 'cde' to be true. Well, 'cde' does turn out to be true and the property passes (Figs. 6.40, 6.41 and 6.42).

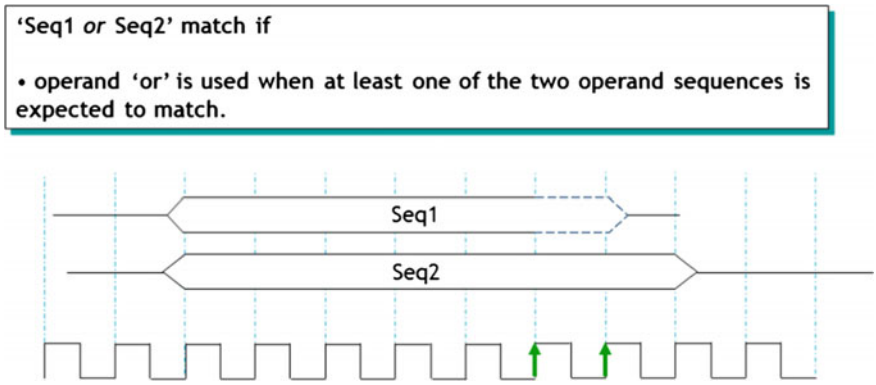


Fig. 6.38 Seq1 or seq2—basics

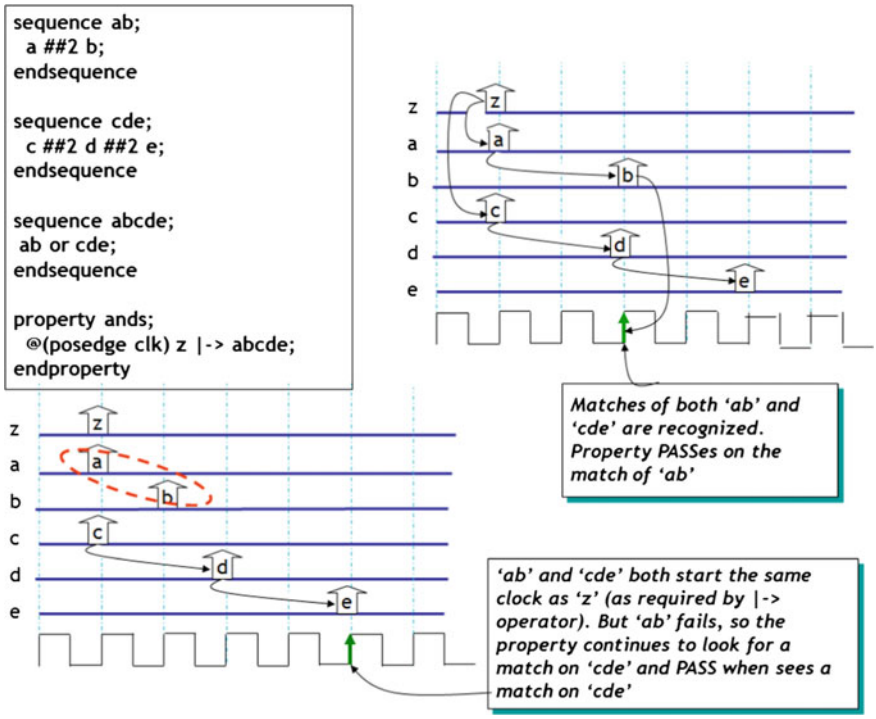
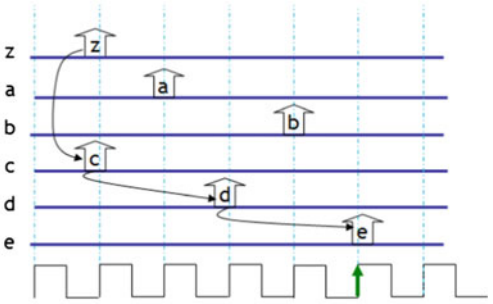


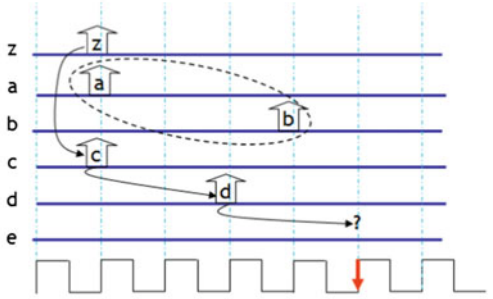
Fig. 6.39 *or* operator—application

```
sequence ab;
  a ##2 b;
endsequence

sequence cde;
  c ##2 d ##2 e;
endsequence

sequence abcde;
 ab or cde;
endsequence

property ands;
  @(posedge clk) z |-> abcde;
endproperty
```



Here 'a' is asserted 1 clock later and 'ab' does satisfy it's requirement, but 'a' was not asserted the same time as 'z' (as required by overlap implication). however, 'c' was indeed asserted when 'z' was asserted, the property is looking for 'cde' to match. Since 'cde' does match, the property passes at the end of 'cde' (and not at the end of 'ab').

```
sequence ab;
  a ##2 b;
endsequence

sequence cde;
  c ##2 d ##2 e;
endsequence

sequence abcde;
 ab or cde;
endsequence

property ands;
  @(posedge clk) z |-> abcde;
endproperty
```



Here, 'ab' does not match; but property keeps looking to see if 'cde' matches. But when 'e' does not follow 2 clocks after d, 'cde' also fails and the property FAILs at that time.

Fig. 6.40 or operator—application II

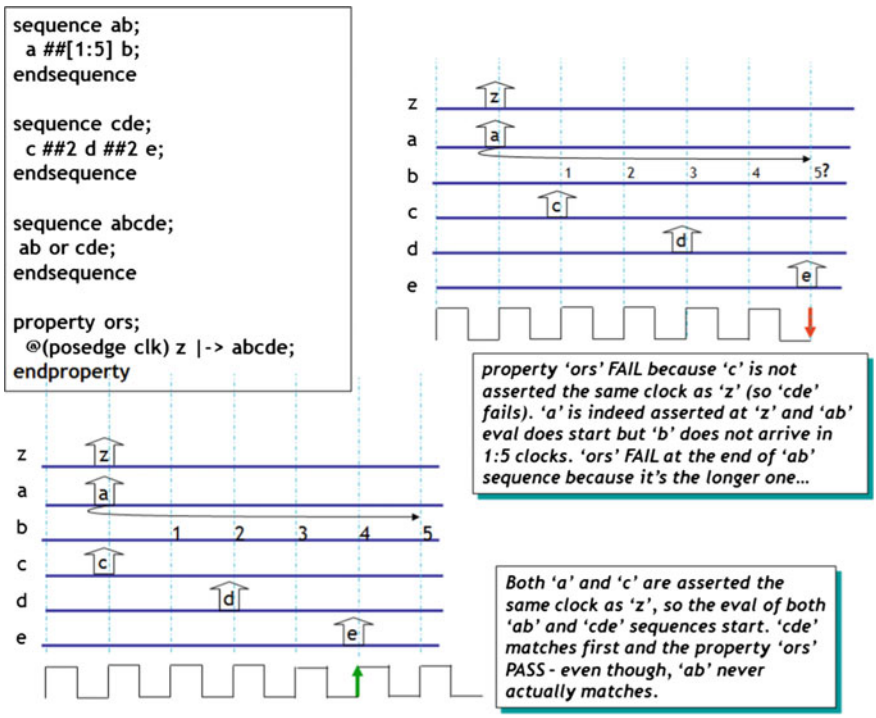


Fig. 6.41 or operator—application III

6.13 Seq1 '*intersect*' Seq2

So, with 'throughout', 'within', 'and' and 'or' operators who needs another operator that also seem to verify that sequences match (Fig. 6.43)?

'Throughout' or 'within' or 'and' or 'or 'does *not* make sure that both the LHS and RHS sequences of the operator are exactly the same. They can be of the same length but the operators do not care as long as the signal/expression or sequence meets their requirements. That's where 'intersection' comes into picture. It makes sure that the two sequences indeed start at the same time *and* end at the same time and satisfy their requirements. In other words, they intersect.

As you can see, the difference between 'and' and 'intersect' is that 'intersect' requires both sequences to be of the same length and that they both start at the same time and end at the same time, while 'and' can have the two sequences of different lengths. I have shown that difference with timing diagrams further down the chapter. But first, some simple examples to understand 'intersect' better.

```
property abcde;
  @(posedge clk) z |-> (a==b) or (c==d);
endproperty
```

```
#       5  CLK # 1 :: clk=1 z=0 a=0 b=0 c=0 d=0
#      15  CLK # 2 :: clk=1 z=1 a=1 b=1 c=0 d=0
#      15    property abcde PASS

#      25  CLK # 3 :: clk=1 z=0 a=0 b=0 c=1 d=1
#      35  CLK # 4 :: clk=1 z=1 a=1 b=0 c=0 d=1
#      35    property abcde FAIL

#      45  CLK # 5 :: clk=1 z=0 a=0 b=1 c=0 d=1
#      55  CLK # 6 :: clk=1 z=1 a=1 b=0 c=1 d=1
#      55    property abcde PASS

#      65  CLK # 7 :: clk=1 z=1 a=1 b=1 c=0 d=1
#      65    property abcde PASS
```

application

Spec : If Write Burst Length is == 2; Write Length can only be 1 or 3 or 7 or 15

```
property BurstLengthRestrict;
   @(posedge clk) disable iff (!rst)
    ( (bLength==2) |->
         (rwlen==1) or (rwlen==3) or (rwlen==7) or (rwlen==15))  ;
  endproperty
  aP: assert property(BurstLengthRestrict);
```

Fig. 6.42 *or* of expressions

6.14 Application: 'intersect' Operator

Figure 6.44 shows two cased of failure with the 'intersect' operator.

Property 'isect' says that if 'z' is sampled true at the posedge clk that sequence 'abcde' should be executed and hold true. I have broken down the required sequence into two subsequences. Sequence 'ab' requires 'a' to be true at posedge clk and then 'b' be true any time from 1 to 5 clks. Sequence 'cde' is a fixed temporal domain sequence which requires c to be true at posedge clk, then d to be true 2 clocks later and 'e' to be true 2 clocks after 'd'.

Top Right timing diagram in Fig. 6.44 shows that both 'ab' and 'cde' meet their requirements but the property fails because they both do not end at the same time (even though they start at the same time). Similarly, the bottom left timing diagram shows that both 'cde' and 'ab' meet their requirements but don't end at the same time and hence the assertion fails.

In Fig. 6.45 we show a PASS case of the same property (repeated here for the sake of convenience). Both 'ab' and 'cde' meet their requirements and end at the same time. Hence the assertion passes.

Now let's look at the example in Fig. 6.46. This one does not use a range operator in sequence 'ab' (as in the above example). It is obvious that without the

'Seq1 *intersect* Seq2' match if

- Both sequences start at the same time
- Both sequences must match
- *The lengths of the two matches of the operand sequences must be the same.*

The end time is when both sequences match and end at the same time.

The main difference between 'and and 'intersect' is the requirement on the length of the two sequences. For 'and' each sequence can be of any length. For 'intersect' they must be of the same length.

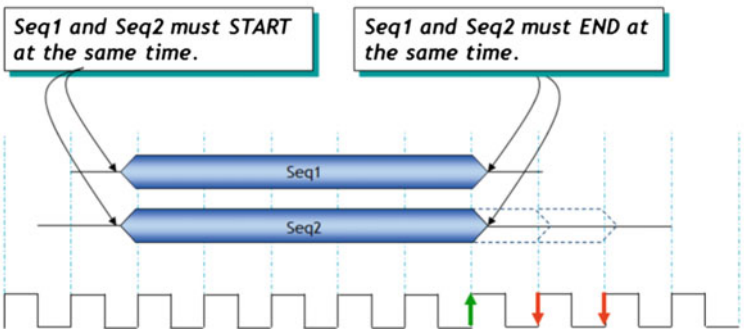


Fig. 6.43 Seq1 intersect seq2

range operator that the two sequences with fixed lengths and ending at different times, the intersect property will never pass. *Hence, it makes sense to use subsequences with a range while using an 'intersect' operator.*

*6.14.1 Application: intersect Operator (*Interesting Application*)*

OK, I admit this property could have been written much simpler, as (Fig. 6.47)

```
@ (posedge clk) $rose(Retry) |- > Retry ##[1:4] $rose(dataRead);
```

So, why are we making it complicated? I just want to highlight an interesting way to use 'intersect'.

When $rose(Retry) is true, the consequent executes. The consequent uses 'intersect' between `true[*1:4] and (Retry ##[1:$] $rose(dataRead). The LHS of 'intersect' says that it will be True for consecutive 4 cycles. The RHS says that $rose(dataRead) should occur anytime (##[1:$]) after 'Retry' has been asserted. Now, recall that 'intersect' requires both the LHS and RHS to be of 'same' length.

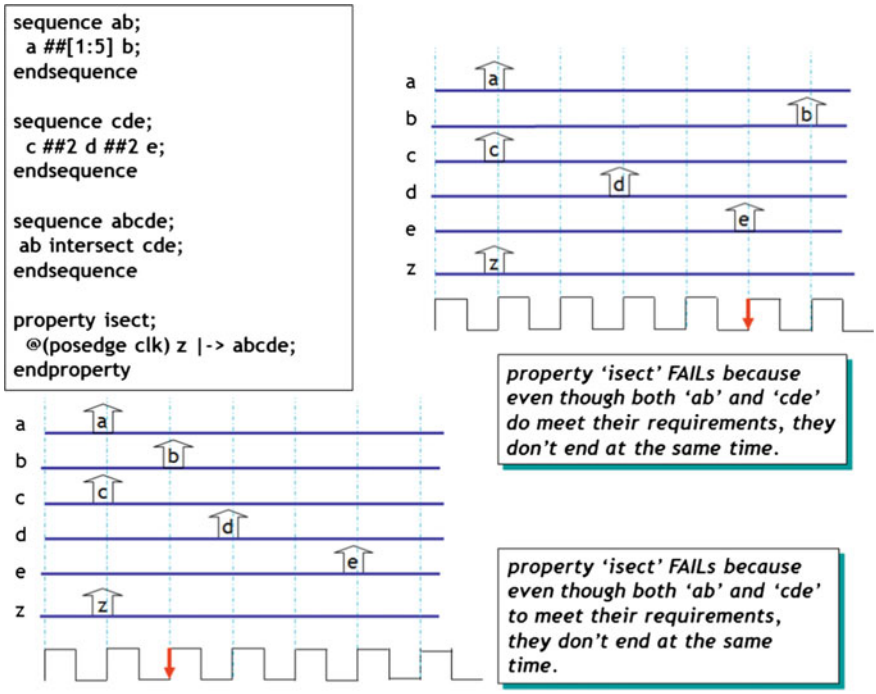


Fig. 6.44 Seq1 'intersect' seq2—application

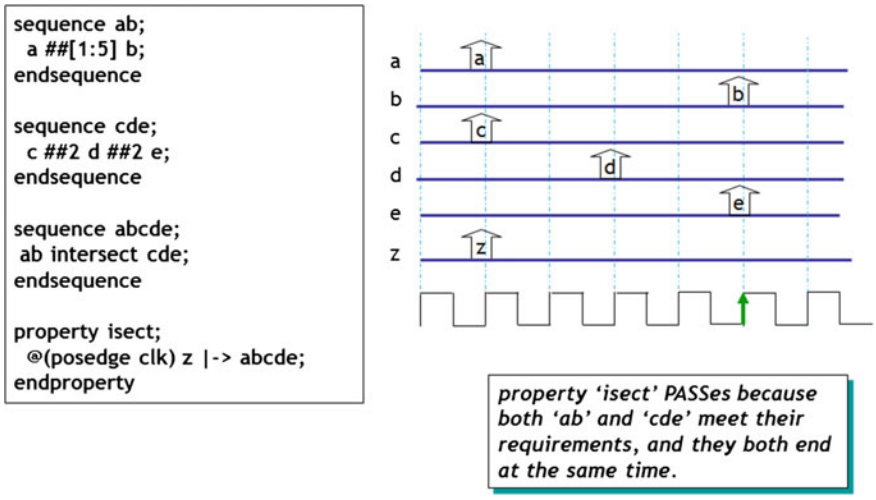


Fig. 6.45 Seq1 intersect seq2—application II

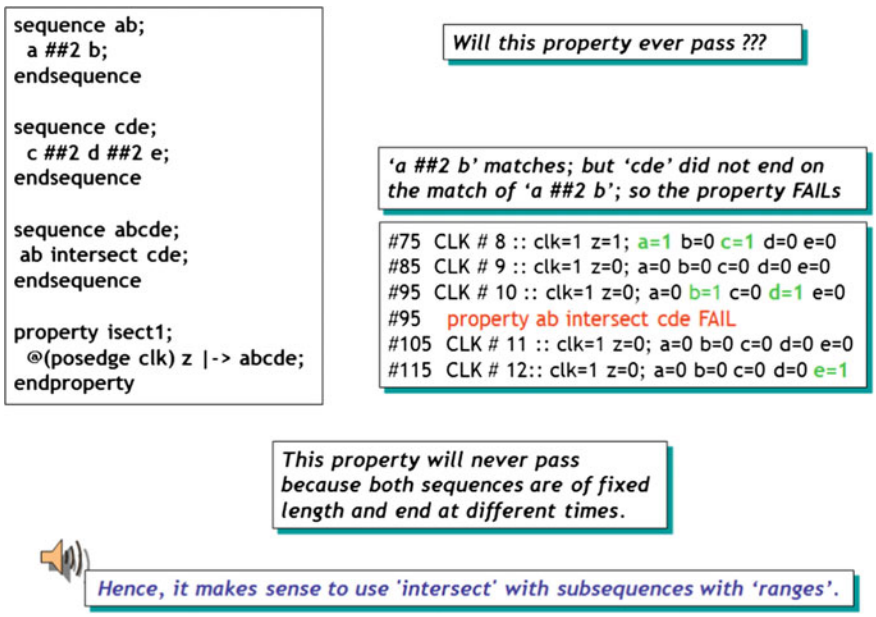


Fig. 6.46 *intersect* makes sense with subsequences with ranges

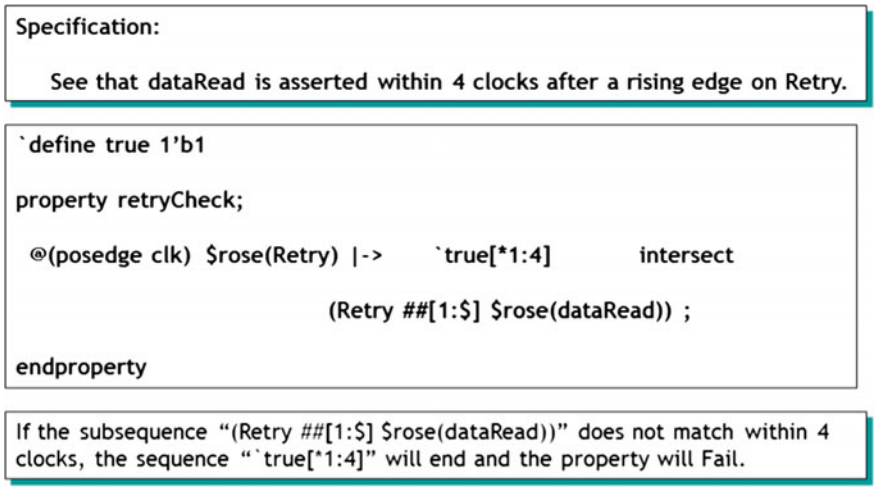


Fig. 6.47 intersect operator: interesting application

If $rose(dataRead) does *not* arrive in 4 cycles, the RHS will continue to execute beyond 4 clocks. But since `true[*1:4] has now completed and since 'intersect' requires both sides to complete at the same time, the assertion will fail.

If $rose(dataRead) does occur within 4 clks, the property will PASS. Why? Let us say $rose(dataRead) occurs on the third clock. That sequence will end and at the same time `true[*1:4] will end as well, since it is `true anytime within the four clocks. This satisfies the requirements of 'intersect' and the property passes.

So, what's the practical use of such a property. Any time you want to contain a large sequence to occur within a certain period, it is very easy to use the above technique. A large sequence may have many time domains and temporal complexities, but with the above method, you can simply superimpose `true construct with 'intersect' to achieve the desired result.

One more example.

Specification:
See that once the CPU starts a cycle (for a certain instruction), that two READs are issued in order and one WRITE is issued. Condition is that the WRITE must complete at the end of the second READ. In other words, the WRITE completion and the second READ completion must happen simultaneously.

Solution:

```
property checkRW;
@(posedge clk)

$rose(CPU_Start) |=>WRITE_complete [->1] intersect READ_complete [-> 2];

endproperty
```

This property will start consequent evaluation at $rose(CPU_Start) and check to see that WRITE_complete occurs at least once and the READ_complete occurs at least twice. And that the completion of the WRITE *intersects* (i.e. ends at the same time) with the completion of second READ.

Continuing with above example, here's another variation of it.

Specification:

Between CPU_Start and CPU_End commands, there must be at least 3 READs and 2 WRITEs.

Solution:

```
property checkNumRW;
@(posedge clk)

  $rose(CPU_start) |-> READ_complete[=3] intersect WRITE_complete[=2]
intersect $rose(CPU_End);

endproperty
```

The example exhibits multiple intersects. 3 READs must intersect with 2 WRITEs which intersect with the end of CPU command (CPU_End).

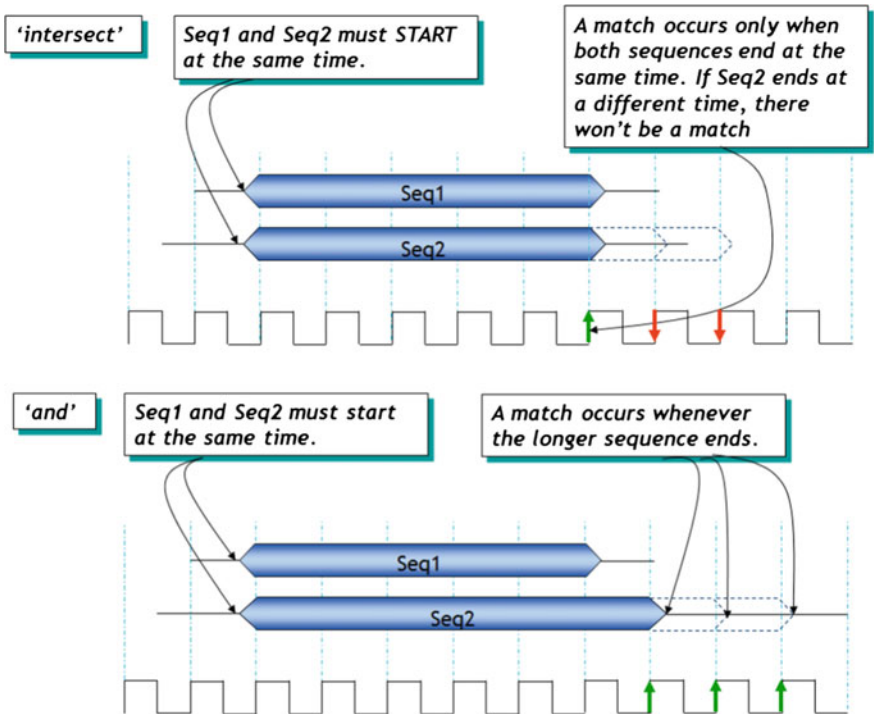


Fig. 6.48 *and* versus *intersect*—what's the difference

6.14.2 'intersect' *and* 'and' *:: What's the Difference?*

See Fig. 6.48.

6.15 first_match

first_match(Seq)
- matches only the first of possibly multiple matches of the eval of Seq.
- useful for detecting the first occurrence in a *delay range*.

6.15.1 Application: first_match

In Fig. 6.49, property 'fms' says that on the first_match of 'bcORef', 'a' should rise. As you notice, the sequence 'bcORef' has many matches because of the range operator and an 'or'. As soon as the *first* match of 'bcORef' is noticed, the property looks for $rose(a). In the top log, that is the case and the property passes. ***Note that rest of the matches are now ignored***. In the bottom log, $rose(a) does not occur and the property fails—even though (and as noted in the log, of Fig. 6.49), (b &&c) is indeed true 3 clocks after d==1 and even though the fact that this is an 'or', the first_match looks for the very first match of either of the sequences in 'bcORef' and looks for $rose(a) right after that. So, as soon as (e &&f) is true, the property looks for $rose(a)—which does not occur and the property fails.

Figure 6.50 further explains 'first_match'. Annotations explain what's going on.

Figure 6.51 application clarifies $first_match further. This is the classic PCI bus protocol application. As the figure shows, the first time frame_ && irdy_ are high (de-asserted) that the bus goes into IDLE state. Note that once frame_ and irdy_ are de-asserted the bus would remain in IDLE state for a long time. But we want the very first time that the bus transaction ends (indicated by frame_ && irdy_ high) that the bus goes into IDLE state. We do not want to evaluate any further busIdle conditions.

So, what would happen if you removed 'first_match' from the above property? The property will continue to look for state==busidle every clock that frame_ && irdy_ is high. Those will be totally redundant checks.

Note that in all the examples above, we have used first_match() in the antecedent. Why? Because *the consequent (RHS) of a property behaves exactly like first_match by definition*. The consequent is not evaluated once its first match is found (without the use of first_match). But the antecedent will keep firing every time there is a match of its expression.

Note the following 'cover' property. That further explains use of first_match.
Problem Statement:
abcProp: **cover property** (@ (posedge clk) a ##[1:4] b ##1 c);

Let us say, 'a' is true at time 10 and 'b' is true at time 20, thus meeting its requirement and then 'c' is true at time 30. The entire sequence matches and will be considered 'covered' (exercised). But if 'b' remains true at time 20, 30, 40 and 'c' remains true at 30, 40, 50, the coverage report will show multiple 'coverage' of this sequence. Something we do not really want. The following will solve the problem.
Solution:

abcProp: **cover property** (@ (posedge clk) first_match (a ##[1:4] ##1 b ##1 c));

In this case, as soon as 'a' is true and 'b' is true the *first time* in ##[1:4] that 'c' is evaluated to be true the next clock, the property is considered covered. No further evaluation of this sequence will take place until 'a' is found asserted again.

In short, there can be many matches of a sequence but you want the evaluation to stop on the very first match that you use 'first_match'.

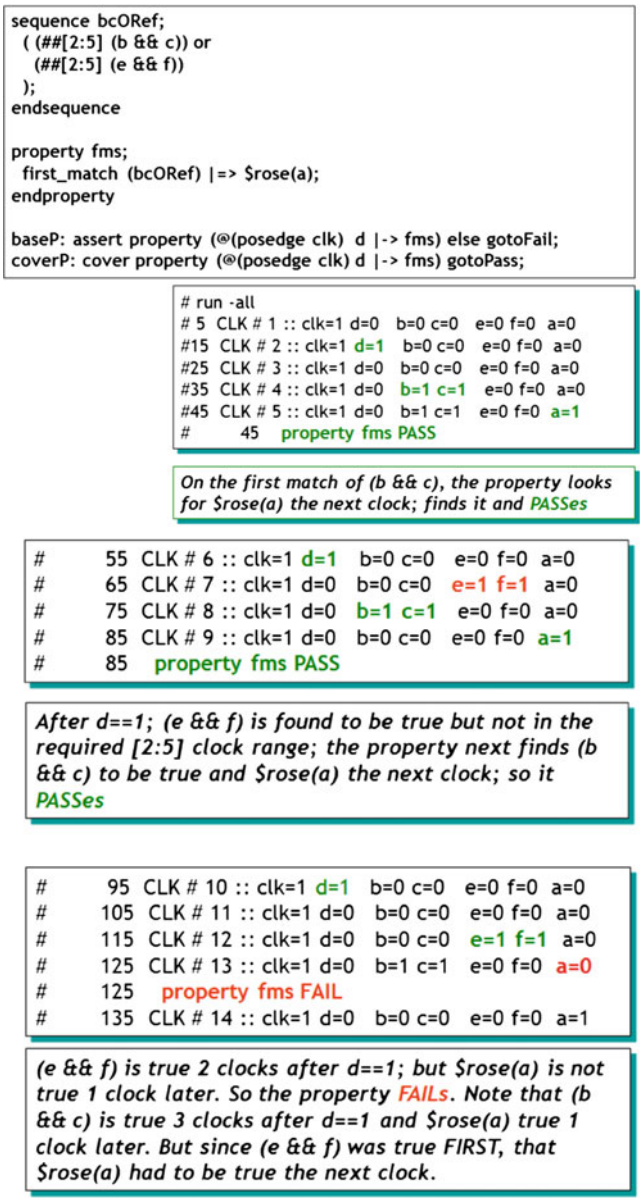


Fig. 6.49 first_match—application

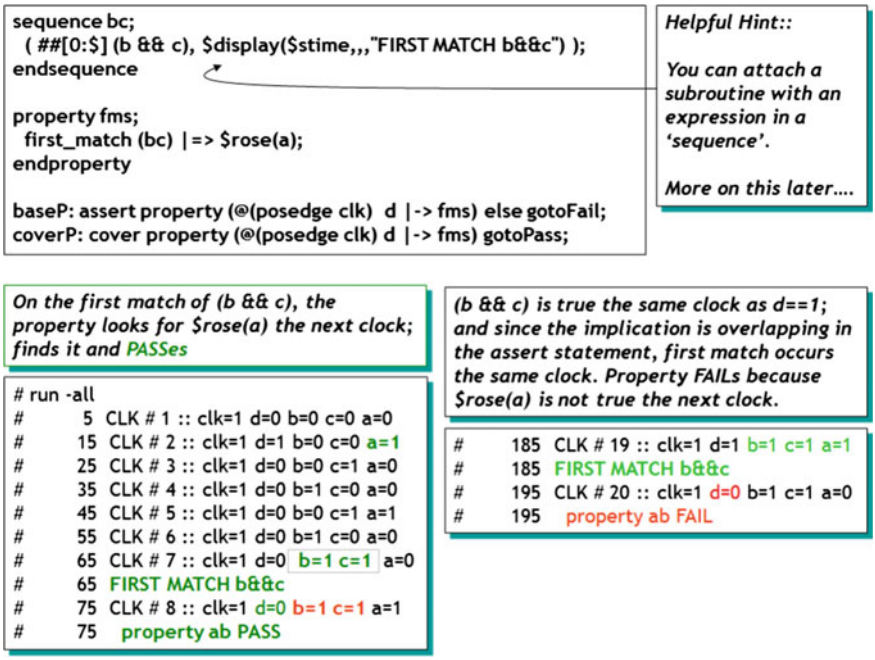


Fig. 6.50 *first_match* application

application

The first time PCI bus goes IDLE, the state machine should transition to busIdle state.

```
sequence busIdleCheck;
  (##[2:$] (frame_ && irdy_));
endsequence

property fms;
    @(posedge clk) first_match (busIdleCheck) |-> (state ==
busIdle);
endproperty
baseP: assert property (fms) ;
```

Fig. 6.51 first_match application

6.16 *not* <property expr>

The 'not' operator seems very benign. However, it could be easily misinterpreted because we are all wired to think positively—correct?

Figure 6.52 shows a use of 'not'. Whenever 'cde' is true the property will fail because of 'not' and pass if 'cde' is not true. Please refer to an example in Fig. 6.53 and then an application thereafter.

6.16.1 Application: not Operator

First, please refer to 'vacuous pass' in the Sect. 14.15 to understand the following application.

Figure 6.53 shows a classic mistake engineers make when using 'not' operator. Without the 'not' in this example if the antecedent "a ##1 b" does not match, the property vacuously passes (*vacuous pass is discussed in* Sect. 14.15) but nothing really happens. The property simply waits for the antecedent to be true so that the consequent can start its execution. However, since the antecedent has a 'not' in front of it, as soon as the property sees that the antecedent does not match it will fail

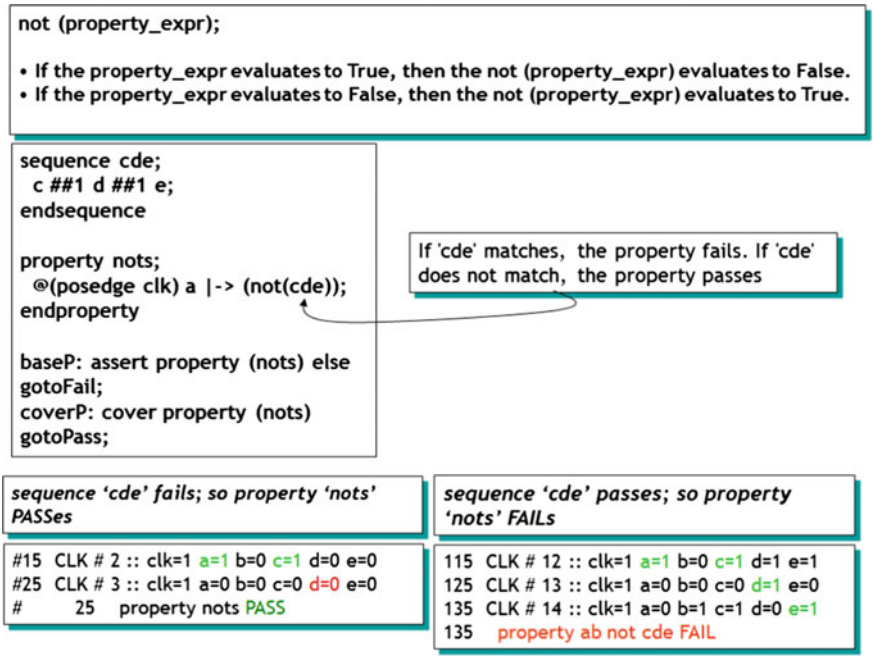


Fig. 6.52 not operator—basics

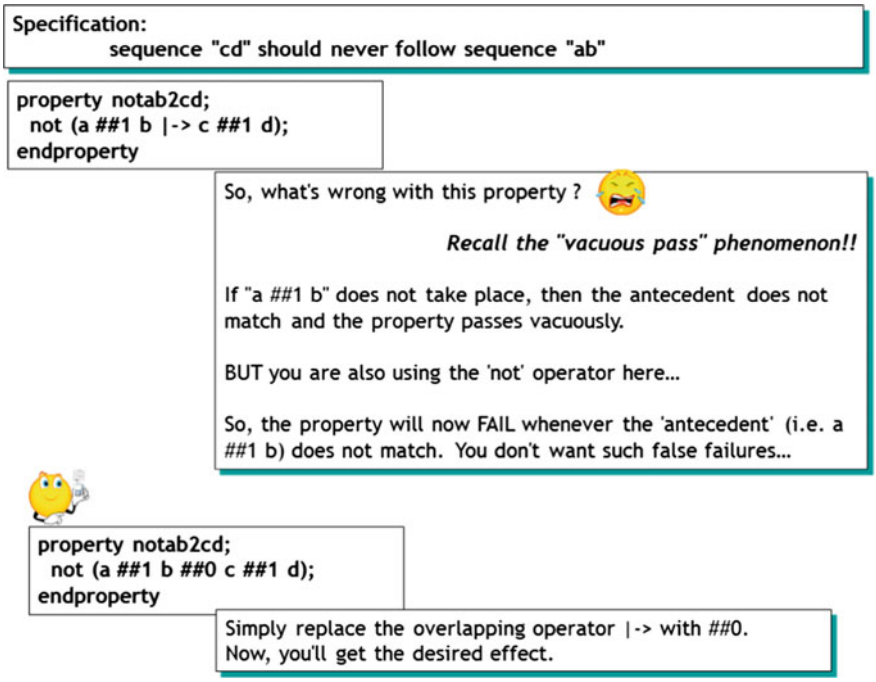


Fig. 6.53 not operator—application

(*not* of vacuous pass). That is indeed detrimental to your design results where many false failures would pop up.

As shown at the bottom of the figure, simply remove the implication operator "|->" and replace it with ##0, which has the same desired effect as the overlapping operator. However, since there is no implication operator, there is no vacuous pass.

Application in Fig. 6.54 is a very useful application. The specification says that once req is asserted (active high) that we must get an ack *before* getting another request. Such a situation occurs in many designs.

Let us examine the assertion. Property strictlyOneAck says that when 'req' is asserted (active high) that !ack[*0:$] remains low until $rose(req). If this matches than the property fails (because of the 'not').

In other words, we are checking to see that ack remains low until the next req, meaning if ack does go high before req arrives that the sequence (!ack[*0:$] ##1 $rose(req)) will fail and the 'not' of it will make it pass. That is the correct behavior since we *do* want an ack before the next req.

Or looking at it conversely (and as shown in the log), if 'ack' does remain low until the next 'req' arrives that the sequence (!ack[*0:$] ##1 $rose(req)) will pass and the 'not' of it will fail. This is correct also, because we do not want ack to remain low until next req arrives. We want 'ack' to arrive before the next 'req' arrives.

```
Specification:
      Once 'req' is asserted that you must get an 'ack' -before- the
next request.
```

```
property strictlyOneAck;
  @(posedge clk) $rose(req) |=> (not (!ack[*0:$] ##1 $rose(req) ) );
endproperty
strictlyOneAckP: assert property (strictlyOneAck)
              else $display($stime,,,"\t Error: strictlyOneAck FAIL");
```

```
KERNEL:         0  clk=1 req=0 ack=0
KERNEL:     10000  clk=1 req=1 ack=0
KERNEL:     20000  clk=1 req=0 ack=0
KERNEL:     30000  clk=1 req=0 ack=0
KERNEL:     40000  clk=1 req=1 ack=0
KERNEL:     40000       Error: strictlyOneAck FAIL
KERNEL:     50000  clk=1 req=0 ack=1
```

Fig. 6.54 *not* operator—application

May seem a bit strange and this property can be written many different ways but this will give you a good understanding of how negative logic can be useful.

Exercise: What is a simpler way to write this property?

6.17 *if* (expression) property_expr1 *else* property_expr2

'if' 'else' constructs are similar to their counterpart in procedural languages and obviously very useful. As the Fig. 6.55 annotates, we are making a decision in consequent based on what happens in the antecedent. The property 'if' states that on 'a' being true, either 'b' or 'c' should occur at least once, *any time* one clock after 'a'. If this antecedent is true, the consequent executes. Consequent expects 'd' to be true if 'b' is true and 'e' to be true if 'b' is false or 'c' is true.

The simulation log in the bottom left of Fig. 6.55 shows that at time 15, 'a==1' and 1 clock later 'b' is true as required. Since 'b' is true, 'd' is true 1 clock later at time 45. Everything works as required and the property passes. In the bottom right simulation log, 'a==1' at time 55 and 'c' goes true at 85. This would require 'e' to be true 1 clock later, but it's not and the property fails. This is just but one way to use if-else and tie in antecedent with consequent.

Based on the analogy of the Fig. 6.55, a practical application is given in Fig. 6.56.

```
if (expression) property_expr1;

OR

if (expression) property_expr1 else property_expr2;
```

```
property ife;
  @(posedge clk) a ##1 (b || c) [->1] |->
  if (b)
    (##1 d)
  else
    (##1 e);
endproperty

baseP: assert property (ife) else gotoFail;
coverP: cover property (ife) gotoPass;
```

The property reads as follows::

@(posedge clk) 'a' followed by at least one 'b' OR 'c';

implies (|->)

that if 'b' is true than 1 clock later 'd' is true else 1 clock later 'e' is true.

```
 5  CLK # 1 :: clk=1 a=0 b=0 c=0 d=0 e=0
15  CLK # 2 :: clk=1 a=1 b=0 c=0 d=0 e=0
25  CLK # 3 :: clk=1 a=0 b=0 c=0 d=0 e=0
35  CLK # 4 :: clk=1 a=0 b=1 c=0 d=0 e=0
45  CLK # 5 :: clk=1 a=0 b=0 c=0 d=1 e=0
45    property PASS
```

```
55  CLK # 6 :: clk=1 a=1 b=0 c=0 d=0 e=0
65  CLK # 7 :: clk=1 a=0 b=0 c=0 d=0 e=0
75  CLK # 8 :: clk=1 a=0 b=0 c=0 d=0 e=0
85  CLK # 9 :: clk=1 a=0 b=0 c=1 d=0 e=0
95  CLK # 10 :: clk=1 a=0 b=0 c=0 d=0 e=0
95    property FAIL
```

Fig. 6.55 *if... else*

Specification :

On a TagCompare,

if there is a TagHit, start mesiCompare
else start an allocRead

application

```
property tagCheck;
  @(posedge clk) (State == TagCompare) ##1 (TagHit || TagMiss) |->

  if (TagHit)
    ##1 (State == mesiCompare)

  else
    ##1 (State == allocRead);

endproperty

baseP: assert property (tagCheck) else gotoFail;
coverP: cover property (tagCheck) gotoPass;
```

Fig. 6.56 if… else—application

6.17.1 Application: if then else

This property is self-explaining. On a TagCompare, if it's a hit, start MESI compare else start a Read Allocation cycle.

6.18 '*iff*' and '*implies*'

p **iff** q is an equivalence operator. This property is true *iff* properties 'p' and 'q' are both true. When 'p' and 'q' are Boolean expressions 'e1' and 'e2', then e1 **iff** e2 is equivalent to e1 **<->** e2.

p **implies** q is an implication property. So, what's the difference between 'implies' and the implication operator '**|->**'? In case of p **|->** q, the evaluation of 'q' starts at the match of 'p'. 'p' is a sequence and 'q' is a property. In case of p **implies** q, both are properties and both 'p' and 'q' start evaluating at the *same* time and the truth results are computed using the logical operator 'implies'. There is no notion of a match of antecedent to trigger the consequent.

For example,

```
x ##2 y |-> a ##2 b;
```

VS.

```
x ##2 y implies a ##2 b;
```

In the case of implication operator "**|->**", evaluation of 'a ##2 b' starts at the match of 'x ##2 y'. In the case of '**implies'**, evaluation of both 'x ##2 y' and 'a ##2 b' start at the same clock tick.

Chapter 7
System Functions and Tasks

Introduction: This chapter discusses in detail the System Functions and Tasks such as $onehot, $onehot0, $isunknown, $countones and Abort System Functions and Tasks such as $assertoff, $asserton, $assertkill. Note that the 2009/2012 LRM introduces quite a few new abort functions such as $assertpasson, $assertfailon, $assertvacuosoff, $assertcontrol, etc. These are described in their entirety in Sects. 16.17 and 16.18.

7.1 $onehot, $onehot0

$onehot and $onehot0 are quite self-explaining as shown in the Fig. 7.1. *Note that if the expression is 'Z' or 'X' that that $onehot or $onehot0 will fail. But will not fail if there are 'x's and 'z's on the bus but—at least—one '1'. Read on.*

A simple application is described in the Fig. 7.1. For any acknowledge of a bus grant there can only be one bus grant. This is very easily accomplished by $**onehot** as shown.

Note the results at time 25 and 35. There is an 'x' and a 'z' on the bus but since there is only one bit '1', it meets $**onehot** requirement and the property passes. What if that's not the result you want. You don't want an 'x' or a 'z' when searching for that one '1'.

The next section covers $**isunknown**, but here's the solution to above dilemma. Write the property as follows and it will guarantee that the bus is indeed in a known state on—all-bits of the bus and that there is only one '1'.

```
property bgcheck;
(@posedge clk) bgack |-> !$isunknown(busgnt) && $onehot(busgnt);
endproperty
```

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_7

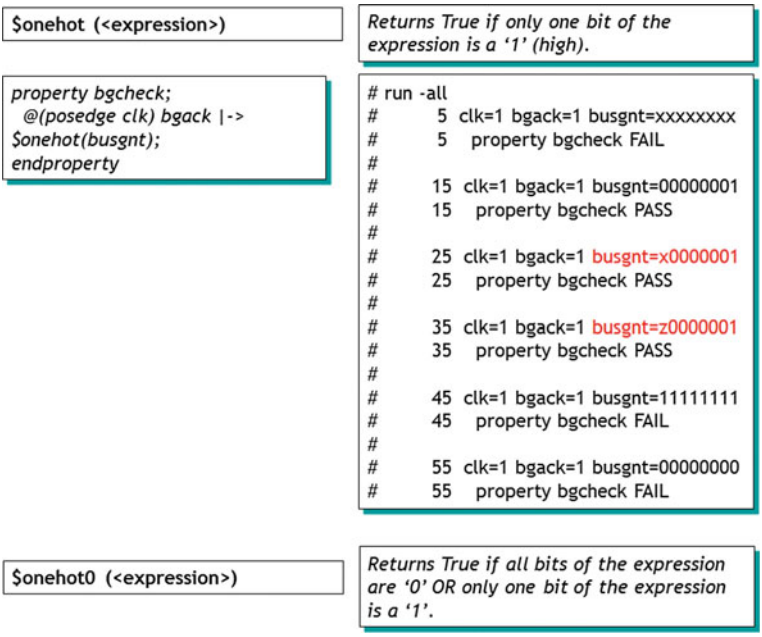


Fig. 7.1 $onehot and $onehot0

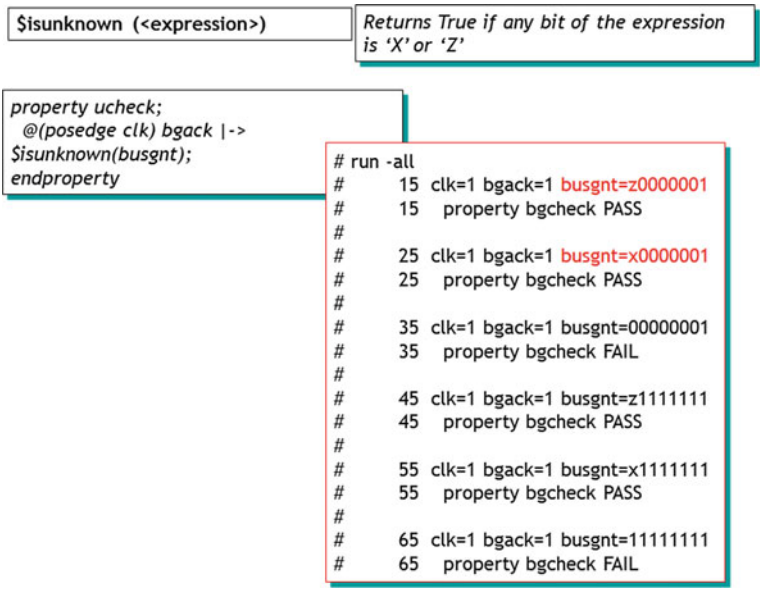


Fig. 7.2 $isunknown

7.2 $isunknown

$isunknown passes if the expression is unknown ('X' or 'Z'). In other words, if the expression is not unknown then the property will fail! *Hence, if you do want a failure on detection of an unknown ('X' or 'Z') then you have to negate the result of $isunknown*. Simple but easy to miss.

Simulation log in Fig. 7.2 clarifies the concept. Property 'ucheck' states that if 'bgack' is true that the 'busgnt' is unknown. What? This is simply to show what happens if you use $isunknown without a 'not'.

Application $isunknown Figure 7.3 shows two simple applications. First is identical to the one we just discussed, but with a 'not'.

```
@ (posedge clk) bgack |-> not ($isunknown(busgnt));
```

It says, if bgack is true (high) that busgnt must not be in unknown state. Again, note that $isunknown returns a true on detection of an unknown. Therefore, if you want a failure, you have to negate the result. This is shown in the simulation log

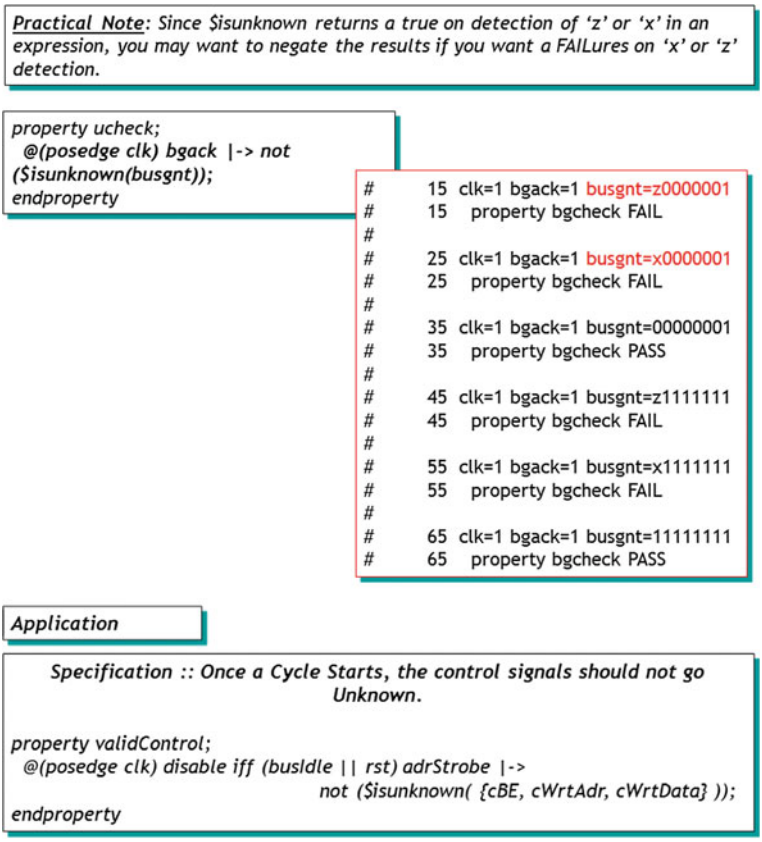


Fig. 7.3 $isunknown application

above. The second application (bottom of Fig. 7.3), says that if 'adrStrobe' is High that the control signals cannot be unknown.

7.3 $countones

$countones is very simple but powerful feature Note that the system function can be used in a procedural block as well as in a concurrent property/assertion.

Figure 7.4 shows an application which states that if there is a bus grant acknowledge (bgack) that there can be only 1 bus grant (busgnt) active on the bus. Note that we are using $countones in a procedural block in this example. Note also that if the entire 'busgnt' is unknown ('X') or tristate ('Z'), the assertion will fail. Figure 7.5 shows a very simple way to check for Gray Code compliancy.

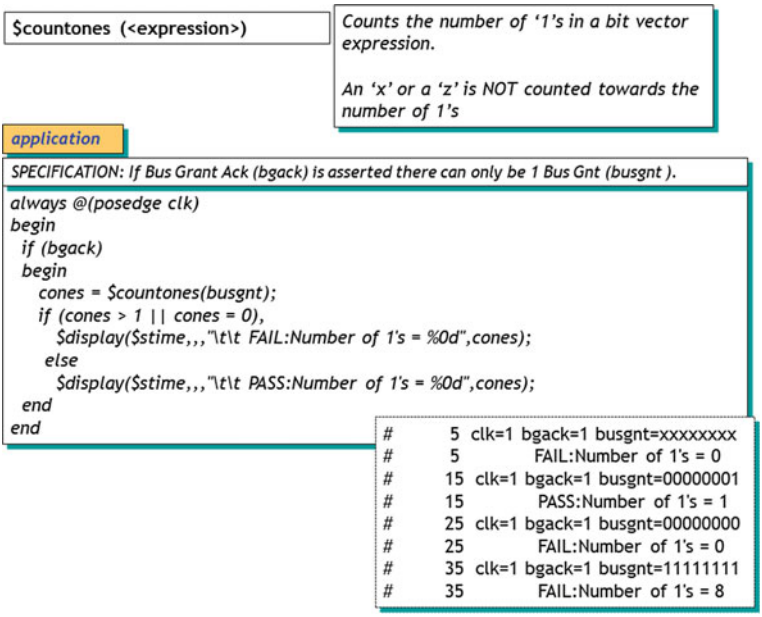


Fig. 7.4 $countones—basics and application

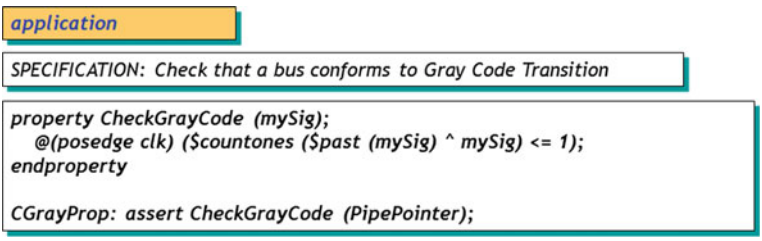


Fig. 7.5 Application $countones

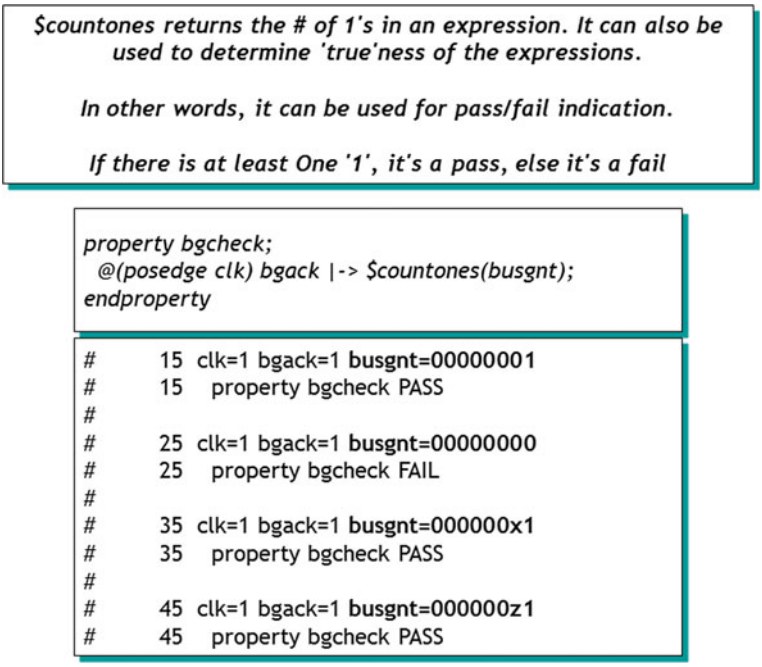


Fig. 7.6 $countones as boolean

7.3.1 $countones (as Boolean)

See Fig. 7.6.

7.4 $assertoff, $asserton, $assertkill

There are many situations when you want to have a global control over assertions both at module and instance level. Recall that 'disable iff' provides you a local control directly at the source of the assertion.

As noted in Fig. 7.7, **$assertoff** temporarily turns off execution of all assertions. Note that if an assertion is under way when $assertoff is executed, the assertion *won't* be killed. You restart assertion execution on a subsequent invocation of $asserton. **$assertkill** will kill *all* assertions in your design *including* already executing assertion. And it won't automatically restart when the next assertion starts

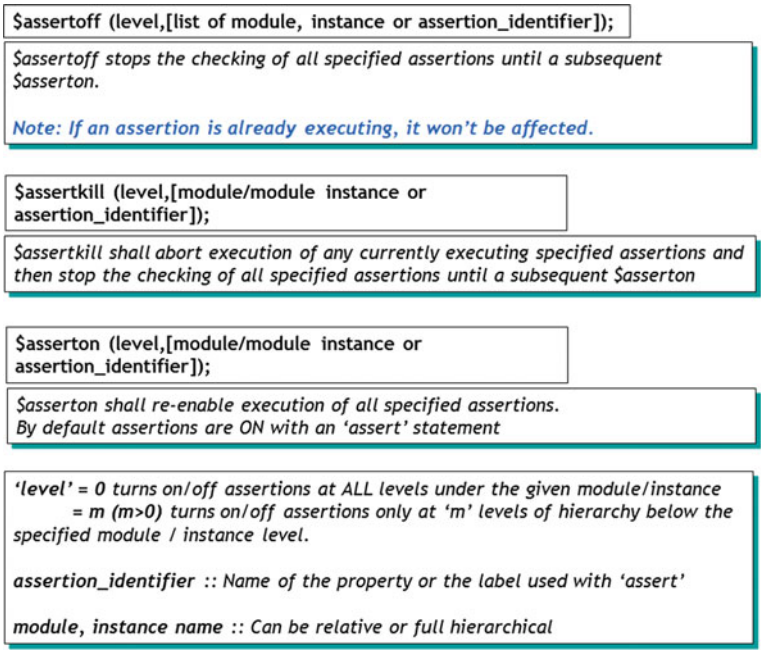


Fig. 7.7 $assertoff, $asserton, $assertkill—basics

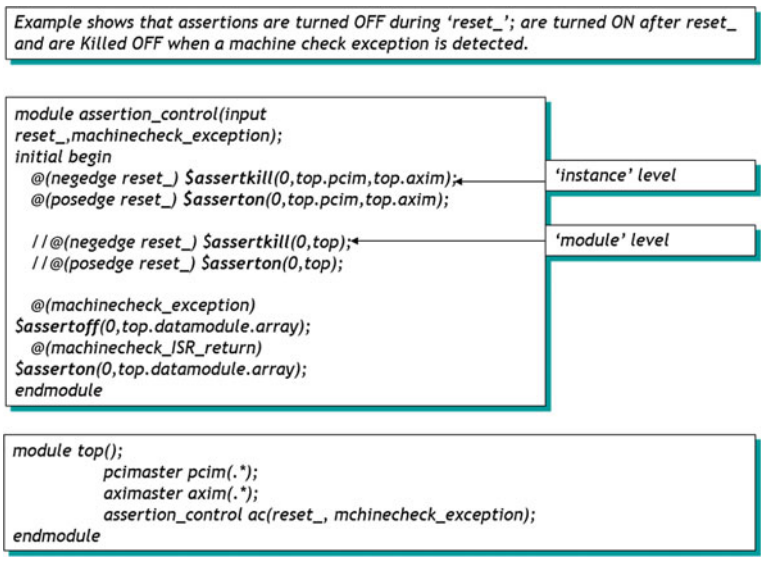


Fig. 7.8 Application assertion control

executing. It will restart executing only on the subsequent $asserton. **$asserton** is the default. It is required to restart assertions after a $assertoff or $assertkill.

Figure 7.8 shows a typical application deployed by projects to suppress execution of assertions during reset or during an exception, if so required.

Chapter 8
Multiple Clocks

Introduction: This chapter describes multiply-clocked sequences and properties and clock flow semantics (how does clock flow from one clock domain to another). It also discusses all the operators that work on multiply-clocked properties such as 'and', 'or', 'not' etc. It also describes nuances of Legal and Illegal conditions of such properties and sequences.

8.1 Multiply-Clocked Sequences and Properties

There are hardly any designs anymore that work only on a single clock domain. So far we have seen properties that work off of a single clock. But what if you need to check for a temporal domain condition that crosses clock boundaries. The so-called CDC (clock domain crossing) issues can be addressed by multiple clock assertions.

We'll thoroughly examine how a property/sequence crosses clock boundary. What's the relationship between these 2 (or more) clocks? How are sampling edges evaluated once you cross the clock domain. Note that in a singly clocked system, the sampling edge is always one—that is mostly the clock posedge or negedge. Since there are two (or more) clocks in multiply clocked system, we need to understand how the sampling edges cross boundary from one clock to another. I think it is best to fully understand the fundamentals before jumping into applications.

Note that there are differences in the way a 'sequence' behaves for multiple clocks and the way a property behaves. Read on …

Figure 8.1 shows a simple multiply clocked sequence. It says at (posedge clk0) A is true and the *very next nearest strictly subsequent* (posedge clk1) B is true. Note the emphasis on '*very next*'. That is because when you join two subsequences each of which runs on a different clock, you can transition only from one clock domain's sampling edge to the next clock domain's very next available sampling edge. This will be clearer when we dive a bit more into detail.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_8

Multiply-clocked sequences are built by concatenating singly-clocked subsequences using the delay concatenation operators ##1 and ##0.

```
sequence mclocks;
  @(posedge clk0) A ##1 @(posedge clk1) B;
  //                ^^^^^^^^^^^^^^^^^
endsequence
```

"##1 @(posedge clk1)" here does -not- necessarily mean a delay of one clock.

It means on a match of 'A' @(posedge clk0), *the ##1 moves the time to the nearest strictly subsequent posedge clk1* and the sequence ends at that point with a match of B.

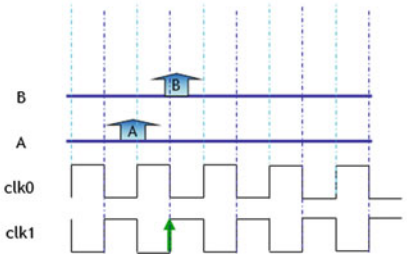


Fig. 8.1 Multiply clocked sequences—basics

Jumping ahead a bit, this '*very next nearest strictly subsequent edge*' semantic is why we use ##1 to cross clock boundaries. So, can you use ##2 when crossing clock boundaries? Keep this question in mind as you learn basics of multiply clocked sequences and properties.

8.1.1 Multiply Clocked Sequences

The timing diagram in Fig. 8.1 shows that at (posedge clk0), 'A' is true. The clocks are out of phase, so the very next clock edge of clk1 is half a clock delayed from posedge clk0. At the posedge of clk1, 'B' is sampled true and the sequence 'mclocks' passes. The point here is that '##1 @ (posedge clk1)' waited only for ½ clk1 and not a full clk1 because the *very next nearest strictly subsequent posedge clk1* arrived within ½ clock period. The next clock can come in any time after clock0 and that will be the 'very next' edge taken as the sampling edge for that subsequence.

Important Note:
The LRM 2005 requirement of ##1 between two subsequences have been removed from 1800-2009/2012. In the 2009/2012 standard you can have both ##1 and ##0 between two subsequences with different clocks. More on this coming up.

```
//ASSUME clk0 is identical to clk1

sequence mclocks;
  @(posedge clk0) A ##1 @(posedge clk1) B;
          //(Or @(posedge clk0) A ##1 @(posedge clk0) B; )
endsequence
```

If both clocks are identical then the clocking event does not change after the ##1 delay and the above sequence is equivalent to

```
@(posedge clk0) A ##1 B;
```

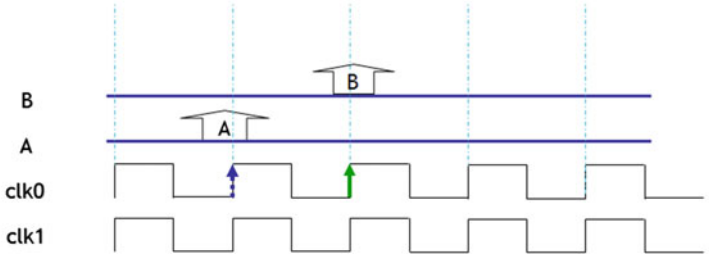


Fig. 8.2 Multiply clocked sequences—identical clocks

So, what happens if clk0 and clk1 are in phase? See Fig. 8.2. The explanation is in the figure itself. As self-evident, the sequence will wait for one full clk before sampling 'B'. But more importantly note that, *if* the clocks on the clock crossing boundary are identical (in phase and same period), then the following is true.

@ (posedge **clk0**) A ##1 @ (posedge **clk1**) B; *is identical to*
@ (posedge **clk0**) A ##1 @ (posedge **clk0**) B; *is identical to*
@ (posedge **clk0**) A ##1 B;

8.1.2 Multiply Clocked Sequences—Legal and Illegal Sequences

Before we move onto multiply clocked properties and based on our observations, let us quickly examine the legal and illegal cases of multiply clocked sequences. ***Again, these cases apply only to sequences and not to properties***.

As LRM puts it, Multiclocked sequences are built by concatenating singly clocked subsequences using the single-delay concatenation operator ##1 or the zero-delay concatenation operator ##0. The single delay indicated by ##1 is understood to be from the end point of the first sequence, which occurs at a tick of the first clock, to the nearest strictly subsequent tick of the second clock, where the second sequence begins. The zero delay indicated by ##0 is understood to be from the end point of the

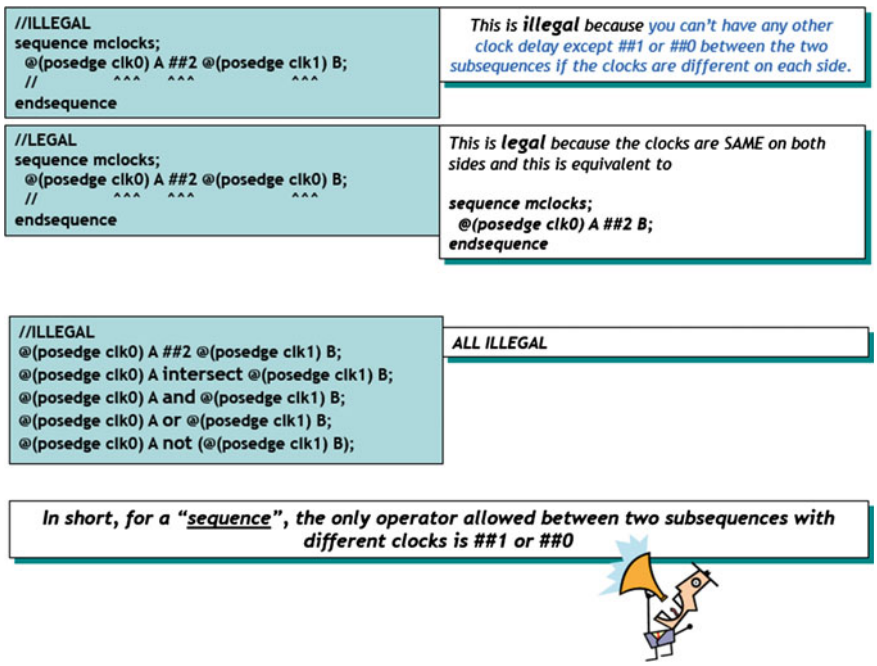


Fig. 8.3 Multiply clocked sequences—illegal conditions

first sequence, which occurs at a tick of the first clock, to the nearest possibly *overlapping* tick of the second clock, where the second sequence begins.

Bottom line is that you can only have ##1 or ##0 between two subsequences with different clocks. If the clocks are the same on both sides, then there is not such restriction. Figure 8.3 makes it clear.

8.1.3 Multiply Clocked Properties—'and' Operator

Note again that here we are discussing multiply clocked 'and' in a *property* and not a *sequence*. As mentioned above, such an 'and' in a sequence is illegal.

The concept of 'and' of two singly clocked properties have been discussed before. But what if the clocks in the properties are different? The important thing to note here is the concept of the very next strictly subsequent edge. In Fig. 8.4, at the posedge of clk0, 'a' is sampled high. That triggers the consequent that is an 'and' of 'b' and 'c'. Note that 'b' is expected to be true at the very next edge of clk1 (after the posedge of clk0). *In other words, even though there is a non-overlapping operator in the property, we don't quite wait for 1 clock*. We simply wait for the very next posedge of clk1 to check for 'b' to be true. The same story applies to 'c'. When both 'b' and 'c' occur as shown in Fig. 8.4, the property will pass. As with

```
property mclocks;
  @(posedge clk1) b and @(posedge clk2) c;
endproperty

baseP: assert property (@(posedge clk0) a |=> mclocks) else
gotoFail;
coverP: cover property (@(posedge clk0) a |=> mclocks) gotoPass;
```

'B' and 'C' must be true at immediate next posedge of clk1 and clk2 respectively after the posedge of clk0

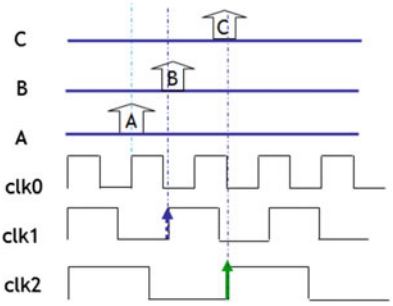


Fig. 8.4 Multiply clocked properties—'and' operator between two different clocks

the singly clocked 'and', the assertion passes at the match of the longest (so to say) sequence 'c'.

Figure 8.5 shows another scenario to solidify the concept of 'and' for multiply clocked assertions. The 'and' is between two subsequences which use the same clock. The behavior is obvious but interesting. This is because "@ (posedge clk1) b **and** @ (posedge clk1) c" acts essentially like "@ (posedge clk1) b **and** c". Hence, both the 'b' and the 'c' must now occur at the very next posedge of clk1.

In short, as we saw before, "@ (posedge clk1) b **and** @ (posedge clk1) c" is identical to "@ (posedge clk1) b **and** c".

8.1.4 Multiply Clocked Properties—'or' Operator

All the rules of 'and' apply to 'or'—except as in singly clocked properties—when either of the sequence (i.e. either the LHS or RHS of the operator) passes that the assertion will pass. The concept of 'the very next strictly subsequent clock edge' is the same as with 'and'.

Please refer to Fig. 8.6 for better understanding of 'or' of multiply clocked properties. The property passes when either @ (posedge clk1) b or @ (posedge clk2) c occurs. In other words, if @ (posedge clk2) c occurs before @ (posedge clk1) b, the property will pass at @ (posedge clk2) c.

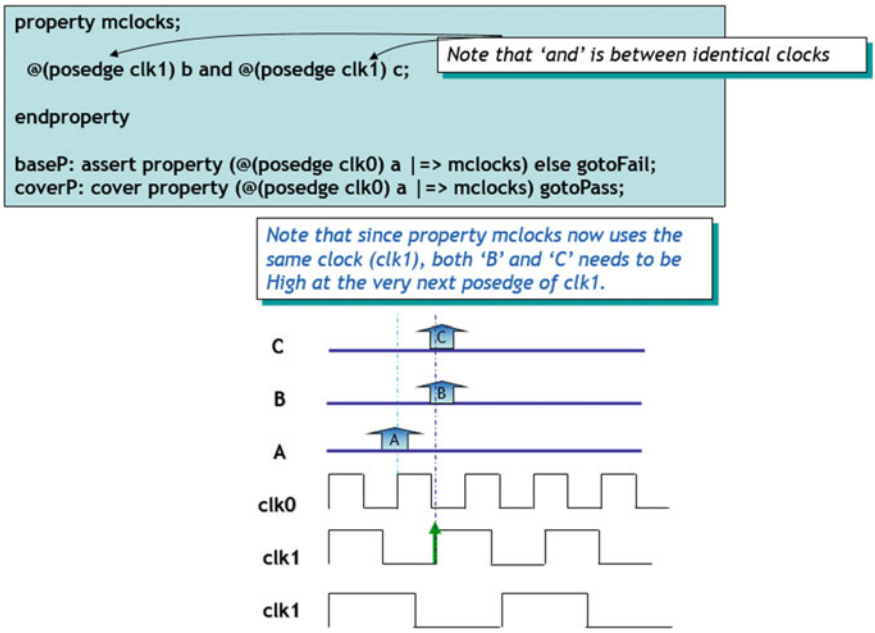


Fig. 8.5 Multiply clocked properties—'and' operator between same clocks

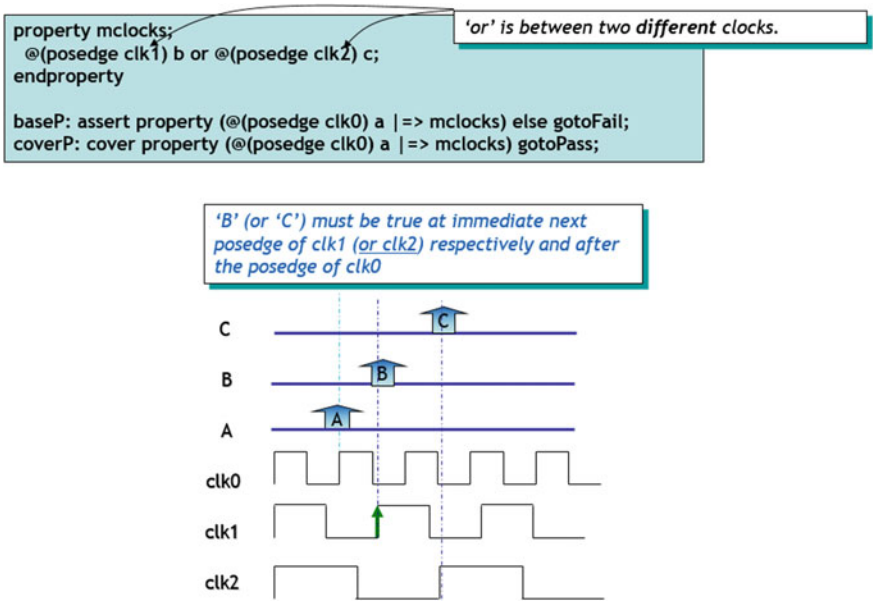


Fig. 8.6 Multiply clocked properties—'or' operator

8.1.5 Multiply Clocked Properties—'not'—Operator

'not' is an interesting operator when it comes to multiply clocked assertions.

The assertion in Fig. 8.7 works as follows. At posedge clk0, 'a' is true which triggers the consequent mclocks. The property mclocks specifies that @ posedge clk1 'b' needs to be true and 'c' should—not—be true @ posedge clk2. The timing diagram shows that 'a' is true at posedge clk0. At the subsequent edge of clk1 'b' should be true and since it is indeed true, the property moves along. Because of an 'and' it looks for 'c' to be *not* true at the very next subsequent posedge clk2. Well, 'c' is indeed true but since we have a 'not' in front of @ (posedge clk2), the property will fail. The concept of 'not' is the same as that of singly clocked properties except for the edge of the clock when it is evaluated. The simulation log clarifies the concept.

8.1.6 Multiply Clocked Properties—Clock Resolution

These rules are important to follow when you are dealing with multiply clocked properties (Fig. 8.8).

Figures 8.9 and 8.10 illustrate other important concepts. How do clocks apply (or flow) from one part of property to another? The description in the figure explains how this works.

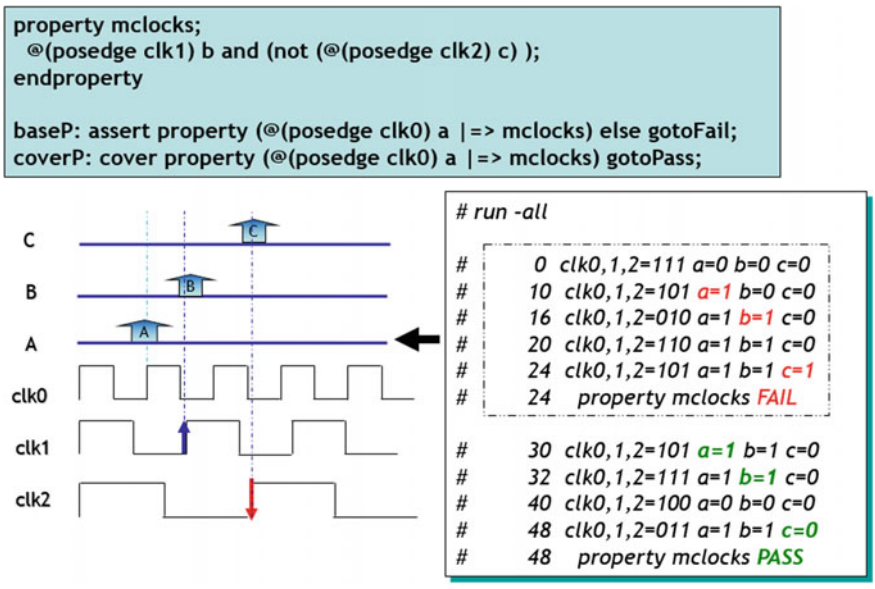


Fig. 8.7 Multiply clocked properties—'not' operator

```
property mclocks;
  @(posedge clk0) A |->
              if (D) @(posedge clk0) B;
endproperty
```

This is equivalent to
@(posedge clk0) A |-> if (D) B;

```
property mclocks;
  @(posedge clk0) A |->
              if (D) @(posedge clk0) B
              else @(posedge clk0) (~B);
endproperty
```

This is equivalent to
@(posedge clk0) A |-> if (D) B else (~B) ;

```
property mclocks;
  @(posedge clk0) A |->
              if (D) @(posedge clk0) B
              ##1 @(posedge clk1) Z
              else @(posedge clk0) (~B);
endproperty
```

This is equivalent to
@(posedge clk0) A |-> if (D) B ##1 @(posedge clk1) Z
else (~B) ;

Fig. 8.8 Multiply clocked properties—clock resolution

```
property mclocks;
 @(posedge clk0) A ##1 (b ##1 @(posedge clk1) C) |=> D;

endproperty
```

Here clk0 flows through 'A' then in the parenthesis to 'B' but not through 'C';
But once out of the parenthesis, it then flows through 'D'

```
//LRM: System Verilog 3.1a, Page 244:
//"The scope of a clocking event flows into parenthesized sub expressions and, if
//the sub expression is a sequence, also flows left-to-right across the parenthesized
//sub expression. However, the scope of a clocking event does not flow out of
//enclosing parenthesis".
```

```
sequence s1;
 @(posedge clk0) b ##1 c;
endsequence

sequence s2;
 @(posedge clk1) d ##1 e;
endsequence

sequence s;
 @(posedge clk) a ##1 s1 ##1 s2 ##1 f;

endsequence
```

Fig. 8.9 Multiply clocked properties—clock resolution—II

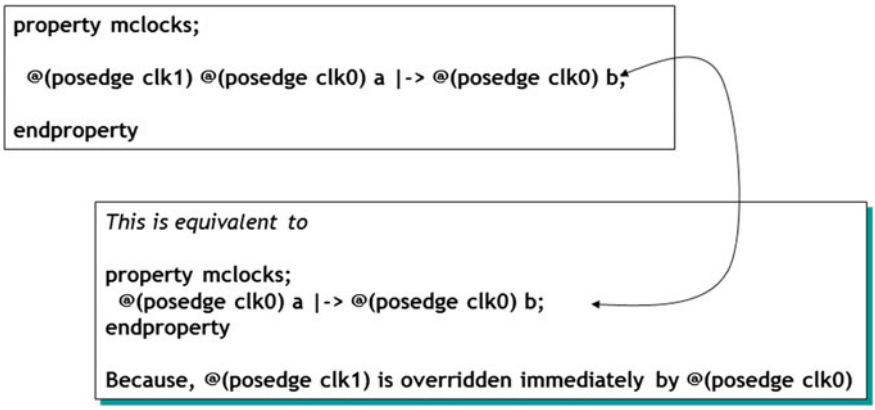


Fig. 8.10 Multiply clocked properties—clock resolution—III

1. In Fig. 8.9, property 'mclocks', (posedge clk0) applies to 'A' as well as 'B' since 'B' does not have an explicit clock. So far so good.
2. Then (posedge clk1) applies to 'C'. That also makes sense.
3. But which clock is applied to 'D' in the consequent since 'D' does not have its own explicit clock?
4. According to the 1800-2005 LRM, 'D' will inherit (posedge clk0) and *not* the (posedge clk1). This is not quite intuitive. But LRM makes it very clear that "*the scope of a clocking event does not flow out of enclosing parenthesis*".
5. In our case, ('B' ##1 @ (posedge clk1) C) is in parenthesis. So once we are out of that parenthesis, (posedge clk1) does not flow forward but (posedge clk0) moves forward to the consequent 'D'.

Similarly, the bottom example in Fig. 8.9 shows how the clock would 'flow' when we have multiple subsequences each with its own clock. Note that there are three different clocks in this sequence. (posedge clk) flows through 'a'. Then 's1' and 's2' use their own clocks as sampling edges (clocks) for their sequences. But once out of 's2', (posedge clk) is applied to 'f'—not—the (posedge clk1) of 's2'.

Figure 8.10 shows another interesting property. What will happen if you need to transition from one clock to another before checking for an expression/sequence? Well, you cannot quite do that. In Fig. 8.10, we show that (posedge clk1) is immediately followed by (posedge clk0). This does *not* mean that the property will wait first for (posedge clk1) then for (posedge clk0) and then apply (posedge clk0) to 'a'. It will simply override (posedge clk1) with (posedge clk0) and directly apply (posedge clk0) to 'a'. This is shown in the bottom of the figure in the equivalent property.

8.1.7 Multiply Clocked Properties—Legal and Illegal Conditions

Figure 8.11 is a recap as an easy reference to legal and illegal semantics of multiply clocked properties.

The top most example shows that it's ok to have different clocks between the antecedent and the consequent, as long as the implication operator is non-overlapping.

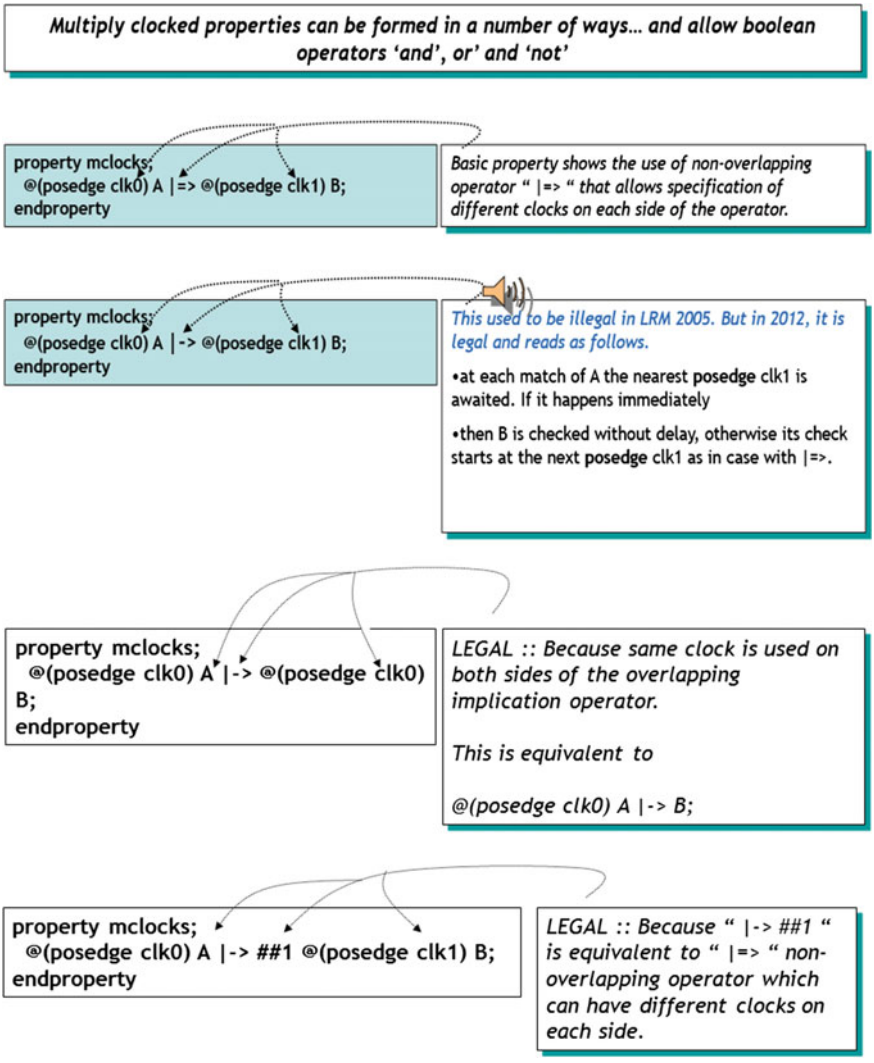


Fig. 8.11 Multiply clocked properties—legal and illegal conditions

In 2012 LRM, the multiclocked overlapping implication |-> has the following meaning: at the end of the antecedent the nearest tick of the consequent clock is awaited. If the consequent clock happens at the end of the antecedent, the consequent starts checking immediately. Otherwise, the meaning of the multiclocked overlapping implication is the same as the meaning of the multiclock nonoverlapping implication.

Also as shown in the third example, overlapping operator is perfectly legal if the clocks on both sides of the overlapping operator are the same.

And the last example is quite intuitive in that "⇒" is equivalent to "|-> ##1". Hence, you can have different clocks on each side of overlapping operator.

Note also that as in the overlapping operator, the semantics of multiclocked if/if-else operators is similar to the semantics of the overlapping implication.

For example,

```
@(posedge clk0) if (b) @(posedge clk1) c else @(posedge clk2) d
```

has the following meaning: the condition b is checked at posedge clk0. If b is true, then 'c' is checked at the nearest, possibly overlapping posedge clk1, else 'd' is checked at the nearest possibly overlapping posedge clk2.

To summarize: clock flow provides that in a multiclocked sequence or property, the scope of a clocking event flows left to right across linear operators (e.g., repetition, concatenation, negation, implication, followed-by, and the nexttime, always, eventually operators) and distributes to the operands of branching operators (e.g., conjunction, disjunction, intersection, if–else, and the until operators) until it is replaced by a new clocking event.

Here's an interesting example of how clock flows and how that makes a significant difference. What's the difference between the following two properties?

```
A1: assert property (
    @(posedge clk1) Frame_ |-> nexttime @(posedge clk2) IRDY);
A2: assert property (
    @(posedge clk1) Frame_ |-> ##1 @(posedge clk2) IRDY;
```

In case of A1, (posedge clk1) flows through to 'nexttime', which means 'nexttime' causes advance of (posedge clk1) to the strictly nearest (posedge clk1), after which it looks for a subsequent nearest (posedge clk2) to evaluate IRDY. So, the clock flow is (posedge clk1) to (posedge clk1) to (posedge clk2).

In case of A2, ##1 is the synchronizer, so after Frame_ is found high at (posedge clk1), the clock flows to (posedge clk2) which means IRDY will be evaluated at the very next strictly nearest (posedge clk2). Here the clock flow is (posedge clk1) to (posedge clk2).

Exercise: Can you figure out clock flow in the following property?

```
A3: assert property (
    @(posedge clk1) Frame_ |-> @(posedge clk2) nexttime IRDY;
```

Here is one more example to nail down the concepts of 'clock flow' in multiply clocked properties.

```
a1: assert property
    (@(posedge clk) en && $rose(req) |=> gnt);
a2: assert property
    (@(posedge clk) en && $rose(req, @(posedge sysclk)) |=> gnt);
```

Both assertions a1 and a2 read: "whenever en is high and 'req' rises, at the next cycle 'gnt' must be asserted." In both assertions, the rise of 'req' occurs if and only if the sampled value of 'req' at the current posedge of clk is 1'b1 and the sampled value of 'req' at a prior point is different from 1'b1. So where do the assertions differ? The assertions differ in the specification of the *prior* point. In a1 the prior point is the preceding posedge of clk, while in a2 the prior point is the *most recent* prior posedge of sysclk.

As another example,

```
always_ff @(posedge clk1)
    Dreg <= $rose(Xreg, @(posedge sysclk));
```

Here, Dreg is updated in each time step in which posedge clk1 occurs, using the value returned from the $rose sampled value function in that time step. $rose compares the sampled value of the LSB of Xreg from the current time step (one in which posedge clk1 occurs) with the sampled value of the LSB of Xreg in the strictly prior time step in which posedge sysclk occurs.

Chapter 9
Local Variables

This chapter is entirely devoted to the dynamic Local Variables. Without the dynamic multi-threaded semantics and features of Local Variables, many of the assertions would be impossible to write. The chapter also lays out how operators such as 'or' and 'and' affect the workings of parallel threads forked off by Local Variables based assertions. There are plenty of examples and applications to help you weed through the semantics.

Local variable is a feature you are likely to use very often. They can be used both in a sequence and a property. They are called *local* because they are indeed local to a sequence and are not visible or available to other sequences or properties. Of course, there is a solution to this restriction, which we will study further into the section. Figure 9.1 points out key elements of a local var. The most important and useful aspect of a local variable is that it *allows multi-threaded application and creates a new copy of the local variable with every instance of the sequence in which it is used.* User does not need to worry about creating copies of local variables with each invocation of the sequence. Above application says that whenever 'RdWr' is sampled high at a posedge clk, that 'rData' is compared with 'wData' 5 clocks later. The example shows how to accomplish this specification. Local variable 'int local_data' stores the 'rData' at posedge of clk and then compares it with wData 5 clocks later. Note that 'RdWr' can be sampled true at every posedge clk. Sequence 'data_check' will enter every clock; create a new copy of local_data and create a new pipelined thread that will check for local_data+'hff with 'wData' 5 clocks later.

Note that the sampled value of a local variable is defined as the current value.

Moving along, Fig. 9.2 shows other semantics of local variables. *Pay close attention to the rule that local variable must be 'attached' to an expression while comparison cannot be attached to an expression*!!

As shown in Fig. 9.2, *a local variable must be attached to an expression when you store a value into it. But when you compare the value stored in a local variable, it must not be attached to an expression.*

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_9

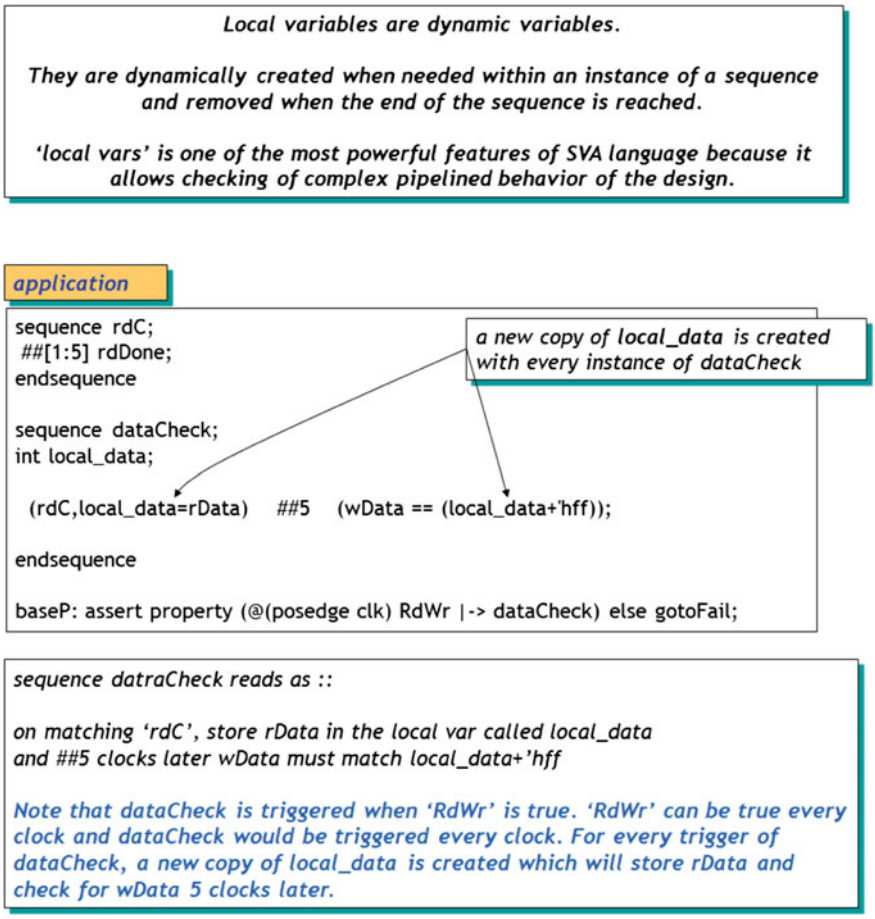


Fig. 9.1 Local variables—basics

In the topmost example, "local_data=rData) is attached to the sequence 'rdC'. In other words, assignment "local_data=rData" will take place only on completion of sequence 'rdC'. Continuing with this story of storing a value into a local variable, what if you don't have anything to attach to the local variable when you are storing a value? Use 1'b1 (always true) as an expression. That will mean whenever you enter a sequence, that the expression is always true and you should store the value in the local variable. Simple!

Note that local variables do not have default initial values. A local variable without an initialization assignment will be unassigned at the beginning of the evaluation attempt. An initialization assignment to a local variable uses the sampled value of its expression in the time slot in which the evaluation attempt begins. The expression of an initialization assignment to a given local variable may refer to a previously declared local variable. In this case the previously declared local variable must itself have an initialization assignment, and the initial value assigned to the

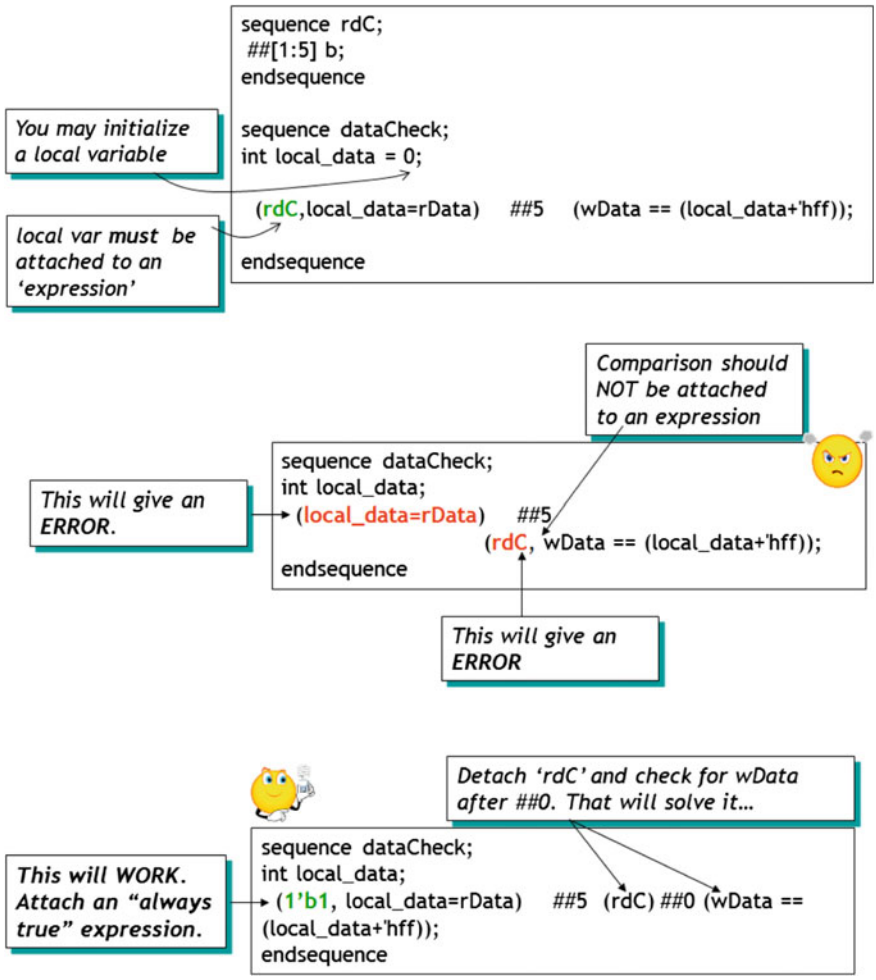


Fig. 9.2 Local variables—do's and don'ts

previously declared local variable will be used in the evaluation of the expression assigned to the given local variable. More on this later.

Ok, so what if you want to compare a value on an expression being true? As shown in Fig. 9.2, you can indeed accomplish this by 'detaching' the expression as shown. The resulting sequence (the last sequence in the Fig. 9.2) will read as "on entering dataCheck, store rData into local_data, wait for 5 clocks and then if 'b' (of sequence rdC) is true within 5 clocks, compare wData with stored local_data + 'hff".

Figure 9.3 points out a couple of other important features. First, there is no restriction in using a local variable in either a sequence or a property. In addition, you cannot declare a local variable as a formal and pass as an actual from another sequence/property. That makes sense, else why would it be called 'local'?

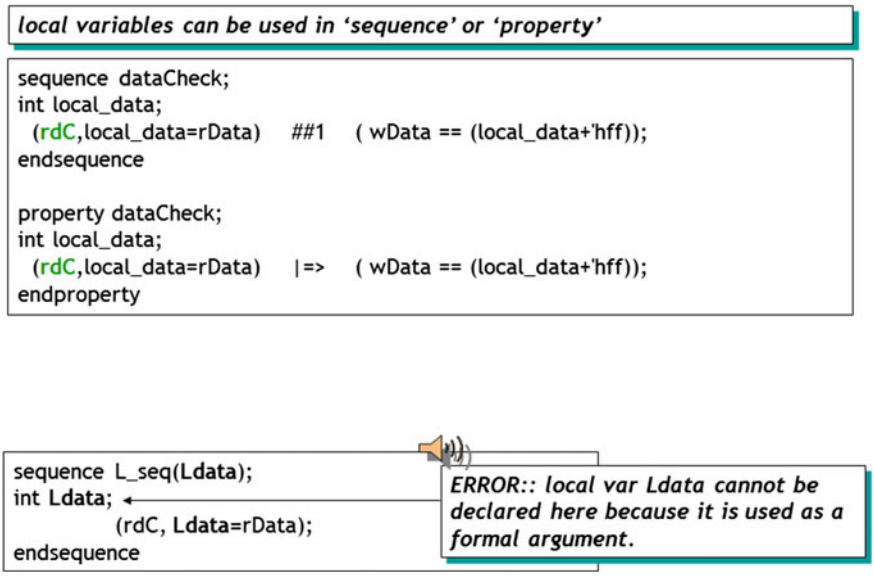


Fig. 9.3 Local variables—and formal argument

In Fig. 9.4, we see that a local variable in a sequence is not visible to the sequence that instantiates it. The solution is quite straightforward. Instead of poking at the local variable directly, simply pass an argument to the sequence that contains the local variable. When the sequence L_seq updates the argument locally, it will be visible to the calling sequence (H_seq). Note that Ldata is not declared as a local variable in sequence L_seq (else that would be an error as we discussed). L_seq simply updates a formal and passes it to the calling sequence, where the actual is declared as a local variable. This is shown in the bottom of Fig. 9.4.

Figures 9.5, 9.6, 9.7, 9.8 and 9.9 shows finer rules. Keep them as reference when you embark upon complex assertions. Annotation in the figure explains the situation(s).

Figure 9.6 describes the semantics governing local variables when they are used in the OR of two sequences. The local variable must be assigned in both the sequences of an OR. However, what if you cannot really do that? There are a couple of solutions presented in Fig. 9.7.

Figure 9.9 describes semantics that govern an 'and' of two sequences. In contrast to an 'or' of two sequences, a local variable must *not* be attached to both sequences involved in an 'and'. The first solution is identical to that for an 'or'. Assign the local variable outside of the 'and' of the two sequences as shown in the figure. Alternatively, simply assign to the local variable in only 1 of two sequences, which is an obvious solution. Figure 9.9 shows solution #2 in addition to the solution #1 in Fig. 9.8.

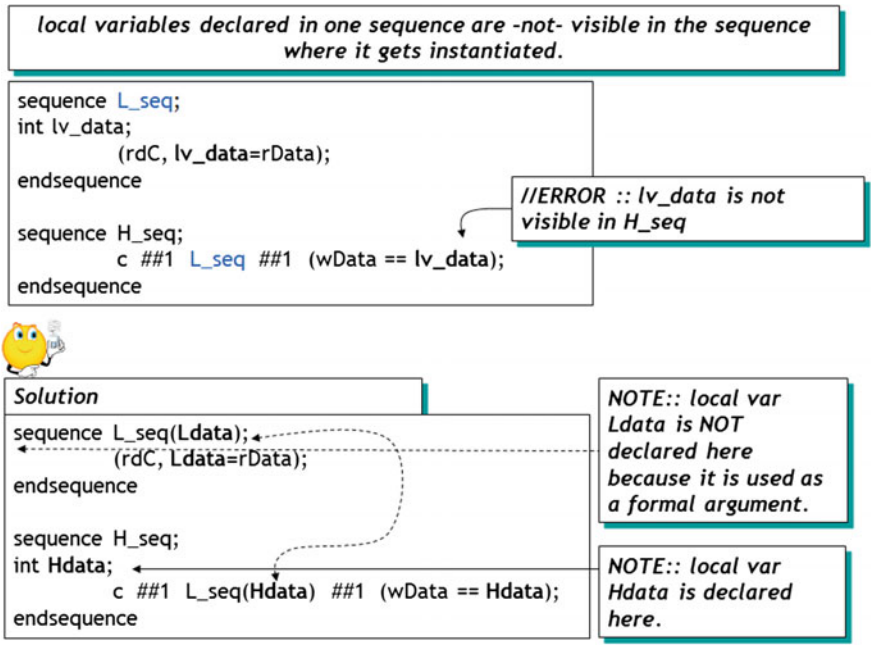


Fig. 9.4 Local variables—visibility

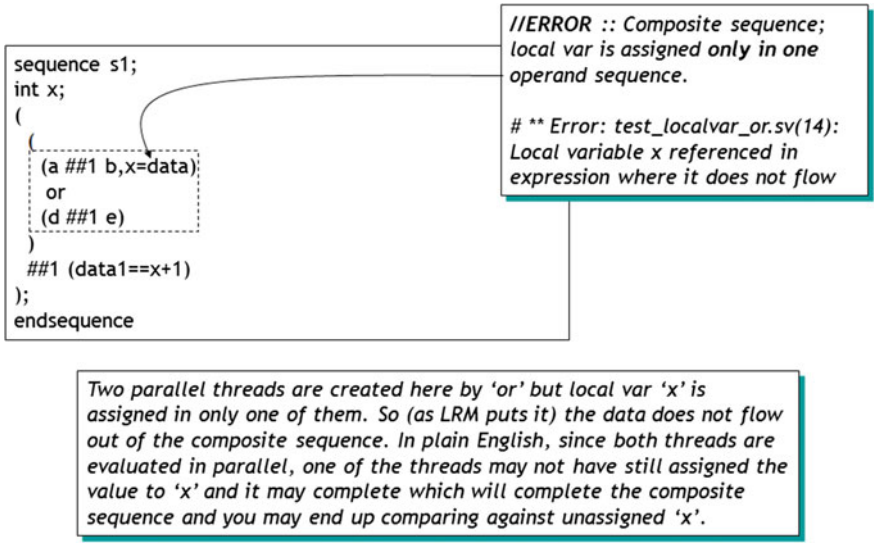


Fig. 9.5 Local variable composite sequence with an 'OR'

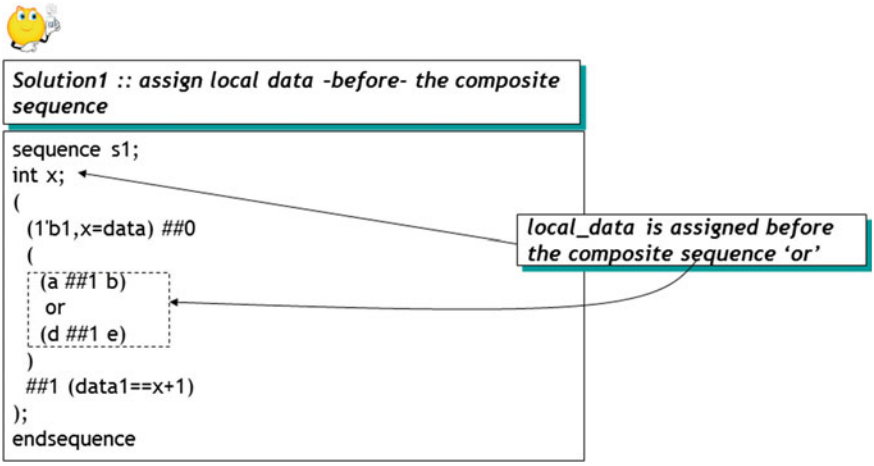


Fig. 9.6 Local variables—for an 'OR' assign local data—before- the composite sequence

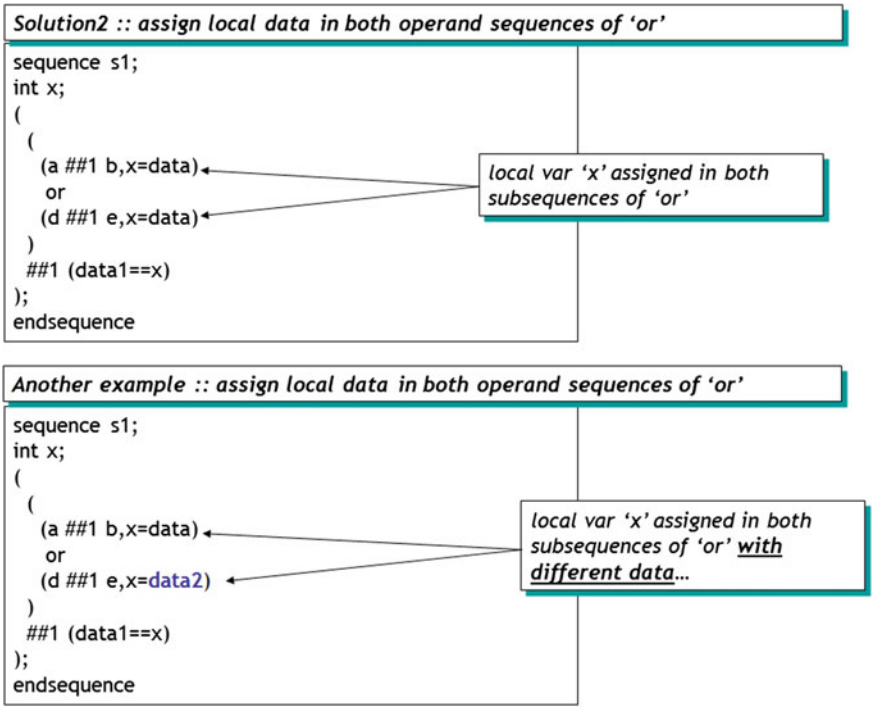


Fig. 9.7 Local variables—assign local data in both operand sequences of 'OR'

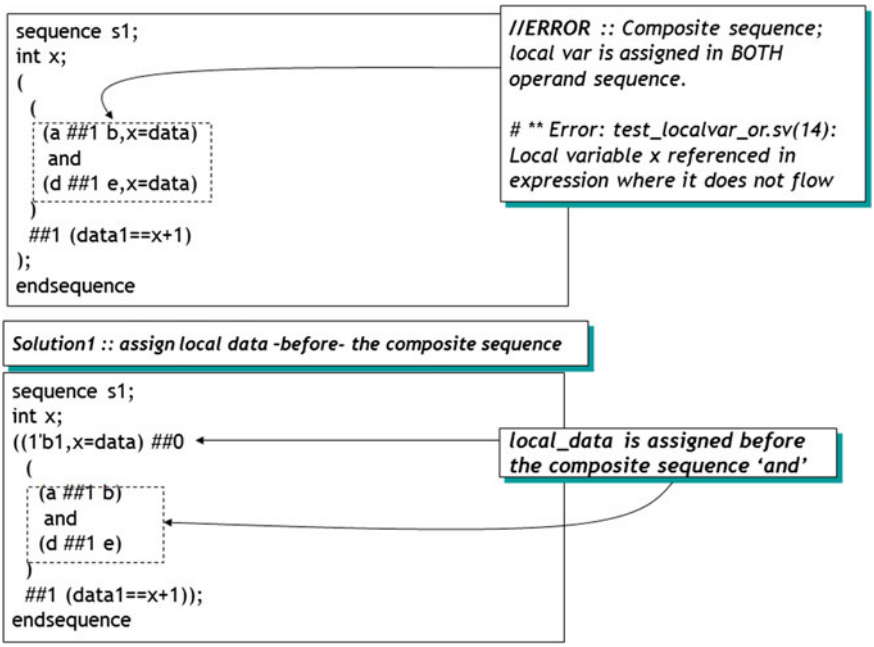


Fig. 9.8 Local variables—'and' of composite sequences

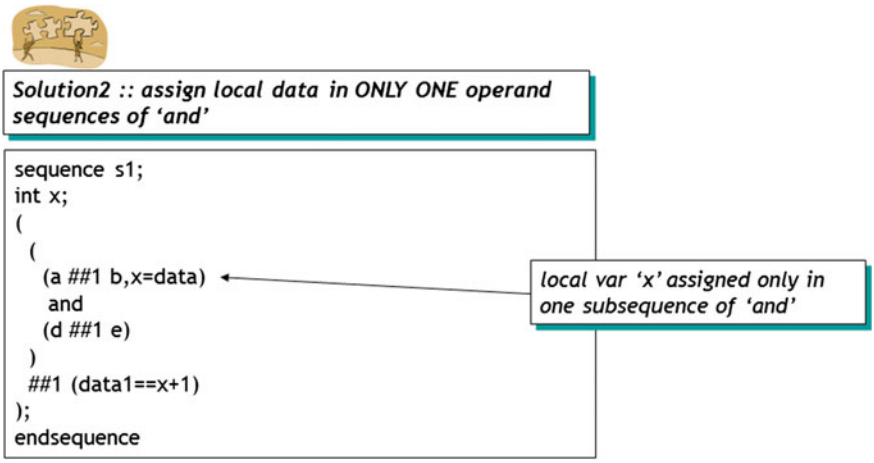


Fig. 9.9 Local variables—finer nuances III

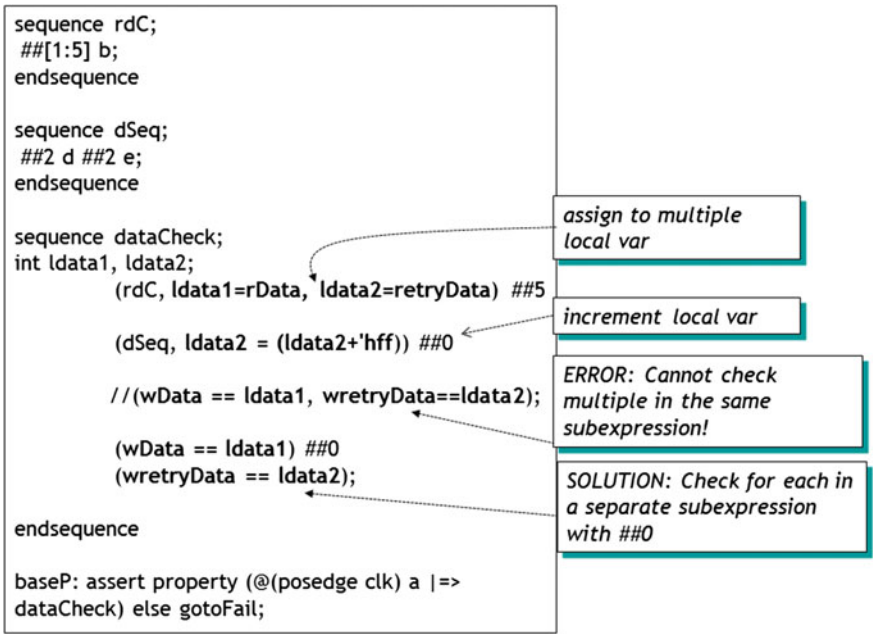


Fig. 9.10 Local variables—further nuances IV

Figure 9.10 describes further rules governing local variables. First, you can assign to multiple local variables, attached to a single expression. Second, you can also manipulate the assigned local data in the same sequence (as is the case for ldata2). But as before, there are differences in assigning to (storing to) local

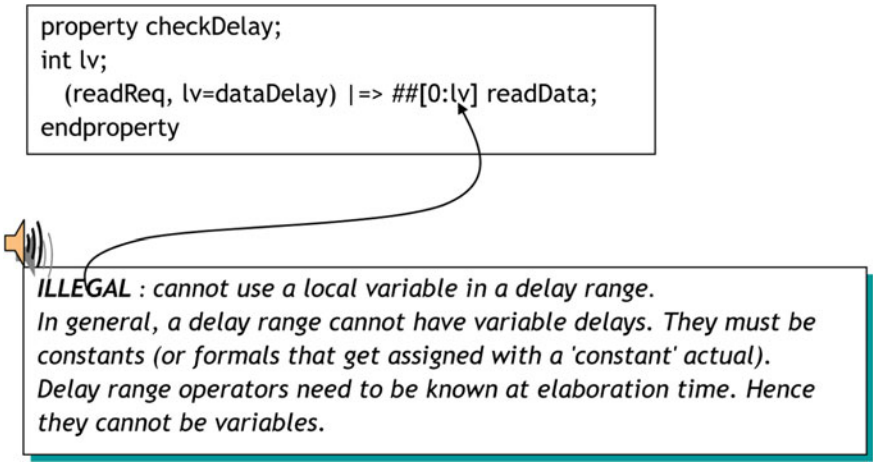


Fig. 9.11 Local variable cannot be used in delay range

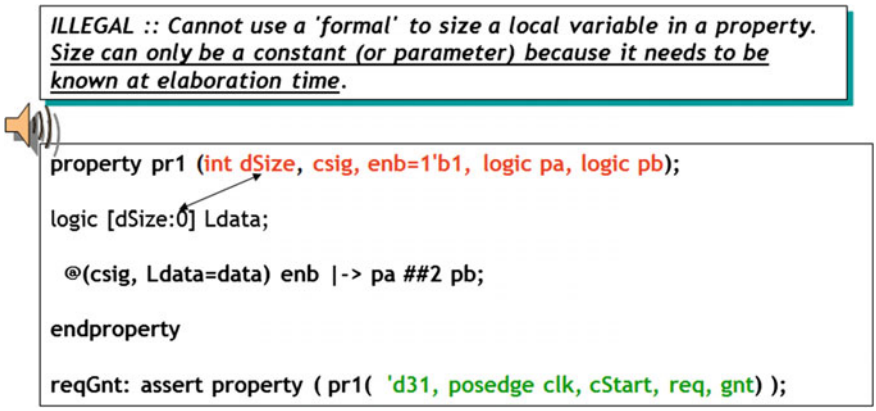


Fig. 9.12 Local variables—cannot use a 'formal' to size a local variable

variables and comparing their stored value. *You cannot compare multiple local variable values in a single expression* in a sequence as is the case in the line "// (wData == ldata1, wretryData==!ldata2)". This is illegal. Of course, there is always a solution as shown in the figure. Simply separate comparison of multiple values in two subsequences with no delay between the two. The 'Solution' annotation in the figure makes this clear.

Figure 9.11 shows that you cannot use a local variable in the range operator. But, it's not the local variables fault. It's the fact that we cannot have variable delay in either #m or #[m:n] delay operators. From software point of view, the delay range operators need to be known at elaboration time. Hence they cannot be 'variables'. From hardware point of view, this is a bummer!

Figure 9.12 shows that you cannot use a 'formal' to size a local variable. Again, 'size' of a vector (bus) declaration can only be a constant. Again, there is a software reason and a hardware reason.

Following points out further cases of legal/illegal declarations of local variables.

```
property illegal_legal_declarations;

data; // ILLEGAL. 'data' needs an explicit data type.
logic data = 1'b0; // LEGAL. Note that unlike SystemVerilog variables, local variables have no default initial value. Also, the assignment can be any expression and need not be a constant value
byte data []; // ILLEGAL – dynamic array type not allowed.

endproperty
```

Also, you can have multiple local data variable declarations as noted above. And a second data variable can have dependency on the first data variable. *But the first data variable must have an initial value assigned*. Here's an example.

```
property legal_data_dependency;

logic data = data_in, data_add = data + 16'h FF;

endproperty
property illegal_data_dependency;

logic data, data_add = data + 16'FF;

endproperty
sequence illegal_declarations (
output logic a, // illegal: 'local' is not specified with direction.
local inout logic b, c = 1'b0, // default actual argument illegal for inout
local d = expr, // illegal: type must be specified explicitly
local event e, // illegal: 'event' type is not allowed
local logic f = g // illegal: 'g' cannot refer to the local variable declared below. It
must be resolved upward from this declaration
);
```

Note one more example of how you can declare a local variable used for initialization directly as an input. First, the sequence with the traditional way of initializing BSize.

```
ONE:
sequence burst (logic FRAME_, BurstSize = 4)
logic abc = 1'b0, BSize = BurstSize;
@(posedge clk)

FRAME_ |=>….

endsequence
```

Since, the BurstSize is solely used for sizing the local variable BSize, you can parameterize it as follows, where now the actual will determine the BSize.

```
TWO:
sequence burst (logic FRAME_, local input logic BSize)
logic abc = 1'b0;
@(posedge clk)

FRAME_ |=> ….

endsequence
```

The keyword 'local' specifies that BSize is an 'argument local variable' (as LRM puts it) while the direction 'input' specifies that BSize will receive its initial value from the actual argument expression.

Note that in the first sequence the BSize initialization takes place @(posedge clk). Here the declaration assignments are performed when the evaluation reaches alignment with @(posedge clk) and at that point the value in the formal argument BurstSize is assigned to BSize as its initial value. In the second sequence, the initialization of BSize takes place when the actual changes its value assigns to formal BSize. Similarly, an 'output' 'argument local variable' outputs its value to the actual argument whenever the sequence matches. For 'inout', it obviously acts both as 'input' and 'output'. So, the rules specified for 'input' applies when it acts as an 'input' and the rules for 'output' apply when it acts as an 'output'.

Note that 'argument local variables' precede the 'body local variables'. If a sequence or property has both 'input' 'argument local variables' and the 'body local variables' with declaration assignments, the initialization assignment of the 'input' 'argument local variables' are performed first.

On the similar line of thought, the following rules also apply. See the 'sequence' below.

```
sequence local_IO (

local byte a;
local inout byte b;
local input logic c;
local output byte d;

);
endsequence
```

Following rules apply to the local variables of sequence local_IO.

1. If a direction is specified for an argument, then the keyword 'local' must also be specified.
2. If the keyword 'local' is specified, then the data type must also be explicitly specified.
3. If the keyword 'local' is specified without a direction, then a default direction of 'input' is understood.
4. An 'input' argument local variable may be declared with an optional default actual argument which can be any expression.
5. An 'output' or 'inout' argument local variable *cannot* be declared with a default actual argument because the actual argument must specify the local variable that will receive the 'output' value.
6. An 'argument local variable' can also be declared as 'output' or 'inout', but *only* in a sequence declaration. An 'argument local variable' of a property must be of direction 'input'.
7. An 'output' 'argument local variable' outputs its value to the actual argument whenever the sequence matches.

8. An ‘input’ receives its initial value from the actual argument.
9. For ‘inout’, it obviously acts both as ‘input’ and ‘output’ and the rules for ‘input’ apply when it is of direction ‘input’ and the rules for ‘output’ apply when it is of direction ‘output’.
10. As stated above, it is important to understand that the ‘sampled’ values are used for all terms that are *not* ‘local’ while the ‘current’ values are used for terms that are ‘local’.
11. The actual argument bound to an ‘argument local variable’ of direction ‘output’ or ‘inout’ must itself be a local variable.

A special note on the use of method ‘.triggered’ with a local variable. A local variable passed into an instance of a named sequence to which sequence method (.triggered) is applied, is not allowed. For example, the following is illegal.

```
sequence check_trdy (cycle_begin);

cycle_begin ##2 irdy ##2 trdy;

endsequence
property illegal_use_of_local_with_triggerd;

bit local_var;
(1'b1, local_var = CB) |-> check_trdy(local_var).triggered;

endproperty
```

Some more rules on referencing local variables.
A local variable can be referenced in expressions such as:

- Array indices
- Arguments of task and function calls
- Arguments of sequence and property instances
- Expressions assigned to local variables
- Boolean expressions
- Bit-select and part-select expressions

However, a local variable cannot be referenced in following:

- Clocking event expressions (even though LRM is ambiguous on this)
- The reset expression of a ‘disable iff’
- The abort condition of a reset operator (accept_on, sync_accept_on and the reject forms of these expressions. See Sect. 16.16)
- Expressions that are compile time constants. E.g. [*n], [→ n], ##n, [=n], etc. and the constant expressions of ranged forms of these operators
- An argument expression to a sampled value function ($rose, $fell, $past, etc.)

9.1 Application: Local Variables

The application in Fig. 9.13 is broken down as follows.

($rose(read),localID=readID

On $rose(read), the readID is stored in the localID.

not (($rose(read) && readID==localID) [*1:$])

Then we check to see if another read ($rose(read)) occurs and it's readID is the same as the one we stored for the previous Read in localID. We continue to check this consecutively until

##0 ($rose(readAck) && readAckID == localID) occurs.

If the consecutive check does result in a match, that would mean that we did get another $rose(read) with the same readID with which the previous read was issued. That's a violation of the specs. This is why we take a 'not' of this expression to see that it turns false on a match and the property would end.

If the consecutive check does not result in a match until *##0 ($rose(readAck) && readAckID == localID)* arrives then we indeed got a readAck with the same readAckID with which the original read was issued. The property will then pass.

In short we have proven that once a 'read' has been issued that another 'read' from the same readID cannot be re-issued until a 'readAck' with the same ID has returned.

Example: This example shows a simple way to track time. Here, on falling edge of Frame_, rising edge of IRDY cannot arrive for at least MinTime.

Once a 'read' has been issued, another 'read' for the same readID cannot be re-issued until a readAck with the same ID has returned.

```
property checkRead;
  int localID;
  ($rose(read),localID = readID) |=>
                    not (($rose(read) && readID==localID) [*1:$]) ##0
                    ($rose(readAck) && readAckID == localID);
endproperty

baseP: assert property (checkRead) else
                    $display($stime,,,"\tproperty FAIL");
```

Fig. 9.13 Local variables—application

Solution:

```
property FrametoIRDY (integer minTime);
integer localBaseTime;

@(posedge clk) ($fall(Frame_), localBaseTime = $time)
|=>
$rose(IRDY) && $time >= localBaseTime + minTIme);

endproperty
measureTime: assert property (FrametoIRDY (.minTIme (MINIMUM_TIME)));
```

Local variable examples are scattered throughout the book. Some are found in following sections.

Section **Clock delay range operator: ##[m:n]: multiple threads** 6.2.1
Section **FIFO TESTBENCH AND ASSERTIONS** 14.1.2
Section **Calling subroutines** 14.3
Section **Building a counter** 14.7
Section **Clock Delay: What if you want** *variable* **clock delay?** 14.8.

Chapter 10
Recursive Property

Recursive property simply states that a condition holds. The property calls itself with a correct non-overlapping implication operator and correct antecedent and consequent relation. As shown in Fig. 10.1 (top "property rc1"), if 'ra' is true *and* at next clock rc1(ra) is true that 'rc1' should recur on itself. Note that the antecedent in 'rc1' is 1'b1, meaning the antecedent is always true. This allows for an easy and correct 'and' of an expression with an antecedent/consequent implication. Note also that you must use a non-overlapping operator in a recursive property.

The topmost example in Fig. 10.1 (baseP: assert property (@ (posedge clk) $fell (rst_) |-> rc1(bStrap)) else gotoFail;) specifies that when $fell(rst_) is true, the consequent rc1(bStrap) is invoked. 'rc1' property takes 'bStrap' as the input and does an 'and' of the input with (1'b1 |=> rc1(ra)). This means that the implication 1'b1 is always true and that rc1(ra) will be called every posedge of clk to recur on itself. The property will continue to go into loop on itself until bStrap is sampled 0. In that case the 'and' will fail and so would the property. In other words, we have checked to see that after $fell(rst_), the 'bStrap' (bootstrap signal) does not get de-asserted (i.e. go Low).

Note: If you use an overlapping operator, the property will recur on itself in zero time, essentially trapping the simulator in a zero delay loop causing simulation to hang.

But, what good does the property in Fig. 10.1 really do. It will check for a signal to hold forever. How do you apply such a model to real world? What you really need to check is that a condition holds until another condition remains true. If that happens, the property passes, else it fails. Now that is more practical. Let us understand this with the following example.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_10

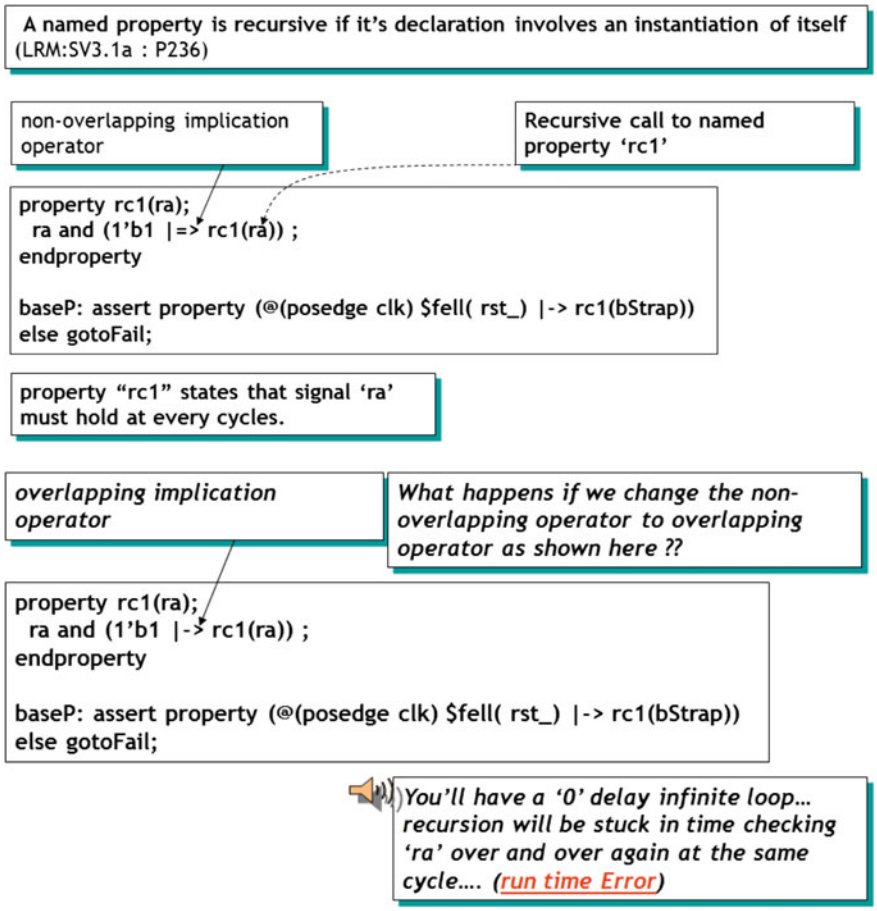


Fig. 10.1 Recursive property—basics

10.1 Application: Recursive Property

As shown in Fig. 10.2, with an 'or' condition, the recursive property becomes useful. The specification says that we need to make sure that 'intr' is held true until 'iack' is asserted. Let's see how that works. property rc1 says that either iack is true or intr is true that the property calls rc1 recursively. Now, if the 'intr' falls (goes low) before an iack, the **(intr and (`true |=> rc1(intr, iack)))** will fail because this is an AND with 'intr'. On the other hand, if iack arrives first, the "iack or …" condition will pass because this is an OR. In other words, we are recursive on 'intr' to see that it holds true until iack arrives.

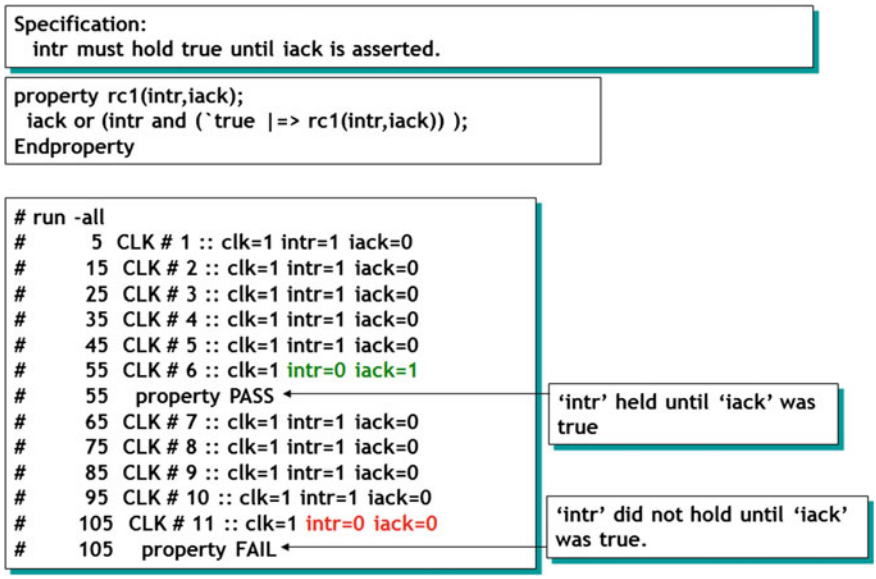


Fig. 10.2 Recursive property—application

At time 55 in the simulation log, iack=1 and intr=0. Since intr was equal to '1' the previous clock, it held itself until iack arrived. Hence, the property passes. On the other hand, at time 105, intr goes '0' *before* iack goes 1. In other words, intr did not hold itself until iack arrived and the property fails. Figure 10.3 shows another interesting application.

10.2 Application: Recursive Property

The specification of this property reads as "on detection of missAlloc, see that wdataH is held until readC is true". In Fig. 10.3, property 's_rc1' checks to see that 'misAlloc' is true, upon which it triggers 'rc1'. Property rc1 in turn holds true (i.e. recursive) until readC is true. If 'wdataH' goes low before readC goes high, the property (and hence the entire assertion) will fail (because wdataH is the LHS of an 'and' condition in sequence rc1). If readC arrives before wdataH, the 'or' condition in the sequence rc1 passes and hence the assertion will pass.

This way we have made sure that wdataH is held until readC completes. If this is not quite apparent at first, please refer to Fig. 10.2 to understand how 'property rc1' works.

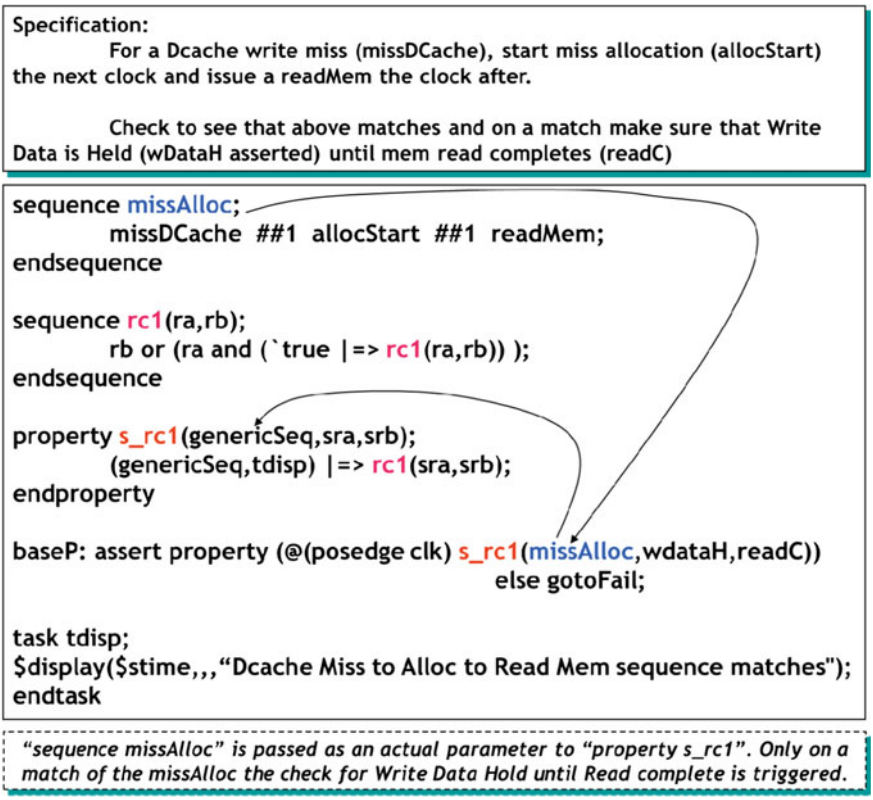


Fig. 10.3 Recursive property—application

Note that we are passing the entire sequence 'missAlloc' as an actual to the formal 'genericSeq' of property s_rc1. This is a very useful way to use a sequence as an actual to sequence formal.

Further nuances are described in Fig. 10.4 with annotations.

Figure 10.5 shows that 'disable iff' is not allowed in a recursive property. Well, there is a simple solution to that problem, which is shown in the bottom of the figure. Separate the requirement of 'disable iff' and the recursive nature of the property in two properties. The recursive property does not contain 'disable iff' and the 'property rillegal' disables rLegal with the 'disable iff' condition.

Figure 10.6 shows that two recursive properties can indeed be mutually recursive. 'cPhase2' calls 'cPhase1' and 'cPhase1' in turn calls 'cPhase2'. Why would this not end up in a zero delay loop? First of all, there is the 1 clock wait because of the non-overlapping operator in both properties. Second, each property has an antecedent and the property will execute only if the antecedent is true. So, if

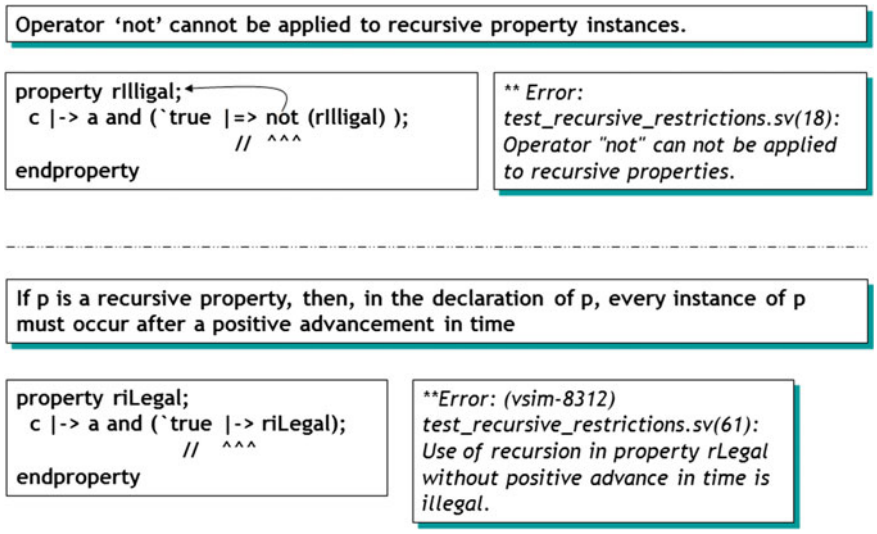


Fig. 10.4 Recursive property—further nuances I

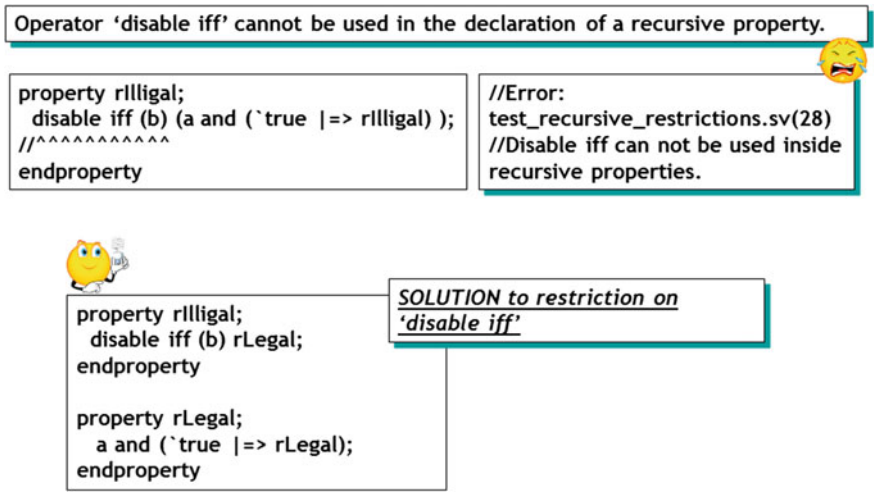


Fig. 10.5 Recursive property—further nuances II

'cPhase2' calls 'cPhase1', then 'cPhase1' will first wait for 'c' to be true and then trigger the recursive part of the property. The same happens when 'cPhase1' calls 'cPhase2'.

Also note that the operators accept_on, reject_on, sync_accept_on, and sync_reject_on (Sect. 16.16) have not been covered yet, but they may be used

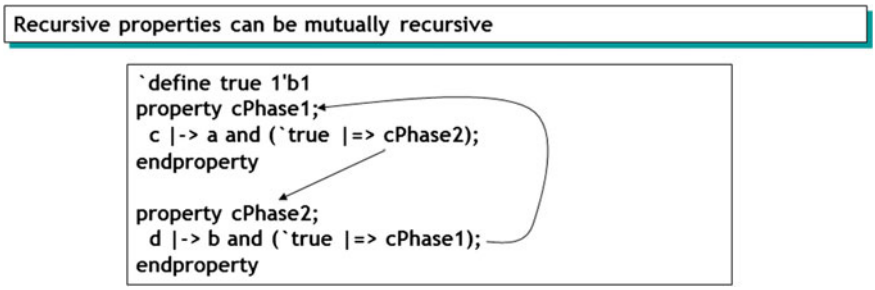


Fig. 10.6 Recursive property—mutually recursive

inside a recursive property. Also, strong operators s_nexttime, s_eventually, s_always, s_until, and s_until_with *cannot* be applied to any property expression that instantiates a recursive property.

Chapter 11
Detecting and Using Endpoint of a Sequence

Introduction: This chapter describes the endpoint sequence detection methods such as .triggered (.ended in 2005 LRM) and .matched and their operation with overlapping and non-overlapping operators. Legal, illegal conditions and plenty of examples and applications are presented.

11.1 .triggered (*Replaced for .ended*)

Before we learn how .triggered works, here's what has changed in the 1800-2009/2012 standard.

The 2009/2012 standard gets rid of .ended and in place supports .triggered. In other words, .triggered has the same exact meaning as .ended, only that .triggered can be used both where .ended gets used as well as where .triggered was allowed in previous versions. In other words, .triggered can be used in a sequence as well as in procedural block and also in level sensitive 'wait' statement.

Following from the LRM:

IEEE Std 1800-2005 17.7.3 required using the .ended sequence method in sequence expressions and the .triggered sequence method in other contexts. Since these two constructs have the same meaning but mutually exclusive usage contexts, in this version of the standard, the .triggered method is allowed to be used in sequence expressions, and the usage of .ended is deprecated and does not appear in this version of the standard.

Note that the entire discussion devoted to .triggered in this chapter applies directly to .ended. In other words you can replace .triggered with .ended in this chapter and you will get the same results.

If you are mainly interested in the *end* of a sequence regardless of when it started, .triggered is your friend. The main advantage of methods that detect the endpoint of a sequence is that you do *not* need to know the start of the sequence. All you care for is, when a sequence ends.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_11

Figure 11.1 shows that behavior. Sequence 'branch' is a complete sequence for a branch to complete. It could have started any time. The property endCycle wants to make sure that the 'branch' sequence has indeed ended when endBranch flag goes high. In other words, whenever $rose(endBranch) is detected to be true that the next clock, branch must end. This is indeed very powerful and useful feature. This makes the assertion intuitive as well.

But what if you simply write the assertion as "at the end of 'branch' see that endBranch goes high" as in "branch(a, b, c, d) |=> $rose(endBranch)". What's wrong with that? Well, what if $rose(endBranch) goes high when the 'branch' is still executing? That $rose(endBranch) would go unnoticed until the end of the sequence 'branch'. The .triggered operator would catch this. If endBranch goes high when sequence 'branch' is still executing, the property will fail. That's because the property endCycle expects 'branch' to have ended when endBranch goes high. Since endBranch could have risen prematurely, the property will see that at $rose(endBranch) the 'branch' sequence has not ended and the property would fail. The forward looking property would not catch this. This is a very important point. Please make a note of it.

Also, note that the clock in both the source and destination sequence must be the same. But what if you want the source and destination clocks to be different? That is what '.matched' does, soon to be discussed.

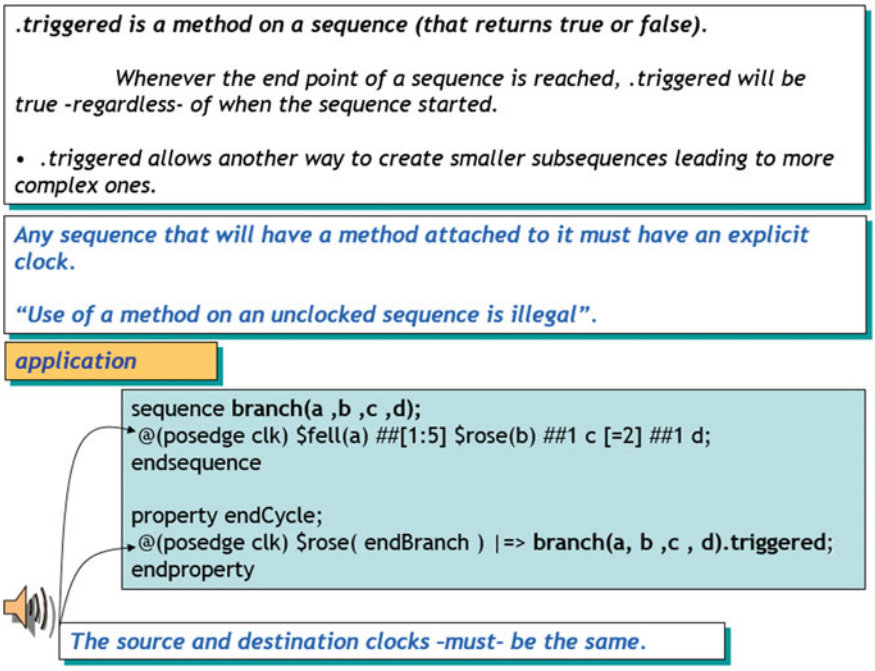


Fig. 11.1 .triggered—end point of a sequence

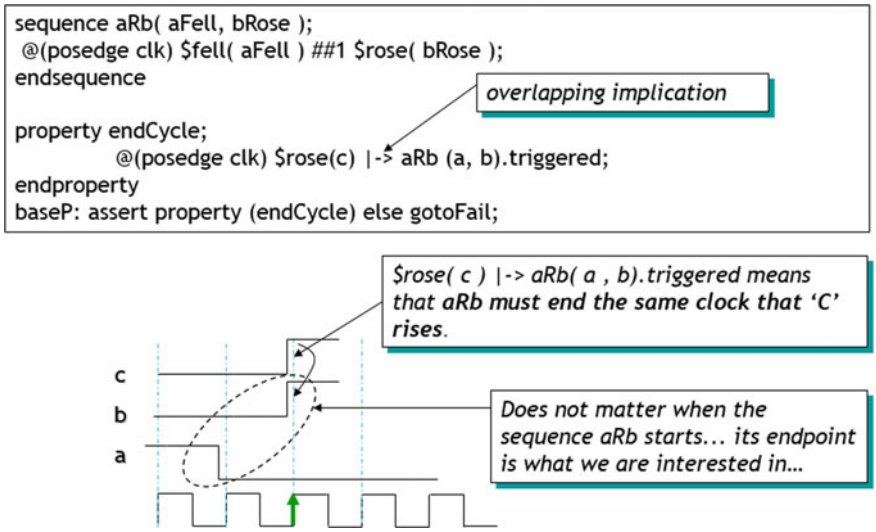


Fig. 11.2 .triggered with overlapping operator

Figure 11.2 explains further nuances.

In the example, $rose(c) implies that the sequence aRb must have ended. Key point to note here is that the implication operator is overlapping. This means that when $rose(c) occurs that at the same clock, sequence aRb should end. As shown in Fig. 11.2, when $rose(c) is true that at the same clock $rose(b) must occur and the previous clock $fell(a).

Figure 11.3 describes the same example but with the non-overlapping operator.

You have to understand this very carefully, as simple as it looks. Non-overlapping states that when $rose(c) is true that *at the next clock* the sequence aRb must end. Hence, as shown is the figure, the property looks for aRb to end one clock *after* $rose(c). This is intuitive but easy to miss.

Let us revisit an example I had presented in Fig. 6.15: **[*m:n] Consecutive Repetition Range—Application**. This figure is again depicted here for easy reference. Note that in this example, we see how a 'consecutive repetition' operator allowed us to design a property for state transition checks. What if you want to model the same property using .triggered?

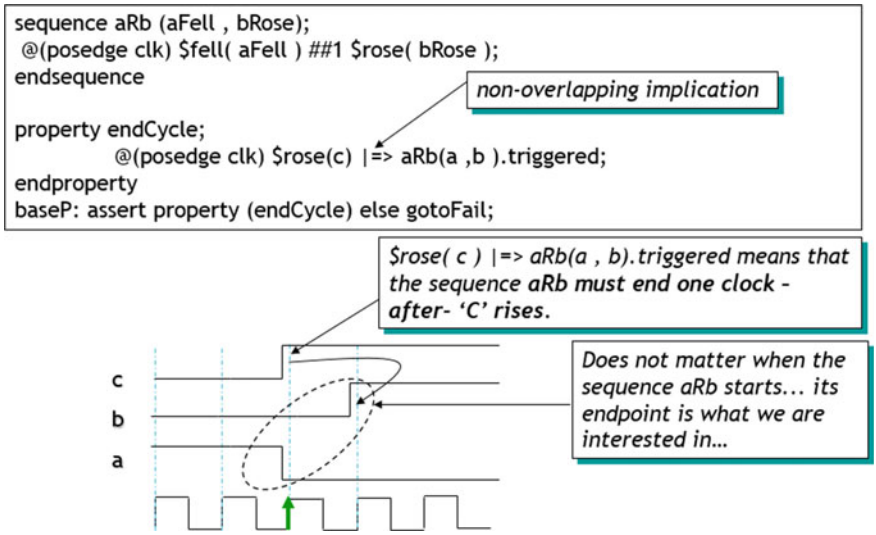


Fig. 11.3 .triggered with non-overlapping operator

First, state transition checks using 'consecutive repetition' operator—same as shown in Fig. 6.15.

application

Specification:

Make sure that the state machine follows the specified transitions

```
`define readStart (read_enb ##1 readStartState)
`define readID (readStartState ##1 readIDState)
`define readData (readIDState ##1 readDataState)
`define readEnd (readDataState ##1 readEndState)

sequence checkReadStates;

  @(posedge clk)
   `readStart              ##1
   `readID         [*1:$]  ##1
   `readData       [*1:$]  ##1
   `readEnd        ;

endsequence
```

Next, the same property written using '.triggered' method.

```
Specification:
Make sure that the state machine follows the specified transitions

sequence readStart; @(posedge clk) read_enb ##1 readStartState; endsequence
sequence readID; @(posedge clk) readStartState ##1 readIDState; endsequence
sequence readData; @(posedge clk) readIDState ##1 readDataState; endsequence
sequence readEnd; @(posedge clk) readDataState ##1 readEndState; endsequence

property checkReadStates;
  @(posedge clk)
          readStart.triggered  ##[1:$]
          readID.triggered     ##[1:$]
          readData.triggered   ##[1:$]
          readEnd
          ;
endproperty

sCheck: assert property (checkReadStates) else $display ($stime,,,"FAIL");
cCheck: cover property (checkReadStates) $display ($stime,,,"PASS");
```

Let us examine the property using .triggered. Instead of using ‘define, we have to explicitly declaring different sequences for state transitions (because the .triggered method can only be attached to a named sequence). Also, note that in each sequence there is an explicit @(posedge clk). This is because .triggered method cannot be attached to a non-clocked sequence. The main property checkReadStates shows how .triggered is used in place of the consecutive repetition operator. We wait for each sequence to ‘end’ and see that the next sequence then ends within 1 clock to whenever, since we don’t know how long will the next state transition (e.g. from readStart to readID) will take place. This is the reason for ##[1:$] after each .triggered sequence. And that each of these ‘ends’ occur in a specified order. This example also illustrates how multiple sequences are used in a property and as I’ve said before, it is best to break down a complex property (as this) in multiple small sequences and then build the master property.

Here’s the entire testbench with the property and the simulation log. Please study the simulation log to solidify your concept of .triggered.

```
module state_transition;
int readStartState, readIDState, readDataState, readEndState;
logic clk, read_enb;

sequence readStart; @(posedge clk) read_enb ##1 readStartState; endsequence
sequence readID; @(posedge clk) readStartState ##1 readIDState; endsequence
sequence readData; @(posedge clk) readIDState ##1 readDataState; endsequence
sequence readEnd; @(posedge clk) readDataState ##1 readEndState; endsequence

property checkReadStates;
 @(posedge clk)
        readStart.triggered     ##[1:$]
        readID.triggered        ##[1:$]
        readData.triggered      ##[1:$]
        readEnd
        ;
endproperty

sCheck: assert property (checkReadStates) else $display ($stime,,,"FAIL");
cCheck: cover property (checkReadStates) $display ($stime,,,"PASS");

initial
begin
        read_enb=1; clk=0;
        @(posedge clk) readStartState=1;
        @(posedge clk) readIDState=1;
        @(posedge clk) @(posedge clk); readDataState=1;
        @(posedge clk) @(posedge clk); readEndState=1;
end
```

```
initial $monitor($stime,,,"clk=",clk,,
                    "read_enb=%0b",read_enb,,,
                    "readStartState=%0b",readStartState,,
                    "readIDState=%0b",readIDState,,
                    "readDataState=%0b",readDataState,,
                    "readEndState=%0b",readEndState);

always #10 clk=!clk;
endmodule

/*
#       10  clk=1 read_enb=1  readStartState=1 readIDState=0 readDataState=0 readEndState=0
#       20  clk=0 read_enb=1  readStartState=1 readIDState=0 readDataState=0 readEndState=0
#       30  clk=1 read_enb=1  readStartState=1 readIDState=1 readDataState=0 readEndState=0
#       40  clk=0 read_enb=1  readStartState=1 readIDState=1 readDataState=0 readEndState=0
#       50  clk=1 read_enb=1  readStartState=1 readIDState=1 readDataState=0 readEndState=0
#       60  clk=0 read_enb=1  readStartState=1 readIDState=1 readDataState=0 readEndState=0
#       70  clk=1 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=0
#       80  clk=0 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=0
#       90  clk=1 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=0
#      100  clk=0 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=0
#      110  clk=1 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=1
#      120  clk=0 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=1
#      130  PASS
#      130  clk=1 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=1
#      140  clk=0 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=1
#      150  PASS
#      150  clk=1 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=1
#      160  clk=0 read_enb=1  readStartState=1 readIDState=1 readDataState=1 readEndState=1
#      170  PASS
*/
```

The .triggered method may be applied to a named sequence instance, with or without arguments, an untyped formal argument, or a formal argument of type sequence, as follows:

sequence_instance.triggered
or
formal_argument_sequence.triggered

When method .triggered is evaluated in an expression, it tests whether its operand sequence has reached its end point at that particular point in time. The result of .triggered does not depend upon the starting point of the match of its operand sequence.

Here's an example of .triggered usage in a procedural assignment using level sensitive control.

```
sequence busGnt;
 @ (posedge clk) req ##[1:5] gnt;
endsequence
initial begin
 wait (busGnt.triggered) $display($stime,,, "Bus Grant given");
 end
 Here's an example that shows the use of .triggered in sequences.
 sequence abc;
  @ (posedge gclk) a ##[1:5] ##1 b [*5] ##1 c;
 endsequence
 sequence myseq;
  @ (posedge gclk) d ##1 abc.triggered ##1 !d;
 endsequence
```

Another example:

Upon detection of IRDY End, Tabort must remain de-asserted until Frame_ End. Solution:

```
sequence IRDY;
        IRDY_start ##2 IRDY_end;
endsequence
sequence Frame_;
        Frame_Start ##[1:16] Frame_end;
endsequence
busyPr: assert property (@(posedge clk)
        IRDY |-> !Tabort until Frame_.triggered;
```

Finally, following is illegal.

```
logic x,y,z;
//....
Xillegal: assert property (@(posedge clk)
        Z |-> (x ## 3).triggered
```

Why is this illegal? Recall that .triggered can only be applied to a 'named sequence instance' or a formal argument. (x ## 3) is neither of the two.

Following is illegal as well because the clocks on two sides of the implication differ. They need to be the same.

```
sequence clk1Seq;
 @(posedge clk1) x ##2 y;
endsequence
Zillegal: assert property (@(posedge clk) z |-> clk1Seq.triggered);
```

Also, note that .triggered can be used as a subexpression. Here's an example:

```
default clocking @(posedge clk); endclocking
sequence ab;
 a ##2 b;
endsequence
sequence cd;
 c ##2 d;
endsequence
sequence cdtr;
 z ##[2:5] cd.triggered;
endsequence
ftr: assert property (ab |-> nogo until cdtr.triggered);
```

The order of execution for the consequent is cd, cdtr, ftr. Note that the order of sequence evaluation is statically determined at compile time. A proper order of evaluation must always be found.

11.2 .matched

The main difference between .triggered and .matched is that .triggered requires both the source and destination sequences have the same clock. .matched allows you to have different clocks in the source and destination sequences.

Since the clocks can be different, understanding of .matched gets a bit complicated. But it follows the same rules as that for multiply clocked properties. As shown in Fig. 11.4, sequence 'e1' uses 'clk' as its clock while sequence 'e2' uses

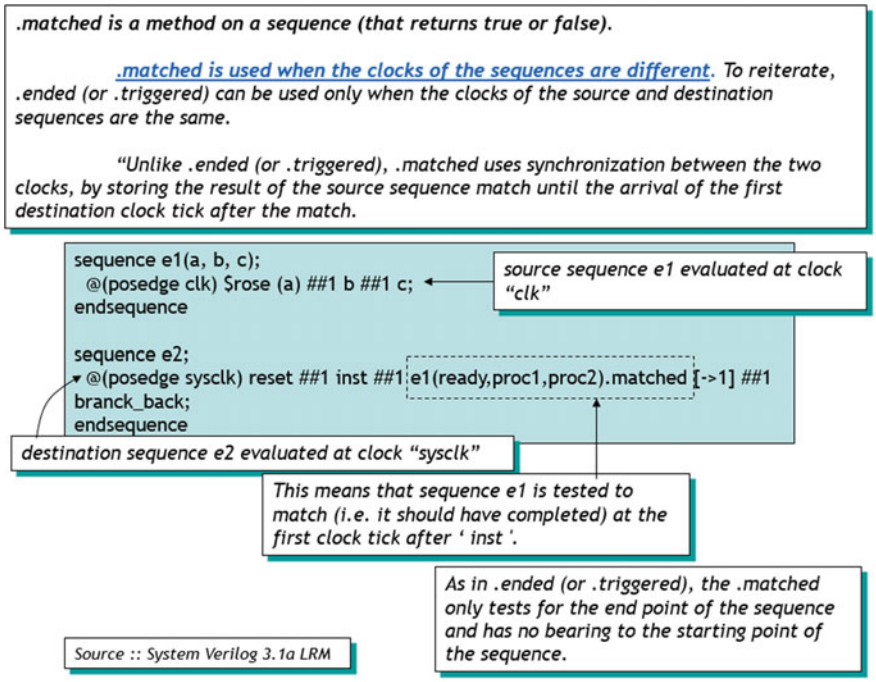


Fig. 11.4 .matched—basics

'sysclk' as it's clock. Sequence 'e2' says that after 'reset'; 1 clock later; 'inst' must be true and 1 clock later sequence 'e1' must match (i.e. end) at least once and 1 clock later branch_back must be true. This is very interesting way of 'inserting' a .matched (or .triggered for that matter) within a sequence. Sequence 'e1' is running on its own. What we really care for is that it matches (ends) when we expect it to.

Let us look at some examples that will make the concept clearer.

Figure 11.5 shows that on the rising edge of 'c' we want to see that aRb sequence matches. Let us look at the timing diagram. When, $rose(c) is true at posedge of clk1 (clk1 = 3), it looks for the end of the sequence 'aRb'. Sequence 'aRb' started during clk = 3 and it ends at clk = 4. Note that the end of 'aRb' [i.e. match of $rose(bRose)] is exactly at the very *next* 'clk' edge after clk1 edge. That's what the property asked for "@ posedge clk1) $rose (c) |=> @ (posedge clk) aRb(a, b).matched;"

The key point to note here (as in multiply clocked properties) is the clock crossing boundary. From clk1 to clk with 1 clock delay in-between (as implied by |=> non-overlapping implication) does not mean 1 full clock. It simply means the *very next edge* of 'posedge clk' after 'posedge clk1'. Please refer to multiply clocked properties (Sect. 8.1) if this concept is not clear.

Figure 11.6 uses overlapped implication and identical (in-phase) clocks (the clocks have to be the same because we are using overlapping implication). Hence when $rose(c) is true, it looks for the very *next* (posedge clk) overlapping with its

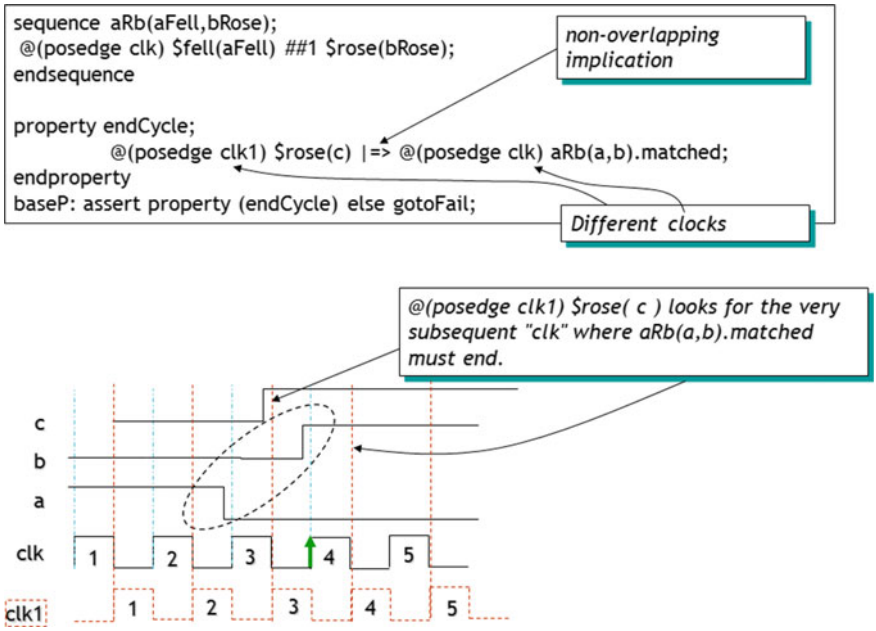


Fig. 11.5 .matched with non-overlapping operator

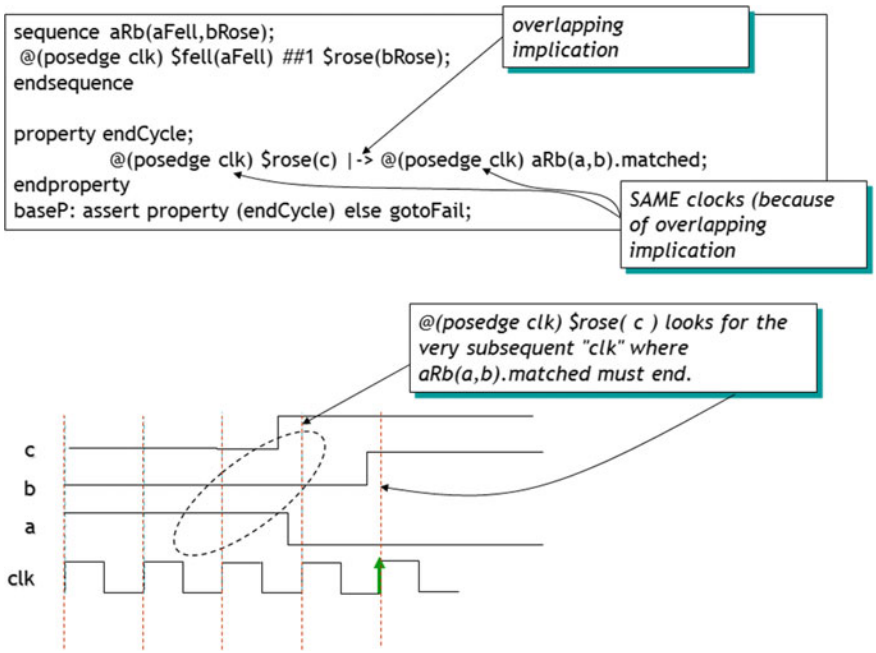


Fig. 11.6 .matched—overlapped operator

(posedge clk). But the very next (posedge clk) is (obviously) 1 clock later. So, 1 clock after $rose(c), the sequence sees that aRb has matched (i.e. ended).

11.2.1 Application: .matched

In Fig. 11.7, sequence 'RdS' uses Busclk while the property checkP uses sysclk. "@ (negedge sysclk) RdS.matched" means that at the negedge of sysclk, the sequence RdS (which is running off of Busclk) must end. 'RdS' could have completed slightly (i.e. when the very preceding posedge of Busclk would have arrived) earlier than the negedge sysclk. That is ok because we are transitioning from Busclk to sysclk (as long as the sequence RdS completes at the *immediately preceding* posedge Busclk).

Let us now look at how clock 'flows' or gets inferred in sequences and properties that use .triggered and .matched methods.

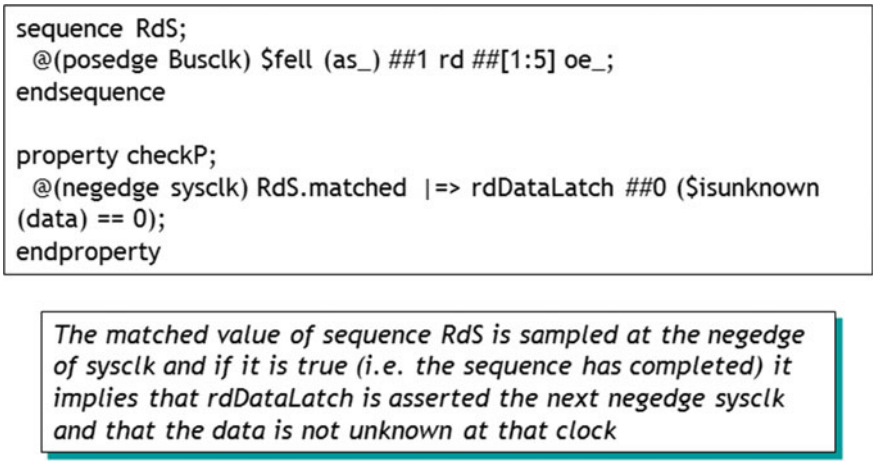


Fig. 11.7 .matched—application

```
module clock_inference;
logic a, b, c, d;

default clocking cb @(posedge clk_d); endclocking

sequence e4;
        $rose(b) ##1 c;
endsequence

a1: assert property (@(posedge clk_a) a |=> e4.triggered);
//Since the leading edge in 'a1' is posedge clk_a, 'e4' infers posedge clk_a as per clock flow rules

sequence e5;
        @(posedge clk_e1) a ##[1:3] e4.triggered ##1 c;
endsequence
/* Note that the leading clock edge here is 'clk_e1'. So, 'e4' will infer posedge clk_e1 as per clock flow rules
   wherever e5 is instantiated (with/without a method)
*/

a2: assert property (@(posedge clk_a) a |=> e5.matched);
/* Here, 'a2' triggers 'e5.matched' with the leading clock as 'posedge clk_a'. 'e5' in turn triggers 'e4.triggered'
   with the leading clock as 'posedge clk_e1'. Hence, by clock flow rules, 'e4' infers 'posedge clk_e1' from
   sequence 'e5'.
*/

sequence e6(f);
        @(posedge clk_e2) f;
endsequence

a3: assert property (@(posedge clk_a) a |=> e6(e4.triggered));
/*Similar to argument above, the clock flows from 'posedge clk_a' to 'posedge clk_e2' and sequence 'e4' infers
'posedge clk_e2'
*/

always @(posedge clk_a) begin
        @(e4);
        d = a;
end

/* This is very interesting and important to understand. Sequence 'e4' infers 'default clocking cb @(posedge
clk_d); endclocking' and not posedge clk_a as there is more than one event control in this procedure.
*/

endmodule
```

Chapter 12
'expect'

Introduction: This chapter describes the procedurally blocking statement 'expect' and its differences with the 'assert' statement.

'expect' takes on the same syntax (not semantics) as 'assert property' in a procedural block. *Note that 'expect' must be used only in a procedural block*. It cannot be used outside of a procedural block as in 'assert' or property/sequence (recall that 'assert property' can be used both in the procedural block as well as outside). So, what's the difference between 'assert property' and 'expect'?

'expect' is a blocking statement while 'assert property' is a non-blocking statement. Blocking means, the procedural block will wait until 'expect' sequence completes (pass or fail). For 'assert property' non-blocking means that the procedural block will trigger the 'assert property' statement and continue with the next statement in the block. 'assert property' condition will continue to execute in parallel to the procedural code. Note that 'assert property' behavior is the same whether it's outside or inside a procedural block. It always executes in parallel on its own thread with the rest of the logic.

Please refer to the simulation log in Fig. 12.1. You notice that the procedural code waits for 'expect' to complete (i.e. blocks execution of procedural code) and only on completion of 'expect' that it executes the subsequent $display. Figure 12.2 highlights further nuances of 'expect' semantics. Annotations in the figure are self-explanatory. *Key point is that 'expect' does not inherit a clock from its preceding procedural clock*. It needs an explicit clock in the sequence (or property) it 'expects' or with its own declaration.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_12

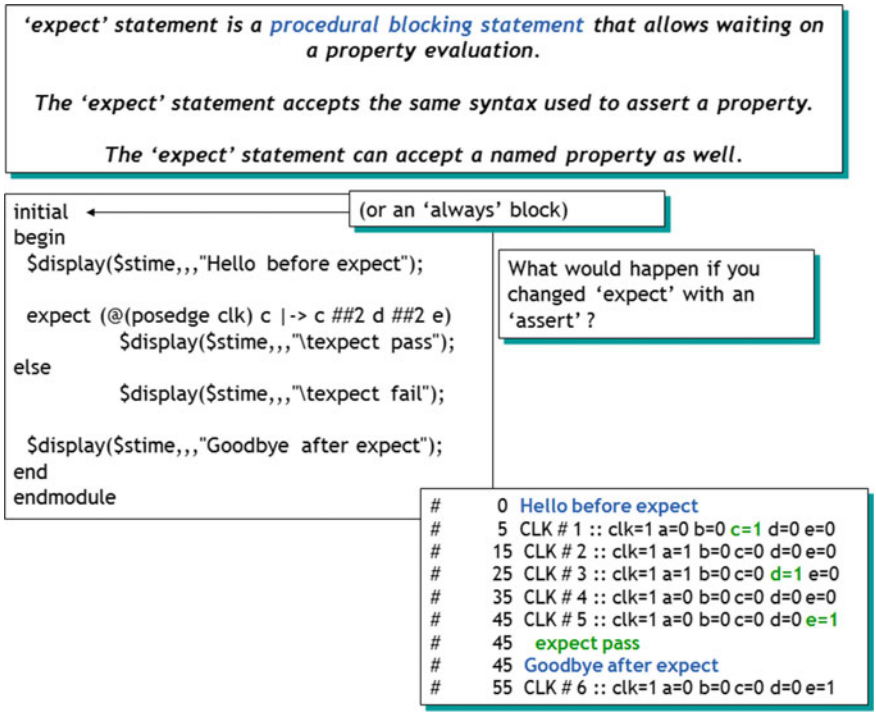


Fig. 12.1 'expect'—basics

Finally,

The expect statement can appear anywhere a wait statement can appear. Because it is a blocking statement, the property can refer to automatic variables as well as static variables.

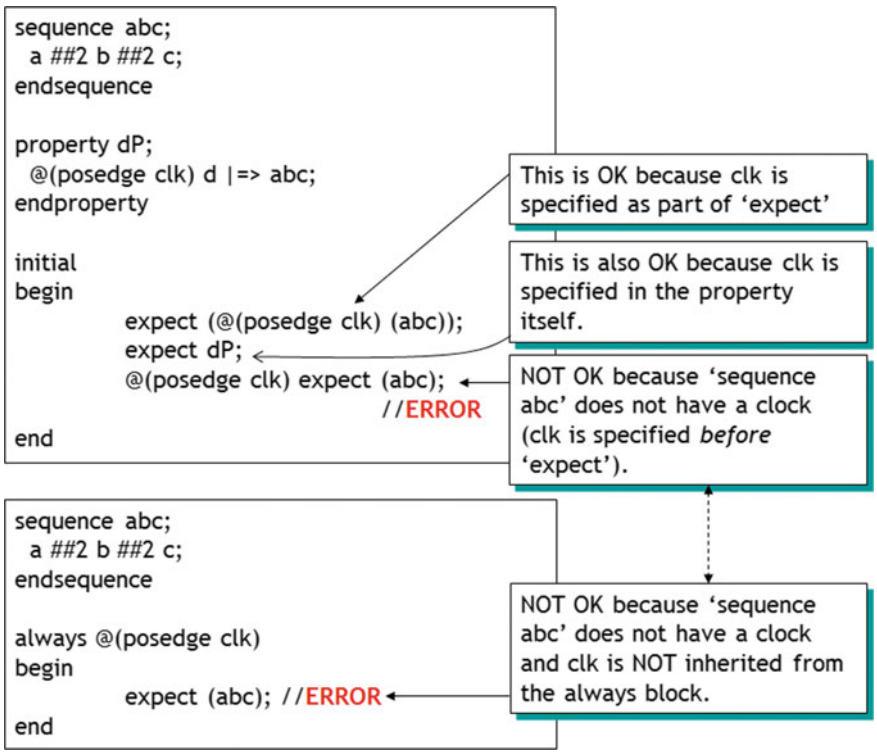


Fig. 12.2 'expect'—error conditions

For example, the task below waits between 1 and 10 clock ticks for the variable data to equal a particular value, which is specified by the automatic argument value. The second argument, 'success', is used to return the result of the expect statement: 1 for success and 0 for failure.

```
integer data;
...
task automatic wait_for (integer value, output bit success);
        expect (@(posedge clk) ##[1:10] data == value) success = 1;
        else success = 0;
endtask

initial begin
        bit ok;
        wait_for (23, ok); // wait for the value 23
...
end
```

Chapter 13
'assume' and Formal (Static Functional) Verification

Introduction: This chapter describes 'assume' statement and its usage for 'static formal' (or 'static functional') and 'constrained random' methodologies.

This is an interesting operator. 'assume' specifies the property as an assumption for the environment. They may be used by simulators to constrain the random generation of free checker variable values or by formal tools to constrain the formal computation. The most useful environment for 'assume' is that of static formal verification. Static formal is a method whereby the formal algorithm exercises all possible combinational and temporal possibilities of inputs to exercise all possible 'logic cones' of a given logic block and checks to see that the assertion holds. During such verification if you do not specify any constraints (i.e. for a 5 input (a, b, c, d, e) block, if you don't specify any constraints such as 'assume' a = 0 and b = 1) then the static formal will try to explore all possible combinations of the 5 input both in combinatorial and temporal (if required) domain. Without any constraints provided via 'assume', the static formal tool may experience something called 'state space explosion' problem. As the description suggests, the tool may give up if too many inputs are unconstrained. This is where the 'assume' statement comes into picture.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_13

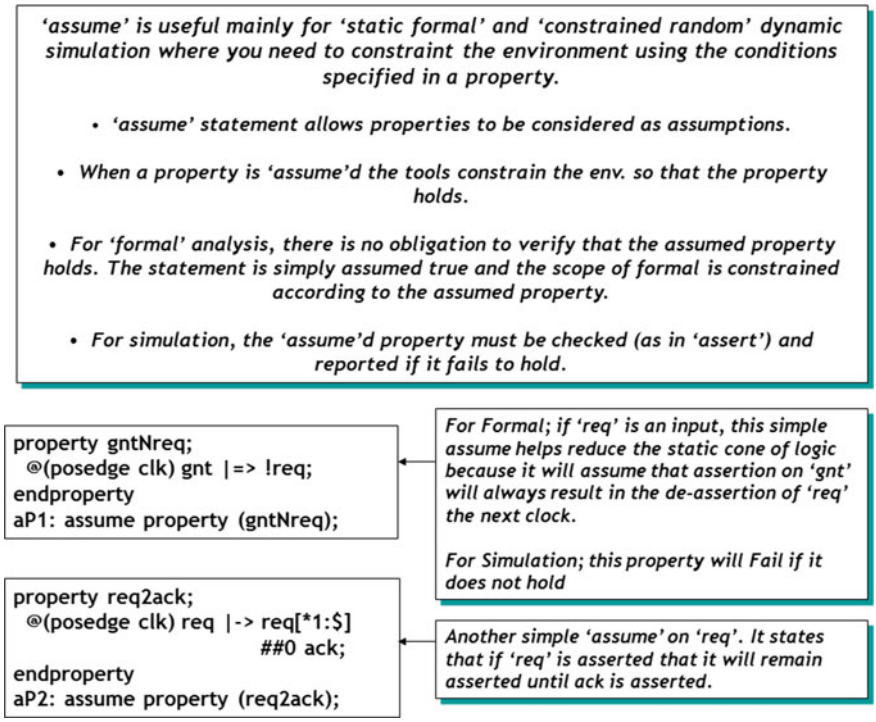


Fig. 13.1 'assume' and formal verification

So, how does it behave in simulation? The examples in Fig. 13.1 simply shows that 'assume' without any other property reliant on assumed property will act like 'assert'. If the property associated with the assume statement is found to be false, the simulation fails.

Chapter 14
Very Important Topics and Applications

Introduction: This chapter addresses many important topics such as Asynchronous FIFO assertions, triggering concurrent assertions from procedural blocks, calling subroutines, sequences as formal arguments, as antecedent and as triggering condition in a sensitivity list. It also describes further nuances such as how to design 'variable delay' using a 'counter', effects of blocking nature of an 'action' (pass/fail) block, cyclic dependencies, vacuous pass of an assertion, empty sequences, etc.

14.1 Asynchronous FIFO Assertions

Asynchronous FIFO (compared to synchronous FIFO) is a difficult proposition when it comes to writing assertions. The Read and the Write clocks are asynchronous which means the most important property to check for is data transfer from Write to Read clock. Other assertions are to check for fifo_full, fifo_empty, etc. conditions.

First we present a comprehensive design of the asynchronous fifo. A bit complicated but you don't need to go into its detail. Next we see a testbench within which I have designed the assertions. Yes, you can have assertions in the design (RTL) (but not recommended), testbench (as in this example), in a systemverilog Interface, in a systemverilog Program and in a file on its own [this is where the 'bind' comes into picture (this is highly recommended)].

This FIFO design and testbench are available on Springer server.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_14

14.1.1 Asynchronous FIFO Design

```
module asynchronous_fifo (
      // Outputs
     fifo_out, full, empty,
      // Inputs
      wclk, wclk_reset_n, write_en,
      rclk, rclk_reset_n, read_en,
      fifo_in
  );

  `define FF_DLY 1'b1
  parameter    D_WIDTH = 20;
  parameter    D_DEPTH = 4;
  parameter    A_WIDTH = 2;

  input                    wclk_reset_n;
  input                    rclk_reset_n;
  input                    wclk;
  input                    rclk;
  input                    write_en;
  input                    read_en;
  input [D_WIDTH-1:0]  fifo_in;

  output [D_WIDTH-1:0]          fifo_out;
  output                        full;
  output                        empty;

  reg [D_WIDTH-1:0]     reg_mem[0:D_DEPTH-1];
  reg [A_WIDTH:0]       wr_ptr;
  reg [A_WIDTH:0]       wr_ptr_gray;
  reg [A_WIDTH:0]       wr_ptr_gray_rclk_q;
  reg [A_WIDTH:0]       wr_ptr_gray_rclk_q2;
  reg [A_WIDTH:0]       rd_ptr;
  reg [A_WIDTH:0]       rd_ptr_gray;
  reg [A_WIDTH:0]       rd_ptr_gray_wclk_q;
  reg [A_WIDTH:0]       rd_ptr_gray_wclk_q2;

  reg                   full;
  reg                   empty;

  wire [A_WIDTH:0]      nxt_wr_ptr;
  wire [A_WIDTH:0]      nxt_rd_ptr;
  wire [A_WIDTH:0]      nxt_wr_ptr_gray;
  wire [A_WIDTH:0]      nxt_rd_ptr_gray;
  wire [A_WIDTH-1:0]    wr_addr;
  wire [A_WIDTH-1:0]    rd_addr;
  wire                  full_d;
  wire                  empty_d;

  assign wr_addr = wr_ptr[A_WIDTH-1:0];
  assign rd_addr = rd_ptr[A_WIDTH-1:0];

  always @ (posedge wclk)
    if (write_en) reg_mem[wr_addr] <= #`FF_DLY fifo_in;

  assign fifo_out = reg_mem[rd_addr];

  always @ (posedge wclk or negedge wclk_reset_n)
    if (!wclk_reset_n) begin
```

```
      wr_ptr <= #`FF_DLY {A_WIDTH+1{1'b0}};
      wr_ptr_gray <= #`FF_DLY {A_WIDTH+1{1'b0}};
    end else begin
      wr_ptr <= #`FF_DLY nxt_wr_ptr;
      wr_ptr_gray <= #`FF_DLY nxt_wr_ptr_gray;
    end

  assign nxt_wr_ptr = (write_en) ? wr_ptr+1 : wr_ptr;
  assign nxt_wr_ptr_gray = ((nxt_wr_ptr>>1) ^ nxt_wr_ptr);

  always @ (posedge rclk or negedge rclk_reset_n)
    if (!rclk_reset_n) begin
      rd_ptr <= #`FF_DLY {A_WIDTH+1{1'b0}};
      rd_ptr_gray <= #`FF_DLY {A_WIDTH+1{1'b0}};
    end else begin
      rd_ptr <= #`FF_DLY nxt_rd_ptr;
      rd_ptr_gray <= #`FF_DLY nxt_rd_ptr_gray;
    end

  assign nxt_rd_ptr = (read_en) ? rd_ptr+1 : rd_ptr;
  assign nxt_rd_ptr_gray = (nxt_rd_ptr>>1) ^ nxt_rd_ptr;

  // check full
  always @ (posedge wclk or negedge wclk_reset_n)
    if (!wclk_reset_n)
      {rd_ptr_gray_wclk_q2, rd_ptr_gray_wclk_q} <= #`FF_DLY {{A_WIDTH+1{1'b0}}, {A_WIDTH+1{1'b0}}};
    else
      {rd_ptr_gray_wclk_q2, rd_ptr_gray_wclk_q} <= #`FF_DLY {rd_ptr_gray_wclk_q, rd_ptr_gray};

  assign full_d = (nxt_wr_ptr_gray == {~rd_ptr_gray_wclk_q2[A_WIDTH:A_WIDTH-1],
rd_ptr_gray_wclk_q2[A_WIDTH-2:0]});

  always @ (posedge wclk or negedge wclk_reset_n)
    if (!wclk_reset_n)
      full <= #`FF_DLY 1'b0;
    else
      full <= #`FF_DLY full_d;

  // check empty
  always @ (posedge rclk or negedge rclk_reset_n)
    if (!rclk_reset_n)
      {wr_ptr_gray_rclk_q2, wr_ptr_gray_rclk_q} <= #`FF_DLY {{A_WIDTH+1{1'b0}}, {A_WIDTH+1{1'b0}}};
    else
      {wr_ptr_gray_rclk_q2, wr_ptr_gray_rclk_q} <= #`FF_DLY {wr_ptr_gray_rclk_q, wr_ptr_gray};

  assign empty_d = (nxt_rd_ptr_gray == wr_ptr_gray_rclk_q2);

  always @ (posedge rclk or negedge rclk_reset_n)
    if (!rclk_reset_n)
      empty <= #`FF_DLY 1'b1;
    else
      empty <= #`FF_DLY empty_d;

endmodule
```

14.1.2 Asynchronous FIFO Testbench and Assertions

```
module test_asynchronous_fifo
(
     fifo_out, full, empty,
     wclk, wclk_reset_n, write_en,
     rclk, rclk_reset_n, read_en,
     fifo_in
);

  parameter   D_WIDTH = 20;
  parameter   D_DEPTH = 4;
  parameter   A_WIDTH = 2;

  output            wclk_reset_n;
  output            rclk_reset_n;
  output            wclk;
  output            rclk;
  output            write_en;
  output            read_en;

  output [D_WIDTH-1:0] fifo_in;

  logic                    wclk_reset_n;
  logic                    rclk_reset_n;
  logic                    wclk;
  logic                    rclk;
  logic                    write_en;
  logic                    read_en;
  logic [D_WIDTH-1:0]  fifo_in;

  input [D_WIDTH-1:0]  fifo_out;
  input                    full;
  input                    empty;

asynchronous_fifo aff1
(
    fifo_out, full, empty,

     wclk, wclk_reset_n, write_en,
     rclk, rclk_reset_n, read_en,
     fifo_in
);

/*
```

Following property checks to see if the FIFO is full that the wr_ptr does not change.

This assertion can be written other ways too (using for example ‘empty’ condition). Please try them out and see if the results match with this assertion.

```
*/
```

```
property check_full;
  @ (posedge wclk) disable iff (!wclk_rstn)
      (full) |=> @ (posedge wclk) aff1.wr_ptr == $past (aff1.wr_ptr);
endproperty
cfull : assert property (check_full) else $display($stime,,,"%m Check wr_ptr full FAIL");
cfullc : cover property (check_full) $display($stime,,,"%m Check wr_ptr full PASS");
```

/*

Following property checks to see that if the FIFO is empty that the rd_ptr does not change. 'empty' means that the rd_ptr remains the same as its value at the last clk—thus guaranteeing that the rd_ptr have not changed. BUT note that we are using !$isunknown and passing it $past as an expression. Why? If FIFO is empty, the rd_ptr in the past could be 'X'. So we make sure that rd_ptr is the same as the past value and that it is not unknown.

This assertion can be written other ways too (using for example 'full' condition or without the use of $past). Please try them out and see if the results match with this assertion.

*/

```
property check_empty;
  @ (posedge rclk) disable iff (! rclk_rstn)
      (empty) |=> @ (posedge rclk)
        if (!$isunknown( $past (aff1.rd_ptr)))
          (aff1.rd_ptr == $past (aff1.rd_ptr));
endproperty
cempty : assert property (check_empty) else $display($stime,,,"%m Check rd_ptr empty FAIL");
cemptyc : cover property (check_empty) $display($stime,,,"%m Check rd_ptr empty PASS");

/*----------------------------------------
  ASYNCHRONOUS DATA TRANSFER CHECK
----------------------------------------*/
```

/*

This is a very important assertion for an asynchronous FIFO. Check that the data that is written at a wr_ptr is the same data that is read when rd_ptr reaches that wr_ptr. Simple! Let us look at the assertion step by step

*/

/*

In the assertion, data_check property checks to see that FIFO is not full. If so, saves wr_ptr into the local variable 'ptr' and the data from fifo into local variable 'data' and display that so that we can easily see how the assertion is progressing during simulation.

If the antecedent is true, the consequent says that the first match of rd_ptr being the same as wr_ptr (note wr_ptr was stored in local variable ptr) that the read data is the same as the write data (note write data were stored in local variable data in the antecedent).

Sequence rd_detect(ptr) is used as an expression to first_match. It says that wait from now until forever until you detect a read and it's rd_ptr is equal to the wr_ptr (which is stored in the local variable 'ptr' in the antecedent).

Please note that these are multi-clocked properties since we have a wr_clk and a rd_clk.

So, in this property, we see

- multi-clock property (wclk and rclk)
- use of local variables for storing/comparing
- first_match
- attaching a subroutine (here a $display) to an expression for effective debugging
- effective use of ##0 to create overlapping condition in a sequence.

Try out different ways to write the assertion. Refer to different operators, sampled value functions etc. and see if you can write an equivalent assertion. There isn't just one way to write an assertion. But there is indeed a right way and a wrong way. You will get that through practice.

*/

```
sequence rd_detect(ptr);
  ##[0:$] (read_en && !empty && (aff1.rd_ptr == ptr));
endsequence

property data_check(wrptr);
integer ptr, data;
  @ (posedge wclk) disable iff (!wclk_reset_n || !rclk_reset_n)
        (write_en && !full, ptr=wrptr, data=fifo_in,
     $display($stime,"\t Assertion Disp wr_ptr=%h data=%h", aff1.wr_ptr, fifo_in))

|=>

  @ (negedge rclk) first_match(rd_detect(ptr),
    $display($stime,,," Assertion Disp FIRST_MATCH ptr=%h Compare data=%h fifo_out=%h", ptr,
data, fifo_out))
        ##0 (fifo_out === data);
endproperty

dcheck : assert property (data_check(aff1.wr_ptr)) else  $display($stime,,,"FAIL: DATA
CHECK");
dcheckc : cover property (data_check(aff1.wr_ptr)) $display($stime,,,"PASS: DATA CHECK");

/*-----------------------------------------
  If FULL -> NOT EMPTY Check
-----------------------------------------*/
```

/*

Following property is quite self-explanatory. But note that this is also a multi-clocked property and since the write and read clocks are different clocks, we must use the non-overlapping operator |=>.

This is a mutex property. What are the different ways you can write this?

*/

```
property full_empty;
 @ (posedge wclk) disable iff (!wclk_reset_n)
        @ (posedge wclk) (full) |=> @ (posedge rclk) (!empty);
        endproperty

        few: assert property (full_empty) else $display($stime,," FAIL: Full and Empty BOTH asserted");
        cfew: cover property (full_empty) $display($stime,," PASS: Full and Empty check ");

/*-----------------------------------------
  If EMPTY -> NOT FULL Check
-----------------------------------------*/
/*
```

Following property is quite self-explanatory. But note that this is also a multi-clocked property and since the write and read clocks are different clocks, we must use the non-overlapping operator |=>.

This is a mutex property. What are the different ways you can write this?

*/

```
property empty_full;
 @ (posedge wclk) disable iff (!wclk_reset_n)
        @ (posedge rclk) (empty) |=> @ (posedge wclk) (!full);
        endproperty

        efw: assert property (full_empty) else $display($stime,,, "FAIL: Full and Empty BOTH
        asserted");

/*-----------------------------------------
  rclk_reset_n Check on rclk
-----------------------------------------*/
/*
```

Following property checks to see that empty pointer is high (i.e. empty) when you reset the FIFO

*/

```
        property reset_n_rclk;
          @ (posedge rclk) !rclk_reset_n |-> empty;
        endproperty

        reset_nrclkA: assert property (reset_n_rclk) else $display($stime,,,"FAIL: FIFO not empty
        during rclk_reset_n");
        reset_nrclkC: cover property (reset_n_rclk) $display($stime,,,"PASS: FIFO empty during
        rclk_reset_n");


 /*-----------------------------------------
   wclk_reset_n Check on wclk
 -----------------------------------------*/

/*
```

Following property checks to see that the FIFO is not full when you reset the FIFO. FIFO can only go Empty during reset, not Full.

```
*/

        property reset_n_wclk;
          @ (posedge wclk)  !wclk_reset_n |-> !full;
        endproperty

        reset_nwclkA: assert property (reset_n_wclk) else $display($stime,,,"FAIL: FIFO FULL during
        wclk_reset_n");
        reset_nwclkC: cover property (reset_n_wclk) $display($stime,,,"PASS: FIFO FULL during
        rclk_wstn");

 endmodule
```

14.1.3 Test the Testbench

module

```
/*
```

Following assertions are important to note. We are checking our own testbench! Yes, that is important. This is a simple testbench but the idea is that as your testbench develops complex code that there is a good chance you will make mistakes. So why not use assertions to catch those errors as well.

For example, in the first assertion we are checking that if FIFO is full that we do not keep writing to it, since this particular FIFO does not have a pushback signal.

Similarly, the second assertion checks to see that if the FIFO is empty that we do not keep reading it!

*/

```
 /*-----------------------------------------
   Checks for the Testbench
 -----------------------------------------*/
          property check_full_write_en;
            @ (posedge wclk) disable iff (!wclk_reset_n)
                  full |-> !write_en;
          endproperty

          check_full_write_enA : assert property (check_full_write_en) else $display($stime,,,"%m FAIL:
          check_full_write_en");
          check_full_write_enC : cover property (check_full_write_en) $display($stime,,,"%m PASS:
          check_full_write_en");

          property check_empty_read_en;
            @ (posedge rclk) disable iff (!rclk_reset_n)
                  empty |-> !read_en;
          endproperty

          check_empty_read_enA: assert property (check_full_write_en) else $display($stime,,,"FAIL: %m
          check_full_write_en");
          check_empty_read_enC: cover property (check_full_write_en) $display($stime,,,"PASS: %m
          check_full_write_en");

 integer i, seed1, wclk_width, rclk_width, loopcount, base;
```

/*

Following is regular Verilog testbench code which you are very familiar with

*/

```
initial begin
  loopcount = 50;
  seed1 = 12345;
  wclk = 1'b1; write_en=1'b1;
  rclk = 1'b0; read_en=1;

  fork
  begin wclk_reset_n = 1'b0; #100; wclk_reset_n = 1'b1; write_en=1'b1; end
  begin rclk_reset_n = 1'b0; #100; rclk_reset_n = 1'b1; read_en=1'b0; end

  for (i=0; i<loopcount; i++)
  begin
    //rclk_width = ({$random} % 40) + 5;
    //wclk_width = ({$random} % 40) + 5;

    rclk_width = 40; wclk_width = 40;
    //rclk_width = 10; wclk_width = 40;
    //rclk_width = 40; wclk_width = 10;

    $display($stime,,,"wclk_width=%0d rclk_width=%0d",wclk_width,rclk_width);
    #1000#i;
  end
  join

  $finish(2);
end

always #rclk_width rclk=!rclk;
always #wclk_width wclk=!wclk;


always @ (wclk,rclk,write_en,read_en,full,empty,aff1.wr_ptr,aff1.rd_ptr,fifo_in,fifo_out)
begin
          $strobe($stime,,,"\t\twclk=%b rclk=%b write_en=%b read_en=%b full=%b empty=%b wr_ptr=%d
rd_ptr=%d fifo_in=%h fifo_out=%h",
                    wclk,rclk,write_en,read_en,full,empty,aff1.wr_ptr,aff1.rd_ptr,fifo_in,fifo_out);
end


always @ (negedge wclk) begin
   if (rclk_reset_n) begin
    if (!full) begin
        write_en=1'b1;
        fifo_in = $random({seed1});
      end
    else
      write_en=1'b0;
   end
end

always @ (posedge rclk) begin
   #2;
   if (rclk_reset_n) begin
    if (!empty) begin
      read_en=1'b1;
      //$strobe($stime,,,"READ: Pointer=%h data=%h",aff1.rd_ptr,fifo_out);
      end
    else
      read_en=1'b0;
   end
end

always @ (wclk_reset_n) $strobe($stime,,,"\twclk_reset_n=%b",wclk_reset_n);
always @ (rclk_reset_n) $strobe($stime,,,"\trclk_reset_n=%b",rclk_reset_n);

endmodule
```

14.2 Embedding Concurrent Assertions in Procedural Code

Yep, you can indeed assert a property from a procedural block. Note that property and sequence itself are declared outside of the procedural block.

Off the bat, what is the difference than between immediate assertion and embedding a concurrent assertion in the procedural block? Well, immediate assertion is invoked with 'assert' while embedded concurrent assertion is invoked with 'assert property'. A concurrent assertion that is embedded in a procedural block is the regular concurrent assertion 'assert property' (i.e. it can be temporal). In other words, an immediate assertion embedded in the procedural code will complete in zero time, while the concurrent assertion may or may not finish in zero time. But is the concurrent assertion in the procedural code blocking or non-blocking? Hang on to this thought for a while.

Figure 14.1 points out that the 'condition' under which you want to fire an assertion is already modeled behaviorally and do not need to be duplicated in an assertion (as an antecedent). The example shows both ways of asserting a property. `ifdef P shows the regular way of asserting the property that we have seen throughout this book. `else shows the same property being asserted from an always

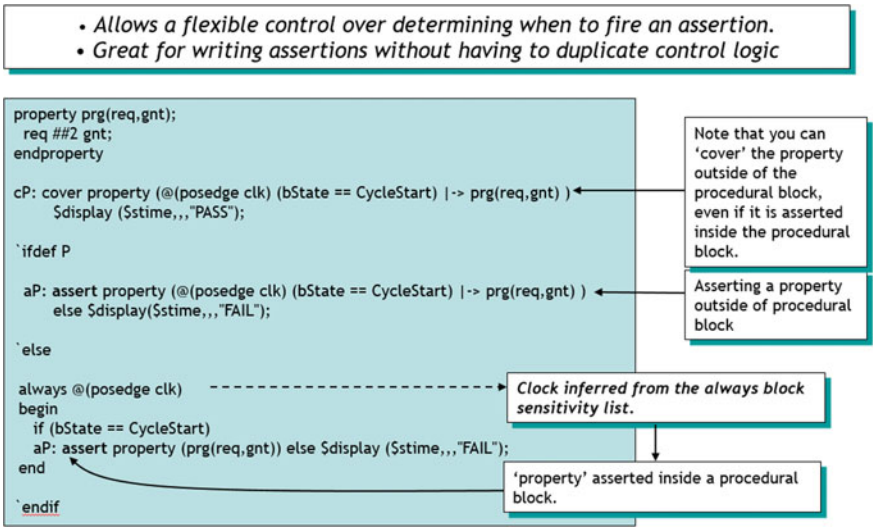


Fig. 14.1 Embedding concurrent assertions in procedural code

block. But note that in the procedural block the assertion is preceded by 'if (bState == CycleStart)' condition. In other words, a condition that could be already in the behavioral code is used to condition an assertion. If the property was 'asserted' outside of the procedural block (as we have done until now in the book) you would have to duplicate the condition in the procedural block as an antecedent in the property.

Let us turn our attention back to embedded concurrent assertion being blocking or non-blocking. What happens when you fire a concurrent assertion from the procedural code and it does not finish in zero time? What happens to the procedural code that follows the concurrent assertion? Will it stall until the concurrent assertion finishes (blocking)? Or will the following code continue to execute in parallel to the fired concurrent assertion (non-blocking)? That's what Fig. 14.2 explains.

There are two properties 'pr1' and 'pr2'. They both 'consume' time, i.e. they advance time. The procedural block 'always @ (posedge clk) asserts both these properties one after another without any time lapse between the two. How will this code execute? The procedural code will encounter 'assert property (pr1 …)' and fire it. 'pr1' will start its evaluation by looking for cstart to be high and follow on through with the consequent. In other words, 'pr1' is waiting for something to happen in time domain. *But the procedural code that fired it won't wait for 'pr1' to*

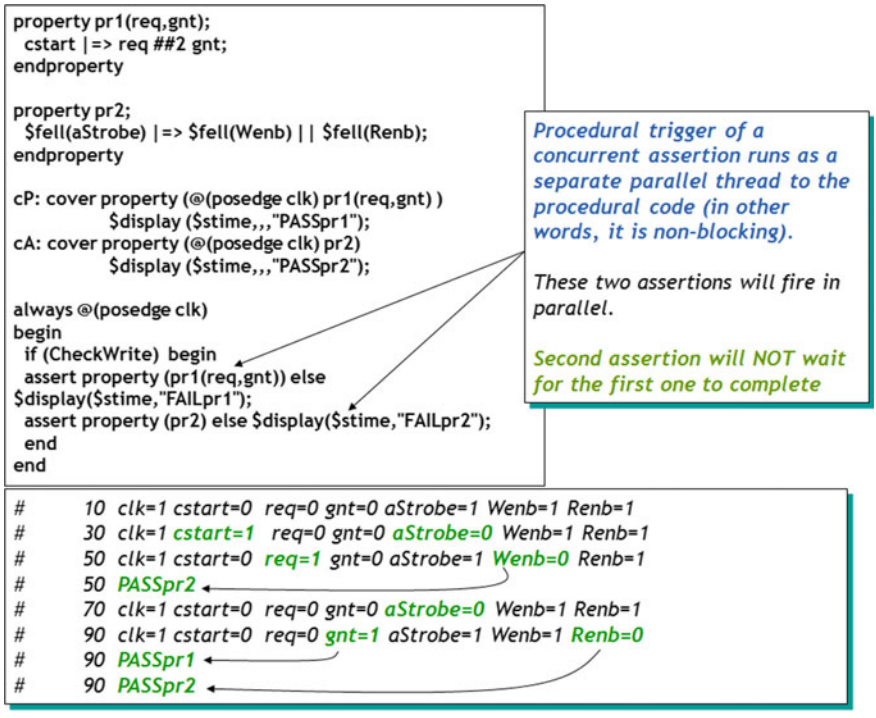


Fig. 14.2 Concurrent assertion embedded in procedural code is non-blocking

complete. It will move on to the very next statement which is 'assert property (pr2.)' and fire it as well. So, now you have 'pr1' that is already under execution and 'pr2' that is just fired both executing in parallel and the procedural code moves on to other code that sequentially follows.

In short, a concurrent assertion in procedural code is non-blocking.

As shown in the simulation log, at time 10, (@ posedge clk) we fire 'pr1'. At the *same* time [since the very next statement is 'assert property (pr2 …)'] we fire 'pr2'. At time 30, 'cstart == 1' and 'aStrob == 0'. This means the antecedent condition of both 'pr1' and 'pr2' have been met. At 50, 'Wenb == 0' which completes the property 'pr2' and the property passes as shown at time 50 in simulation log. Therefore, the first thing you notice here is that even though 'pr1' was fired first, 'pr2' finished first. In other words, since both properties were non-blocking and executing on their own parallel threads, there is no temporal relationship between them or among 'pr1', 'pr2' and the procedural code. Following the same line of thought, see why both 'pr1' and 'pr2' pass at the same time at 90.

See the rules on inferring clock edge for the assertion in the procedural block. In addition, other nuances of semantic are noted in Fig. 14.3. In short, the clock inference for the embedded assertion comes from the 'always' block edge

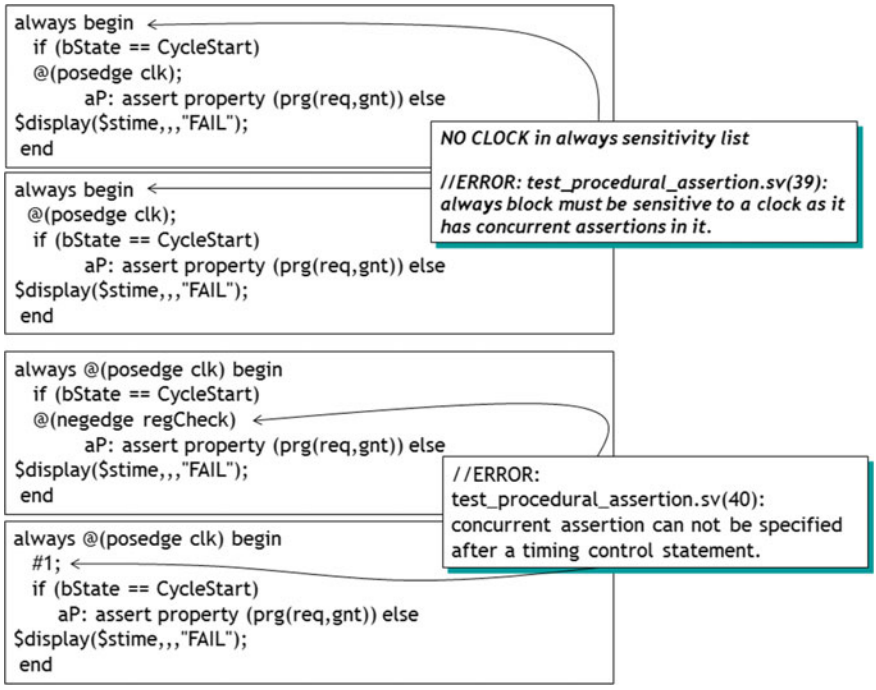


Fig. 14.3 Embedding concurrent assertions in procedural code—further nuances

sensitivity. It is not derived from any other temporal domain condition (edge or level) that is embedded in the procedural code.

Note that procedural code in context of placing concurrent assertion means only the 'initial' and the 'always' (including 'always_comb') blocks.

One note on 'automatic' variables in the procedural code and their use in an assertion (in contrast to 'static' variables that we have used so far). An 'automatic' variable is sampled *at the time* the assertion attempt is started. This is in contrast to 'static' variable, which in the assertion expression are always *sampled* at the sampling edge as with any other concurrent assertion that we have seen so far. To reiterate, since this is a very important point, the value of an 'automatic' variable is *not* sampled at the sampling edge rather captured at the time it arrives at the assertion attempt that value of the variable is then used throughout the assertion evaluation.

Note further Legal and Illegal conditions when it comes to embedding concurrent assertions in procedural code. Concurrent assertions can only be placed in an 'initial' or an 'always' block.

```
property q1;
        $rose(a) |-> ##[1:5] b;
endproperty

property q2;
        @(posedge clk) q1;
endproperty


property q5;
        @(negedge clk) b[*3] |=> !b;
endproperty

always @(negedge clk)
begin
        a1: assert property ($fell(c) |=> q1); // legal: contextually inferred leading clocking event,
@(negedge clk)
        a3: assert property ($fell(c) |=> q2); // illegal: multiclocked property with contextually inferred
leading clocking event

        a4: assert property (q5); // legal: contextually inferred leading clocking event, @(negedge clk)
end
```

Following are all illegal. Their clocks cannot be inferred.

```
always @(clk) begin

        a = b+c;

        a1: assert property (z |=> d | e);

        .....

end
```

In the above example, we don't have any edge operators (posedge or negedge). Hence, a leading clock cannot be inferred, i.e. "clk" will not be applied to property a1. An event expression without an edge operator is not allowed.

```
always @(posedge (dma_intr | intr) begin

        intrIn = intr;

        a1: assert property (z |=> d | e);

        ....

end
```

Here, 'intr' is reused in the procedural code. Hence, @(posedge dma_intr|intr) cannot be considered the leading clock for the assertion. Clock is not inferred in this case.

```
always @(posedge dma_intr or posedge intr) begin

        ........

        a1: assert property (z |=> d | e);

        ........

end
```

Here there are two edges. Hence which one should be the leading clock? That cannot be determined and there won't be any clock inference. Only one valid event expression can be specified in the event control for the inferred clock.

Here's a quick note on how procedural concurrent assertions work in a 'for loop'.

```
logic [7:0] a;
logic [15:0] b;

always @(posedge clk) begin

        If (!reset) begin

                for (int i = 0; i < 4; i++) begin

                ....

                z1: assert property (a [i] ##[1:16] b[i]);

                end

        end

end
```

In this example, four assertions, z1_1, z1_2, z1_3 and z1_4 are initiated when 'for loop' starts executing. Clock inferred is @(posedge clk).

14.3 Calling Subroutines

Attaching a subroutine to an expression is an excellent feature and a great boon to debugging effort and other applications. The subroutine calls, like local variable assignments, appear in the comma-separated list that follows the sequence. The subroutine calls are said to be attached to the sequence. It is an error to attach a subroutine call to a sequence that admits an empty match.

For example, if you'd like to know exactly when an expression is executed in a complex sequence (this is just but one example), you can 'attach' a Verilog task to the expression and display the conditions you are interested in. Figure 14.4 explains this scenario.

As shown in the figure, you can attach a subroutine to an expression, a sequence or to an expression or subsequence within a sequence. Note that the attached subroutine will execute on *successful* completion of either the expression or the sequence to which it is attached. For example, (not(cde,tdisp1)) in the topmost example of Fig. 14.4 means that 'tdisp1' will execute when 'cde' reaches its true conclusion. Else, it won't execute.

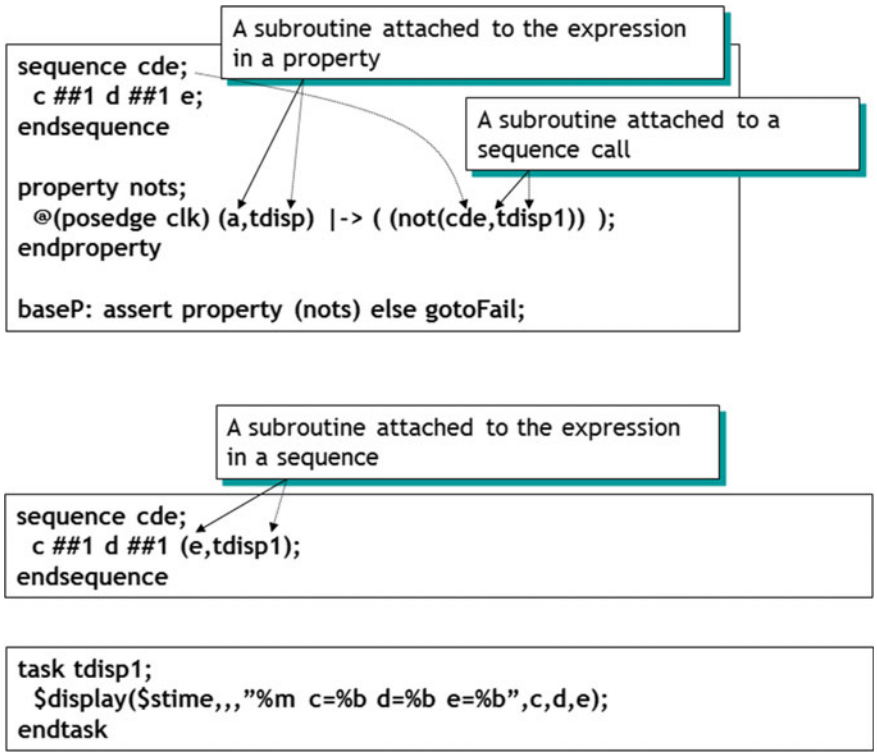


Fig. 14.4 Calling subroutines

```
sequence lvar_seq(pin,pout);
 int local_data;
 ($rose(ptrue),local_data = pin,$display($stime,,,"pin=%0d",pin) )
//                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//          This will be executed when $rose(ptrue) is detected..
  ##5
 (pout == (local_data+5),$display($stime,,,"pout=%0d",pout) );
//                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//                       This will be executed ONLY IF
//                       the sequence MATCHES.
endsequence

property lvar;
 @(posedge clk) cStart |-> lvar_seq(pipe_in,pipe_out);
endproperty
```

Fig. 14.5 Calling subroutines—further nuances

Figure 14.5 further explains when a subroutine is executed. First, at the top of the figure you notice that we 'attach' a local variable as well as a subroutine ($display task). Since $rose(ptrue) is the sequence to which the subroutine is attached, it will execute only when $rose(ptrue) is true. Similarly, the next part of the figure shows (pout == (local_data + 5), $display (…)), where (pout == (local_-data + 5)) is the expression to which the $display subroutine is attached. Again, the subroutine $display will execute only if (pout == (local_data + 5)) is true.

You will also notice that $display in this figure (for most part) displays local variables. This is one of the important use of $display as a subroutine because local variables cannot be accessed from an action block (pass or fail).

In Fig. 14.6, the subroutine is a Verilog 'task'—lvar_seq_trigger which in turn contains a $display. But *I don't want you to run away with the idea that the only subroutine you can attach is the one that $displays something!* For example, you can use a subroutine to collect coverage information (using covergroups and cover-points). We will see this with an example when we discuss Functional Coverage. *Since the attached subroutine can be task, you can think of many possible applications.*

The idea behind attaching a 'task' to the expression is that you can do whatever that Verilog allows you to do in a 'task' except that you cannot access the local variables of the sequence that invoked the 'task'. But you can indeed pass a local variable as an argument to the 'task' as shown in Fig. 14.6.

'sequence lvar_seq' has a local variable called 'local_data'. This local variable is passed to 'lvar_seq_trigger' as an actual argument. 'task lvar_seq_trigger' in turn uses that as an input 'ldata' and displays it. This is one way you can pass a local variable to the attached subroutine.

```
sequence lvar_seq(pin,pout);
  int local_data;
  ($rose(ptrue),local_data = pin,lvar_seq_trigger(local_data))
   ##5
  (pout == (local_data+5),lvar_seq_match(pin,pout,local_data));
endsequence

property lvar;
  @(posedge clk) ptrue |-> lvar_seq(pipe_in,pipe_out);
endproperty

baseP: assert property (lvar) else gotoFail;
coverP: cover property (lvar) gotoPass;

task lvar_seq_trigger;
input ldata;
  $display($stime,,,"%m  ldata=%0d",ldata);

  //$display($stime,,,"%m  ldata=%0d",lvar_seq.pin);
  // ** Error: Hierarchical access to formal parameter pin' of 'lvar_seq' is illegal.
endtask

task lvar_seq_match;
input tpin,tpout,ldata;
  $display($stime,,,"%m  pin=%0d pout=%0d ldata=%0d",tpin,tpout,ldata);
endtask
```

Fig. 14.6 Application: calling subroutines and local variables

But note that you cannot access a variable (local or not) hierarchically from the attached subroutine (for example, task lvar_seq_trigger). This is shown with an ERROR in the figure. Here we tried to hierarchically access variable 'pin' in 'sequence lvar_seq' by using 'lvar_seq.pin'. That is a violation.

To recap:

All subroutine calls attached to a sequence are executed at every successful match of the sequence.
For each successful match, the attached calls are executed in the order they appear in the list.
Assertion evaluation does *not* wait on or receive data back from any attached subroutine. The subroutines are scheduled in the Reactive region, like an action block.
Actual argument expressions that are passed by value use sampled values of the underlying variables and are consistent with the variable values used to evaluate the sequence match.
An automatic variable may be passed as a constant input for a subroutine call from an assertion statement in procedural code. An automatic variable cann*ot* be passed

by reference nor passed as a nonconstant input to a subroutine call from an assertion statement in procedural code.
Local variables can be passed into subroutine calls attached to a sequence. Any local variable that is assigned in the list following the sequence, but before the subroutine call, can be used in an actual argument expression for the call. If a local variable appears in an actual argument expression, then that argument must be passed by value.
Tasks, task methods, void functions, void function methods, and system tasks can be called at the end of a successful non-empty match of a sequence.
Finally, arguments passed to a subroutine must be by value or by reference ('ref' or 'const ref').

14.4 Sequence as a Formal Argument

System Verilog assertions are indeed powerful as evident from this feature. *You can send an entire sequence as an actual argument to a property or another sequence.* One obvious advantage is that you may reuse a sequence in different properties as an actual to the property's formal argument. One example of this is the Reset Sequence as shown in Fig. 14.7. Reset sequence is often used in different properties as an antecedent. Write it once and pass it to different properties as an actual

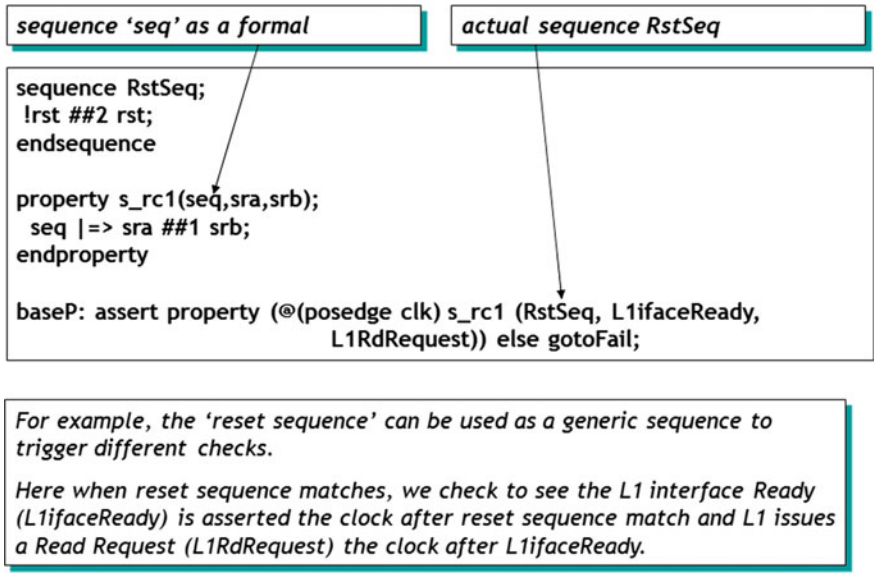


Fig. 14.7 Sequence as a formal argument

argument. That is reusability with observability and debuggability. The sequence that is passed to a property can (obviously) be used on both the antecedent and consequent side.

14.5 Sequence as an Antecedent

Since a sequence can be passed as an actual argument there are many advantages. We saw one in Fig. 14.7. Here's another. Here, we define a simple sequence 'seq' and pass it to property 's_rc1'. In this property, we use 'seq' (i.e. c_seq in the property) as an antecedent (Fig. 14.8).

As with any antecedent, the property will wait for antecedent to be true and then imply the consequent. Now, with many operators (e.g. 'throughout') we have observed that the LHS and RHS of the operator is equally responsible for failure. If either side fails that the operator and hence the sequence/property fails. Important thing to note here is that in the cases of 'throughout' the operator was used in the consequent and not in antecedent. Anything that fails in consequent causes the property to fail. Here, a sequence is used in the 'antecedent' meaning even if the

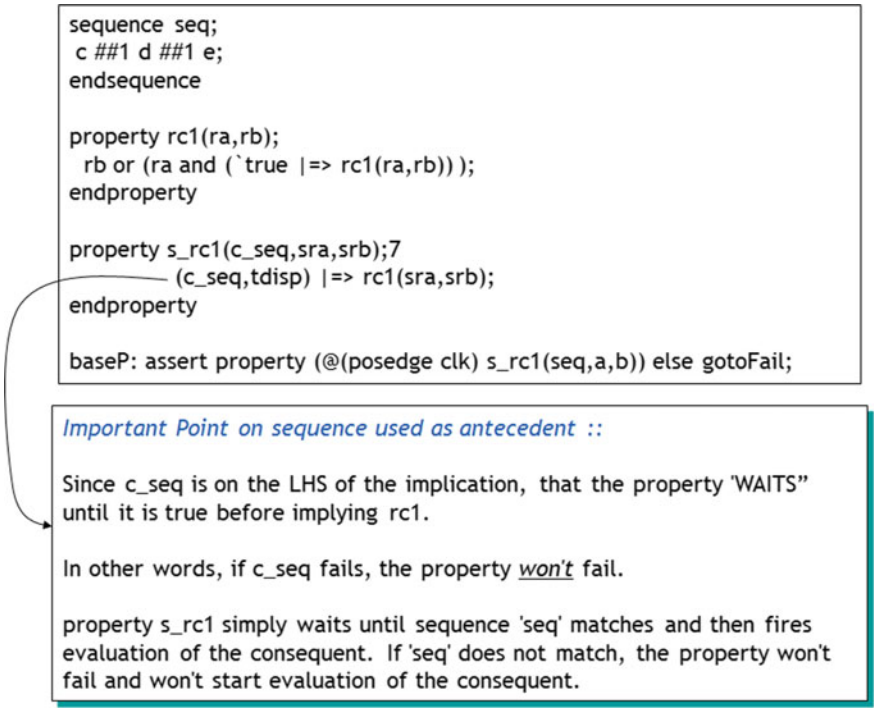


Fig. 14.8 Sequence as an antecedent

sequence in antecedent fails, the property will not fail. Instead, the property will simply wait for the sequence 'c_seq' to be eventually true, after which it will execute the consequent. This makes sense because consequent fires only when antecedent is sampled to be true.

Short end of the story is that no matter what you have in the antecedent, it will not cause a failure. Antecedent's job is to evaluate its expression/sequence and on sampling it to be true, imply the consequent.

14.6 Sequence in Sensitivity List

Let us take the use of a 'sequence' even further. Guess what? You can use a sequence for event control either in the sensitivity list or as an explicit edge sensitive control in an initial block. You can use this feature very effectively because designing a temporal domain condition in SVA is far easier than using behavioral Verilog. You can design certain condition as a sequence and then use it in your procedural behavioral Verilog code as shown in Figs. 14.9 and 14.10.

Figure 14.9 shows that the 'always' block waits for sequence 'sr1' to complete after which it display its PASS result. Can you figure out why there is no FAIL report? Was "always @ (sr1)" triggered when 'gnt' did not follow 'req' after 2 clocks? Please experiment and see what happens.

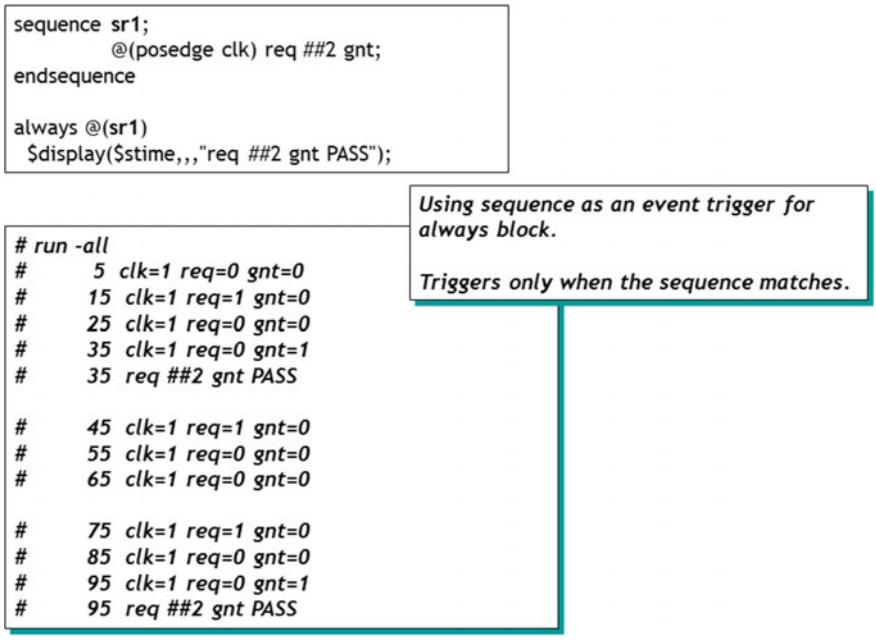


Fig. 14.9 Sequence in procedural block sensitivity list

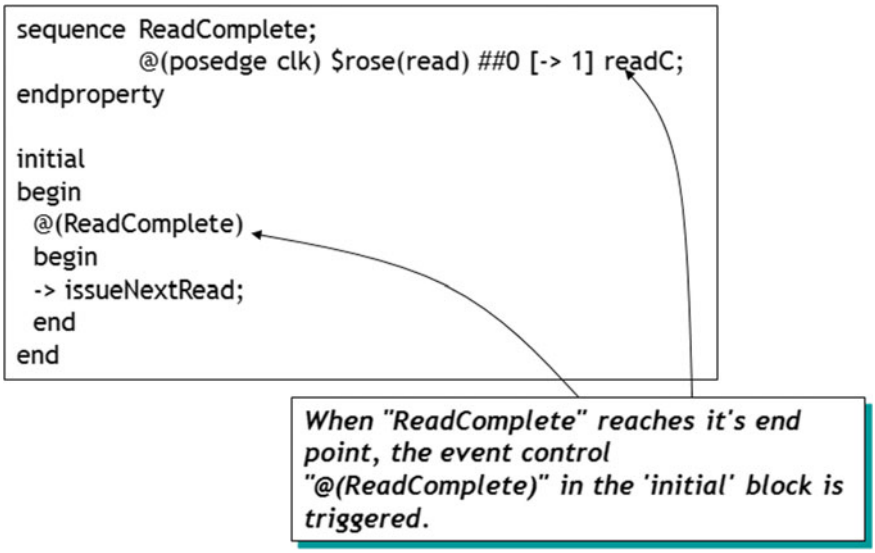


Fig. 14.10 Sequence in 'sensitivity' list

The sequence in Fig. 14.10 says that @ (posedge clk) if $rose(read) is sampled high (edge) that there should be at least 1 readC (read complete). The 'initial' block waits for this sequence to complete (using @ (ReadComplete)) and then issues the next Read.

As you can see, this feature is extremely powerful in using the power of sequence in designing your SystemVerilog testbench code.

14.7 Building a Counter

Ok, that is enough of sequences (for a while, at least). Let us see how we can effectively use local variables and the consecutive repetition operator to build something!

Let us build a counter. Why? There are many applications where you will be able to use this example. For example, you want to make sure that an incoming packet on a network generates an interrupt when it's payload reaches a maximum threshold.

The property checkCounter declares a local 'int' called 'LCount'. It waits for a rising edge on 'startCount' and on this rising edge, it stores 'initCount' into 'LCount' (initCount is defined elsewhere in your procedural code).

It then waits for 1 clock and increments LCount by 1 and continues to do so at every clock until LCount reaches maxCount. The consecutive repetition operator [*0:$] does the counting. In other words, "LCount = LCount + 1" repeats at every

posedge clk until you reach "LCount == maxCount". Once the maxCount is reached, the antecedent implies at the same clock (overlapping implication) that the intr be asserted.

Quick Note: In this book, as you have noticed, instead of giving large applications and then describe a limited set of SVA features, I have chosen to describe each operator with simple applications so that you clearly understand the workings of the operator and apply the operator features to design your assertions.

14.8 Clock Delay: What if You Want *Variable* Clock Delay?

Figure 14.11 built a counter which can be used to create a variable delay model. Note that SVA allows only constant fixed delays with its delay operator. So, the example in Figs. 14.12 and 14.13 demonstrates a strategy to get around this limitation.

Figure 14.12 describes a typical specification. We need to check for variable latency based on the position of a 'read' in the read queue. In other words, if the 'read' is at the end of the queue, 'read' will complete with maximum latency. On the contrary if it's at the beginning of the queue, it will complete with minimum latency. Or with a latency anywhere in the middle.

Since the delay (or range delay) operator does not allow variable delay, how would you model this with one generic assertion? You do not want to create a separate assertion for each of the fixed latency. That is the problem statement. Now let us see how we solve this. Consider this example as an idea generator.

Following is an example of how to build a counter using the consecutive repetition operator [m]*

```
property checkCounter;
int LCount;
  @(posedge clk) disable iff (!rst_n)

          (
                    ($rose(startCount), LCount=initCount ) ##1

                    (1, LCount = LCount+1)[*0:$] ##1 (LCount == maxCount) |->

                    (Intr == 1'b1)
          );
endproperty
assert property (checkCounter);
```

Fig. 14.11 Application: building a counter using local variables

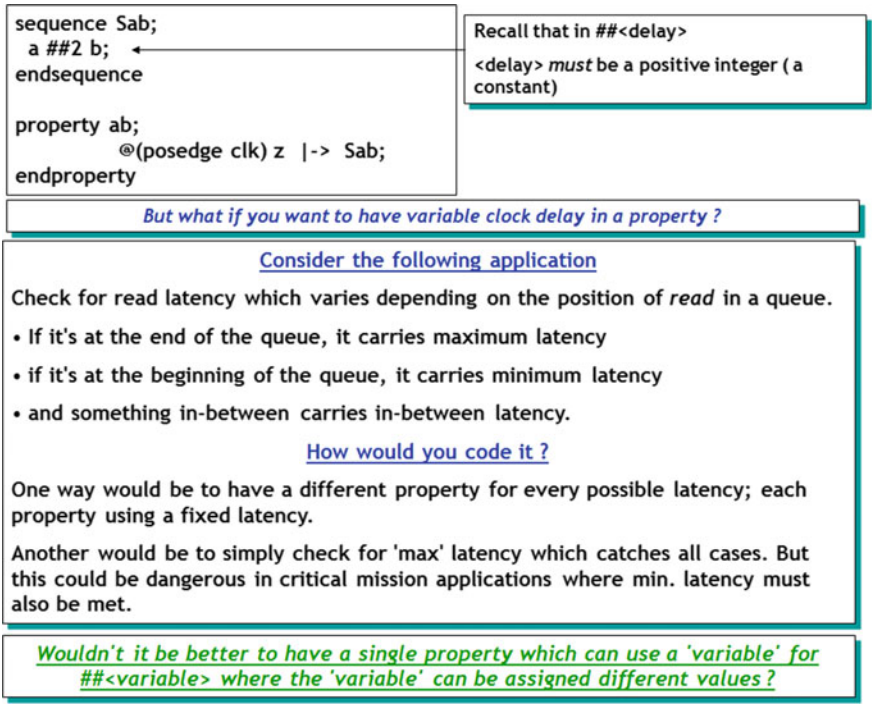


Fig. 14.12 Variable delay—problem statement

The concept in Fig. 14.13 is identical to building a counter example. Here instead of incrementing the local variable we decrement it until it reaches zero. In this application readLatency is defined in your procedural code and it changes based on the position of Read in the read queue. That part of the code is not shown here.

When the property 'read_latecncy_check' is asserted, it will assign the readLatency to Ldelay on assertion ($fell(rd_)) and decrement it at every posedge clk, until it reaches 0.

(1, Ldelay = Ldelay-1)[*0:$] ##1 (Ldelay ==0)

Ldelay is decremented consecutively at every posedge clk until Ldelay == 0. Need for '1' in (1, Ldelay = Ldelay − 1)? Why? Recall that we can assign to a local variable when that assignment is attached to an expression. Since we do not have any explicit expression, we simply use 'always true' as an expression.

You can continue to change readLatency from procedural code based on the position of 'read' in your read queue and use the same property to check for different latencies of read in the read queue.

This simple example has very powerful application capabilities.

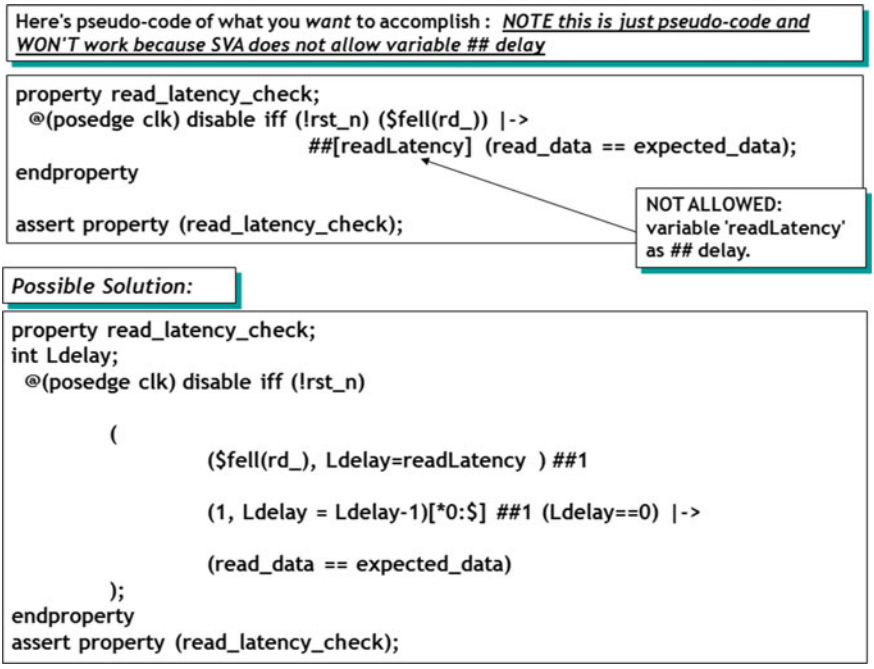


Fig. 14.13 Variable delay—solution

14.9 What if the 'Action Block' Is Blocking?

```
property pr1;
  @(posedge clk) req |-> ##2 gnt ;
endproperty

reqGnt: assert property (pr1) else failtask;

task failtask;
    $display($stime,,,"FROM  failtask - 0");

    @(posedge clk) $display($stime,,,"FROM  failtask - 1");
    @(posedge clk) $display($stime,,,"FROM  failtask - 2");
    @(posedge clk) $display($stime,,,"FROM  failtask - 3");
    @(posedge clk) $display($stime,,,"FROM  failtask - 4");

endtask
```

We have seen that the assertion of a property ("assert property") allows you two 'action' blocks. One is triggered when the property passes and the other when it fails.

This action block can contain any procedural code that SystemVerilog supports. The procedural block can have temporal domain 'delay' (e.g. @ (posedge cc) or 'wait sig', etc.). That is when you need to carefully weigh in the consequences. If there is no 'delay' in the block, life is straightforward. The "assert property" triggers the block without any delay; the block executes in 0 time; returns and the property moves along with its execution. But if there are delays in the action block, here's what happens.

The property at top left of Fig. 14.14 says "assert property (pr1) else failtask". If the property fails, call a task called 'failtask'. 'failtask' in turn waits for 4 @ (posedge clk) and returns from the task. The 4 clock delay is just an example for temporal delay.

Now let us look at the simulation log. At time 30, req = 1, so the property pr1 moves along. At time 70, gnt = 0 which is a failure condition because gnt should have been '1' at that time. Since there is a failure, the 'failtask' is invoked at time 70 (the time of failure). The 'failtask' waits for 4 posedge clks, which are displayed in the log file at time 90, 110, 130, 150. While the 'failtask' was waiting for its clocks to complete, at time 110, req goes high again. So, the property starts executing and expects 'gnt' to be high at 150. And again gnt is '0', so the property should fail. But it does not!! Why? Because at time 150, the 'failtask' was still completing its 4th clock wait. Since the 4th clock wasn't over by the time the 'gnt' based failure came

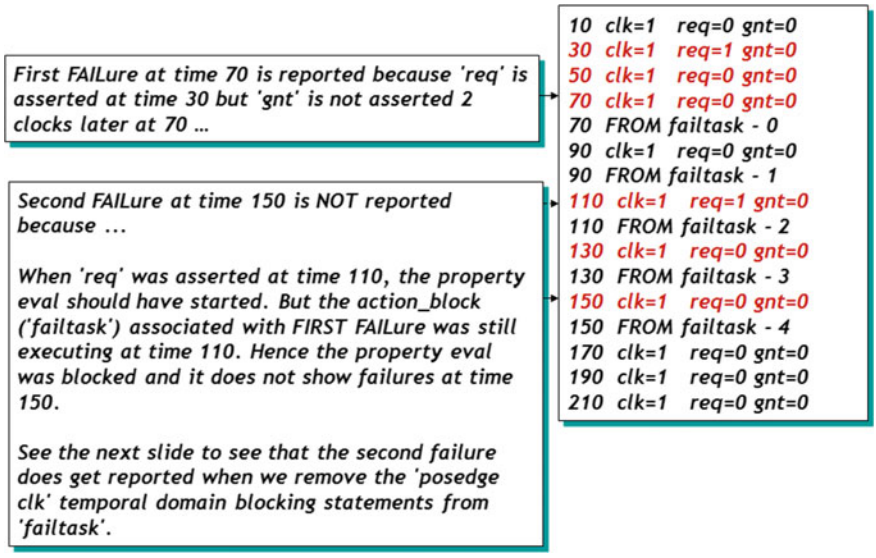


Fig. 14.14 Blocking action block

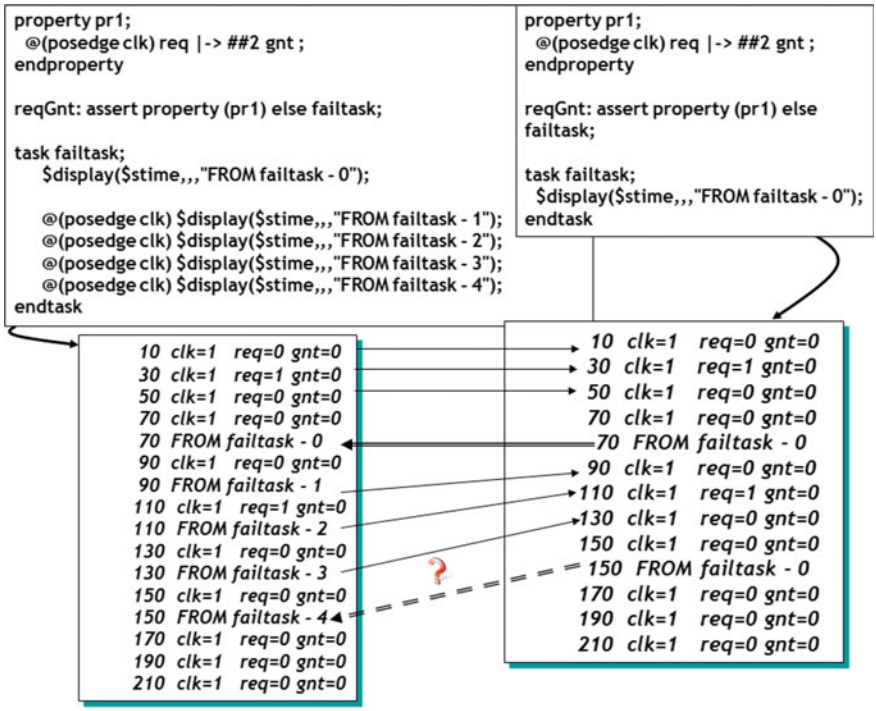


Fig. 14.15 Blocking versus non-blocking action block

along at 150, the failure got suppressed because first invocation of 'failtask' wasn't over.

The point is, if you call a procedural block that does not complete in '0' time, and if the next trigger of the property antecedent comes along causing another pass/fail, the next trigger of the action block associated with pass/fail won't happen. So, be careful in using time lapse in your action block(s).

Figure 14.15 simulation logs highlight the same point. There is one example with time lapse in the action block and another without.

As shown in Fig. 14.15, the properties on LHS and RHS are identical, except that LHS action block calls a 'failtask' that elapses time (4 clocks). The RHS on the other hand calls 'failtask' that does not elapse any time.

The 'req' condition is also identical in both simulation logs. But the RHS shows two invocations of 'FROM failtask—0' while the LHS log shows only one invocation (as explained above for Fig. 14.14) because the action block is 'blocking' and does not allow reentry into an already executing action block.

14.10 Interesting Observation with Multiple (Nested) Implications in a Property. Be Careful

Can you have multiple implications in a property? Sure you can. However, you need to very carefully understand the consequences of multiple implications in a property. Some also call, multiple implications as nested implications. You decide. Let us look an example and understand how this works.

In Fig. 14.16, the property mclocks (at first glance) looks very benign. But play close attention and you will see two implications. @ (posedge clk) if 'a' is true that implies "'bSeq' ##1 c", which implies 'dSeq'. One antecedent implies a consequent which acts as the antecedent for another consequent.

```
sequence bSeq;
 ##[1:5] b;
endsequence

sequence dSeq;
 ##2 d ##2 e;
endsequence

property mclocks;
  @(posedge clk) a |-> bSeq ##1 c |-> dSeq;
endproperty
```

```
#      165  CLK # 17 :: clk=1 a=0 b=0 c=0 d=0 e=1
#      175  CLK # 18 :: clk=1 a=1 b=0 c=1 d=0 e=0 //a=1 so start
#      185  CLK # 19 :: clk=1 a=0 b=1 c=0 d=0 e=0
                        //b=1 next clock; so 'bSeq' matches

#      195  CLK # 20 :: clk=1 a=0 b=0 c=0 d=0 e=0
                        //But c NE 1 the next clock
                        //and the property does NOT fail.

#      205  CLK # 21 :: clk=1 a=0 b=0 c=0 d=0 e=0
#      215  CLK # 22 :: clk=1 a=0 b=0 c=1 d=0 e=0

//So, when 'c' does go '1' a couple of clocks later, the 'dSeq' seq
//won't start eval at that time. The fact that 'c' did not go '1'
//the very next clock after 'bSeq' matches that the entire
//property will simply not get evaluated for pass or fail; until //'a' is asserted
again.

#      225  CLK # 23 :: clk=1 a=0 b=0 c=0 d=0 e=0
#      235  CLK # 24 :: clk=1 a=0 b=0 c=0 d=1 e=0
#      245  CLK # 25 :: clk=1 a=0 b=0 c=0 d=0 e=0
#      255  CLK # 26 :: clk=1 a=0 b=0 c=0 d=0 e=1
#      265  CLK # 27 :: clk=1 a=0 b=0 c=0 d=0 e=1
```

Fig. 14.16 Multiple implications in a property

Now, let us look at the simulation log. At time 175, a = 1, so the property starts evaluation and implies "bSeq ##1 c". At time 185 'bSeq' matches, so the property now looks for ##1 c. At time 195, c is –not- equal to '1' but the property does not fail. Wow! Reason? Note that 'bSeq ##1 c' is now an antecedent for 'dSeq' and as we know if there is no match on antecedent that the consequent won't be evaluated and the property won't fail. Here that seems to apply even though 'bSeq ##1' is a consequent, it is also an antecedent. Language anomaly? Not really, but the behavior of such properties is not quite intuitive. Since 'bSeq ##1c' did not match, the entire property is discarded and the property again waits for the next "a == 1" to start all over again.

Confusing? Well, it is. Hence, please don't use such multiple implication properties unless you are absolutely sure that that's what you want. I've seen engineers use it because the logic seems intuitive, but the behavior is not.

My suggestion is to use only a single implication. It will keep your code unambiguous. For example, following two properties are equivalent.

```
P1: assert property (@posedge clk) req |-> ##2 gnt |-> ##2 gntAck;

P1: assert property (@posedge clk) req ##2 gnt |-> ##2 gntAck;

Following two are equivalent too.

P2: assert property (@posedge clk) req |=> gnt |=> gntAck;

P2: assert property (@posedge clk) req ##1 gnt |=> gntAck;

And finally following two are equivalent

P3: assert property (@posedge clk) a |-> b |-> c;

P3: assert property (@posedge clk) a ##0 b |-> c;
```

So, as you can see, it is indeed possible to write properties with a single implication operator. Keep it simple.

14.11 Subsequence in a Sequence

A sequence can be embedded in another sequence. The embedded sequence can be called a subsequence. Figure 14.17 shows that sequence 'abc' is embedded into sequence 'abcRule'. The embedded subsequence infers the clock from the parent sequence, if the subsequence does not have an explicit clock of its own.

Also, as shown in Fig. 14.18, a sequence can be used both as an antecedent and/or a consequent.

As shown, the antecedent is sequence 's1', which implies sequence 's2' as consequent. Here, each sequence has its own explicit clock. However, if that were not the case, the subsequences would inherit (inference) the clock from property s1 to s2.

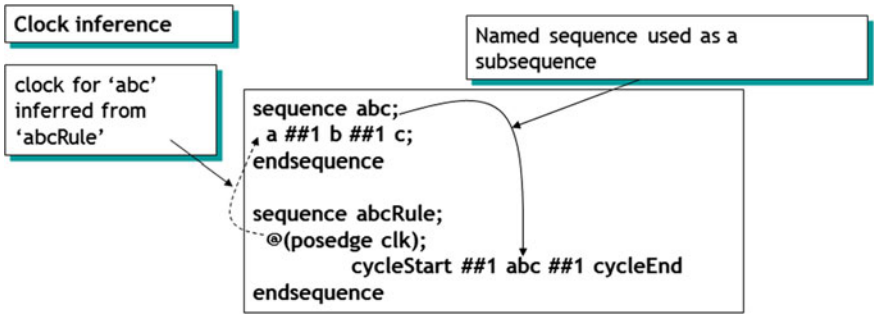


Fig. 14.17 Subsequence in a sequence—clock inference

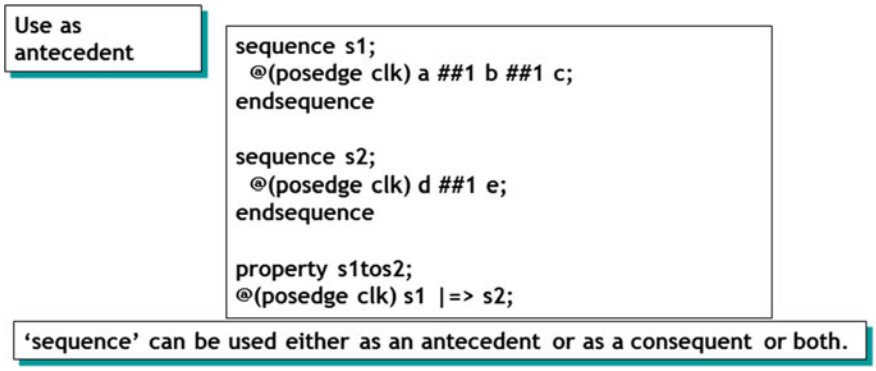


Fig. 14.18 Subsequence in a sequence

The reason for pointing out this usage is, again, to emphasize that it's best to break down a property into smaller sequences and then build the larger overall property. The smaller the sequence, the better it is for debuggability and controllability.

14.12 Cyclic Dependency

As shown in Fig. 14.19, you can indeed have cyclic dependency between properties but *not* among sequences. But *note that the cyclic dependency between properties is only between consequent of the property, not the antecedent*.

What is the use of this feature? If you want to check for continuous toggle between two states of a state machine, you can use this property. The property as shown in this example will never complete until simulation ends.

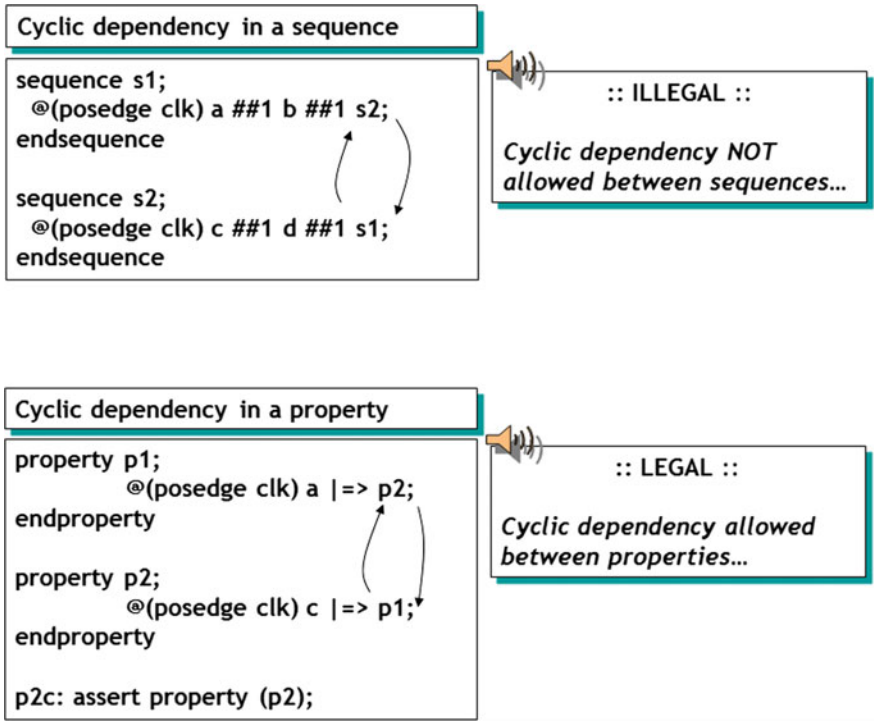


Fig. 14.19 Cyclic dependency

Note also that you cannot do something like "c|=>d ##1 e ##1 p1". You cannot use another property as a subsequence for cyclic dependency. You will get the following Error

**Error: massert.v(42): Illegal SVA property value in RHS of '##' expression.

14.13 Refinement on a Theme

See Fig. 14.20

14.14 Simulation Performance Efficiency

In Fig. 14.21, the top property rdyProtocol says that if rdy is true then you must get a rdyAck. We have designed that using the constant delay range. Nothing wrong with that, but (as seen from simulation results), the infinite range based design runs

sequence 'abc' states that 'a' is followed by 'b' followed by 'c' (all with a single clock delay). Simple enough ....

```
sequence abc;
        @(posedge clk) a ##1 b ##1 c;
endsequence
```

In many applications, however, it maybe required that 'a' remains asserted when b is asserted and then 'a and b' remain asserted when 'c' is asserted. The sequence here will do the trick ...

```
a ##1 a & b ##1 a & b & c;

OR (with good use of parenthesis)

a ##1 (a & b) ##1 (a & b &c);
```

OR if it is required that 'a' and 'b' must get deasserted the very next clock after they are found asserted, then this sequence will do the trick.

```
a ##1 !a & b ##1 !a & ! b & c;

OR

a ##1 (!a & b) ##1 (!a & !b & c);
```

Fig. 14.20 Refinements on a theme

```
property rdyProtocol;
  @(posedge clk) rdy |-> ##[1:$] rdyAck;
endproperty
assert property(rdyProtocol);
```

Avoid long or infinite time ranges.

```
property rdyProtocol;
  @(posedge clk) rdy |-> rdyAck [-> 1];
endproperty
assert property(rdyProtocol);
```

A more simulation efficient way of expressing the 'infinite' range requirement...

Fig. 14.21 Simulation performance efficiency

slower than the one that does not use such a range. This by no means prohibits use of ##[1:$], but if you can find a better way to solve the problem, you will get better simulation efficiency. The bottom part of Fig. 14.21 shows the alternate way. It uses the 'goto' operator, which models the same behavior, namely, that there will be at least 1 rdyAck after a rdy.

14.15 It's a Vacuous World! Huh?

This section could have gone much earlier in the book but I did not want the reader to get confused from the get go. Once you go through this example, you will see why the 'implication' operator is (almost) a must in an assertion. The example figures below have detailed annotation for ease of understanding.

Here we go.

14.15.1 Concurrent Assertion—Without—An Implication

Let us examine Fig. 14.22.

There is no implication operator in property pr1. As Fig. 14.22 shows, property 'pr1' reads as "@ (posedge clk) req should be true and 2 clocks later gnt should be true". Note that we have not used implication operator in the property. Hence, read the property carefully. It does-not—say that "*if*" req is true that the property should check for gnt. It simply says that 'req' be true at the posedge clk and 2 clks later gnt be true. Hence, every clock that req is *not* true the property FAILs. Is that what we really want? I don't think so. That's where an implication operator comes into picture.

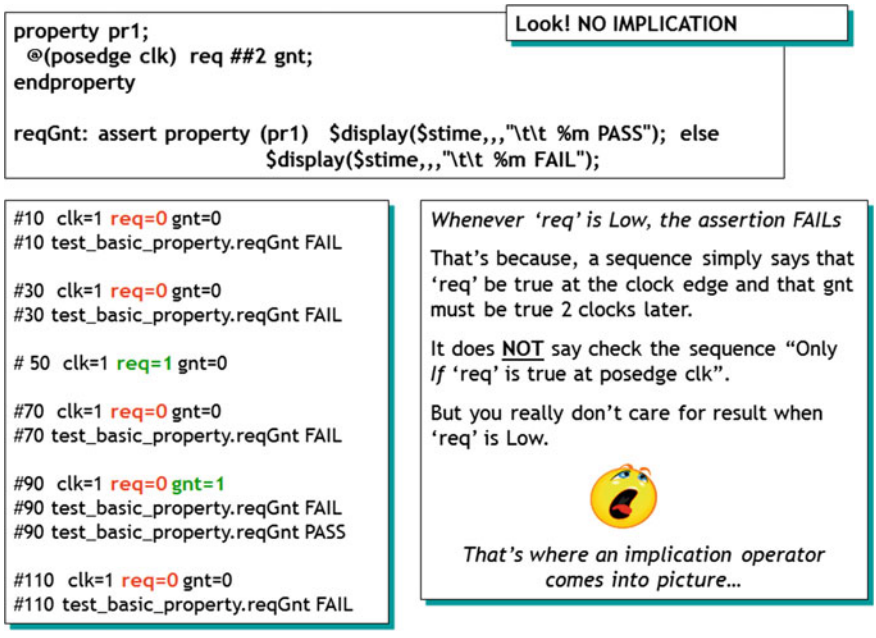


Fig. 14.22 Assertion without implication operator

More importantly, do you notice that at time 90, the property PASSes as well as FAILs!! Amazing! The property passes because at time 50, req = 1 so the property looks for gnt = 1 at 90. It does find gnt = 1 at 90, so it PASSes. But since req = 0 at 90, it also FAILs. Amazing, again!

My suggestion, do NOT use properties without implication, unless you are absolutely sure of what you are doing. Read On. The story does not quite end with implication either … but there is hope.

14.15.2 Concurrent Assertion—With—An Implication

Ok, so we decide to add an implication operator as in "@ (posedge clk) req|->##2 gnt;" so that we don't get false failures. But wait! See the simulation log in Fig. 14.23 carefully. Now the assertion passes whenever req = 0. What's going on?

Everything seems Ok. If the antecedent is not true, the consequent won't fire. But when the antecedent is not true, the properties PASS action block triggers and tells us that the property PASSes. Read on.

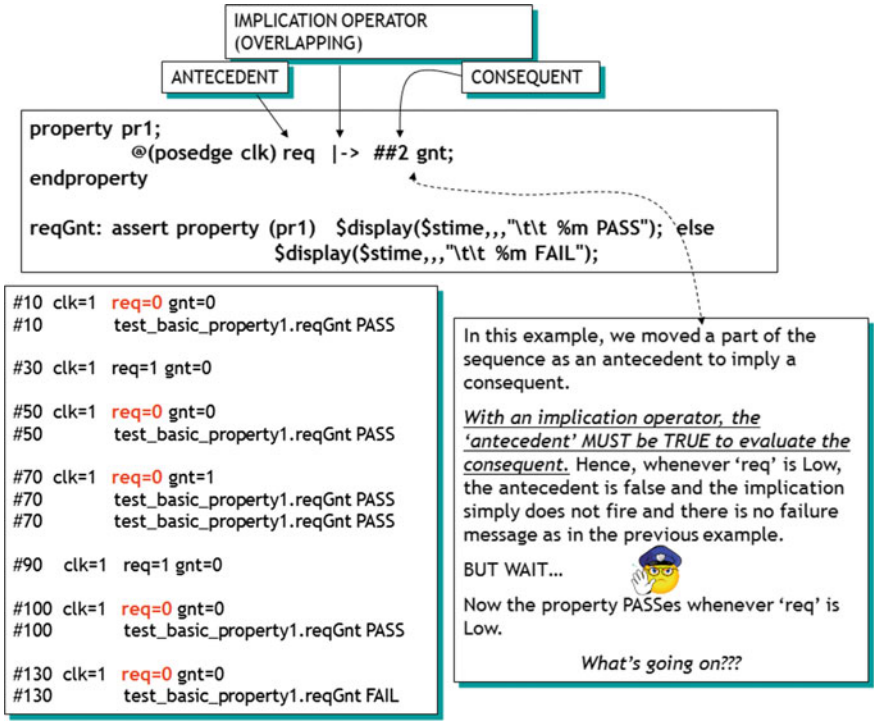
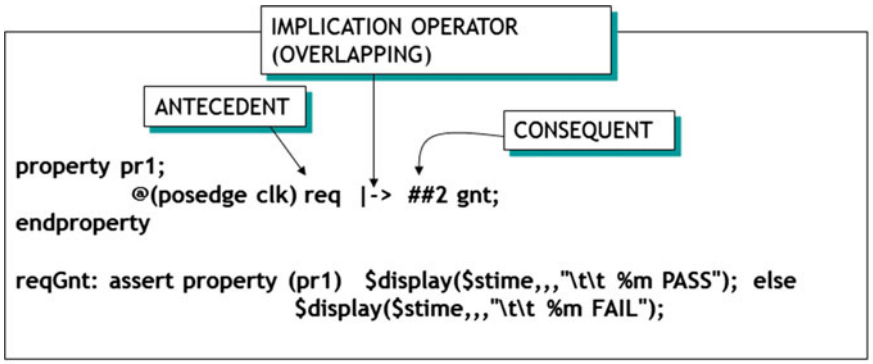


Fig. 14.23 Assertion resulting in vacuous pass

14.15.3 Vacuous Pass. What?



LRM 3.1a (Page 232) ::

"If there is no match of the antecedent sequence_expr, then evaluation of the implication succeeds vacuously and returns true"

A couple of ways to get around this...

One is to simply not use the action_block associated with 'pass' (duh...) of the property, so that you don't get pass indication vacuously

But what if you do want to know when the property passes...

The reason we get a PASS on the antecedent failure is that according to the LRM "If there is no match of the antecedent sequence_expr, then evaluation of the implication succeeds vacuously and returns true". Hence, whenever you see 'req' low, you get a 'vacuous' pass which triggers the PASS action block and we get the PASS display.

Ok, so what is the solution? Why did not we see this behavior until now with all the examples we have been through? Read on …

14.15.4 Concurrent Assertion–with 'Cover'

There are two ways to overcome this behavior that we do not want. One is to simply ignore the PASS action block on 'assert' of a property, i.e. simply do not have a PASS action block. That way if there is a vacuous pass, our log will not be cluttered with misleading PASS messages. Note that ignoring the so-called vacuous pass is harmless. This is the obvious solution.

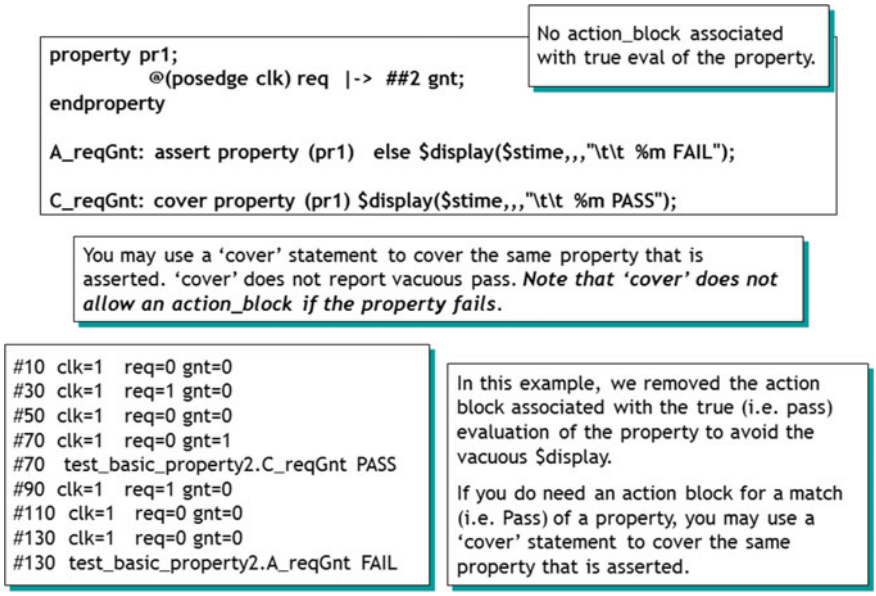


Fig. 14.24 Assertion with 'cover' for PASS

But what if you do want to know when the property PASSes? That's where 'cover' comes into picture.

The solution with 'cover' allows us to see if the property is indeed covered (i.e. exercised). 'cover' does *not* have the vacuous pass property. It indicates at the *end* of the assertion if it has been covered. When it is covered, it triggers a PASS action block. In this action block you may put a $display statement to indicate that the property has been covered or that it has 'passed'.

Note that 'cover' simply does *not* have a FAIL action block and does not have the vacuous pass property.

This way, with 'assert' and 'cover', we have a method to code an assertion that gives us the required FAIL and PASS indication without any other message. Please read the simulation log carefully in Fig. 14.24 to see the behavior of the property.

NEW TO IEEE-1800 2012 LRM. So, for all this 'vacuous' pass dilemma, 2012 LRM came up with a clean solution. They have introduced a new System Task called $assertcontrol. I have devoted complete Sects. 16.18 and 16.17 to describe this task. But here's an example of how you can turn OFF the vacuous pass indication.

$assertcontrol (VACUOUSOFF, CONCURRENT|EXPECT);

This systasks affect the whole design so no modules are specified. Disable vacuous pass action for all the concurrent asserts and 'expect' in the design. Also disable vacuous pass action for expect statements.

The 2012 LRM also introduces a specific system task simply to disable vacuous pass indication.

$**assertvacuousoff** system task turns off the PASS indication based on a vacuous success. An assertion that is already executing is not affected. By default, we get a PASS indication on vacuous pass.

Note also the use of "%m" in the $display task. This good old Verilog feature displays the entire path to the assertion. This is one way to distinguish two properties with same name in two separate scopes.

14.16 Empty Sequence

In Fig. 14.25, we are using the consecutive operator '*' but with '0' repetition [*0]. In other words, we are saying 'b' should not repeat *ever*. In yet other words, that means 'b' simply does not exist (empty) even though it is part of the property. Hence, "b[*0] ##1 !a" simply means check for empty sequence 'b' and 1 clock later check for '!a'. Since, empty sequence does not mean anything, we are basically checking for '!a' 1 clock after $rose(a).

Following examples are given for the sake of completeness. Regard them as Reference Material.

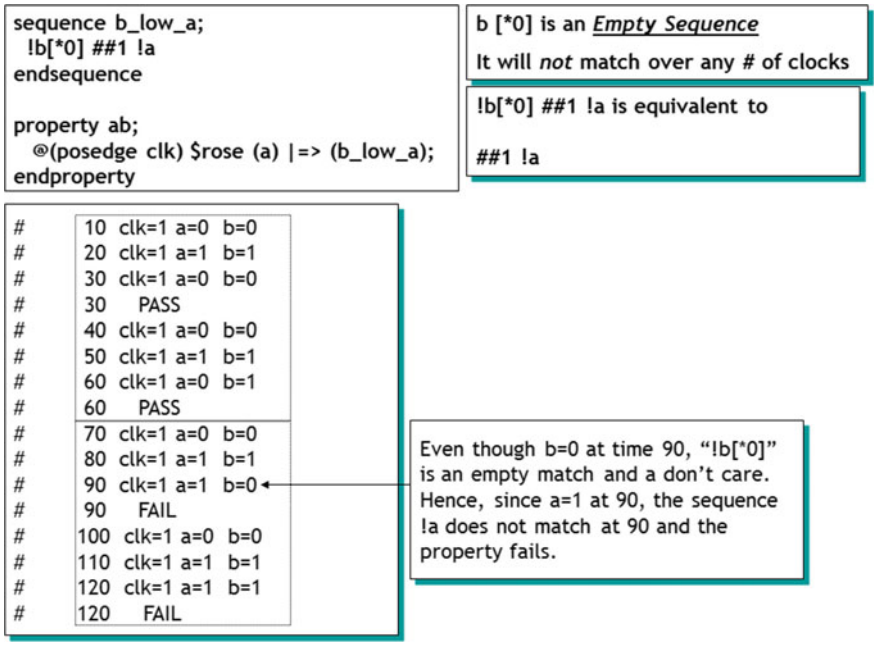


Fig. 14.25 Empty match [*m] where m = 0

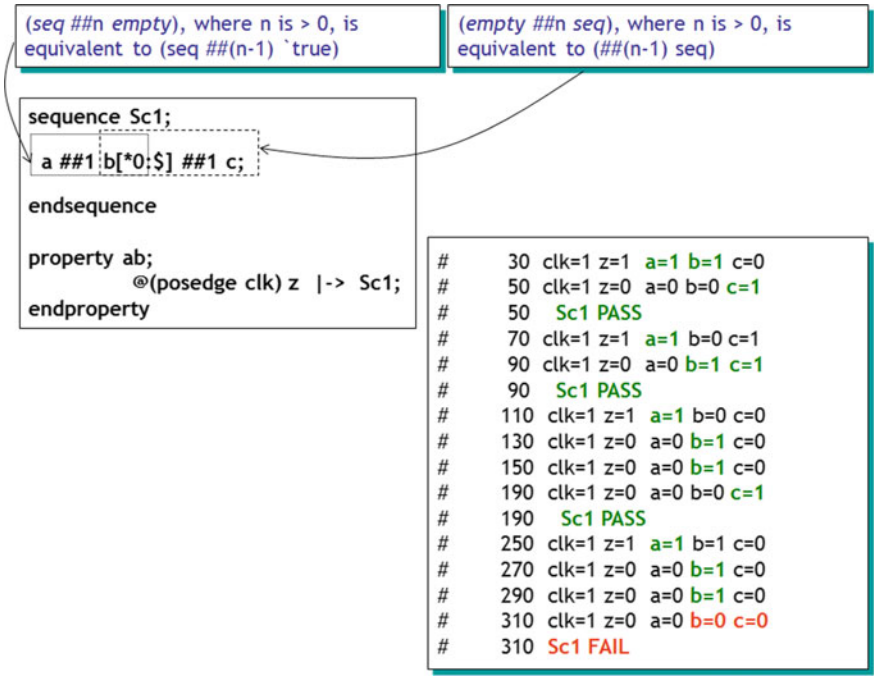


Fig. 14.26 empty match—example

Figure 14.26 is indeed interesting. LRM provides two rules on how to interpret (seq ##n *empty*) and (*empty* ##n seq) both with n > 0. The explanation is noted at the top of Fig. 14.26. To make that clear, let us go through the example.

Property 'ab' says that if the antecedent 'z' is true that the consequent sequence 'sc1' should execute. Sequence 'sc1' says that 'a' be true when 'z' is true; then (according to the LRM rule (seq ##n *empty* == seq##(n − 1) `true), the sequence can be read as 'a' be true; then 'b' may not be true (i.e. empty—does not exist) at all or will continue to repeat forever until c ==1.

Let us look at the simulation log to see if the new definition holds.

At 30, z = 1 so the property looks for 'a' = 1 at the same time. 'a' = 1 at 30. 'b' is also equal to '1'—which does not really matter because 'b' can have zero match, as long as there is c == 1 at 1 clock after the *last* 'b' or 'a'. In our case which started at 30, we do have c == 1 at 50 which is one clock after 'a == 1' as well as 'b == 1'. Hence the property passes.

At 70, z = 1, a = 1 but b = 0. That's an empty match. Hence, the property looks for c == 1 after the last 'a'. It finds that at 90 and the property passes.

At 110, z = 1, a = 1, b = 0. Next clock at 130, a = 0 and c is also equal to '0'—but 'b' = 1. As we saw before, 'b' may not match or may continually match forever until "c == 1". Since c == 0 at time 130, the property continues to look for b == 1

```
property ab;
          @(posedge clk) a |=> b [=0];
endproperty
```

b [=0]

means that the signal 'b' should *never* be true (in other words, it means that the # of non-consecutive occurrences are ZERO).

```
#       5  clk=1 a=1 b=0
#      15  clk=1 a=0 b=0
#      15          property ab PASS

#      35  clk=1 a=1 b=0
#      45  clk=1 a=0 b=0
#      45          property ab PASS
#      55  clk=1 a=0 b=1

#      65  clk=1 a=1 b=0
#      75  clk=1 a=0 b=1
#      75          property ab FAIL
```

Fig. 14.27 empty match example—II

until c == 1. That happens at time 150 (b == 1) and 190 (c == 1) and the property passes.

At 250, z = 1, a = 1 and b = 1. The next clock at 270, c is still zero, so the property continues to see that b remains '1'. At 290, b == 1, so we move on. But at time 310, 'b' does not remain asserted and 'c' is not equal to '1' either. 'c' should have been '1' (to satisfy b[*0:$] ##1 c)—or—'b' should have remain asserted. Neither happens and the property fails.

SystemVerilog 3.1a LRM; Page 210

(*empty* ##0 *seq*) does not result in a match

(*seq* ##0 *empty*) does not result in a match

(*empty* ##n *seq*), where n is > 0, is equivalent to (##(n-1) seq)

(*seq* ##n *empty*), where n is > 0, is equivalent to (seq ##(n-1) `true)

Examples:

b ##1 (a[*0] ##0 c) - will never produce a match

b ##1 a[*0:1] ##2 c is equivalent to
(b ##2 c) or (b ##1 a ##2 c)

Fig. 14.28 Empty sequence. Further rules

The example in Fig. 14.27 can be used effectively when you want to check that if a certain sequence takes place that another never takes place. You can do that with non-zero consecutive repetition operator also, but the [*0] or [=0] makes it that much easier.

For example, in the following example,

```
@ (posedge clk) a |=>b[= 0];
```

means that on 'a' being true, one clock later, 'b' should never occur. In other words, 'b' should be negated (zero) forever. This is quite straightforward to read/interpret. The behavior is shown in simulation log in Fig. 14.27.

So, how else would you write this property?

```
@ (posedge clk) a |=>!b[*1:$];
```

Same meaning. Once 'a' is true that starting next clock 'b' should remain '!b' consecutively forever.

Following is purely reference material. Keep it in your back pocket. It will be useful on a rainy day ! (Fig. 14.28.)

Chapter 15
Asynchronous Assertions!!!

Introduction: This chapter is solely devoted to Asynchronous Assertions, meaning the sampling edge of the assertion is not a synchronous clock rather an asynchronous edge. Special focus is on pitfalls of using an asynchronous assertion both as the sampling edge as well as an antecedent expression variable.

So far in the book we have always used a synchronous clock edge as the sampling edge for the assertion. That is for good reason. The example presented here uses an asynchronous edge (perfectly legal) as the sampling edge. The problem statement goes something like whenever (i.e. asynchronously) L2TxData == L2ErrorData that L2Abort is asserted. Now that looks very logical to implement without the need for a clock. So, we write a property as shown in the Fig. 15.1. We simply say that @ (L2TxData) (i.e. whenever L2TxData changes) that we compare L2TxData == L2ErrorData and if that matches we imply that L2Abort ==1.

This sounds very logical. What is wrong with it? Hmm… many things. The annotation in Fig. 15.1 takes you systematically on what is going on. Please study it carefully to see why there is a problem. We will not repeat that explanation of the figure again. But here's the high level hint on the problem. *The 'sampling edge' (namely, @ (L2TxData)) is also used in the comparison expression (L2TxData == L2ErrorData*). Since the value of variables in an expression are always sampled in the preponed region that the value of L2TxData in the expression won't be the same as when @ (L2TxData) changed. In other words, @ (L2TxData) uses the 'current' value of L2TxData, while the L2TxData in the expression (L2TxData == L2ErrorData), is the 'sampled' value. When such is the case, I strongly recommend against using an asynchronous assertion.

We continue the analysis in Fig. 15.2. The annotation explains the reasons.

In order to circumvent the problem, we just described in Fig. 15.2, we can continue with the asynchronous sampling edge, only that we put all the comparison expressions/variables as part of asynchronous sampling edge. This is shown in Solution 1 that will take care of the problems we first encountered. Think through and you will see why (Fig. 15.3).

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_15

EXAMPLE ON ASYNCHRONOUS ASSERTIONS ::

Whenever (i.e. asynchronously) L2TxData == L2ErrorData that L2Abort is asserted (*i.e. you want to check this condition irrespective of the clock*).

You may be tempted to write the property as follows:

```
property CheckData;
        @(L2TxData) (L2TxData == L2ErrorData) |-> (L2Abort == 1);
endproperty
```

:: Let us analyze how this property works ::

- *First, recall that the values of expression variables in an assertion are evaluated in the pre-poned region (i.e., variable value is that which existed a delta before the sampling edge (i.e. clock edge))*
- Now, let us assume that L2TxData changes and is now equal to L2ErrorData. This is what will happen

@(L2TxData) is triggered

and (L2TxData == L2ErrorData) expression is evaluated.

BUT

- The value of L2TxData compared in this expression is the value -before- L2TxData became equal to L2ErrorData. So, the expression won't match and implication won't trigger.

OK, so let's move on

- Now when L2TxData changes again (i.e. now it is NOT equal to L2ErrorData, assuming L2ErrorData did not change)

@(L2TxData) is triggered again

- and again, the value of L2TxData and L2ErrorData used in the expression are the ones -before- L2TxData changed (when they *did* actually match). So now the experssion will match and the implication will trigger checking for L2Abort == 1.
- But, what have you really proven? Read on ....

Fig. 15.1 Asynchronous assertion—problem statement

Why do we have assign #1 in solution 2? That way when L2TxData or L2ErrorDataW or L2ABortW change that there is a 1 time unit delay which will allow the new value to settle down and 'sample' the settled value—before—you check for (L2TxData == L2ErrorData).

This is a (convoluted) way to get around the check of these variables in the preponed region. If all this looks confusing, do not be daunted. I strongly advice you against using asynchronous edges as sampling edges when the same edge/expression is also used in the antecedent or the consequent. Again, if you are

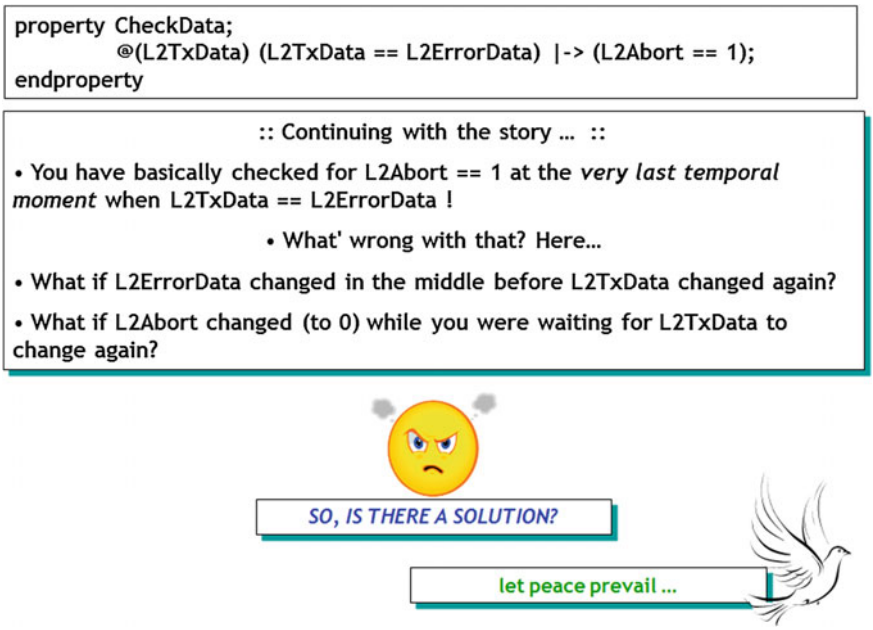


Fig. 15.2 Asynchronous assertion—problem statement analysis continued

comfortable with using them, please do so, but be careful. Refer to the example above to help you with the behavior of asynchronous sampling edge. Note that I have shown all three solutions with asynchronous assertion (so much for my opposition to it!). How would you model this as a synchronous assertion? Please try and see if you succeed. Assume 'posedge clk' as your sampling edge.

Solution 1 ::

```
property CheckData;
        @(L2TxData or L2ErrorData or L2Abort) (L2TxData == L2ErrorData)
                                        |-> (L2Abort == 1);
endproperty
```

Solution 2 ::

```
wire L2TxDataW, L2ErrorDataW, L2AbortW;

assign #1 L2TxDataW = L2TxData;
assign #1 L2ErrorDataW = L2ErrorData;
assign #1 L2AbortW = L2Abort;

property CheckData;
          @(L2TxDataW or L2ErrorDataW or L2AbortW) (L2TxData ==
                                        L2ErrorData) |-> (L2Abort == 1);
endproperty
```

Solution 3 :: Good old Verilog comes to rescue ...

```
always @(L2TxData or L2ErrorData or L2Abort)
begin
  if (L2TxData == L2ErrorData)
  begin
    if (L2Abort) $display("L2Abort Check PASS")
    else $display("L2Abort Check FAIL");
  end
end
```

Fig. 15.3 Asynchronous assertion—solution

Solution 3 uses a procedural block to determine when you do the check. Notice that I have not used assertions in this solution. Point being, sometimes it is better and ok to simply use Verilog which will be more intuitive and give the results you desire. I recommend pure Verilog for asynchronous assertions and SVA for all other types of assertions.

Chapter 16
IEEE-1800-2009/2012 Features

Introduction: This chapter describes all the new features of the 2009/2012 LRM. In that sense, it is a long chapter. It describes features such as 'strong' and 'weak' properties, abort system tasks, deferred immediate assertions, past and future global clock based sampling functions such as $rose_gclk, $fell_gclk, $rising_gclk, $falling_gclk, etc. It further covers 'followed by' property operators, 'always', 'eventually', 'until', 'nexttime', 'case', $inferred_clock and $inferred_disable. Finally, it goes into detail of the 'let' construct and 'checkers'.

16.1 Strong and Weak Sequences

IEEE-1880-2009/2012 adds the notion of a strong and weak operator applied to sequence expressions. The idea behind these 'strengths' is very simple.

Here's an example

```
property a_wait_b;
    @ (posedge clk) A |-> (A ##[1:$] B);
endproperty
awb: assert property (strong(a_wait_b)) else $display($stime,"a_wait_b FAIL"); //default 'weak'
awbc: cover property (weak(a_wait_b)) $display($stime,"a_wait_b PASS"); //default 'strong'
```

'strong' sequence means that if you run out of simulation ticks (at the end of simulation, for example), the 'strong' sequence will FAIL. In other words, 'strong' will evaluate to true only if there is a non-empty match of the sequence expression.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_16

And in yet other words, the 'strong' operator requires 'enough' ticks to witness a success. In our example, if 'B' never arrives until the end of simulation, the property will FAIL. By default, an 'assert' (or 'assume') is 'weak' and as we have seen so far, if you run out of simulation ticks the sequence will *not* fail (a simulator may still give an indication of an incomplete sequence).

On the other hand, 'cover' is strong by default. Analogous to 'assert', if the property does not complete, the evaluation of sequence expression does not succeed and the 'cover' will be considered to FAIL (i.e. not covered). In other words, an incomplete 'cover' sequence will not give us a 'PASS' or 'cover' indication, because there haven't been enough ticks to reach a 'success' state. That is exactly what we want because we do not want an incorrect 'cover' of a sequence that never completes. On the other hand, if you use 'weak' operator with 'cover' and the sequence never completes, the 'cover' will be considered to have completed or covered (this is simulator dependent from author's experience, so take the description of cover with a 'weak' operator with a grain of salt).

In short, by default, a property is weak in the context of an 'assert' (or an 'assume') and is strong in the context of a 'cover'.

16.2 Deferred Immediate Assertions

Deferred immediate assertions are a type of 'immediate' assertions. Recall that 'immediate' assertions evaluate immediately without waiting for variables in its combinatorial expression to settle down. This also means that the immediate assertions are very prone to simulation glitches as the combinatorial expression settles down (for example, an expression evaluates to '0' then to '1' then back to '0' to settle down on '0'), the immediate assertion may fire multiple times. On the other hand, deferred assertions do not evaluate their sequence expression until the end of time tick when all values have settled down (or in the reactive region of the time tick).

The syntax for deferred immediate assertion is "assert #0" or "assert final". It's the #0 (or 'final') that distinguishes deferred immediate assertion from the immediate assertion.

Let us examine the following example (2009 LRM).

```
assign not_a = !a;
always_comb begin:b1
 a1: assert (not_a != a)  //immediate
 a2: assert #0 (not_a != a);   //Deferred immediate
 a3: assert final (not_a !=a)  //Deferred immediate
end
```

Let us examine the difference between immediate and deferred immediate assertions in this example. As soon as 'a' changes, always_comb wakes up and both the immediate and deferred assertions fire right away. When the immediate assertion fires, the continuous assignment "not_a = !a" may not have completed its assignment. In other words, 'a' has not been inverted yet. But the immediate assertion expects an inverted 'a' on 'not_a'. The assertion will fail. This is why immediate assertions are known to be glitch prone.

On the other hand, the deferred assertion will wait until all expressions in the given time stamp have settled down (in other words, it was put in the deferred assertion report queue). In our case, the continuous assign would have completed its evaluation by the end of time stamp (meaning when the deferred assertion queue will be flushed); and 'not_a' would indeed be = !a. This is the value the deferred assertion will take into account when evaluating its expression. The deferred assertion will pass.

To reiterate, in a simple immediate assertion, pass and fail actions take place immediately upon assertion evaluation. In a deferred immediate assertion, the actions are delayed until later in the time step, providing some level of protection against unintended multiple executions on transient or "glitch" values.

Note that there is a limitation on the action block that a deferred immediate assertion has. The action block can only be a single subroutine (a task or a function). The requirement of a single subroutine call also implies that no begin-end block can surround the pass or fail statements, as begin is itself a statement that is not a subroutine call. A subroutine argument may be passed by value as an input or passed by reference as a ref or const ref. Actual argument expressions that are passed by value, including function calls, will be fully evaluated at the instant the deferred assertion expression is evaluated. It is also an error to pass automatic or dynamic variables as actuals to a ref or const ref formal.

For example, following action block is illegal with deferred assertions because either they contain more than one statement or are not subroutine calls.

```
frameirdy: assert #0 (!frame_ == irdy) else begin interrupt=1; $error ("FAILure"); end //ILLEGAL
frameirdy: assert #0 (!frame == irdy) else begin $error("FAILure"); end //ILLEGAL
```

Following are legal.

```
frameirdy: assert #0 (!frame == irdy) else $error("FAILure"); //LEGAL (no begin-end)
frameirdy: assert #0 (!frame == irdy) $info("PASS"); else $error("FAILure"); //LEGAL
frameirdy: assert #0 (!frame == irdy); //LEGAL – no action block
```

Another feature of deferred immediate assertion to note is that it can be declared both in the procedural block as well as outside of it (recall that immediate assertion

can only be declared in procedural block). For example, following is legal for deferred immediate assertion but not for immediate assertion.

```
module (x,y,z);
        .....
        z1: assert #0 (x == y ||z);
endmodule

This is actually equivalent to

module (x,y,z);
        always_comb begin
        z1: assert #0 (x == y||z);
        end
endmodule
```

Analogous to ‘assert’, we also have deferred ‘cover’ and ‘assume’. Again, the idea is the same as that for immediate ‘cover’ and ‘assume’ you want the final values of the combinatorial logic before evaluating ‘cover’ or ‘assume’.

A deferred ‘assume’ will often be useful in cases where a combinational condition is checked in a function, but needs to be used as an assumption rather than a proof target by formal tools. A deferred cover is useful to avoid crediting tests for covering a condition that is only met in passing by glitch values.

```
    assign a = c || d;

    assign b = e || f;

    always_comb begin:b1

     a1: cover (b !=a)  //immediate

     a2: cover #0 (b != a);   //deferred cover

     a3: cover final (b !=a)  //deferred cover

    end

    and for ‘assume’

    assign a = c || d;

    assign b = e || f;

    always_comb begin:b1

     a1: assume (b !=a)  //immediate

     a2: assume #0 (b != a);   //deferred assume

     a3: assume final (b !=a)  //deferred assume
end
```

Some more nuances of deferred assertions are further explained below.

Disabling a deferred assertion.

The following example illustrates how user code can explicitly flush a pending assertion report. In this case, failures or successes of 'a1' are only reported in time steps where 'NoGo' does not settle at a value of 1.

```
always @(NoGo or Go)
begin : b1
        a1: assert final (Go) else $fatal(1, "Sorry");
        if (NoGo)
        begin
                disable a1;
        end
end
```

On the similar line of thought, the following example illustrates how user code can explicitly flush *all* pending assertion reports on the deferred assertion queue of process 'b2':

```
always @(a or b or c)
begin : b2
        if (c == 8'hff)
        begin
                a2: assert final (a && b);
        end
        else begin
                a3: assert final (a || b);
        end
end

always @(NoGo)
begin : b3
        disable b2;
end
```

Finally, unlike immediate assertions, the deferred immediate assertion can be placed outside of procedural code. When declared outside a procedural block, the deferred immediate assertion is treated semantically as if the assertion were enclosed with an always_comb block.

For example, here's the code with always_comb:

```
assign Frame_ = !Frame_ && IRDY;
assign ACK = Req && Gnt;
always_comb a1: assert #0 (Frame_ || ACK);
```

The same deferred immediate assertion can be written as follows:

```
assign Frame_ = !Frame_ && IRDY;
assign ACK = Req && Gnt;
a1: assert #0 (Frame_ || ACK);
```

16.3 $changed

SystemVerilog 2009/2012 adds $changed sampled value function (Fig. 16.1) in addition to the ones we have already seen such as $past, $rose, $fell and $stable.

Here's a simple example of where $changed is helpful.

Specification: Make sure that 'toggleSig' toggles every clock. In other words, see that 'toggleSig' follows the pattern 101010… or 010101…

Solution: First inclination will be to write the assertion as follows.

```
tP: assert property (@(posedge clk) toggleSig ##1 !toggleSig);
```

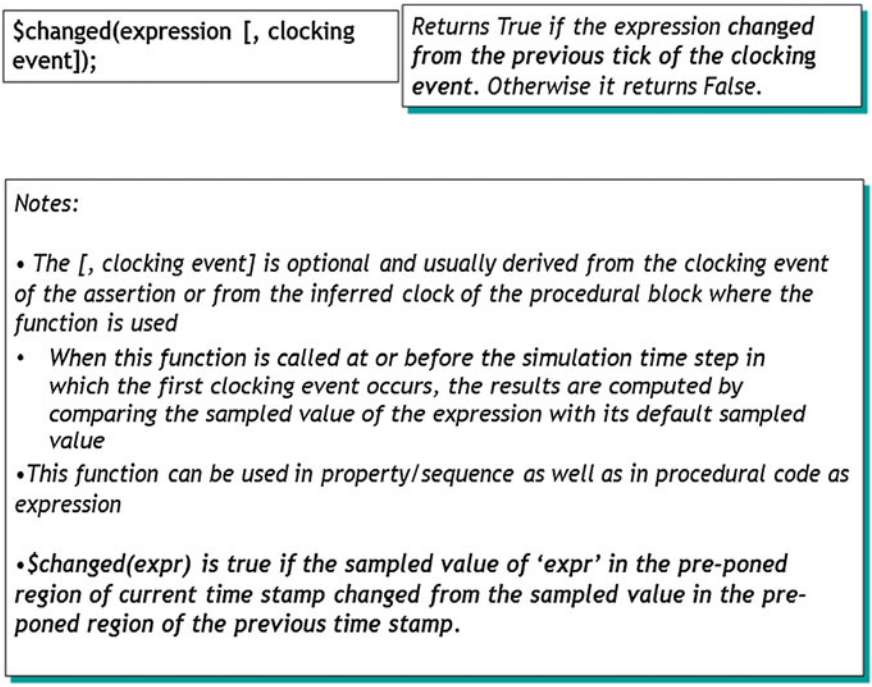
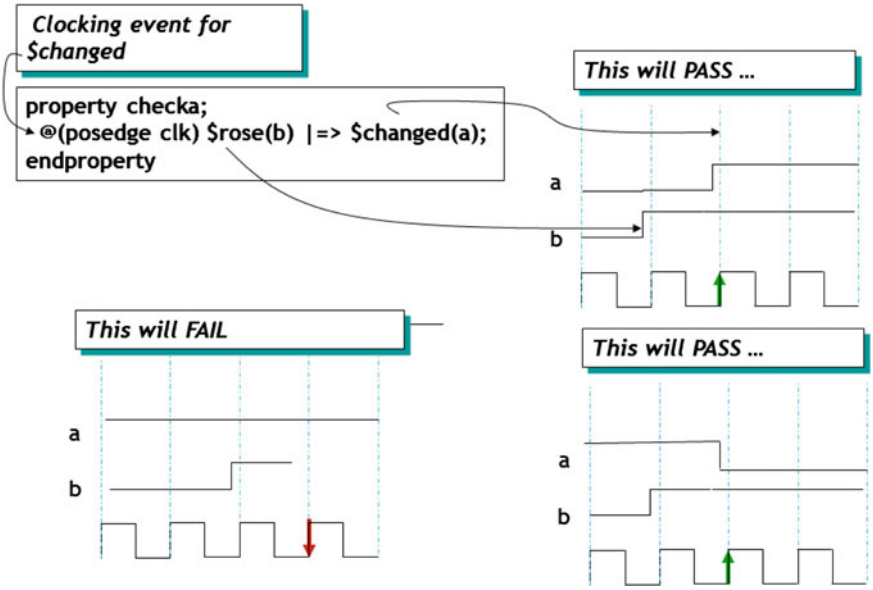


Fig. 16.1 $changed

But will this work? No. This property simply states that toggleSig be true every clock that it is false the next clock. What that also means is that the next clock, we are checking for toggleSig to be both true and false at the same time! Totally contradictory.



Here's where $changed comes to rescue. Following property will verify the toggle specification.

```
tP: assert property (@(posedge clk) ##1 $changed(toggleSig));
```

16.4 $sampled

$sampled simply does—explicitly—what we have seen assertions do. In other words, the expressions in an assertion are always *sampled* in the preponed region of a time stamp. $sampled does exactly the same. It returns the value of an expression sampled in the preponed region of the simulation time stamp in which the function is called.

So, *for concurrent assertions $sampled function is redundant*. Following two are equivalent

```
z1 : assert property (@ (posedge clk) $sampled(a) == $sampled(b));
z1 : assert property (@ (posedge clk) a == b));
```

The reason they are the same is that in the concurrent assertion (as we have seen throughout the book) the expressions are always sampled in the preponed region of the time stamp in which they are sampled. $sampled function also returns value of expression in the preponed region.

However, there are places where $sampled can be useful for debug purpose. For example, in a simultaneously changing event situation, you can find out the sampled value of an expression (i.e., the value in the preponed region). Let us say, you have an assertion where you want to make sure that 'gnt' is asserted on a posedge clk. If 'gnt' and posedge clk went high at the same time, the property will fail. That's because the sampled value of 'gnt' is 0 in the preponed region. In such a case you can have a $display("gnt=%b", $sampled(gnt)); This will tell you right away that the sampled value was '0' which is why the assertion failed.

Important note: The assertion system function $sampled does *not* use a clocking event (note that in the 2005 standard, the explicit clocking event was indeed required but that requirement was removed from the 2012 standard). And as we know, for a sampled value function other than $sampled, the clocking event will be explicitly specified as an argument or inferred from the code where the function is called.

From LRM:

IEEE Std 1800-2005 required that an explicit or inferred clocking event argument be provided for the $sampled assertion system function. In the 2012 version of the standard, the semantics of $sampled have been changed to a form that does not depend on a clocking event. Therefore, the syntax for defining the clocking event argument to $sampled is deprecated and does not appear in this version of the standard.

Also, the use of $sampled in a 'disable iff' clause is meaningful since *the disable condition by default is not sampled*. 'disable iff' by default is asynchronous. So what if you want to make it synchronous, i.e. sampled on a clock edge? Here's how

```
property syncIff;
(@posedge clk) disable iff ($sampled(rst)) a |=> b;
endproperty
```

16.5 $past_gclk, $rose_gclk, $fell_gclk, $stable_gclk, $changed_gclk, $future_gclk, $rising_gclk, $falling_gclk, $steady_gclk, $changing_gclk

As the name suggests, these sampled value functions work off of a global clock. Before we look into these functions, let us see what a global clock is.

```
module top_pd;
  logic clk;
  global clocking sys_clk @ (clk); endclocking
  ....
endmodule
```

'sys_clk' is now considered a global clocking event. It is defined to occur (trigger) if there is a change in 'clk'. You can access global clock using the system function $global_clock. This system function does not take any arguments but returns the event expression specified in the global clocking declaration. Note that the specification of the name sys_clk in the global clocking declaration is optional since the global clocking event may be referenced by $global_clock. The $global_clock system function will be used to explicitly refer to the event expression in the effective global clocking declaration.

A reference to $global_clock is understood to be a reference to a *clocking_event* defined in a global clocking declaration. A global clock behaves just as any other clocking event. Thus, in the following example:

```
global clocking @clk; endclocking
assert property(@$global_clock a);
```

the assertion states that a is true at each tick of the global clock. This assertion is logically equivalent to:

```
assert property(@clk a);
```

Global clocking is a SystemVerilog feature and a detailed explanation is beyond the scope of this book. However, here are some high level points.

- A clocking block may be declared as global clocking for all or part of the design hierarchy. In other words, such a specification may be done for a whole design, or separately for different subsystems in a design.
- Global clocking may be declared in a module, an interface, a checker, or a program. A given module, interface, checker, or program can contain at most one global clocking declaration.
- Although more than one global clocking declaration may appear in different parts of the design hierarchy, at most one global clocking declaration is effective at each point in the elaborated design hierarchy.

Let us revert to the system functions. *These functions can only be used if there is a global clock defined in your testbench* (hence the suffix _gclk). They are sampled value functions as we have seen before and they sample their expression value at the global clock tick that you have defined. These sampled value functions are divided into two groups. One group looks in the past (these are identical in functionality with the sampled value functions we have seen previously only that the _gclk sampled value functions work off a global clock event). The other group

has 'future' sampled value functions. They sample the value of the expression in the time step that subsequently (and immediately) follows the time step in which the function is called. Note that the 'past' sampled value functions have a non-global clock counterpart as we have seen. *However, for the* '*future*' *sampled value functions, there is no non-global clock counterpart*. They work only if you have defined a global clock.

The *past* sampled value functions are

$past_gclk (expression)
$rose_gclk (expression)
$fell_gclk (expression)
$stable_gclk(expression)
$changed_gclk(expression)

The *future sampled value functions* are

$future_gclk(expression)
$rising_gclk(expression)
$falling_gclk(expression)
$steady_gclk(expression)
$changing_gclk(expression)

As mentioned before the globally clocked *past* sampled value functions work the same way as the non-global clocking sampled value function.

If you recall, these past sampled value functions take an explicit clocking event. So, $rose_gclk (expr) is equivalent to $rose(expr, @ $global_clock). Please refer to the non-global clocking past sampled value functions to understand how these functions work (Chap. 5).

The future sampled value functions are also similar except that they use the subsequent (future) value of the expression. Here's a brief explanation of each.

$future_gclk(expression) returns the sampled value of expression at the next global clocking tick.
$rising_gclk(expression) returns a Boolean True if the sampled value of the *least significant bit* of the expression changes to 1 at the next global clocking event. Else it returns false.
$falling_gclk(expression) returns a Boolean True if the sampled value of the *least significant bit* of the expression changes to 0 at the next global clocking event. Else it returns false.
$steady_gclk(expression) returns a Boolean True if the sampled value of the expression does not change at the next global clocking event. Else it returns false
$changing_gclk (expression) returns a Boolean True if the sampled value of the expression changes at the next global clocking event. Else it returns false.

An example:

Specification: Frame_ signal should be stable between two consecutive clock ticks. Frame_ can change only at the positive edge of clock – no glitch in-between.

Solution:

```
aframe_ : assert property (@ $global_clock disable iff (!irdy_)
$changing_gclk(Frame_) |-> $rising_gclk(Frame_));
```

Exercise: Can you write the same assertion using $changed and $rose? Please experiment to solidify your understanding of past sampled value functions and future sampled value functions.

Execution of the action block of an assertion containing global clocking future sampled value functions is *delayed* until the global clocking tick that follows the last tick of the assertion clock for the attempt.

Following are illegal conditions for the global clocking future sampled value functions:

1. *The* '*future*' *sampled value functions cannot be used outside of concurrent assertions.*
2. They cannot be nested (for example, $future_gclk ($falling_gclk(gnt_) && req)). But do not confuse this with the following which is legal
   F1: assert property (@$ global _clock $rising_gclk(sig1) |-> $falling_gclk (sig2));
3. They cannot be used in a 'reset' condition (for example, "disable_iff ($falling_gclk (reset_)").
4. The global clocking future sampling functions cannot be used in an assertion *action block (pass or fail).*

16.6 'followed by' Properties #-# and #=#

The *followed by* properties has the following form.

sequence_expression #-# property_expression
sequence_expression #=# property_expression

#-# is the overlapped property and #=# is the non-overlapped, just as in |-> and |=> but there are differences between the *implication* operators and the *followed by* operators.

For '*followed by*' to succeed *both* the antecedent sequence_expression and the consequent property_expression must be true. *If the antecedent sequence_expression does not have any match, then the property fails*. If the sequence_expression has a match, then the consequent property_expression must match.

This is the fundamental difference between the implication operators (|-> and |=>) and the followed by operators. Recall that with implication operators, if the antecedent does not match, you get a vacuous pass and not a fail.

For overlapped followed-by, there must be a match for the antecedent sequence_expr, where the end point of this match is the start point of the evaluation

of the consequent property_expr. For nonoverlapped followed-by, the start point of the evaluation of the consequent property_expr is the clock tick after the end point of the match.

Obviously, #-# being an overlapped operator, it starts the consequent evaluation the same time that the antecedent match ends (and succeeds). Consequently, the # = # non-overlapped operator will start the consequent evaluation the clock after the antecedent match ends and succeeds.

Here's a simple example

```
property p(a, b)
@ (posedge clk) a #=# b;
endproperty
assert property (p(req[*5],gnt));
```

Request need to remain asserted (high) for 5 consecutive clocks. One clock later gnt must be asserted (high). If request does *not* remain asserted for 5 consecutive clocks, the assertion will fail. If it does remain asserted for 5 clocks and the next clock gnt is not asserted, the assertion will fail. If both the antecedent and consequent match in the required temporal domain, the property will pass.

If you think about it, a #-# b behaves pretty much like a ##0 b. So what's the difference? a ##0 b requires that 'b' is a sequence while a#-#b allows 'b' as a property. The same discussion applies to a #=# b, which is equivalent to a ##1 b, except that 'b' can be a property.

16.7 'always' and 's_always' Property

'always' property behaves exactly as you would expect. The syntax for 'always' (and its variations) is

1. **always property_expression** (weak form)
2. **always [cycle delay constant range expression] property_expression** (weak form with *unbounded* range)
3. **s_always [constant_range] property_expression** (strong form with *bounded* range)

So, let us see how 'always' works. As LRM puts it "A property 'always property_expression' evaluates to true if and only if the property_expression holds at every current and future clock tick". Rather self-explaining. Here's a simple example.

```
property reset_always;
@ (posedge clk) POR[*5:10] |=> always !reset;
endproperty
```

The property says that once POR (power on reset) signal has remained high for minimum 5 clocks or maximum 10, that starting next clock, reset would remain de-asserted 'always' (forever).

'always' makes it simple to specify the continuous longevity of an assertion.

Next, let us see how always [cycle delay constant range] works.

```
property p1;

@ (posedge clk) a |-> always [3:$] b;

endproperty
```

property p1 says that if 'a' is true that 'b' will be true 3 clocks *after* 'a' and will remain true 'always' (forever) after the 3 clocks. Note that 'always[n:m]' allows an unbounded range.

In contrast 's_always' allows only bounded range.

So, let us see what s_always does

```
property p2;

@ (posedge clk) a |->s_always [3:10] b;

endproperty
```

The property says that if 'a' is true that 'b' remains true from 3rd clock to 10th clock after 'a' was detected true. This is a 'strong' property. Recall strong property that we discussed earlier. This 's_' property also works the same way. In other words, if you run out of simulation ticks for 's_always', the property will indeed fail.

BUT, why do we need '**always'**? Don't the concurrent assertions always execute at every clock tick? The answer is yes which means we do not always need an '**always**' operator with a concurrent assertion. It is redundant. For example, in the following, '**always**' is redundant.

```
P1: assert property p1p (@ (posedge clk) always bstrap1==0);
```

There is no reason for an 'always' in the above concurrent assertion. It is the same as

```
P1: assert property p1p (@ (posedge clk) bstrap1==0);
```

'always' can be useful in 'initial' block, however. See the example below

```
initial

begin

        P1: assert p1p (@ (posedge clk) always bstrap1==0);

end
```

Note that the immediate assertion noted above is slated to execute only once. In our case though once it is asserted, it will then look for bstrap1==0 at every posedge clock.

As LRM puts it: The concept of weak and strong operators is closely related to an important notion of safety properties. Safety properties have the characteristic that all their failures happen at a finite time. For example, the property *always a* is a safety property since it is violated only if after finitely many clock ticks there is a clock tick at which *a* is false, even if there are infinitely many clock ticks in the computation.

16.8 'eventually', 's_eventually'

There are two types of this operator, the 'weak' kind ('**eventually**') and the 'strong' kind ('**s_eventually**'). Here are three forms of these two properties.

s_eventually property_expr (*strong* property without range)
s_eventually [cycle_delay_constant_range] property_expr (*strong* property with range)

- the constant_range can be unbounded

eventually [constant_range] property_expr (*weak* property with range)

- the constant_range must be bounded

Some examples

```
property p1;
s_eventually $fell(frame_);
endproperty
```

Eventually PCI cycle will start with assertion of frame_ (frame_ goes low). If frame_ does not assert until the end of simulation time, the property will fail since this is a *strong* property. Note that frame_ can be true in current clock tick or any future clock tick.

```
property p2;
s_eventually [2:5] $fell(frame_);
endproperty p2;
```

A new PCI cycle must start (frame_ goes low) within the range of 2 clocks from now (i.e. frame_ cannot assert earlier than the 2nd clock) and eventually by 5th clock (2nd and 5th clock inclusive). Note that as with any *strong* property, s_eventually[n:m] property_expr evaluates to true if, and only if, there exist at least n+1 ticks of the clock of the eventually property, including the current time step,

and property_expr evaluates to true beginning in one of the n+1 to m+1 clock ticks, where counting starts at the current time step.

Exercise: Is the following property 'p3' equivalent to 'p2' above? Hint: Simulate from 'initial' condition to know the subtle difference.

```
property p3;
  frame_ |-> ##[2:5] $fell(frame_);
endproperty
```

Following is s_eventually with unbounded range.

```
property p4;
s_eventually [2:$] $fell(frame_);
endproperty
```

A new PCI cycle must start (from the current clock tick) 2 clocks from now (i.e. frame_ cannot assert earlier than the 2nd clock) or any time after that.

```
property p4;
  eventually [2:$] $fell(frame_); //ILLEGAL. Weak property must be bound.
endproperty
property p4;
  s_eventually always a;
endproperty
```

'a' (Boolean) will eventually (starting current clock tick) go high and then remain high at every clock tick after that until the end of simulation.

16.9 'until', 's_until', 'until_with' and 's_until_with'

There are 4 forms of 'until' property

1. property_expression1 **until** property_expression2 (weak form—non-overlapping)
2. property_expression1 **s_until** property_expression2 (strong form—non-overlapping)
3. property_expression1 **until_with** property_expression2 (weak form—overlapping)
4. property_expression1 **s_until_with** property_expression2 (strong form—overlapping)

Let us start with 'until'.

```
property p1;
   req until gnt;
endproperty
```

property p1 is true if 'req' is true until 'gnt' is true. In other words, 'req' must remain true as long as 'gnt' is false. 'req' need not be true at the clock tick when 'gnt' is found to be true (but it can be). In other word, **until** is nonoverlapping. An **until** property of the non-overlapping form evaluates to true if 'req' evaluates to true at every clock tick beginning with the starting clock tick of the evaluation attempt and continuing until at least one tick *before* a clock tick where 'gnt' evaluates to true. If 'gnt' is never true, 'req' will remain true at every current and future clock tick. Since **until** is of weak form, if this property never completes (i.e. 'gnt' is never true), the property will *not* fail.

```
property p1;
   req s_until gnt;
endproperty
```

s_until is identical to **until** except that if 'gnt' never arrives and you run out of simulation time, the property will fail.

To reiterate the difference between strong and weak properties, an 'until' property of one of the strong forms requires that a current or future clock tick *exists* at which 'gnt' evaluates to true, while an 'until' property of one of the weak forms does not make this requirement. Strong properties require that some terminating condition happen in the future, and this includes the requirement that the property clock ticks enough time to enable the condition to happen. Weak properties do not impose any requirement on the terminating condition, and do not require the clock to tick.

```
property p1;
   req until_with gnt;
endproperty
```

property p1 is true if 'req' is true until and *including* a clock tick when 'gnt' is true. In other words, 'req' must remain true as long as 'gnt' is false. 'req' must be true at the *same* clock tick when 'gnt' is found to be true. If 'gnt' is never true, 'req' will remain true at every current and future clock tick. In other words, **until_with** is an overlapping property. Since 'until_with' is of weak form, if this property never completes (i.e. 'gnt' is never true), the property will *not* fail.

In short, property '**until_with**' requires 'req' and 'gnt' to be true at the *same* clock tick when 'gnt' is found to be true. '**until'** does not have this requirement.

```
property p1;
   req s_until_with gnt;
endproperty
```

Same as **until_with** but if you run out of simulation tick (end of simulation, for example), and if 'gnt' is never found to be true, this property will fail.

16.10 'nexttime' and 's_nexttime'

'nexttime' property_expression evaluates to true, if property_expression is true at time t+1 clock tick.

There are 4 forms of 'nexttime'.

nexttime property_expression (weak form)

The weak nexttime property **nexttime** *property_expr* evaluates to true if, and only if, either the *property_expr* evaluates to true beginning at the next clock tick or there is no further clock tick.

s_nexttime property_expression (strong form)

The strong nexttime property **s_nexttime** *property_expr* evaluates to true if, and only if, there exists a next clock tick and *property_expr* evaluates to true beginning at that clock tick.

nexttime [constant_expression] property_expression (weak form)

The indexed weak nexttime property **nexttime** [*constant_expression*] *property_expr* evaluates to true if, and only if, either there are not *constant_expression* clock ticks or *property_expr* evaluates to true beginning at the last of the next *constant_expression* clock ticks.

s_nexttime [constant_expression] property_expression (strong form)

The indexed strong nexttime property **s_nexttime** [*constant_expression*] *property_expr* evaluates to true if, and only if, there exist *constant_expression* clock ticks and *property_expr* evaluates to true beginning at the last of the next *constant_expression* clock ticks.

Let us examine the following simple example.

```
property p1;
  @ (posedge clk) nexttime req;
endproperty
```

The above property says that the property will pass, if the clock ticks once more and 'req' is true at the next clock tick (t+1). In addition, since this is the weak form, if you run out of simulation ticks (i.e. there is no t+1), this property will not fail.

Some examples.

What if you want to check to see that 'req' remains asserted for all the clocks following the next clock? Following will do the trick.

```
property p1;
  @ (posedge clk) nexttime always req;
endproperty
```

Or what if you want to see that starting next clock, 'req' will eventually become true? Following will do the trick.

```
property p1;
  @ (posedge clk) nexttime eventually req;
endproperty
```

What if you want to see that 'req' is true after a certain exact # of clocks? Following will do the trick

```
property p1;
  @ (posedge clk) nexttime[5] req;
endproperty
```

This property says that 'req' shall be true at the fifth future clock tick (provided that there are indeed 5 clock ticks in future, of course).

```
property p1;
  @ (posedge clk) s_nexttime req;
endproperty
```

Same as 'nexttime' except that if you run out of simulation ticks after the property is triggered (i.e. there is no (t+1), the property will fail. Other way to look at this is that there exists a next clock and 'req' should be true at that next clock, else the property will fail.

Similarly, the following property says that there must be at least 5 clock ticks and that 'req' will be true at the fifth future clock tick.

```
property p1;
  @ (posedge clk) s_nexttime [5] req;
endproperty
property
  @ (posedge clk) (seq 1.matched nexttime seq_expr == 'hff);
endproperty
```

When 'seq 1' ends(matches) at 't' that the next time tick ('t+1') 'seq_expr' must be equal to 'hff'.

One more real life issues we face that can be solved with **nexttime**. Initial 'x' condition can always give us false failures. This can be avoided with the use of **nexttime**. For example,

Let us say you are using $past to do a simple 'xor' of past value and present value (Gray encoding).

```
property
@(posedge clk)

$onehot (fifocntr ^ $past (fifocntr);

endproperty
```

At time 'initial' $past (fifocntr) will return 'x' (unknown) and the 'xor' would fail right away. This is a false failure and you may spend unnecessary time debugging it. Here's how **nexttime** can solve that problem.

```
property
@(posedge clk)

nexttime $onehot (fifocntr ^ $past (fifocntr);

endproperty
```

nexttime will avoid the initial $past value of fifocntr and move the comparison to the next clock tick when (hopefully) you have cleared the fifocntr and the comparison will not fail due to the initial 'x'.

Similarly, if you want to know that a signal stays stable forever (e.g. bootstrap signals). You may write a property as follows

```
property
@(posedge clk)

$stable (bstrap);

endproperty
```

BUT, this will sample the value 'x' (e.g. for a 'logic' type which has not been explicitly initialized) at time tick 0 and then continue to check to see that it stays at 'x'. You end up checking for a stable 'x'. Completely opposite of what you want to accomplish. Again, **nexttime** comes to rescue.

```
property
@(posedge clk)

nexttime $stable (bstrap);

endproperty
```

This will ensure that you start comparing the previous value of bstrap with the current value, starting *next* clock tick. Obvious, but easy to miss.

We discussed multi-clock properties in Chap. 8. Here's an example of how **nexttime** can be used in a multi-clock property.

```
N1: assert property
@(posedge clk1) x |-> nexttime @(posedge clk2) z;
```

It is very important to understand how this property works. The 'posedge clk1' flows through to 'nexttime'—in other words, 'nexttime' does not use '@(posedge clk2)' to advance time to next tick. So, when 'x' is true at ('posedge clk1'), the 'nexttime' causes advance to the next occurrence of 'posedge clk1' strictly after when 'x' was detected true before looking for a concurrent or subsequent occurrence of 'posedge clk2' at which to evaluate 'z'.

Exercise: How would the following property contrast with the one above?

```
N1: assert property
@(posedge clk1) x |-> @(posedge clk2) nexttime z;
```

16.11 'case' Statement

The *case* statement in assertions is the same as the one we use in systemverilog language. There is no difference, only that in systemverilog assertions, you use the *case* statement in a property. The *case* property statement is a multi-way decision making mechanism that tests a Boolean expression and sees if it matches one of a number of Boolean expressions. On a match, it will take action specified for that case statement. We are all familiar with this functionality of *case*. The 'default' statement is optional.

Here is a simple example.

```
property CycleCase (logic [1:0] CycleType);
  case (CycleType)
    2'b00: $fell(frame_) ##1 (cmd==READ);
    2'b01: $fell(frame_) ##1 (cmd==WRITE);
    2'b10: $fell(frame_) ##1 (cmd==TABORT);
    2'b11: $fell(frame_) ##1 (cmd==MABORT);
   default: $fell(frame_) ##1 (cmd==ILLEGAL); //default is optional
endcase
endproperty
```

Note that if the default statement is not given and all of the comparisons fail, then none of the case item property statements are evaluated. In addition, as we know if "assert property ()" antecedent does not evaluate to true that we get a vacuous pass. The same applies here.

If there is no default and no case branch match, we get a vacuous pass.

16.12 $inferred_clock and $inferred_disable

Many times while developing assertion logic, we define default blocks for clock and reset. These default blocks based clock and reset are then available in properties that follow. Please refer to the Sect. 4.3.1 on Default Clocking for further understanding of a default-clocking block.

The inferred clocking event expression is the current resolved event expression that can be used in a clocking event definition. It is obtained by applying clock flow rules to the point where $inferred_clock is called. If there is no current resolved event expression when $inferred_clock is encountered then an error is issued.

The inferred disable expression is the disable condition from the default disable declaration whose scope includes the call to $inferred_disable. If the call to $inferred_disable is not within the scope of any default disable declaration, then the call to $inferred_disable returns 1'b0 (false).

– $inferred_clock returns the expression of the inferred clocking event.
– $inferred_disable returns the inferred disable expression.

Let us say you have the following default blocks:

```
module (…,clk, rst,…);
default clocking @ (negedge clk); endclocking
default disable iff rst;
```

One of the ways to use this default clocking and reset blocks is as follows.

```
property inferB(a, b, c, clk=$inferred_clock, reset=$inferred_disable);

@ (clk) disable iff (reset) a |=> b || c;

endproperty
assert property (inferB(x, y, z));
```

The formal parameters of property inferB uses default clocking and reset from their respective default blocks. In other words, "@ (clk)" is now "@ (negedge clk)". Similarly, "disable iff (reset)" is now "disable iff (rst)".

Note that if property 'inferB' is invoked as follows, the $inferred_clock will not take effect—but the actual clocking event 'posedge clk' will take effect.

```
assert property inferB (a, b, c, posedge clk, reset);
```

@ (clk) in property inferB will be '@ (posedge clk)' and *not* '@ (negedge clk)' as in the default clocking block. In other words, the actual overwrites the formal argument, as always.

From the above we can see that the inferred clocking event expression is the current resolved event expression that can be used in a clocking event. Of course, if you use $inferred_clock and there is no default clocking block defined, you will get an Error.

Here's simple Verilog code that exemplifies above description.

```
module inferred_clock;
logic a,b,c,rst,clk,reset,x,y,z;
default clocking negclock @ (negedge clk); endclocking
default disable iff rst;
property inferB (a, b, c, clk=$inferred_clock, reset = $inferred_disable);
   @ (clk) disable iff (reset) a |=> b || c;
endproperty
assert property (inferB (x, y, z, clk, reset)) else $display ($stime,,,"FAIL");
cover property (inferB (x, y, z, clk, reset)) $display ($stime,,,"PASS");
initial
begin
   clk=0; reset=0;
   x=1; y=0; z=0;
   #40; y=1;
end
always #10 clk = !clk;
initial $monitor($stime,,,"clk=",clk,,,"reset=",reset,,,"a=",x,,"b=",y,,"c=",z);
endmodule

/*
#       0  clk=0  reset=0  a=1 b=0 c=0
#      10  clk=1  reset=0  a=1 b=0 c=0
#      20  FAIL
#      20  clk=0  reset=0  a=1 b=0 c=0
#      30  FAIL
#      30  clk=1  reset=0  a=1 b=0 c=0
#      40  FAIL
#      40  clk=0  reset=0  a=1 b=1 c=0
#      50  PASS
#      50  clk=1  reset=0  a=1 b=1 c=0
#      60  PASS
#      60  clk=0  reset=0  a=1 b=1 c=0
*/
```

Here are the nuances:

A call to an inferred expression function may only be used as the entire default value expression for a formal argument to a property or sequence declaration.

A call to an inferred expression function cannot appear within the body expression of a property or sequence declaration.

If a call to an inferred expression function is used as the entire default value expression for a formal argument to a property or sequence declaration, then it is replaced by the inferred expression as determined at the point where the property or sequence is instantiated. Therefore, if the property or sequence instance is the top-level property expression in an assertion statement, the event expression that is used to replace the default argument $inferred_clock is that as determined at the location of the assertion statement. If the property or sequence instance is not the top-level property expression in the assertion statement, then the event expression determined by clock flow rules up to the instance location in the property expression is used as the default value of the argument.

16.13 'let' Declarations

We have all used the compiler directive `define (as a global text substitution macro). Note the word *global*—it is truly global spanning across *all* scopes of your design modules and files. For example, `define intr 3'b111 will substitute `intr with 3'b111 where-ever it sees `intr either within the local scope or global scope. This can be good and bad. Good is that you have to define it only once and it will span across module/file boundaries. Bad is you cannot redefine `intr (well actually you can, but with consequences). For example, if you change the definition of `intr in a package, you will get a warning and also the new definition will overwrite all the previous ones. Also, `define cannot be parameterized.

That is where 'let' comes into picture. 'let' not only allows local scope but also allows parameterization (or as LRM puts it, it has 'ports') (as in a sequence or a property). Parameterization is a big advantage as you can imagine towards developing modular and reusable code.

To reiterate, let declarations can be used for customization and can replace the text macros in many cases. The 'let' construct is safer because it has a local scope, while the scope of compiler directives is global within the compilation unit. A 'let' declaration defines a template expression (a let body), customized by its ports (aka parameters). A 'let' construct may be instantiated in other expressions.

The syntax for 'let' is

let_declaration ::= let let_identifier [([let_port_list]) = expression;

Let us see each feature of 'let' one by one.

16.13.1 let: Local Scope

First let us see an example using ‘let’

```
module example;

logic r1,r2, r3,r4,clk,clk1;

let exDefLet = r1 || r2;

always @ (posedge clk) begin: ablock

  let exDefLet = r1 & r2; //exDefLet has a local scope of ‘ablock’

  r3=exDefLet;

end

always @ (posedge clk1) begin: bblock

  r4=exDefLet; // exDefLet will take the definition from the scope that is visible to it. Here

                //it is the outer most scope definition of (r1 || r2);

end

endmodule
```

We have defined ‘exDefLet’ in two different scopes. One in the always block ‘ablock’ and another at the outermost scope ‘module example’. Note that their definition (expression) is different in each block. Since ‘let’ can have local scope, each of the definition of ‘let’ will be preserved in its local block. The above code will look like the following after ‘let’ substitutions take place

```
module example;
logic r1,r2, r3,r4,clk,clk1;
always @ (posedge clk) begin :ablock

  r3=r1 & r2;

end
 lways @ (posedge clk1) begin: bblock

  r4=r1 || r2;

end
endmodule
```

If the same design was modeled using `define, here’s how the code would look like

```
module example;
logic r1,r2, r3,r4,clk,clk1;
`define exDefLet r1 || r2;
```

```
always @ (posedge clk) begin :ablock

`define exDefLet r1 & r2;
r3=`exDefLet;

end
always @ (posedge clk1) begin: bblock

   r4=`exDefLet;

end
endmodule
```

In this example, since there are two `define for the same variable, the compiler will complain right off the bat and use the second definition r1&r2 (it's the latest in lexical order) as the global definition of exDefLet. The above code will look like the following after `define substitutions

```
module example;
logic r1,r2, r3,r4,clk,clk1;
always @ (posedge clk) begin: ablock

   r3=r1 & r2;

end
always @ (posedge clk1) begin: bblock

   r4=r1&r2;

end
endmodule
```

As you see 'let' is very useful from scoping point of view. It follows the normal scoping rules. You can parameterize it (or as LRM puts it, it can have 'ports') and reuse the 'let' expression repeatedly with different parameters. Let us now see some more usage/advantages of 'let'.

16.13.2 let: With Parameters

As mentioned before, 'let' can be parameterized which is its significant advantage over `define which is purely a text substitution macro (global compiler directive). Note that instantiation of 'let' is quite different from a 'parameterized function'. With 'let' you replace the instance with *entire* 'let' body. With 'function' you simply pass the parameters and the function executes using those parameters. Function does not replace the instance of function call.

Ok, let us see a simple example.

```
module abc;

logic clk, x, y, j;

logic [7:0] r1;

let lxor (p, q=1'b0) = p^q;

always @ (posedge clk) begin

  for (i = 0; i <= 256; i++) begin

    r1 = lxor(i); //After expanding the 'let' instance, this will be r1 = i ^ 1'b0;

  end

end

endmodule
```

For each value of 'i' r1 will get the 'xor' of 'i' and 'q = 1'b0'. Note that the formal parameter 'q' is assigned a default value of '1'b0'. That being the case, when "r1 = lxor(i)" is executed, the actual 'i' replaces the formal 'p' in lxor and 'q' takes on its assigned default value of '1'b0'. You could have also specified "r1 = lxor(i, j)" and the formal 'q' will now take the value of 'j' (whatever 'j' is).

Note that some rules apply to the formal arguments.

1. Note again that the 'let' body gets expanded with the actual arguments (which replace the formal arguments) and the body (RHS of 'let') will replace the instance of 'let'. That being the case, once the body of 'let' replaces the instance of 'let', all required semantic checks will take place to see that the expanded 'let' body with the actual arguments is legal.
2. The formal arguments can have a default value (as we saw in the example above).
3. The formal arguments can be typed or un-typed. The typed arguments will force type compatibility between formal and actual (cast compatibility). In other words, the actual argument will be cast to the type of the formal argument before being substituted. Un-typed formal in that case is more flexible.
4. If the formal argument is of 'event' type, then the actual argument must be an event_expression. Each reference to the formal argument shall be in a place where an event_expression may be written.
5. The self-determined result type of the actual argument must be cast compatible with the type of the formal argument. The actual argument must be cast to the type of the formal argument before being substituted for a reference to the formal argument.

   If the variables used in 'let' are not formal arguments to the 'let' declaration, they will be resolved according to the scoping rules of the scope in which 'let' is declared.

16.13.3 let: In Immediate and Concurrent Assertions

Yes, 'let' can be used in an immediate ('assert') as well as concurrent ('assert property') assertions in a procedural block.

Let us start with a very simple example of 'let' usage in a sequence. *Note that 'let' expression can only be structural or with sampled value function (as in $past).* We will see 'let' with sampled value function in the next section.

```
module abc;
logic req, gnt;
let reqack = !req && gnt;
sequence reqGnt;

   reqack;

endsequence
endmodule
```

After expanding the 'let' instance

```
module abc;
logic req, gnt;
sequence reqGnt;

   !req && gnt; //

endsequence
endmodule
```

Here's another example.

```
module abc;
logic clk, r1,r2,req,gnt;
let xxory (x,y) = x ^ y; //bit wise xor
let rorg = req || gnt;
....
P1: assert property (@ (posedge clk) (rorg)); //concurrent assertion
always_comb begin
a1: assert (xxory (r1,r2)); //immediate assertion
a2: assert (rorg);
end
endmodule
```

After expansion (showing complete hierarchical scope)

```
module abc;
logic clk, r1,r2,req,gnt;
P1: assert property (@ (posedge clk) (abc.req || abc.gnt)); //concurrent assertion
always_comb begin
```

```
a1: assert (abc.r1 ^ abc.r2)); //immediate assertion
a2: assert (abc.req || abc.gnt);
end
endmodule
```

Now, here's an example that uses the sampled value functions $(rose) and $(fell).

```
module abc;
logic clk,r1,r2,req,gnt,ack,start;
let arose(x) = $rose(x);
let afell(y) = $fell (y);
always_comb begin
if (ack) s1: assert(arose(gnt));
if (start) s2: assert(afell(req));
end
```

Another intended use of let is to provide shortcuts for identifiers or subexpressions. For example, (LRM):

```
task write_value;
        input logic [31:0] addr;
        input logic [31:0] value;
...
endtask
...
let addr = top.block1.unit1.base + top.block1.unit2.displ;
...
write_value(addr, 0);
```

But note that hierarchical references to 'let' expressions are *not* allowed. For example, following is illegal.

```
assign e = Top.CPU.my_let(a)); //Illegal
```

Also, Recursive 'let' instantiations are not permitted.

Here's an example of how the 'let' arguments bind in the declarative context.

```
module sys;
logic req = 1'b1;
logic a, b;
let y = req;
...
always_comb begin
        req = 1'b0;
        b = a | y;
end
endmodule: sys
```

The effective code after expanding let expressions:

```
module sys;
logic req = 1'b1;
logic a,b;
...
always_comb begin
        req = 1'b0;
        b = a | (sys.req);   //NOTE: y binds to preceding definition of 'req' in the declarative context of 'let'
end
endmodule : top
```

Following is an example of 'let' with typed formal arguments. The example shows how type conversion works when the type of a formal is different from the type of an actual (partially from LRM).

First, module 'm' with 'let' declarations. Note the typed formals in 'let' declarations and the mismatch between 'let' instance 'actuals' and 'formals'.

```
module m(input clock);
logic [15:0] a, b;
logic c, d;
typedef bit [15:0] bits;
...
let ones_match(bits x, y) = x == y;
let same(logic x, y) = x === y;

always_comb
        a1: assert(ones_match(a, b));
                //Note: the actuals 'a' and 'b' are of type 'logic', while the formals 'x', 'y' are of type 'bits'

property toggles(bit x, y);
        same(x, y) |=> !same(x, y);
                //Note: the actuals 'x', 'y' are of type 'bit', while the formals 'x', 'y' are of type 'logic'

endproperty

a2: assert property (@(posedge clock) toggles(c, d));
endmodule : m

After expanding the 'let' macro, the code looks as follows:

module m(input clock);
logic [15:0] a, b;
logic c, d;
typedef bit [15:0] bits;
...
// let ones_match(bits x, y) = x == y;
// let same(logic x, y) = x === y;

always_comb
        a1:assert((bits'(a) == bits'(b)));

property toggles(bit x, y);
        (logic'(x) === logic'(y)) |=> ! (logic'(x) === logic'(y));
endproperty

a2: assert property (@(posedge clock) toggles(c, d));
endmodule : m
```

Finally, here's where a 'let' can be declared:

- A module
- An interface
- A program
- A checker
- A clocking block
- A package
- A compilation-unit scope
- A generate block
- A sequential or parallel block
- A subroutine

16.14 'restrict' for Formal Verification

The 'restrict' property is strictly for Formal (static functional) verification. *Simulators do not check this property*. Since we are not covering Formal Verification in this book, this property is noted here for the sake of completeness. Note that we have immediate 'assert', 'cover' and 'assume' but *there is no immediate 'restrict' assertion statement*. As we saw with the 'assume' property, formal verification requires some assumption or restriction in order for it to restrict the logic cones to process and not explode in the state space. 'restrict' has the same semantics as 'assume', only that '*restrict' does not have an action block*. Here is the syntax.

restrict property (property_spec); //Note, no action block.

For example, for Formal Verification you need to restrict the checking of an assertion which has 5 inputs (a, b, c, d, e) and 2 control bits (x, y). If {x, y} = 2'b00, the inputs a, b are of no use for the static formal check. So, we restrict the property as follows:

```
restrict property (@ (posedge clk) {x, y} == 2'b00);
```

Note again that the property is ignored by simulation. In other words, {x,y} == 2'b00 is not enforced during simulation.

Please refer to the Chap. (13) on 'assume' to further understand usage of 'restrict'.

16.15 Abort Properties: reject_on, accept_on, sync_reject_on, sync_accept_on

Recall 'disable_iff' disable condition (Sect. 4.6), which preempts the entire assertion, if true. 'disable_iff' is an asynchronous abort (or reset) condition for the entire assertion. It is also asynchronous in that it's expression is *not* sampled in the pre-poned region but the expression is evaluated at every time stamp

(i.e. in-between clock ticks as well as at the clock ticks) and whenever the 'disable_iff' expression turns true that the entire assertion will be abandoned (no pass or fail).

With that background, 1800-2009/2012 adds four more abort conditions. 'reject_on' and 'accept_on' are asynchronous abort conditions (as in disable_iff) and 'sync_reject_on' and 'sync_accept_on' are synchronous (i.e. sampled) abort condition. Note that 'accept_on' is an abort condition for PASS, even though that may seem a bit counterintuitive at first. In other words, if 'accept_on' aborts an evaluation, the result is a PASS. If 'reject_on' aborts an evaluation, the result is FAIL.

The syntax for all four is the same.

accept_on (abort condition expression) property_expression

sync_accept_on (abort condition expression) property_expression

reject_on (abort condition expression) property_expression

sync_reject_on (abort condition expression) property_expression

Before we see examples, here are high-level points to note

1. One note off the bat to distinguish 'disable_iff' from the abort properties is that 'disable_iff ' works at the 'entire concurrent assertion' level while these abort properties work at the 'property' level. Only the property_expression associated with the abort property will get 'aborted'—not the entire assertion as with 'disable_iff'. More on this later.
2. The operators 'accept_on' and 'reject_on' work at the granularity of simulation time step (i.e. asynchronously).
3. In contrast, the operators 'sync_accept_on' and 'sync_reject_on' do *not* work at the granularity of simulation time-step. They are sampled at the simulation time step of the clocking event.
4. You can nest the four abort operators 'accept_on', 'reject_on', 'sync_accept_on', 'sync_reject_on'. Note that nested operators are in the lexical order 'accept_on', 'reject_on', 'sync_accept_on' and 'sync_reject_on' (from left to right). While evaluating the inner abort property, the outer abort property takes precedence over the inner abort condition in case both conditions occur at the same time tick.
5. Abort condition cannot contain any reference to local variables or the sequence methods .triggered and .matched.

Now let us look at some examples to nail down the concepts.

```
property p1;
  @ (posedge clk) $fell(bMode) |-> reject_on(bMode) data_transfer[*4];
endproperty
assert property (p1);
```

The above example specifies that on the falling edge of burst Mode (bMode), data_transfer should remain asserted for 4 consecutive clocks and that the bMode should –not- go high during those 4 data transfers. The way the property reads is; look for the falling edge of bMode and starting that clock *reject* (fail) the property

"(data_transfer[*4])" if at any time (i.e. asynchronously—even between clock ticks) it sees bMode going high. As noted before, 'reject_on' abort means failure. Hence consequent will FAIL and so will the property p1.

The important thing to note here is that the evaluation of the abort property namely "data_transfer[*4]" and the reject condition reject_on(bMode) start at the *same* time. In other words, (as shown below), this is like a 'throughout' operator where the LHS is checked to see if it holds for the entire duration of RHS. Similarly, here we check to see that while we are monitoring "data_transfer[*4]" to hold that 'bMode' should not go high. If it does go high at any time during "data_transfer[*4]" the property will be rejected i.e. it will fail.

The same property can be written using sync_reject_on, only that the "bMode" will not be evaluated asynchronously (any time including in-between clock ticks) but will be sampled only at the sampling edge, clock tick.

Note that the above property can be written using 'throughout' as well. Please refer to Sect. 6.9 on 'throughout' operator to see a similar example.

```
property p1;

  @ (posedge clk) $fell(bMode) |-> !(bMode) throughout data_transfer[*4];

endproperty
assert property (p1);
```

Let us look at an example of 'accept_on'

```
property reqack;

  @ (posedge clk) accept_on(cycle_end) req |-> ##5 ack;

endproperty
assert property (reqack);
```

This property uses "accept_on(cycle_end)" as the abort condition on the property "req |-> ##5 ack". When 'req' is sampled high on a posedge clk, the property "req |-> ##5 ack" starts evaluating waiting for ack to arrive after 5 clocks. At the same time, 'cycle_end' is also monitored to see if it goes high. Here are the scenarios that take place.

'cycle_end' arrives within the 5 clocks that the property is waiting for 'ack'. The accept_on condition will be true in that case and the property will be considered to pass. The next evaluation will again start the next time 'req' is sampled high on posedge clk.

'cycle_end' does not arrive within 5 clocks when the property is waiting for 'ack'. The property will evaluate as with any concurrent assertion and if 'ack' does not come in high at 5th clock, the property will fail. If 'ack' does come in high at 5th clock, the property will pass.

'cycle_end' arrives exactly the same time as 'ack' within the 5 clocks. The abort condition takes precedence. Since in this case, both 'ack' arrived and the accept_on was triggered at the same time, the accept_on aborts the evaluation with a pass and

so the assertion will pass. What if we used ‘reject_on’ instead of ‘accept_on’ in such a scenario?

In short, the property evaluation aborts on ‘accept_on’ (and passes) or ‘reject_on’ (and fails) OR it will finish on its own (and pass/fail) if the abort condition does not arrive.

Here are some more examples courtesy 1800-2009/2012 LRM.

property p; (**accept_on**(a) p1) and (**reject_on**(b) p2); **endproperty**

Note that we are using an ‘and’ operator here between two properties p1 and p2. Note that for an ‘and’ to succeed both the LHS and RHS of ‘and’ must complete and pass. If ‘a’ becomes true, then p1 will abort and pass. But since there is an ‘and’ we will wait for the second property to complete as well. If ‘b’ becomes true during the evaluation of p2, p2 will be aborted and considered to fail and since this is an ‘and’ the property ‘p’ will fail. What if we used an ‘or’ instead?

property p; (**accept_on**(a) p1) or (**reject_on**(b) p2); **endproperty**

Recall that ‘or’ requires either the LHS or the RHS to complete and pass. In the same scenario as above, if ‘a’ becomes true first during the evaluation of p1, p1 is aborted and will be pass (i.e. accepted) and the property ‘p’ will pass.

Note that nested operators are in the lexical order ‘accept_on’, ‘reject_on’, ‘sync_accept_on’ and ‘sync_reject_on’ (from left to right). If two nested operator conditions become true in the same time tick during the evaluation of the property, then the outermost operator takes precedence.

property p; **accept_on**(a) **reject_on**(b) p1; **endproperty**

Note there is no operator between accept_on and reject_on.

If ‘a’ goes high first, the property is aborted on accept_on and will pass. If ‘b’ goes high first, the reject_on succeeds and the property p will fail. If both ‘a’ and ‘b’ go high at the same time during the evaluation of p1, then accept_on takes precedence and ‘p’ will pass.

Another simple example.

```
Frame_accept_reject: assert property (

    @(posedge clk)

        accept_on (Frame_)

        Cycle_start |=> reject_on(Tabort)

    )

    else $display (“Frame_ FAIL);
```

This is another example of nested asynchronous aborts. The outer abort (accept_on (Frame_)) has the scope of the entire property of the concurrent

assertion. The inner abort (reject_on(Tabort)) has the scope of the consequent of |=>. The inner abort does not start evaluation until Cycle_start is true. Note that the outer abort takes precedence over the inner abort.

Exercise: Try the same property with s_accept_on and s_reject_on and note the differences. Try both examples with different triggers of Frame_ and Tabort and see when/how the property passes and fails. I'll leave this for you as an exercise.

Finally, note the following points

- A disable condition (disable iff) may use the method .triggered (attached to the sequence used in disable condition). But an abort condition (the ones described above) cannot use .triggered method
- Either disable or abort properties cannot refer to local variables
- Either of the reset conditions may not use the method .matched attached to the sequence used in reset conditions.

16.16 $assertpassoff, $assertpasson, $assertfailoff, $assertfailon, $assertnonvacuouson, $assertvacuousoff

These system tasks add further control over assertion execution during simulation. We have seen $asserton, $assertoff and $assertkill (refer to Sect. 7.4) before. Here's a brief explanation of what these new system tasks do.

$**assertpassoff** : This system task turns off the action block associated with PASS of an assertion. This includes PASS indication because of both the vacuous and non-vacuous success (Sect. 14.15). To re-enable the PASS action block, use $**assertpasson**. It will turn on the PASS action of both the vacuous and non-vacuous successes (Sect. 14.15). If you want to turn on only the non-vacuous PASS, then use $**assertnonvacuouson** system task. Note that these system tasks do not affect an assertion that is already executing.
$**assertfailoff**: This system task turns off the action block associated with the FAIL of an assertion. In order to turn it on, use $**assertfailon**. Here also, these system tasks do not affect an assertion that is already executing.
$**assertvacuousoff** system task turns off the PASS indication based on a vacuous success (Sect. 14.15). An assertion that is already executing is not affected. By default, we get a PASS indication on vacuous pass.

All the system tasks take arguments, as we have seen before with $asserton, $assertoff and $assertkill (refer to Sect. 7.4). The first argument indicates how many level of hierarchy below each specified module instance to apply the system tasks. The subsequent arguments specify which scopes of the model to act upon (entire modules or instances).

16.17 $assertcontrol

LRM IEEE-1800 (2012) introduces a new system task $assertcontrol. This system task can be used in lieu of above mentioned individual system tasks.

The $assertcontrol system task controls the evaluation of assertions. The $assertcontrol system task can also be used to control the execution of assertion action blocks associated with assertions and expect statements.

This system task provides the capability to enable/disable/kill the assertions based on assertion type or directive type. Similarly, this task also provides the capability to enable/disable action block execution of assertions and expect statements based on assertion type or directive type.

The violation reporting for *unique*, *unique0* and *priority if* and *case* constructs can also be controlled using these tasks. I will refrain from explain unique, unique0 and priority if and case since these are SystemVerilog constructs and not assertions constructs. Please refer to the SystemVerilog LRM for their description.

Here's the syntax:

$assertioncontrol (control_type [, [assertion_type] [, [directive_type] [, [levels] [, list_of_scopes_or_assertions]]]]);

where:

– **control_type**: This argument controls the effect of the $assertcontrol system task. This argument is an integer expression. The valid values for this argument are

Control_type value	Effect
1	Lock
2	Unlock
3	On
4	Off
5	Kill
6	PassOn
7	PassOff
8	FailOn
9	FailOff
10	NonvacuousOn
11	VacuousOff

Now let us see what does each of the 'effect' mean (LRM):

– Lock: A value of 1 for this argument *prevents status change* of all specified assertions, expect statements, and violation reports until they are unlocked. Once an $assertcontrol with control_type of value 1 (Lock) is applied to an assertion, expect statement, or violation report, it becomes locked and no

$assertcontrol will affect it until the locked state is removed by a subsequent $assertcontrol with a control_type value of 2 (Unlock).
- Unlock: A value of 2 for this argument will remove the locked status of all specified assertions, expect statements, and violation reports.
- On: A value of 3 for this argument will re-enable the execution of all specified assertions. A value of 3 for this argument will also re-enable violation reporting from all the specified violation report types. This control_type value does not affect expect statements.
- Off: A value of 4 for this argument will stop the checking of all specified assertions until a subsequent $assertcontrol with a control_type of 3 (On). No new attempts will be started.

Attempts that are already executing for the assertions, and their pass or fail statements, are *not* affected. In the case of a deferred assertion (Sect. 16.2), currently queued reports are not flushed and may still mature, though further checking is prevented until a subsequent $assertcontrol with a control_type of 3 (On). In the case of a pending procedural assertion instance, currently queued instances are not flushed and may still mature, though no new instances may be queued until a subsequent $assertcontrol with a control_type of 3 (On). A value of 4 for this argument will also disable the violation reporting from all the specified violation report types.

Currently queued violation reports are not flushed and may still mature, though no new violation reports will be added to the pending violation report queue until a subsequent $assertcontrol with a control_type value of 3 (On). The violation reporting can be re-enabled subsequently by $assertcontrol with a control_type value of 3 (On). This control_type value does not affect expect statements.

- Kill: A value of 5 for this argument will abort execution of any currently executing attempts for the specified assertions and then stop the checking of all specified assertions until a subsequent $assertcontrol with a control_type of 3 (On). This also flushes any queued pending reports of deferred assertions or pending procedural assertion instances that have not yet matured. A value of 5 for this argument will also abort violation reporting from all the specified violation report types. Currently queued violation reports that have not yet matured are also flushed, and no new violation reports shall be added to the pending violation report queue until a subsequent $assertcontrol with a control_type value of 3 (On). This control_type value does not affect expect statements.
- PassOn: A value of 6 for this argument will enable execution of the pass action for vacuous and nonvacuous success of all the specified assertions and expect statements. An assertion that is already executing, including execution of the pass or fail action, is not affected. This control_type value does not affect violation report types.
- PassOff: A value of 7 for this argument will stop execution of the *pass action* for vacuous and nonvacuous success of all the specified assertions and expect statements. Execution of the pass action for both vacuous and nonvacuous successes can be re-enabled subsequently by $assertcontrol with a control_type

value of 6 (PassOn), while the execution of the pass action for only nonvacuous successes can be enabled subsequently by $assertcontrol with a control_type value of 10 (NonvacuousOn). An assertion that is already executing, including execution of the pass or fail action, is not affected. This control_type value does not affect violation report types.

- FailOn: A value of 8 for this argument will enable execution of the fail action of all the specified assertions and expect statements. An assertion that is already executing, including execution of the pass or fail action, is not affected. This task also affects the execution of the default fail action block. This control_type value does not affect violation report types.
- FailOff: A value of 9 for this argument will stop execution of the fail action of all the specified assertions and expect statements until a subsequent $assertcontrol with a control_type value of 8 (FailOn). An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the fail action is executed. This task also affects the execution of default fail action block, i.e., $error, which is called in case no else clause is specified for the assertion. This control_type value does not affect violation report types.
- NonvacuousOn: A value of 10 for this argument will enable execution of the pass action of all the specified assertions and expect statements on nonvacuous success. An assertion that is already executing, including execution of the pass or fail action, is not affected. This control_type value does not affect violation report types.
- VacuousOff: A value of 11 for this argument will stop execution of the pass action of all the specified assertions and expect statements on vacuous success until a subsequent $assertcontrol with a control_type value of 6 (PassOn). An assertion that is already executing, including execution of the pass or fail action, is not affected. By default, the pass action is executed on vacuous success. This control_type value does not affect violation report types.

The assertion action control tasks or $assertcontrol with control_type values of 6 (PassOn) to 11 (VacuousOff) do not affect statistics counters for the assertions. The details related to the behavior of $assertcontrol for assertions referring to global clocking future sampled value functions are explained in Sect. 16.5.

- **assertion_type**: This argument selects the assertion types and violation report types that are affected by the $assertcontrol system task. This argument shall be an integer expression. The valid values for this argument are

Assertion_type value	Types of assertions affected
1	Concurrent
2	Simple Immediate
4	Observed Differed Immediate
8	Final Differed Immediate
16	Expect
32	Unique

(continued)

(continued)

Assertion_type value	Types of assertions affected
64	Unique0
128	Priority

Note that multiple assertion_type values can be specified at a time by OR-ing different values. For example, a task with assertion_type value of 3 (which is the same as Concurrent|SimpleImmediate) will apply to concurrent and simple immediate assertions. Similarly, a task with assertion_type value of 96 (which is same as Unique|Unique0) will apply to unique and unique0 if and case constructs.

If assertion_type is not specified, then it defaults to 255 and the system task applies to all types of assertions, expect statements, and violation reports.

– **directive_type**: This argument selects the directive types that are affected by the $assertcontrol system task. This argument will be an integer expression.

The valid values for this argument are

Directive_type values	Type of directives affected
1	Assert directives
2	Cover directives
3	Assume directives

Multiple directive_type values can be specified at a time by OR-ing different values. For example, a task with directive_type value of 3 (which is same as Assert|Cover) shall apply to assert and cover directives.

If directive_type is not specified, then it defaults to 7 (Assert|Cover|Assume) and the system task applies to all types of directives.

– **levels**: This argument specifies the levels of hierarchy, consistent with the corresponding argument to SystemVerilog $dumpvars system task. If this argument is not specified, it defaults to 0. This argument will be an integer expression.
– **list_of_scopes_or_assertions**: This argument specifies which scopes of the model to control. These arguments can specify any scopes or individual assertions.

Now, let's see how $assertcontrol maps to the individual controls that were described in the previous section. Note that these individual system tasks are still supported in 2012 LRM for backward compatibility.

– $**asserton**[(levels[, list])] is equivalent to $assertcontrol(3, 15, 7, levels [,list])
– $**assertoff**[(levels[, list])] is equivalent to $assertcontrol(4, 15, 7, levels [,list])
– $**assertkill**[(levels[, list])] is equivalent to $assertcontrol(5, 15, 7, levels [,list])

And as I just mentioned, assertion action control tasks $assertpasson, $assertpassoff, $assertfailon, $assertfailoff, $assertvacuousoff, and $assertnonvacuouson are also provided for backward compatibility. Following example from LRM.

These tasks can be defined as follows:

- $**assertpasson**[(levels[, list])] is equivalent to $assertcontrol(6, 31, 7, levels [,list])
- $**assertpassoff**[(levels[, list])] is equivalent to $assertcontrol(7, 31, 7, levels [,list])
- $**assertfailon**[(levels[, list])] is equivalent to $assertcontrol(8, 31, 7, levels [,list])
- $**assertfailoff**[(levels[, list])] is equivalent to $assertcontrol(9, 31, 7, levels [,list])
- $**assertnonvacuouson**[(levels[, list])] is equivalent to $assertcontrol(10, 31, 7, levels [,list])
- $**assertvacuousoff**[(levels[, list])] is equivalent to $assertcontrol(11, 31, 7, levels [,list])

Examples:

```
$assertcontrol (VACUOUSOFF, CONCURRENT | EXPECT);
```

This systasks affect the whole design so no modules are specified. Disable vacuous pass action for all the concurrent asserts, covers and assumes in the design. Also disable vacuous pass action for expect statements.

```
$assertcontrol (OFF);
```

Disable concurrent and immediate asserts and covers. This system task does not affect *expect* statements as control type is Off using default values of all the arguments after first argument. This will also disable violation reporting.

```
$assertcontrol (ON, CONCURRENT|S_IMMEDIATE|D_IMMEDIATE, ASSERT|COVER|
ASSUME, 0);
```

This system task enables assertions. This will not enable violation reporting.

```
$assertcontrol (KILL, CONCURRENT, ASSERT, 0);
```

Kill currently executing concurrent assertions but do not kill concurrent covers/assumes and immediate/deferred asserts/covers/assumes using appropriate values of second and third arguments.

```
$assertcontrol (LOCK, ALL_ASSERTS, ALL_DIRECTIVES, 0, a1);
$assertcontrol (ON); //enable all the assertions except a1
$assertcontrol (UNLOCK, ALL_ASSERTS, ALL_DIRECTIVES, 0, a1);
```

Enable all the assertions except a1. To accomplish this, first we'll lock a1 and then we'll enable all the assertions and then unlock a1 as we want future assertion control tasks to affect a1.

16.18 Checkers

Checkers provide a way to group several assertions together into a bigger block which acts with its well defined functionality and interfaces providing modularity and reusability. In addition to bundling assertions, you may also put modeling code in these blocks that the assertions or covergroups need. A checker allows you to place all such logic in a well-defined block. One of the intended use of checkers is to serve as verification library units.

But wait. Don't we have 'modules' and 'interfaces' that do the same thing? Sure, you can have a 'module' or 'interface' which can keep assertions separate from RTL code and bind them 'externally'. But there are significant advantages to keeping assertions grouped into a checker.

1. A checker can be instantiated from a procedural block as well as from outside procedural code as with concurrent assertions. On the other hand, and as we are familiar with, a module cannot be instantiated in a procedural block. It can only be instantiated outside of a procedural block.
2. The formal arguments of a checker can be sequences, properties or other edge sensitive events. Module I/O ports do not allow this.
3. Synthesis tools normally ignore the entire checker block while in a module you have to use conditional compile if you have synthesizable code mixed with assertions.

Here's the syntax for a checker. A checker is declared using the keyword '*checker*' followed by a name and optional formal argument list, and ending with the keyword '*endchecker*'.

checker checker_identifier [([checker_port_list])];

{checker_or_generate_item}

endchecker [: checker_identifier]

Let us start with a simple example where we show the advantages of grouping assertions in a 'checker' versus a 'module'.

1. module: First we'll define a '*module'* which holds properties and sequences for a simple bus protocol.
2. module testbench: Then we'll define a testbench module that instantiates this '*module*'.
3. Checker: Then we'll see how to put all these properties in a '*checker*' (instead of a '*module*').
4. Checker Testbench: And finally we'll see how the testbench 'instantiates' this '*checker*' from procedural code.

ONE: Assertions in a '*module*'

```
module checkerModule  #(burstSize =4) ( dack_ , oe_, bMode, bMode_in, clk, rst);
input dack_, oe_, bMode , bMode_in, clk, rst;
        sequence data_transfer;
            ##2 ((dack_==0) && (oe_==0)) [*burstSize];
        endsequence
        sequence checkbMode;
          (!bMode) throughout  data_transfer;
        endsequence
        property pbrule1;
         @ (posedge clk) disable iff (rst) bMode_in |-> checkbMode;
        endproperty
        checkBurst: assert property(pbrule1) else $display($stime,,, "Burst Rule Violated");
endmodule
```

module checkerModule is a simple bus protocol checker that is fashioned on the PCI bus. When bMode(burst mode) is asserted (active low) for 2 clocks consecutively that we need to make sure that it remains low *throughout* data_transfer which is 4 clocks long. We have seen a very similar model in (Sect. 6.9) while studying throughout. Please refer to the AC specs (timing diagrams) for this *module* in that section. Note that we have parameterized the 'burstSize' which is a practical way to model a property that can be reused for different burst lengths.

Now let us see how do we instantiate this checkerModule *module* from our testbench.

TWO:

```
module test_checkerModule;
        logic dataAck_, outputEn_;
        logic bMode, bMode_send, rst, clk;
        ..............
        always @ (posedge clk or negedge rst) begin
         if (!rst) begin
           dataAck_=1'b0; outputEn_=0; bMode=0;
        end
         /*Following block generates a 'bMode' that is Low for 2 clocks consecutively. If so, we send
        'bMode_send' to the checkerModule module. */
        always @ (posedge clk && rst) begin
           if (!bMode) begin
                 @ (posedge clk);
                If (!bMode) bMode_send=bMode;
          end
        end
        //Now let us instantiate module 'checkerModule'
        checkerModule  (#8)
        ck1(.dack_(dataAck_),.oe_(outputEn_),.bMode(bMode),.bMode_in(bMode_send),
        .clk(clk), .rst(rst));
endmodule
```

A few things to note here.

1. We had to explicitly create a 'sequence' using an 'always' block (to check that 'bMode' is Low for 2 consecutive clocks) in behavioral code since we cannot pass sequences to a module
2. We have to explicitly pass clk and rst to the module checkerModule since a module instance won't infer clk or rst from its context. In other words, clk and rst cannot be inferred from the module test_checkerModule
3. You cannot pass an edge control to a module (we all know how modules/instances work… so some of these points are probably obvious). Since an edge cannot be passed to a port, you have to make sure that you send the right polarity on these ports (clk for posedge clk) and (!clk for negedge clk)

Now let us model the same checkerModule '*module'* as a '*checker*'

THREE:

```
checker checkerM  #(burstSize =4) (dack_ , oe_, bMode, bMode_in, rst, event clk=$inferred_clock);
        input dack_, oe_, bMode ,bMode_in, rst;
        sequence data_transfer;
            ##2 ((dack_==0) && (oe_==0)) [*burstSize];
        endsequence
        sequence checkbMode;
          (!bMode)  throughout  data_transfer;
        endsequence
        property pbrule1;
          @ (clk) disable iff (rst) bMode_in |-> checkbMode;
        endproperty
        checkBurst: assert property (pbrule1) else $display($stime,,, "Burst Rule Violated");
endchecker
```

Note that 'clk' is now inferred from the context from which checkerM is instantiated (see the next module test_checkerM). Also, a sequence 'bMode_Sequence' will be explicitly assigned to port 'bMode_in' from the test_checkerM module. Neither of these two features are possible if we model our assertions in a Verilog *module*.

Here's the test_checkerM module that calls the 'checker checkerM' module FOUR

```
module test_checkerM;
logic dataAck_, outputEn_, bMode;
logic cycle_start, rst, clk;
.............
/*Following block generates a bMode that is Low for 2 consecutive clocks. */
sequence bMode_Sequence;
 !bMode[*2]
endsequence
.....................
//Now let us call the checker 'checkerM' from a procedural block
always @ (posedge clk or negedge rst) begin
 if (!rst) begin
   dataAck_=1'b0; outputEn_=0;bMode=0;
end
else
  checkerM  (#8) ck1 (.dack_(dataAck_),  .oe_(outputEn_),  .bMode(bMode),
.bMode_in(bMode_Sequence) .rst(rst));
endmodule
```

Note

1. We did not explicitly pass 'clk' to the *checker* checkerM. The clk was inferred from the context of the procedural block from which it was called (just as in concurrent assertion that is called from a procedural block). We could have done the same for 'rst'.
2. We passed a sequence bMode_Sequence to checkerM on bMode_in port.

As you noticed, it is much more practical, modular and easier to code and bundle assertions in a *checker* than in a *module*.

Now let us study further language features and nuances of a 'checker'.

Once again, the clock and reset (disable iff) contexts are inherited from the scope of the checker instantiation. Here is another simple example.

```
module test;
        default clocking @ clk; endclocking
        default disable iff reset;
        checker test_bMode;
                //directly inherits @ clk and 'reset' from the higher level context of module test
        endchecker
        checker test_cMode; //Note this is a new checker
        //Redefines the default blocks. Point is that you can infer/inherit or redefine what is
        //inherited
                default clocking @ clk1; endclocking //Note that the default clocking block is for @ clk1
                default disable iff reset_system; //The default disable iff condition is 'reset_system'
        endchecker
endmodule
```

The example shows a testbench called 'module test' which defines a clk and a reset at 'module test' level. The first checker 'test_bMode' inherits the clk and reset from its higher-level scope (which is 'module test'). The second checker 'test_cMode' defines its own clk1 and reset_system. This enables it to have its own local definition of clk1 and reset_system. It will not inherit the default clk and reset from the top level module 'module test'.

A checker body may contain the following elements:

- Declarations of 'let' constructs, sequences, properties, and functions
- Deferred assertions (see Sect. 16.2)
- Concurrent assertions (see Chap. 4)
- Checker declarations
- Other checker instantiations
- Covergroup declarations and instances
- 'always' ('always_comb','always_latch','always_ff'), 'initial' and 'final' procedures
- 'generate' blocks
- 'default clocking' and 'default disable iff' statements
- Checker variable declarations and assignments

A checker body *cannot* contain the following elements:

- 'if', 'case', 'for', 'continuous assignment' etc. type of procedural conditional and loop statements are *not* allowed.
- 'immediate' assertions are not allowed, as such. Immediate assertions are allowed only in the 'action' blocks of assertions and its final procedural blocks.
- 'initial' block can only contain concurrent or deferred assertions and a procedural event control statement '@'. All other statements are forbidden in the 'initial' block.
- An 'always' block also has similar restrictions. It can only contain concurrent or deferred assertions, checker variable assignments and procedural timing control statement '@'. All other statements are forbidden in the 'always' block.
- modules, interfaces, programs and packages cannot be declared inside a checker.

Further illegal constructs are described in coming sections.

16.18.1 Nested Checkers

As mentioned earlier, a checker can embed another checker thus making checkers nested. Here is an example following the examples above.

```
checker ck1(irdy, trdy, frame_, event clk=$inferred_clock, event reset = $inferred_disable);
default clocking @ clk; endclocking
default disable iff reset;
        property check1;
          irdy |-> ##2 trdy;
        endproperty
        property check2;
         $rose(irdy) |=> frame_;
        endproperty
        checker ck2; //nested checker
                property check1; //Redefinition of check1 within the local scope of checker ck2
                  $rose(trdy) |-> irdy;
                endproperty
                property check3;
                  $fell(irdy) |-> !frame_;
                endproperty
                checkp1: assert property check1;
                checkp3: assert property check3;
                checkp2: assert property check2;
        endchecker : ck2
ck2 ck2i; //instantiate ck2
endchecker : ck1
```

Points to note:

1. Checker ck1 properties are visible to checker ck2. Hence checker ck2 is able to 'assert' check2 of checker ck1
2. Checker ck2 redefines property check1 for its local scope use. Since ck2 is instantiated in ck1, property check1 of checker ck2 is not directly visible to checker ck1
3. The inferred clk and reset of checker ck1 are visible to checker ck2

16.18.2 Checkers: Illegal Conditions

Following is–*not*–allowed in the checker body (this is as far as the author knowledge permits, since the simulators did not support checkers as of this writing to validate the following)

- A checker *cannot* be instantiated in a concurrent procedural construct such as fork..join, fork…join_any or fork…join_none.
- Continuous assignment.
- Procedural conditional and loop statements (if, case, for, etc.) are not allowed in checkers!
- Declaring *nets* in a checker body is illegal.
- Using blocking procedural assignments to checker variables is illegal. Such assignments can only be non-blocking.
- 'initial' procedural block may only contain concurrent, deferred and event control @. Nothing else.

  - For example, following is illegal

```
checker myCheck;
    bit myBit;
    initial begin myBit=1'b1; //'initial' assignment to a variable is ILLEGAL
    end
endchecker
```

We assigned a checker variable in the initial block – that is illegal.

Following will work

```
checker myCheck;
    bit myBit=1'b1; //This is Legal
endchecker
```

Following is illegal as well!

```
checker myCheck (a, b, c);
        bit myBit;
        ....
endchecker
module myMod;
        ...
        mycheck mck1(a, b, c);
        $display(mck1.myBit);//Hierarchical reference to checker variable is
                              //ILLEGAL
endmodule
```

- Following is illegal as well !!

```
checker myCheck(a,b,c);
    logic myBus[7:0];
    always @ (posedge clk) begin
    myBus[1:0] = 2'b0;
    myBus[7:1] = 6'b1;
            // Multiple assignments to the same variable is ILLEGAL. Bit
            // myBus[1] is common and assigned twice.
    end
endchecker
```

BUT the following is legal

```
checker myCheck(a,b,c);
    logic myBus[7:0];
    always @ (posedge clk) begin
    myBus[1:0] = 2'b0;
    myBus[7:2] = 6'b1; //Multiple assignments to bits of myBus is assigned
                       //only once.
    end
endchecker
```

- Following is illegal as well!

```
checker myCheck(bus,i);
        bit [3:0] bus, i;
        initial begin
                @ (posedge clk) i=0;
                bus[i]=4'b1111;
        end
checker
module myMod
        logic [3:0] busIndex;
        logic [3:0] datafromBus;
        .....
        myCheck m1 (datafromBus,busIndex);
                                //busIndex is non-constant ILLEGAL
        myCheck m2(datafromBus,4'b0); //busIndex is constant - LEGAL
endmodule
```

- So, with all these restrictions on checker variable assignments what is one supposed to do? One of the solution is to use functions, as in the example below.

```
checker myCheck(a, b);
        bit a, b;
        initial begin
                @ (posedge clk);
                a=returnAvalue;
        end
function (bit a) returnAvalue;
         return a+1;
endfunction
endchecker
```

In this example, since we cannot assign a value to 'a' directly in the 'initial' block, we called the function 'returnAvalue' to accomplish the same. Note that 'a' is visible in the function 'returnAvalue'. Since function 'returnAvalue' is within the scope of checker mycheck, all the variables available to 'mycheck' are also visible to 'returnAvalue'. As evident, procedural control statements are allowed in a function.

16.18.3 Checkers: Important Points

1. A checker can be declared in a
   a. module
   b. interface
   c. program
   d. checker (nested checkers)
   e. package
   f. generate block
   g. compilation unit scope
2. 'type' and 'data' declarations within the checker are local to the checker scope and are static.
3. Clock and 'disable iff' contexts are inherited from the scope of the checker declaration.
4. You can modify/access DUT variables from a 'checker'! But my suggestion is to not overdo it. Checker code will not be portable and may result in spaghetti code. Try to keep a checker modular and reusable.
5. Checker formal arguments cannot be of type 'local'.
6. Checker formal argument cannot be an 'interface'.

7. The connectivity between the actual arguments and formal arguments of a checker follow exactly the same rules as those for modules, namely,

    a. positional association
    b. explicit named association
    c. implicit named association
    d. wildcard name associations.

    Author leaves it to the reader to know of these techniques since they are age old Verilog.
8. A checker body may contain the following elements (LRM 1800-2009/2012)

    a. Declarations of 'let' constructs, sequences, properties and functions
    b. Deferred assertions
    c. Concurrent and Deferred assertions

        i. A checker can contain only concurrent and deferred assertions. *Immediate assertions are allowed only in the 'action' blocks of assertions and in the final procedural blocks*

    d. Nested checkers are allowed
    e. Covergroup declarations and assignments. One or more 'covergroup' declarations are permitted in a checker. These declarations and instances cannot appear in any procedural block in a checker. A 'covergroup' may reference any variable visible in its scope, including checker formal arguments and checker variables. But it is indeed an Error if a formal argument referenced by a 'covergroup' has a 'const' actual argument. Please refer to Chap. 20 on Functional Coverage to see how 'covergroups' are defined. The same definition can be directly embedded in a checker.
    f. Default clocking and disable iff declarations are allowed
    g. initial, always and final procedural blocks are allowed in a 'checker' body.

        i. An 'initial' procedure in a checker body may contain *let* declarations, *immediate*, *deferred*, and *concurrent* assertions, and a procedural timing control statement using an event control only. Similarly, an 'always' procedural block also may contain concurrent, deferred assertions, variable assignments and event control '@'. Nothing else.
        ii. The following forms of 'always' procedures are allowed in checkers: always_comb, always_latch, and always_ff. Checker 'always' procedures may contain the following statements:

            1. Blocking assignments;
            2. Nonblocking assignments;
            3. Loop statements;
            4. Timing event control,
            5. subroutine calls,
            6. 'let' declarations.

iii. Except for the variables used in 'event' control, all other expressions in 'always_ff' procedures are 'sampled' (in preponed region). Similarly, the expressions in immediate and deferred immediate procedural blocks are also 'sampled'.
iv. However, the expressions in 'always_comb' and 'always_latch' procedures are *not* 'sampled' and the assignments in these procedures use the 'current' value of their expressions. This is an important point to note because it will matter when you debug your results.
For example, (partially, courtesy LRM):

```
checker mycheck(a, b, c, clk, rst);
  logic x,y,z,v,t;
  assign x=a;       //current value of 'a'
  always_ff @(posedge clk or negedge rst)   //current values of 'clk' and 'rst'
  begin
        a1: assert (b);    //sampled value of 'b'
        If (rst)           //current value of 'rst'
              z <= b;  //sampled value of 'b'
        else
              z <= c;  //sampled value of 'c'
  end
  always_comb
  begin
        a2: assert (b);    //current value of 'b'
        If (a)             //current value of 'a'
              v = b;   //current value of 'b'
        else
              v = c;   //current value of 'c'
  end
endchecker
```

h. Generate blocks, containing any of the above elements are allowed.

9. Checker variables can be 'rand' (free variables).
10. The semantics of 'checker' formal arguments is similar to the semantics of 'property' arguments. Almost all formal argument types allowed in properties are also allowed in 'checkers'. BUT they cannot have 'local' qualifier.

11. Just as in properties, you can also use inference system functions such as $inferred_clock and $inferred_disable for checker argument initialization.

```
E.g.,
    checker checker_args
            (sequence start,
             property end,
             string message = "",
             event clk = $inferred_clock,
             rst = $inferred_disable
            );
```

12. You can use 'let' declarations in a checker. (see Sect. 16.14 for 'let')
13. The RHS of a checker variable assignment may contain the sequence method. triggered

```
checker myCheck(a, b, c);
        ...
        sequence busSeq ; ...; endsequence
        always @ (posedge clk) begin a <= busSeq.triggered; end
endchecker
```

16.18.4 Checker: Instantiation Rules

We have already covered Nested Checker rules. This section provides guidelines on checker instantiation rules.

As noted above, a checker can be instantiated anywhere a concurrent assertion can be, except that a checker cannot be instantiated in a procedural construct such as fork..join, fork…join_any or fork…join_none. The mechanism for passing actual arguments to the formal arguments of a checker is the same as that for passing actual arguments to a 'property'. It is important to note that it's the 'sampled' value (i.e. the value in the preponed region) of an actual that is assigned to the formal of the checker (this rule is also the same as that for a property).

Here are the rules for 'formal' and 'actual' of a checker and checker instantiation. Again, they are similar to those applied to a 'property' or a 'sequence'. But some are repeated here for the sake of completeness.

- A 'formal' argument of a checker can be optionally preceded by a direction qualifier: 'input' or 'output'.
- If no direction is specified explicitly then the direction of the previous argument will be inferred.

- If the direction of the first checker argument is omitted, it will default to 'input'
- Obviously, an 'input' checker formal argument cannot be modified by a checker.
- The legal data types of a checker formal arguments are the same as those legal for a property.
- The type of an 'output' argument cannot be of type 'untyped', 'sequence' or 'property'.
- You cannot omit the type of a formal argument, if you have assigned an explicit direction qualifier.
- If you do omit the type of a checker formal argument and if it's the first argument of the checker, then it will be assumed to be 'input untyped'.
- If you do omit the type of a checker formal argument and it is *not* the first argument, then the type of the 'previous' formal argument will be inferred.
- A checker declaration may specify a 'default' value for each singular input.
- A checker declaration may also specify an initial value for each of its singular output using the same syntax as the default value specification for input arguments.

There is a difference between a checker instantiation inside a procedural block or outside. Let us study this via an example

```
`define MAX_SUM 256
checker c1( logic[7:0] a, b);
        logic [7:0] add;
        always @ (posedge clk) begin
            add <= a + 1'b1;
            p0: assert property (add < `MAX_SUM);
        end
        p1: assert property (@ (posedge) clk add  <  `MAX_SUM);
        p2: assert property (@ (posedge clk) clk a != b);
endchecker
module m(input logic rst, clk, logic en, logic[7:0] in1, in2, in_array [20:0]);
       c1 check_outside(in1, in2);  //Concurrent (static) instantiation of 'c1' checker
       always @ (posedge clk) begin
         automatic logic [7:0] v1=0;
         if (en) begin
              c1 check_inside(in1, v1); //Procedural instantiation of 'c1'
         end
         for (int i = 0; i < 4; i++) begin
               v1 = v1+5;
          if (i != 2) begin
               c1 check_loop (in1, in_array [v1]); //Procedural (Loop) instantiation of 'c1'
          end
         end
       end
endmodule: m
```

Points to note in the above example

1. *check_outside* is a static instantiation, while *check_inside* and *check_loop* are procedural. Total of three instantiations of 'c1'.
2. Each of the three instantiation of 'c1' has its own copy of 'add' – which is rather obvious because without it, one instance of 'add' would clobber the 'add' of another instance. This copy of 'add' is updated at every positive clock edge, regardless of whether that instance was visited in procedural code. Even in the case of check_loop, there is only one instance of 'add', and it will be updated using the sampled value of 'in1'.
3. Each of the three instantiations will queue an evaluation of 'p0' at every posedge of the clock. This evaluation will report a violation during any time step when 'add' is not less than MAX_SUM, regardless of the behavior of the procedural code in module 'm'.
4. For checker instance 'check_outside', 'p1' and 'p2' are checked at every posedge clock. For checker instance 'check_inside', 'p1' and 'p2' are queued for evaluation anytime 'en' is true (on posedge clk).
5. For check_loop, three procedural instances of 'p1' and 'p2' are queued (for I = 0,1,3) and they will evaluate at every posedge clk. For 'p1', all three instances are identical using the sampled value of 'add'; but for 'p2', the three instances compare the sampled value of 'in1' to the sampled value of 'in_arry' indexed by constant 'v1' values of 5,10,20 respectively.
6. Since 'c1' (check_outside) instance is static (concurrent), the assertion statements in 'checker c1' are continually monitored and begin execution on any time step when their sampling edge (clock event) occur.

Please note: Checkers for Formal verification are not covered in this book since it is beyond the scope of the book. Specifically, 'assume property' is not explored beyond its context in simulation.

Chapter 17
SystemVerilog Assertions LABs

Introduction: This chapter through six labs with increasing difficulty to solidify the practical features of properties and sequences. The LABs are as follows:

1. 'bind' and implication operators
2. Overlap and non-overlap operators
3. Synchronous FIFO
4. Counter
5. Data Transfer Protocol
6. PCI Read Protocol

Each of these completely self-contained LABs is included on the Springer server whose information is provided with the book. Each LAB includes, the DUT/Testbench models, LAB Questions, 'run' scripts for both Linux and Windows and of course, the .solution directory with all required models so that you can simply execute the 'run' scripts and understand the results and answer the LAB questions.

17.1 LAB1: Assertions with/Without Implication and 'bind'

Please note again that everything noted below (and for all the LABs) is provided on the Springer server. You do not need to rewrite any of the following to run the LABs. The overall view of LAB objectives/questions is given here. Required run scripts/logs/etc. are on the server. The answers for each LAB are included in the . solution directory. The answers are included in the book as well.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_17

17.1.1 LAB1: 'bind' DUT Model and Testbench (Fig. 17.1)

```
************************************************************
LAB1 :: Objective
************************************************************
How to bind a design module to a property module that carries assertions for the design module.
And further confirm your understanding of writing a property with/without implication.

************************************************************
LAB1 :: What you will do...
************************************************************
        1) 'bind' the design module 'dut' with its property module 'dut_property'
        2) Compile/simulate according to instructions given below.
        3) Study simulation log.
        4) Answer questions embedded in the simulation log file
************************************************************
LAB1 :: Database
************************************************************
FILES:
        dut.v :: Verilog module that drives a simple req/gnt protocol.
        dut_property.sv :: File that contains 'dut' properties/assertions.
        test_dut.sv :: Test-bench for the 'dut'.
                This is the file in which you'll 'bind' the 'dut' with 'dut_property'

************************************************************
dut.v
************************************************************
```

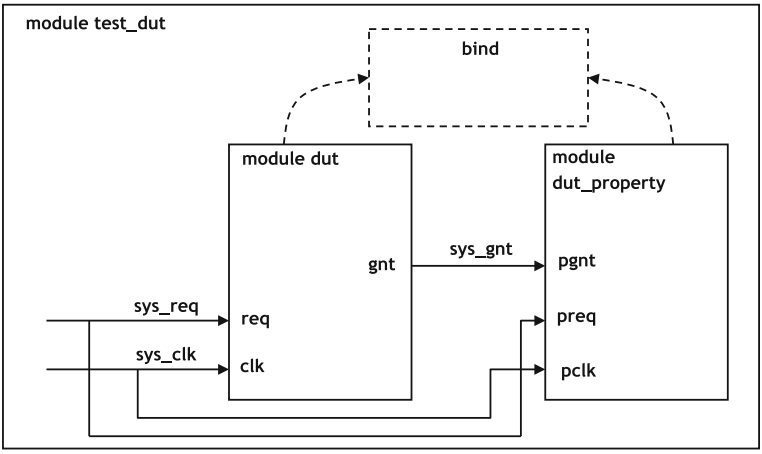


Fig. 17.1 LAB1: 'bind' assertions. Problem definition

```
/* Behavioral Verilog model that acts as the DUT driving a simple req/gnt protocol
*/


module dut(clk, req, gnt);
input logic clk,req;
output logic gnt;

initial
begin
  gnt=1'b0;
end

initial
begin
      @ (posedge req);
        @ (negedge clk); gnt=1'b0;
        @ (negedge clk); gnt=1'b1;

      @ (posedge req);
        @ (negedge clk); gnt=1'b0;
        @ (negedge clk); gnt=1'b0;
end

endmodule

************************************************************
```

dut_property.sv

```
************************************************************

module dut_property(pclk,preq,pgnt);
input pclk, preq, pgnt;

`ifdef no_implication
property pr1;
  @ (posedge pclk) preq ##2 pgnt;
endproperty
preqGnt: assert property (pr1) $display($stime,,,"\t\t %m PASS");
      else $display($stime,,,"\t\t %m FAIL");

`elsif implication
property pr1;
  @ (posedge pclk) preq |-> ##2 pgnt;
endproperty

preqGnt: assert property (pr1) $display($stime,,,"\t\t %m PASS");
      else $display($stime,,,"\t\t %m FAIL");

`elsif implication_novac
property pr1;
  @ (posedge pclk) preq |-> ##2 pgnt;
endproperty


preqGnt: assert property (pr1) else $display($stime,,,"\t\t %m FAIL");

property pr2;
  @ (posedge pclk) preq ##2 pgnt;
endproperty
cpreqGnt: cover property (pr2) $display($stime,,,"\t\t %m PASS");
`endif

endmodule
```

17.1.2 LAB1: Questions

```
************************************************************
test_dut.sv
************************************************************

module test_dut;
bit sys_clk,sys_req;
wire sys_gnt;

/* Instantiate 'dut' */

dut dut1 (
            .clk(sys_clk),
            .req(sys_req),
            .gnt(sys_gnt)
      );

//--------------------------------------
// LAB EXERCISE - START
//--------------------------------------
//
// Add your code to bind 'dut' with 'dut_property' here.

// You need to know the names of the ports in the design as well as the
// property module to be able to bind them. So, here they are:

// ----------------------
// Design module (dut.v)
// ----------------------
// module dut(clk, req, gnt);
//            input logic clk,req;
//            output logic gnt;

// ---------------------------------
// Property module (dut_property.sv)
// ---------------------------------
//module dut_property(pclk,preq,pgnt);
//input pclk,preq,pgnt;

//--------------------------------------
// LAB EXERCISE - END
//--------------------------------------

always @ (posedge sys_clk)
  $display($stime,,,"clk=%b req=%b gnt=%b",sys_clk,sys_req,sys_gnt);

always #10 sys_clk = !sys_clk;

initial
begin
                      sys_req = 1'b0;
  @ (posedge sys_clk) sys_req = 1'b1; //30
  @ (posedge sys_clk) sys_req = 1'b0; //50
  @ (posedge sys_clk) sys_req = 1'b0; //70
  @ (posedge sys_clk) sys_req = 1'b1; //90
  @ (posedge sys_clk) sys_req = 1'b0; //110
  @ (posedge sys_clk) sys_req = 1'b0; //130

@ (posedge sys_clk);
@ (posedge sys_clk); $finish(2);
end

endmodule
```

```
************************************************************
LAB1 - Q U E S T I O N S (based on simulation log)
************************************************************
/* +define+no_implication

run -all
KERNEL:          10  clk=1 req=0 gnt=0
KERNEL:          10               test_implication FAIL
KERNEL:          30  clk=1 req=1 gnt=0
KERNEL:          50  clk=1 req=0 gnt=0
KERNEL:          50               test_implication FAIL
KERNEL:          70  clk=1 req=0 gnt=1
KERNEL:          70               test_implication FAIL
KERNEL:          70               test_implication PASS

            Q: WHY DOES THE PROPERTY FAIL -AND- PASS AT TIME (70) ??

KERNEL:          90  clk=1 req=1 gnt=0
KERNEL:         110  clk=1 req=0 gnt=0
KERNEL:         110               test_implication FAIL
KERNEL:         130  clk=1 req=0 gnt=0
KERNEL:         130               test_implication FAIL
KERNEL:         130               test_implication FAIL

            Q: WHY ARE THERE 2 failures AT TIME (130) ??
*/
/* +define+implication

run -all
KERNEL:          10  clk=1 req=0 gnt=0
KERNEL:          10               test_implication PASS
KERNEL:          30  clk=1 req=1 gnt=0
KERNEL:          50  clk=1 req=0 gnt=0
KERNEL:          50               test_implication PASS
KERNEL:          70  clk=1 req=0 gnt=1
KERNEL:          70               test_implication PASS
KERNEL:          70               test_implication PASS

            WHY ARE THERE 2 PASSes AT TIME 70 ??

KERNEL:          90  clk=1 req=1 gnt=0
KERNEL:         110  clk=1 req=0 gnt=0
KERNEL:         110               test_implication PASS
KERNEL:         130  clk=1 req=0 gnt=0
KERNEL:         130               test_implication FAIL
KERNEL:         130               test_implication PASS

            WHY IS THERE A PASS -and- a FAIL AT TIME 130 ??
*/

/* +define+implication_novac

run -all
KERNEL:          10  clk=1 req=0 gnt=0
KERNEL:          30  clk=1 req=1 gnt=0
KERNEL:          50  clk=1 req=0 gnt=0
KERNEL:          70  clk=1 req=0 gnt=1
KERNEL:          70               test_implication PASS
KERNEL:          90  clk=1 req=1 gnt=0
KERNEL:         110  clk=1 req=0 gnt=0
KERNEL:         130  clk=1 req=0 gnt=0
KERNEL:         130               test_implication FAIL

*/
```

17.2 LAB2: Overlap and Non-overlap Operators

17.2.1 LAB2 DUT Model and Testbench

****

LAB: Objective

****

1) Learn how overlapping implication operator works
2) Learn how non-overlapping implication operator works
3) Learn how pipelined threads work in SVA.

****

LAB: Database

****

test_overlap_nonoverlap.sv :: This file contains the properties, sequences and the test-bench required to simulate the DUT.

****

test_overalp_nonoverlap.sv

****

```
module test_overlap_nonoverlap;
bit clk,cstart,req,gnt;

always @ (posedge clk)
  $display($stime,,, "clk=%b cstart=%b req=%b gnt=%b",clk,cstart,req,gnt);

always #10 clk = !clk;

sequence sr1;
  req ##2 gnt;
endsequence

`ifdef overlap
property pr1;
  @ (posedge clk) cstart |-> sr1;
endproperty

//Note that if a simulator supports filter on vacuous pass for a 'cover'
//the following property is not needed. You can simply use "property pr1"
//for 'cover' as well.

property pr1_for_cover;
  @ (posedge clk) cstart ##0 sr1;
endproperty

`elsif nonoverlap
property pr1;
  @ (posedge clk) cstart |=> sr1;
endproperty

//Note that if a simulator supports filter on vacuous pass for a 'cover'
//the property pr1_for_cover is not needed. You can simply use "property pr1"
//for 'cover' as well.

property pr1_for_cover;
  @ (posedge clk) cstart ##1 sr1;
endproperty
`endif
```

```
reqGnt: assert property (pr1) else $display($stime,,,"\t\t %m FAIL");
creqGnt: cover property (pr1_for_cover) $display($stime,,,"\t\t %m PASS");

initial
begin
  {cstart,req,gnt}=3'b000;
end

initial
begin
  @ (negedge clk); {cstart,req,gnt}=3'b100;
  @ (negedge clk); {cstart,req,gnt}=3'b110;
  @ (negedge clk); {cstart,req,gnt}=3'b000;
  @ (negedge clk); {cstart,req,gnt}=3'b001;

  @ (negedge clk); {cstart,req,gnt}=3'b110;
  @ (negedge clk); {cstart,req,gnt}=3'b110;
  @ (negedge clk); {cstart,req,gnt}=3'b111;
  @ (negedge clk); {cstart,req,gnt}=3'b010;
  @ (negedge clk); {cstart,req,gnt}=3'b000;
  @ (negedge clk); {cstart,req,gnt}=3'b001;

  @ (negedge clk); $finish(2);
end

endmodule
```

17.2.2 LAB2: Questions

```
*********************************************************
LAB Questions based on simulation log
*********************************************************
*********************************************************
Simulation Log - Q U E S  T I O N S
*********************************************************

/* +define+overlap

run -all
KERNEL:          10  clk=1 cstart=0 req=0 gnt=0
KERNEL:          30  clk=1 cstart=1 req=0 gnt=0
KERNEL:          30            test_overlap_nonoverlap FAIL

             Q: WHY DOES THE PROPERTY FAIL at 30?

KERNEL:          50  clk=1 cstart=1 req=1 gnt=0
KERNEL:          70  clk=1 cstart=0 req=0 gnt=0
KERNEL:          90  clk=1 cstart=0 req=0 gnt=1
KERNEL:          90            test_overlap_nonoverlap PASS
```

```
                Q: WHY DOES THE PROPERTY PASS at 90?

KERNEL:         110  clk=1 cstart=1 req=1 gnt=0
KERNEL:         130  clk=1 cstart=1 req=1 gnt=0
KERNEL:         150  clk=1 cstart=1 req=1 gnt=1
KERNEL:         150            test_overlap_nonoverlap PASS

                Q: WHY DOES THE PROPERTY PASS at 150?

KERNEL:         170  clk=1 cstart=0 req=1 gnt=0
KERNEL:         170            test_overlap_nonoverlap FAIL

                Q: WHY DOES THE PROPERTY FAIL at 170?

KERNEL:         190  clk=1 cstart=0 req=0 gnt=0
KERNEL:         190            test_overlap_nonoverlap FAIL

                Q: WHY DOES THE PROPERTY FAIL at 190?

KERNEL:         210  clk=1 cstart=0 req=0 gnt=1

*/

/* +define+nonoverlap

run -all
KERNEL:          10  clk=1 cstart=0 req=0 gnt=0
KERNEL:          30  clk=1 cstart=1 req=0 gnt=0
KERNEL:          50  clk=1 cstart=1 req=1 gnt=0
KERNEL:          70  clk=1 cstart=0 req=0 gnt=0
KERNEL:          70            test_overlap_nonoverlap FAIL

                Q: WHY DOES THE PROPERTY FAIL at 70?

KERNEL:          90  clk=1 cstart=0 req=0 gnt=1
KERNEL:          90            test_overlap_nonoverlap PASS

                Q: WHY DOES THE PROPERTY PASS at 90?

KERNEL:         110  clk=1 cstart=1 req=1 gnt=0
KERNEL:         130  clk=1 cstart=1 req=1 gnt=0
KERNEL:         150  clk=1 cstart=1 req=1 gnt=1
KERNEL:         170  clk=1 cstart=0 req=1 gnt=0
KERNEL:         170            test_overlap_nonoverlap FAIL

                Q: WHY DOES THE PROPERTY FAIL at 170?

KERNEL:         190  clk=1 cstart=0 req=0 gnt=0
KERNEL:         190            test_overlap_nonoverlap FAIL

                Q: WHY DOES THE PROPERTY FAIL at 190?

KERNEL:         210  clk=1 cstart=0 req=0 gnt=1
KERNEL:         210            test_overlap_nonoverlap PASS

                Q: WHY DOES THE PROPERTY PASS at 210?
*/
```

17.3 LAB3: Synchronous FIFO Assertions

17.3.1 LAB3: DUT Model and Testbench

```
************************************************************
LAB3
************************************************************
```

We saw an example of an asynchronous FIFO in Sect. 14.1 and assertions thereof. For this LAB, I have chosen a simpler Synchronous FIFO for which you will exercise writing assertions. This way you will be familiar with writing assertions for both styles of FIFO. Note that one of the most important set of assertions that you may write for your project are the FIFO assertions. Like it or not, FIFOs always give trouble! (Fig. 17.2).

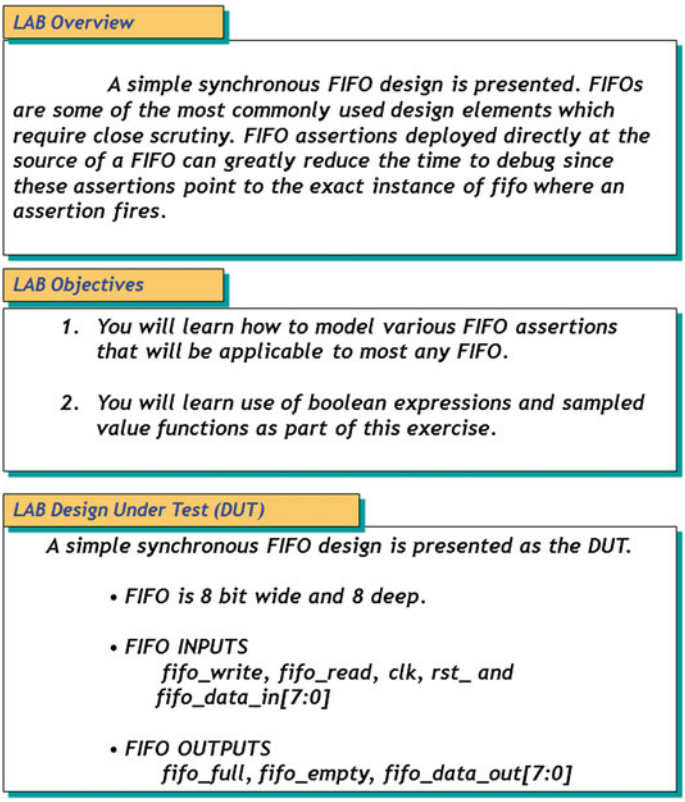
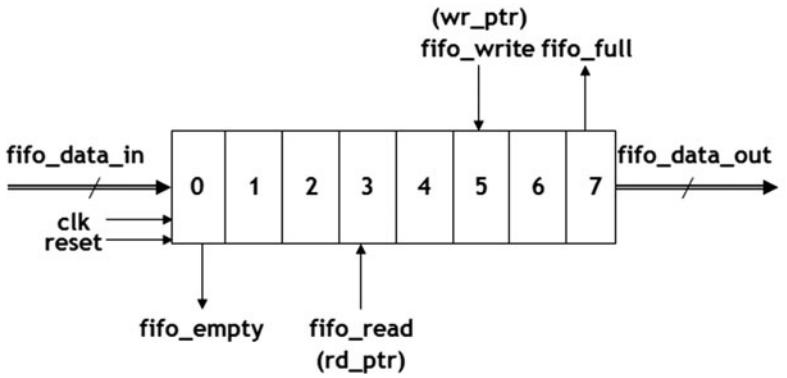


Fig. 17.2 LAB3: Synchronous FIFO: problem definition

FIFO Specs

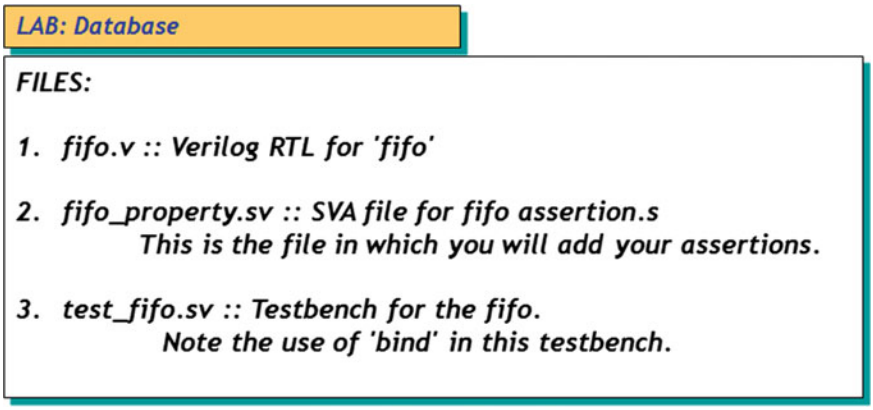
- *fifo maintains a wr_ptr and a rd_ptr*
  - *wr_ptr increments by 1 everytime a write is posted to the fifo on a fifo_write request*
  - *rd_ptr increments by 1 everytime a read is posted to the fifo on a fifo_read request*
- *fifo maintains a 'cnt' that increments on a write and decrements on a read. It is used to signal fifo_full and fifo_empty conditions as follows*
  - *When fifo 'cnt' is >= 7, fifo_full is asserted*
  - *When fifo 'cnt' is 0, fifo_empty is asserted.*



****

LAB3 :: Database

****



****

fifo.v

****

```
module fifo (fifo_data_out,fifo_full,fifo_empty,fifo_write,fifo_read,clk,rst_,
             fifo_data_in);

parameter fifo_depth=8;
parameter fifo_width=8;

output logic [(fifo_width-1):0] fifo_data_out;
output logic fifo_full, fifo_empty;
input  logic fifo_write, fifo_read, clk, rst_;
input  logic [(fifo_width-1):0] fifo_data_in;

logic [(fifo_width-1):0] fifomem [0:(fifo_depth-1)];

logic [3:0] wr_ptr, rd_ptr;
logic [3:0] cnt;

always @ (posedge clk or negedge rst_)
  if (!rst_) begin
    rd_ptr  <= 0;
    wr_ptr  <= 0;
    cnt   <= 0;
`ifndef check1
    fifo_empty <= 1;
```

```
`endif
    fifo_full  <= 0;
  end
  else begin
    case ({fifo_write, fifo_read})
      2'b00: ;     // everyone's sleeping!
      2'b01: begin // read
               if (cnt>0) begin
                 rd_ptr <= rd_ptr + 1;
                 cnt  <= cnt  - 1;
               end
`ifdef check2
               if (cnt==0) fifo_empty <= 1;
`else
`ifdef check5
               if (cnt==1) fifo_empty <= 1;
             rd_ptr <= rd_ptr+1;
`else
               if (cnt==1) fifo_empty <= 1;
`endif
`endif
               fifo_full <= 0;
             end
      2'b10: begin // write
               if (cnt< fifo_depth) begin
                 fifomem[wr_ptr] <= fifo_data_in;
                 wr_ptr <= wr_ptr + 1;
                 cnt  <= cnt  + 1;
               end
`ifdef check3
               if (cnt>(fifo_depth-1)) fifo_full <= 1;
`else
`ifdef check4
               if (cnt>=(fifo_depth-1)) fifo_full <= 1;
             wr_ptr <= wr_ptr+1;
`else
               if (cnt>=(fifo_depth-1)) fifo_full <= 1;
`endif
`endif
               fifo_empty <= 0;
             end
      2'b11: // write && read
             //You cannot write if cnt is full; so read only
               if (cnt>(fifo_depth-1)) begin
                 rd_ptr  <= rd_ptr + 1;
                 cnt   <= cnt  - 1;
               end
            //You cannot read if cnt is empty; so write only
               else if (cnt<1) begin
                 fifomem[wr_ptr] <= fifo_data_in;
                 wr_ptr  <= wr_ptr + 1;
                 cnt   <= cnt  + 1;
               end
```

```
                //else write and read both
                   else begin
                     fifomem[wr_ptr] <= fifo_data_in;
                     wr_ptr  <= wr_ptr + 1;
                     rd_ptr  <= rd_ptr + 1;
                   end
      endcase
    end

assign fifo_data_out = fifomem[rd_ptr];

endmodule
```

17.3.2 LAB3: Questions

LAB: Assertions to Code

Code assertions to check for the following conditions in the 'fifo' design.

CHECK # 1. Check that on reset
rd_ptr=0; wr_ptr=0; cnt=0; fifo_empty=1 and fifo_full=0

CHECK # 2. Check that fifo_empty is asserted when fifo 'cnt' is 0.
Disable this property 'iff (!rst)'

CHECK # 3. Check that fifo_full is asserted any time fifo 'cnt' is greater than 7.
Disable this property 'iff (!rst)'

CHECK # 4. Check that if fifo is full and you attempt to write (but not read) that the wr_ptr does not change.

CHECK # 5. Check that if fifo is empty and you attempt to read (but not write) that the rd_ptr does not change.

CHECK # 6. Write a property to Warn on write to a full fifo

CHECK # 7. Write a property to Warn on read from an empty fifo

****
LAB3: Questions – Assertion Questions embedded in the fifo_property.sv
****

I have provided the fifo_property.sv file. All you have to do is write your assertions in this file and simulate. I have coded '***dummy***' properties so that you can compile the code. The .solution directory contains correct assertions and simulation log against which you can compare your results. Note that there is not just but one way to write an assertion and your assertion could look different from the one you see in the .solution directory. But the results must match with the simulation log in the .solution directory

```
`define rd_ptr test_fifo.fi1.rd_ptr
`define wr_ptr test_fifo.fi1.wr_ptr
`define cnt test_fifo.fi1.cnt

module fifo_property (
      input logic [7:0] fifo_data_out,
      input logic       fifo_full, fifo_empty,
      input logic       fifo_write, fifo_read, clk, rst_,
      input logic [7:0] fifo_data_in
                  );

parameter fifo_depth=8;
parameter fifo_width=8;

//      ------------------------------------------
//        1. Check that on reset,
//           rd_ptr=0; wr_ptr=0; cnt=0; fifo_empty=1 and fifo_full=0
//      ------------------------------------------

`ifdef check1
property check_reset;
  @ (posedge clk) !rst_ |-> `rd_ptr==0; //DUMMY... remove this line and
                                        //replace it with correct check
endproperty
check_resetP: assert property (check_reset) else $display($stime,"\t\t
FAIL::check_reset\n");
`endif




//      ------------------------------------------
//        2. Check that fifo_empty is asserted the same clock that fifo 'cnt'
//           is 0. Disable this property 'iff (!rst)'
//      ------------------------------------------

`ifdef check2
property fifoempty;
  @ (posedge clk) !rst_ |-> `rd_ptr==0; //DUMMY... remove this line and
                                        //replace it with correct check
endproperty
fifoemptyP: assert property (fifoempty) else $display($stime,"\t\t FAIL::fifo_empty
condition\n");
`endif

//      ------------------------------------------
//        3. Check that fifo_full is asserted any time fifo 'cnt'
//           is greater than 7. Disable this property 'iff (!rst)'
//      ------------------------------------------

`ifdef check3
property fifofull;
  @ (posedge clk) !rst_ |-> `rd_ptr==0; //DUMMY... remove this line and
```

```
                                        //replace it with correct check
endproperty
fifofullP: assert property (fifofull) else $display($stime,"\t\t FAIL::fifo_full
condition\n");
`endif

//      ----------------------------------------
//       4. Check that if fifo is full and you attempt to write (but not read)
//           that the wr_ptr does not change.
//      ----------------------------------------

`ifdef check4
property fifo_full_write_stable_wrptr;
  @ (posedge clk) !rst_ |-> `rd_ptr==0; //DUMMY... remove this line and
                                        //replace it with correct check
endproperty
fifo_full_write_stable_wrptrP: assert property (fifo_full_write_stable_wrptr)
      else $display($stime,"\t\t FAIL::fifo_full_write_stable_wrptr condition\n");
`endif

`ifdef check5

//      ----------------------------------------
//      5. Check that if fifo is empty and you attempt to read (but not write)
//          that the rd_ptr does not change.
//      ----------------------------------------

property fifo_empty_read_stable_rdptr;
  @ (posedge clk) !rst_ |-> `rd_ptr==0; //DUMMY... remove this line and
                                        //replace it with correct check
endproperty
fifo_empty_read_stable_rdptrP: assert property (fifo_empty_read_stable_rdptr)
      else $display($stime,"\t\t FAIL::fifo_empty_read_stable_rdptr condition\n");
`endif

//      ----------------------------------------
//        6. Write a property to Warn on write to a full fifo
//    This property will give Warning with all simulations
//      ----------------------------------------

`ifdef check6
property write_on_full_fifo;
  @ (posedge clk) !rst_ |-> `rd_ptr==0; //DUMMY... remove this line and
                                        //replace it with correct check
endproperty
write_on_full_fifoP: assert property (write_on_full_fifo)
      else $display($stime,"\t\t WARNING::write_on_full_fifo\n");
`endif

//      ----------------------------------------
//        7. Write a property to Warn on read from an empty fifo
//    This property will give Warning with all simulations
//      ----------------------------------------
```

```
`ifdef check7
property read_on_empty_fifo;
  @ (posedge clk) !rst_ |-> `rd_ptr==0; //DUMMY... remove this line and
                                        //replace it with correct check
endproperty
read_on_empty_fifoP: assert property (read_on_empty_fifo)
      else $display($stime,"\t\t WARNING::read_on_empty_fifo condition\n");
`endif

endmodule

*********************************************************
```

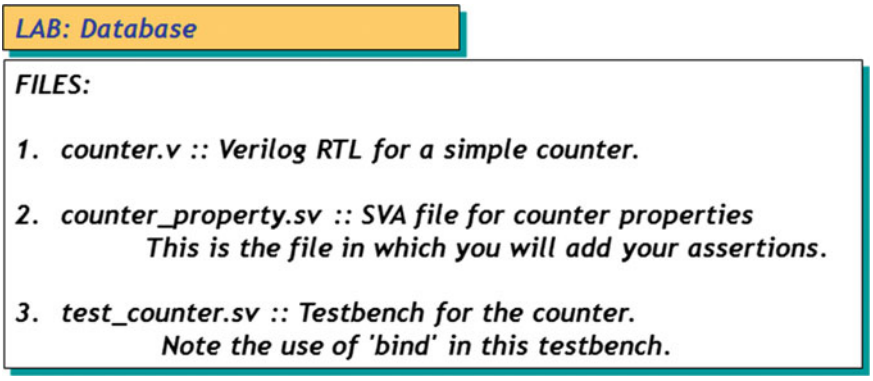
test_fifo.sv

```
*********************************************************
module test_fifo;

wire  [7:0] fifo_data_out;
wire        fifo_full, fifo_empty;
logic       fifo_write, fifo_read, clk, rst_;
logic [7:0] fifo_data_in;

parameter fifo_depth = 8, fifo_width = 8;

fifo #(fifo_depth,fifo_width) fi1
(fifo_data_out,fifo_full,fifo_empty,fifo_write,fifo_read,clk,rst_,
                 fifo_data_in);

bind fifo fifo_property #(fifo_depth,fifo_width) fi1bind
      (fifo_data_out,fifo_full,fifo_empty,fifo_write,fifo_read,clk,rst_,fifo_data_i
n);

initial
begin
  clk=0;
  fiforeset;
  fifowrite(10);
  fiforead(9);
  @ (posedge clk);
  @ (posedge clk);
  @ (posedge clk); $finish(2);
end

always #5 clk=!clk;

task fiforeset;
  fifo_write=0; fifo_read=0; rst_=1;
  @ (negedge clk); rst_=0;
  @ (negedge clk);
  @ (negedge clk); rst_=1;
endtask

task fifowrite;
```

```
input int nwrite;
  fifo_read=0;
  for (int i=0; i<=nwrite-1; i++)
  begin
    @ (negedge clk); fifo_write=1; fifo_data_in=i;
      //$display($stime,,,"fifo Write Data = %0d",fifo_data_in);
  end
endtask

task fiforead;
input int nread;
  fifo_write=0;
  repeat(nread)
  begin
    @ (negedge clk); fifo_read=1;
      //$display($stime,,,"fifo Read Data = %0d",fifo_data_out);
  end
endtask

always @ (posedge clk)
  $display($stime,,,"rst_=%b clk=%b fifo_write=%b fifo_read=%b fifo_full=%b
fifo_empty=%b wr_ptr=%0d rd_ptr=%0d cnt=%0d",
      rst_,clk,fifo_write,fifo_read,fifo_full,fifo_empty,fi1.wr_ptr,fi1.rd_ptr,fi1.
cnt);

endmodule
```

17.4 LAB4: Counter

See Fig. 17.3.



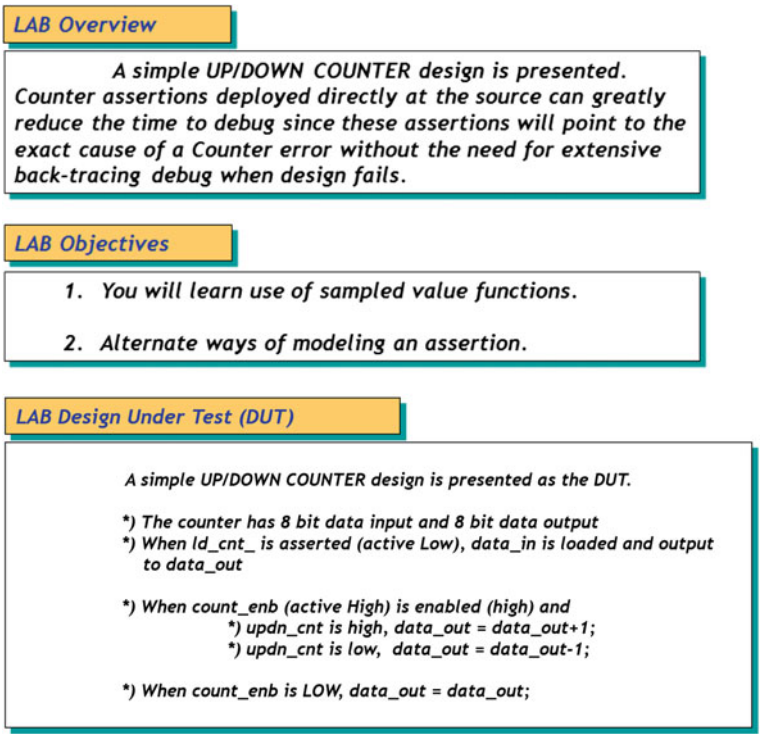
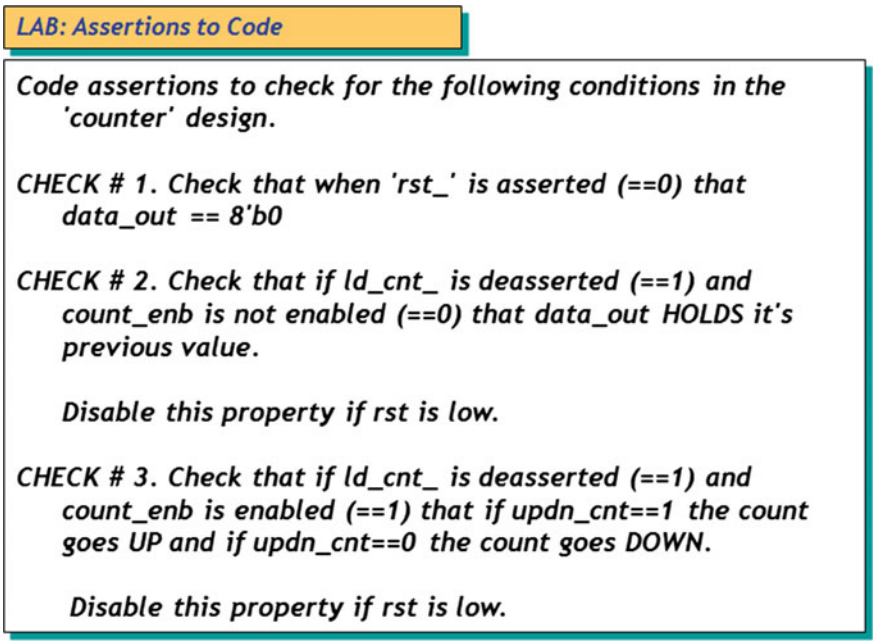


Fig. 17.3 LAB4: counter: problem definition

```
************************************************************
counter.v
************************************************************

module counter (
  input clk, rst_, ld_cnt_, updn_cnt, count_enb,
  input [7:0] data_in,
  output logic [7:0] data_out
  );

always @ (posedge clk or negedge rst_)
begin
  if (!rst_)
    begin
    `ifndef check1
       data_out <= 0;
    `endif
     end
  else
  begin

    //LOAD DATA
    if (!ld_cnt_)
      data_out <= data_in;

    //HOLD DATA
    `ifndef check2
     else if (!count_enb)
      data_out <= data_out;
    `endif

    //COUNT DATA
    `ifdef check3
    else
      case (updn_cnt)
        1'b1: data_out <= data_out - 1;
        1'b0: data_out <= data_out + 1;
      endcase
    `else
    else
      case (updn_cnt)
        1'b1: data_out <= data_out + 1;
        1'b0: data_out <= data_out - 1;
      endcase
    `endif

  end
end
endmodule
```



17.4.1 LAB4: Questions

```
************************************************************
LAB4: Questions embedded in counter_property.sv
************************************************************
module counter_property (
  input clk, rst_, ld_cnt_, updn_cnt, count_enb,
  input [7:0] data_in,
  input logic [7:0] data_out
  );

//-------------------------------------
//     CHECK # 1. Check that when 'rst_' is asserted (==0) that
//     data_out == 8'b0
//-------------------------------------

`ifdef check1
property counter_reset;
  @ (posedge clk) data_in |=> data_out; // DUMMY - REMOVE this line and code
                                        // correct assertion
endproperty
```

```
counter_reset_check: assert property(counter_reset)
  else $display($stime,,,
                "\t\tCOUNTER RESET CHECK FAIL:: rst_=%b data_out=%0d \n",
                rst_,data_out);
`endif

//-------------------------------------
// CHECK # 2.Check that if ld_cnt_ is de-asserted (==1) and count_enb is not
// enabled (==0) that data_out HOLDS its previous value.
// Disable this property 'iff (!rst)'
//-------------------------------------

`ifdef check2
property counter_hold;
  @ (posedge clk) data_in |=> data_out; // DUMMY - REMOVE this line and code
                                        //correct assertion
endproperty

counter_hold_check: assert property(counter_hold)
  else $display($stime,,,"\t\tCOUNTER HOLD CHECK FAIL:: counter HOLD \n");
`endif




//-------------------------------------
//CHECK # 3. Check that if ld_cnt_ is de-asserted (==1) and count_enb is
//           enabled(==1) that if updn_cnt==1 the count goes UP and if
//           updn_cnt==0 the count goes DOWN.
//           Disable this property 'iff (!rst)'
//-------------------------------------

`ifdef check3
property counter_count;
  @ (posedge clk) data_in |=> data_out; // DUMMY - REMOVE this line and code
                                        //correct assertion
endproperty

counter_count_check: assert property(counter_count)
  else $display($stime,,,
          "\t\tCOUNTER COUNT CHECK FAIL:: UPDOWN COUNT using $past \n");
`endif

endmodule

************************************************************
```

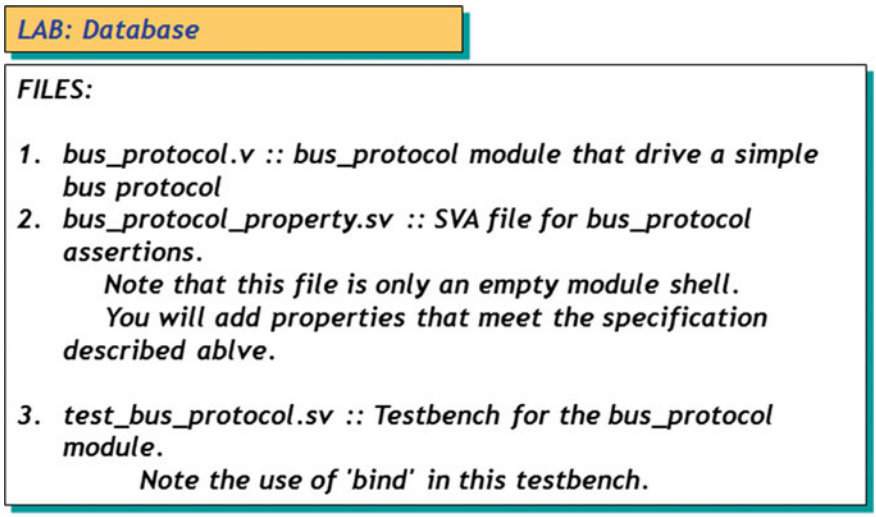
test_counter.sv

```
************************************************************
module test_counter;

logic clk, rst_, ld_cnt_, updn_cnt, count_enb;
logic [7:0] data_in;
wire [7:0] data_out;

int seed1;
```

```
counter  upc(
  clk, rst_, ld_cnt_, updn_cnt, count_enb,
  data_in,
  data_out
  );

bind counter counter_property bind_inst (
  clk, rst_, ld_cnt_, updn_cnt, count_enb,
  data_in,
  data_out
  );

initial
begin
  clk=1'b0;
  counter_init;
      count_up(100,10);
  repeat (2) @ (posedge clk);
      count_down(100,10);
  repeat (2) @ (posedge clk);
  @ (posedge clk); $finish(2);
end

always @ (posedge clk)
   $display($stime,,,"rst_=%b clk=%b count_enb=%b ld_cnt_=%b updn_cnt=%b DIN=%0d
DOUT=%0d",
   rst_, clk, count_enb, ld_cnt_, updn_cnt, data_in, data_out);

always #5 clk=!clk;

task counter_init;
  rst_=1'b1; ld_cnt_=1'b1; count_enb=1'b0; updn_cnt=1'b1;

  @ (negedge clk); rst_=1'b0;
  @ (negedge clk);
  @ (negedge clk); rst_=1'b1;
  @ (negedge clk); data_in=8'b0; ld_cnt_=1'b0;
  @ (negedge clk);
endtask

task count_up;
input logic [7:0] din;
input int count;
  @ (negedge clk); data_in=din; ld_cnt_=1'b0;
  @ (negedge clk); ld_cnt_=1'b1; count_enb=1'b1; updn_cnt=1'b1;
  repeat (count-1) @ (negedge clk);
  @ (negedge clk); count_enb=1'b0;
endtask




        task count_down;
        input logic [7:0] din;
        input int count;
          @ (negedge clk); data_in=din; ld_cnt_=1'b0;
          @ (negedge clk); ld_cnt_=1'b1; count_enb=1'b1; updn_cnt=1'b0;
          repeat (count-1) @ (negedge clk);
          @ (negedge clk); count_enb=1'b0;
        endtask

        endmodule
```

17.5 LAB5: Data Transfer Protocol

See Fig. 17.4.



LAB Objectives

Bus interfaces are common to any design and this lab will show you how to model assertions for common bus protocol specification.

You will learn

1. *Modeling temporal domain assertions for bus interface type logic.*
2. *Reinforce understanding of Edge sensitive and sampled value functions, consecutive repetition, boolean expressions, etc.*

LAB: Assertions to Code

Code assertions to check for the following conditions in the 'bus protocol' design.

CHECK # 1. Check that once dValid goes high that it is consecutively asserted (high) for minimum 2 and maximum 4 clocks

CHECK # 2. Check that data is not unknown and remains stable after dValid goes high and until dAck goes high.

CHECK # 3. Check that 'dAck' and 'dValid' relationship is maintained to complete the data transfer.

In other words,

'dack' going high signifies that target have accepted data and that master must de-assert 'dValid' the clock after 'dack' goes high.

Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and that it must not remain low for more than 3 clocks (because data must transfer in max 4 clocks).

LAB Overview

Specification for a simple data transfer protocol.

- dValid must remain asserted for minimum of 2 clocks but no more than 4 clocks.
- 'data' must be known when 'dValid' is High.
- 'dack' going high signifies that target have accepted data and that master must de-assert 'dValid' the clock after 'dack' goes high.
  - *Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and that it must not remain low for more than 3 clocks (because data must trasnfer in max 4 clocks).*

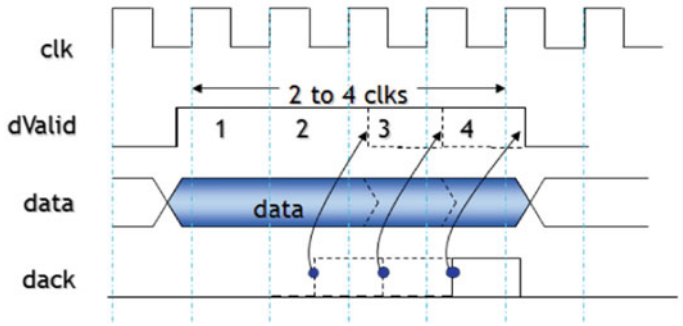


Fig. 17.4 LAB5: data transfer protocol: problem definition

```
*********************************************************
bus_protocol.sv
*********************************************************


/* bus_protocol.v module

   This module drives the bus protocol
   timing diagram.

   This module acts as the bus interface unit of
   your design whose protocol you are trying to verify.

*/
module bus_protocol (input bit clk, reset,
                     output bit dValid, dAck,
                     output logic [7:0] data
);

initial
begin
  $display("SCENARIO 1");
  @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
  @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
  @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
  @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
  @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
  @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
```

```
    $display("\n");

    $display("SCENARIO 2");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");

    $display("SCENARIO 3");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");

  `ifdef nobugs
    $display("SCENARIO 4");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
```

```
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");
  `else
    $display("SCENARIO 4");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");
  `endif

  `ifdef nobugs
    $display("SCENARIO 5");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");
  `else
```

```
    $display("SCENARIO 5");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");
  `endif

  `ifdef nobugs
    $display("SCENARIO 6");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");
  `else
    $display("SCENARIO 6");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");
```

```
  `endif


  `ifdef nobugs
    $display("SCENARIO 7");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;
    $display("\n");
  `else
    $display("SCENARIO 7");
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'hx; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;


    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h1; dAck=1'b0;
    @(negedge clk); dValid=1'b1; data=8'h0; dAck=1'b1;
    @(negedge clk); dValid=1'b0; data=8'h0; dAck=1'b0;


    @(negedge clk);
    @(negedge clk);
    $display("\n");
  `endif


    @(negedge clk);
    $finish(2);
  end
  endmodule
```

17.5.1 LAB5: Questions

```
*********************************************************
bus_protocol_property.sv : Questions embedded in the file
*********************************************************

/* Properties (assertions) for bus_protocol.v
*/
module bus_protocol_property (input bit clk, dValid, dAck, reset,
                              input logic [7:0] data
);

   /*-------------------------------------------------
    CHECK # 1. Check that once dValid goes high that it is consecutively
    asserted (high) for minimum 2 and maximum 4 clocks.
    Check also that once dValid is asserted (high) for 2 to 4 clocks that
    it does de-assert (low) the very next clock.
    -------------------------------------------------*/

`ifdef check1
        property checkValid;
        @ (posedge clk) dValid |-> dValid; //DUMMY - REMOVE this line and code
                                           //correct assertion
        endproperty
        assert property (checkValid) else
                                     $display($stime,,,"checkValid FAIL");
`endif

   /*-------------------------------------------------
    CHECK # 2. Check that data is not unknown and remains stable after
             dValid goes high and until dAck goes high.
   -------------------------------------------------*/

`ifdef check2
        property checkdataValid;
         @ (posedge clk) disable iff (reset)
         @ (posedge clk) dValid |-> dValid; //DUMMY - REMOVE this line and
                                            //code correct assertion
        endproperty
        assert property (checkdataValid) else
                                   $display($stime,,,"checkdataValid FAIL");
`endif
```

```
 /*-----------------------------------------------------
  CHECK # 3.
 'dack' going high signifies that target have accepted data and that master
  must de-assert 'dValid' the clock after 'dack' goes high.

  Note that since data must be valid for minimum 2 cycles, that 'dack' cannot
  go High for at least 1 clock after the transfer starts (i.e. after the
  rising edge of 'dValid') and that it must not remain low for more than 3
  clocks (because data must transfer in max 4 clocks).
 -----------------------------------------------------*/

`ifdef check3
     property checkdAck;
       @ (posedge clk) dValid |-> dValid; //DUMMY - REMOVE this line and code
                                          //correct assertion
     endproperty
     assert property (checkdAck) else $display($stime,,,"checkdAck FAIL");
`endif

Endmodule

************************************************************
test_bus_protocol.v
************************************************************

module test_bus_protocol (output bit clk, reset,
                          input logic dValid, dAck,
                          input logic [7:0] data);

bus_protocol bp1(.*);
bind bus_protocol bus_protocol_property bpb1 (.*);

initial begin clk=1; reset=1; end
always #5 clk=!clk;

initial
begin
  @ (negedge clk); reset=1;
  @ (negedge clk); reset=0;
end

always @ (posedge clk)
  $display($stime,,,"clk=%b dValid=%b data=%h dAck=%b",
               clk,dValid,data,dAck);

endmodule
```

17.6 LAB6: PCI Read Protocol

See Fig. 17.5.

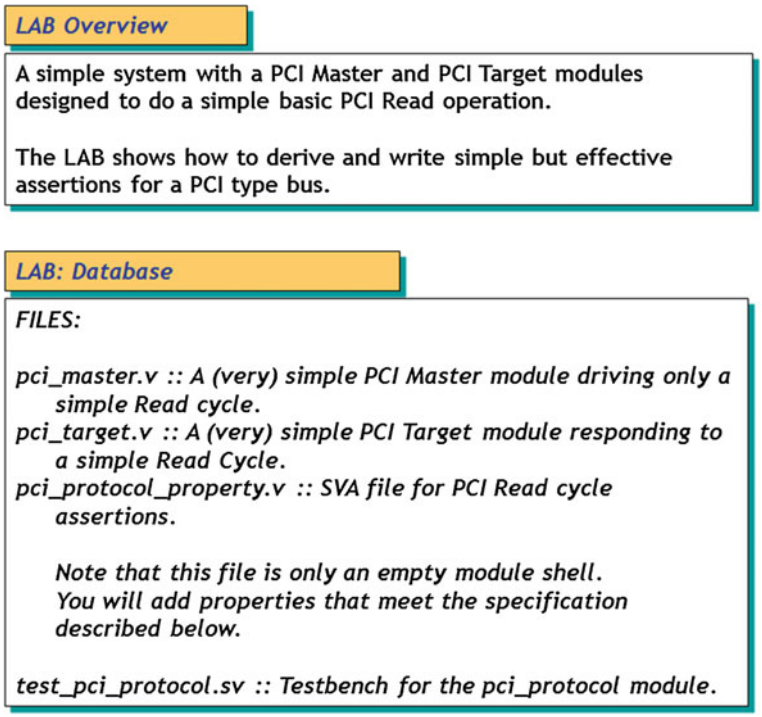
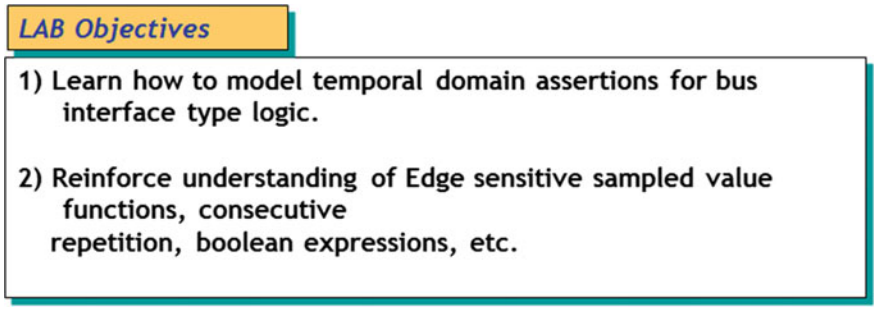
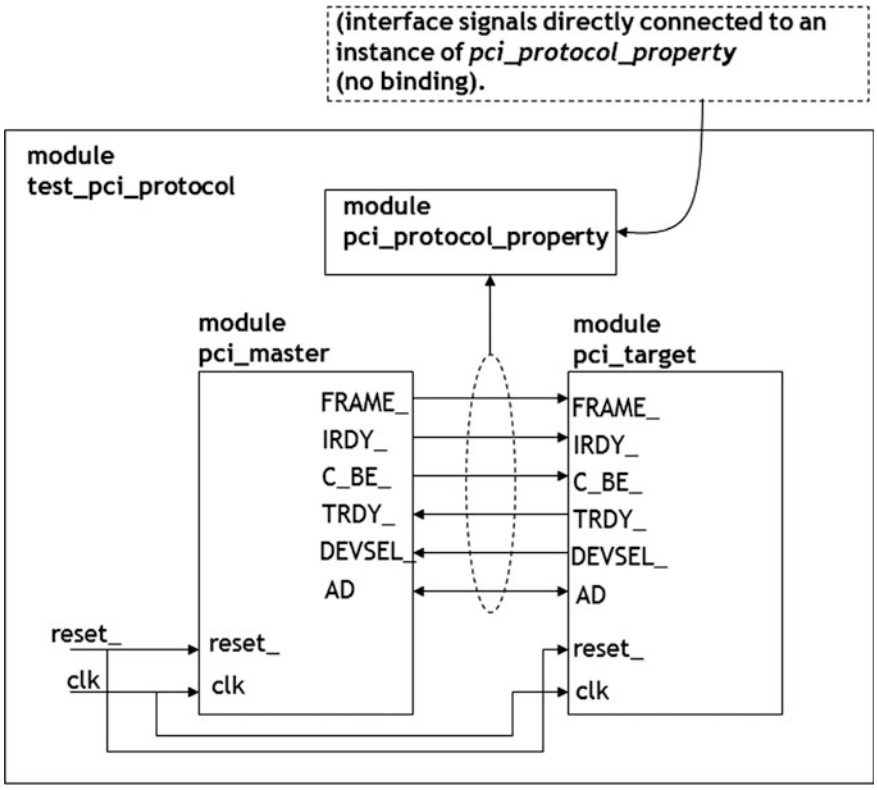
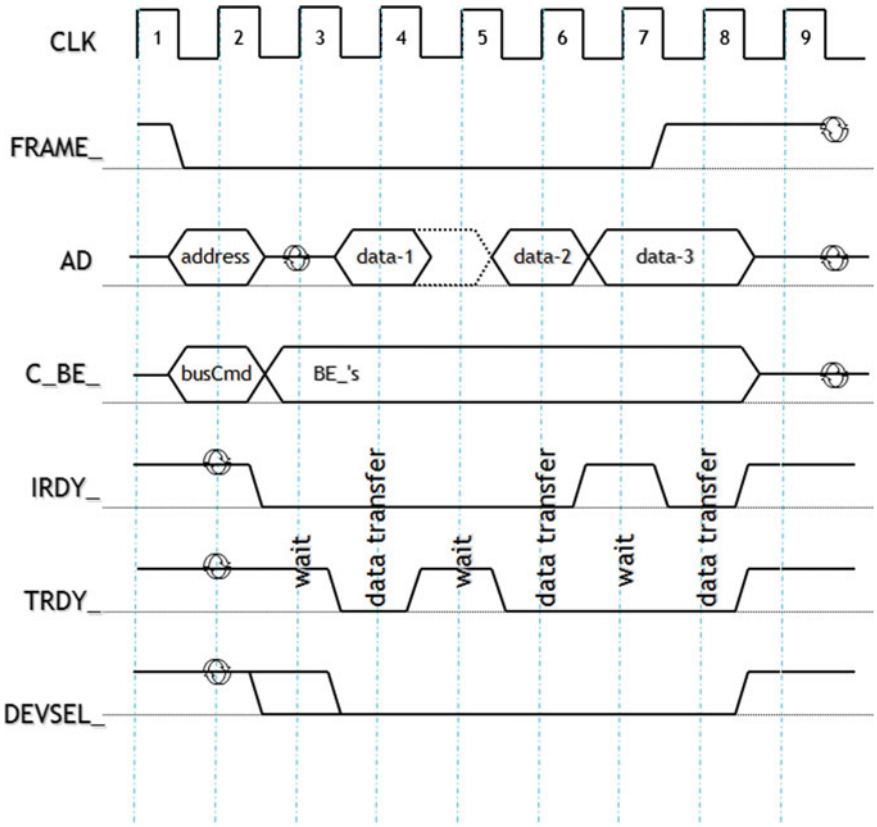


Fig. 17.5 LAB6: PCI protocol: problem definition




(interface signals directly connected to an instance of pci_protocol_property (no binding).
module test_pci_protocol
module pci_protocol_property
module pci_master
module pci_target
FRAME_
IRDY_
C_BE_
TRDY_
DEVSEL_
AD
reset_
clk


CLK
1
2
3
4
5
6
7
8
9
FRAME_
AD
address
data-1
data-2
data-3
C_BE_
busCmd
BE_'s
IRDY_
TRDY_
DEVSEL_
wait
data transfer
wait
data transfer
wait
data transfer

17.6.1 LAB6: Questions

LAB: Assertions to Code

Property Name	Description
checkPCI_AD_CBE (check1)	On falling edge of FRAME_, AD or C_BE_ bus cannot be unknown
checkPCI_DataPhase (check2)	When both IRDY_ and TRDY_ are asserted, AD or C_BE_ bus cannot be unknown
checkPCI_Frame_Irdy (check3)	FRAME can be de-asserted only if IRDY_ is asserted
checkPCI_trdyDevsel (check4)	TRDY_ can be asserted only if DEVSEL_ is asserted
checkPCI_CBE_during_trx (check5)	Once the cycle starts (i.e. at FRAME_ assertion) C_BE_ cannot float until FRAME_ is de-asserted.

```
************************************************************
pci_protocol_property.sv- LAB6 Questions embedded in code
************************************************************

module pci_protocol_property (input logic clk, reset_, TRDY_, DEVSEL_, FRAME_,
                                          IRDY_,
                              input logic [3:0] C_BE_,
                              input logic [31:0] AD
);

  /*-----------------------------------------------------
        CHECK # 1. On falling edge of FRAME_, AD or C_BE_
                   cannot be unknown.
   -----------------------------------------------------*/

`ifdef check1
        property checkPCI_AD_CBE;
          @ (posedge clk) disable iff (!reset_)
            FRAME_ |-> 1'b1; //DUMMY -REMOVE this line and code correct
                            //assertion
        endproperty
        assert property (checkPCI_AD_CBE) else
$display($stime,,,"CHECK1:checkPCI_AD_CBE FAIL\n");
```


The information contained herein is the exclusive property of Ashok B. Mehta, and shall not be distributed, reproduced, or disclosed in whole or in part without prior written permission of Ashok B. Mehta.

```
`endif

 /*-------------------------------------------------
  CHECK # 2. When IRDY_ and TRDY_ are asserted (low) AD
            or C_BE_ cannot be unknown.
  -------------------------------------------------*/

`ifdef check2
       property checkPCI_DataPhase;
         @ (posedge clk) disable iff (!reset_)
           FRAME_ |-> 1'b1; // DUMMY - REMOVE this line and code correct
                            //assertion
       endproperty
       assert property (checkPCI_DataPhase) else
$display($stime,,,"CHECK2:checkPCI_DataPhase FAIL\n");
`endif

/*-------------------------------------------------
CHECK # 3. FRAME_ can go High only if IRDY_ is asserted.
           In other words, master can signify end of cycle
           only if IRDY_ is asserted.
-------------------------------------------------*/

`ifdef check3
       property checkPCI_Frame_Irdy;
         @ (posedge clk) disable iff (!reset_)
           FRAME_ |-> 1'b1; // DUMMY - REMOVE this line and code correct
                            //assertion
       endproperty
       assert property (checkPCI_Frame_Irdy) else
$display($stime,,,"CHECK3:checkPCI_frmIrdy FAIL\n");
`endif




/*-------------------------------------------------
 CHECK # 4. TRDY_ can be asserted (low) only if DEVSEL_
            is asserted (low)
 -------------------------------------------------*/

`ifdef check4
       property checkPCI_trdyDevsel;
         @ (posedge clk) disable iff (!reset_)
           FRAME_ |-> 1'b1; // DUMMY - REMOVE this line and code correct
                            //assertion
       endproperty
       assert property (checkPCI_trdyDevsel) else
$display($stime,,,"CHECK4:checkPCI_trdyDevsel FAIL\n");
`endif

/*-------------------------------------------------
 CHECK # 5. Once the cycle starts (i.e. at FRAME_ assertion)
            C_BE_ should not float until FRAME_ is de-asserted
 -------------------------------------------------*/

`ifdef check5
       property checkPCI_CBE_during_trx;
         @ (posedge clk) disable iff (!reset_)
           FRAME_ |-> 1'b1; // DUMMY - REMOVE this line and code correct
                            //assertion
       endproperty
       assert property (checkPCI_CBE_during_trx) else
$display($stime,,,"CHECK5:checkPCI_CBE_during_trx FAIL\n");
`endif
Endmodule
```

Chapter 18
SystemVerilog Assertions—LAB Answers

This chapter provides answers to all the LAB questions posed in previous chapter, namely, answers for the following LABs are presented.

1. 'bind' and implication operators
2. Overlap and non-overlap operators
3. Synchronous FIFO
4. Counter
5. Data Transfer Protocol
6. PCI Read Protocol

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_18

18.1 LAB1: Answers: ‘bind’ and Implication Operators

See Figs. 18.1, 18.2 and 18.3.

LAB 1 : Code snippet from test_dut.sv showing 'bind' between 'dut' and 'dut_property'

```
bind dut dut_property dut_bind_inst (
            .pclk(clk),
            .preq(req),
            .pgnt(gnt)
      );
```

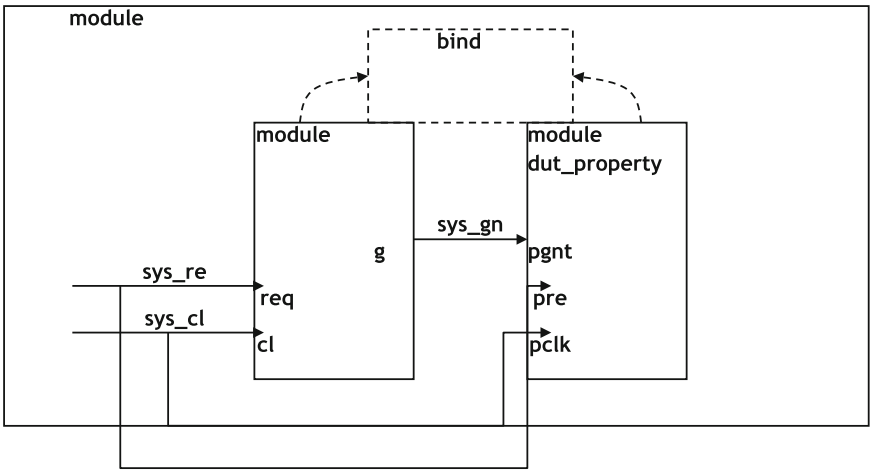


Fig. 18.1 LAB1: ‘bind’ assertions (answers)

LAB 1 : Code snippet for "no_implication"

```
property pr1;
  @(posedge clk) req ##2 gnt;
endproperty
reqGnt: assert property (pr1) $display($stime,,,"\t\t  %m PASS");
              else $display($stime,,,"\t\t  %m FAIL");
```

LAB 1 : Q&A on "no_implication"

```
/* +define+no_implication

run -all
KERNEL:       10  clk=1 req=0 gnt=0
KERNEL:       10            test_implication FAIL
KERNEL:       30  clk=1 req=1 gnt=0
KERNEL:       50  clk=1 req=0 gnt=0
KERNEL:       50            test_implication FAIL
KERNEL:       70  clk=1 req=0 gnt=1
KERNEL:       70            test_implication FAIL
KERNEL:       70            test_implication PASS

              Q: WHY IS THERE A FAIL -AND- A PASS AT TIME (70) ??

              A: The FAIL at 70 is for the thread starting at time 70.
                 At 70, req==0 and since there is no implication, the property fails
                 because without an implication there is no antecedent to match before
                 the check begins. Whenever at posedge clk, 'req' is detected low,
                  the property will fail.

                 The PASS at 70 is for the thread that starts at 30.
                 At 30, req==1, so property eval proceeds.
                 At 70 (i.e. 2 clocks later) gnt==1 as required by the property and the
                 property PASSes.
```

Fig. 18.2 LAB1: Q&A on 'no_implication' operator (answers)

LAB 1 : Code snippet for "no_implication"

```
property pr1;
  @(posedge clk) req ##2 gnt;
endproperty
reqGnt: assert property (pr1) $display($stime,,,"\t\t  %m PASS");
              else $display($stime,,,"\t\t  %m FAIL");
```

LAB 1 : Q&A on "no_implication"

```
KERNEL:       90  clk=1 req=1 gnt=0
KERNEL:      110  clk=1 req=0 gnt=0
KERNEL:      110            test_implication FAIL
KERNEL:      130  clk=1 req=0 gnt=0
KERNEL:      130            test_implication FAIL
KERNEL:      130            test_implication FAIL

            Q: WHY ARE THERE 2 FAILs AT TIME (130) ??

            A: The first failures is for the thread starting at time 90
              At 90, req==1, so property eval proceeds.
              At 130 (i.e. 2 clocks later) gnt==0 which violates the property and the
              property FAILs.

              The second failure is for the thread starting at time 130.
              At 130, req==0 and since there is no implication, the property fails
              because without an implication there is no antecedent to match before
              the check begins. Whenever at posedge clk, 'req' detected low,
               the property will fail.
```

Fig. 18.3 LAB1: Q&A on ‘implication’ operator (answers)

LAB 1 : Code snippet for "implication"

```
property pr1;
  @(posedge clk) req |-> ##2 gnt;
endproperty

reqGnt: assert property (pr1) $display($stime,,,"\t\t  %m PASS");
              else $display($stime,,,"\t\t  %m FAIL");
```

LAB 1 : Q&A on "implication"

```
run -all
KERNEL:       10  clk=1 req=0 gnt=0
KERNEL:       10            test_implication PASS
KERNEL:       30  clk=1 req=1 gnt=0
KERNEL:       50  clk=1 req=0 gnt=0
KERNEL:       50            test_implication PASS
KERNEL:       70  clk=1 req=0 gnt=1
KERNEL:       70            test_implication PASS
KERNEL:       70            test_implication PASS

              Q: WHY ARE THERE 2 PASSes AT TIME 70 ??

              A: The first pass is for the thread starting at time 30.
                 At 30, req==1, so property eval proceeds.
                 At 70 (i.e. 2 clocks later) gnt==1 as required by the property and the
                 property PASSes.

                 The second pass is for the thread starting at time 70.
                 At 70, req==0 and since there is implication, the consequent eval won't
                 start. However, there is a PASS action_block associated with the
                 property which triggers because of the vacuous pass phenomenon. In
                 other wods, whenever 'req' is low, the antecedent won't match and the
                 property will pass vacuosly.
```

Fig. 18.3 (continued)

LAB 1 : Code snippet for "implication"

```
property pr1;
  @(posedge clk) req |-> ##2 gnt;
endproperty

reqGnt: assert property (pr1) $display($stime,,,"\t\t  %m PASS");
              else $display($stime,,,"\t\t  %m FAIL");
```

LAB 1 : Q&A on "implication"

```
KERNEL:       90  clk=1 req=1 gnt=0
KERNEL:      110  clk=1 req=0 gnt=0
KERNEL:      110            test_implication PASS
KERNEL:      130  clk=1 req=0 gnt=0
KERNEL:      130            test_implication FAIL
KERNEL:      130            test_implication PASS

            Q: WHY IS THERE A PASS -and- a FAIL AT TIME 130 ??

            A: The failure is for the thread starting at time 90.
              At 90, req==1, so property eval proceeds.
              At 130 (i.e. 2 clocks later) gnt==0 which violates the property and the
              property FAILs.

              The pass is for the property stating at 130.
              At 130, req==0 and since there is implication, the consequent eval won't
              start. However, there is a PASS action_block associated with the
              property which triggers because of the vacuous pass phenomenon. In
              other words, whenever 'req' is low, the antecedent won't match and the
              property will pass vacuosly.
```

18.2 LAB2: Answers: Overlap and Non-overlap Operators

See Figs. 18.4 and 18.5.

LAB 2 : Code snippet with "overlap" operator

```
sequence sr1;
  req ##2 gnt;
endsequence

property pr1;
  @(posedge clk) cstart |-> sr1;
endproperty

property pr1_for_cover;
  @(posedge clk) cstart ##0 sr1;
endproperty
```

LAB 2 : Q&A on "overlap" operator

```
run -all
KERNEL:      10  clk=1 cstart=0 req=0 gnt=0
KERNEL:      30  clk=1 cstart=1 req=0 gnt=0
KERNEL:      30            test_overlap_nonoverlap FAIL

          Q: WHY DOES THE PROPERTY FAIL at 30?

            A: At time 30, cstart=1; so antecedent matches and consequent eval starts
              At time 30, req is NOT equal to 1 as required by overlapping implication
              and the consequent fails right away and the property FAILs.

KERNEL:      50  clk=1 cstart=1 req=1 gnt=0
KERNEL:      70  clk=1 cstart=0 req=0 gnt=0
KERNEL:      90  clk=1 cstart=0 req=0 gnt=1
KERNEL:      90            test_overlap_nonoverlap PASS

          Q: WHY DOES THE PROPERTY PASS at 90?

            A: At time 50, cstart=1; so antecedent matches and consequent eval starts
              At time 50 (i.e, the same clock as required by overlapping implication),
              req=1; so consequent eval continues
              At time 70, gnt=1 as required by the property and the consequent
              matches and the property PASSes.
```

Fig. 18.4 LAB1: Q&A on 'overlap' operator (answers)

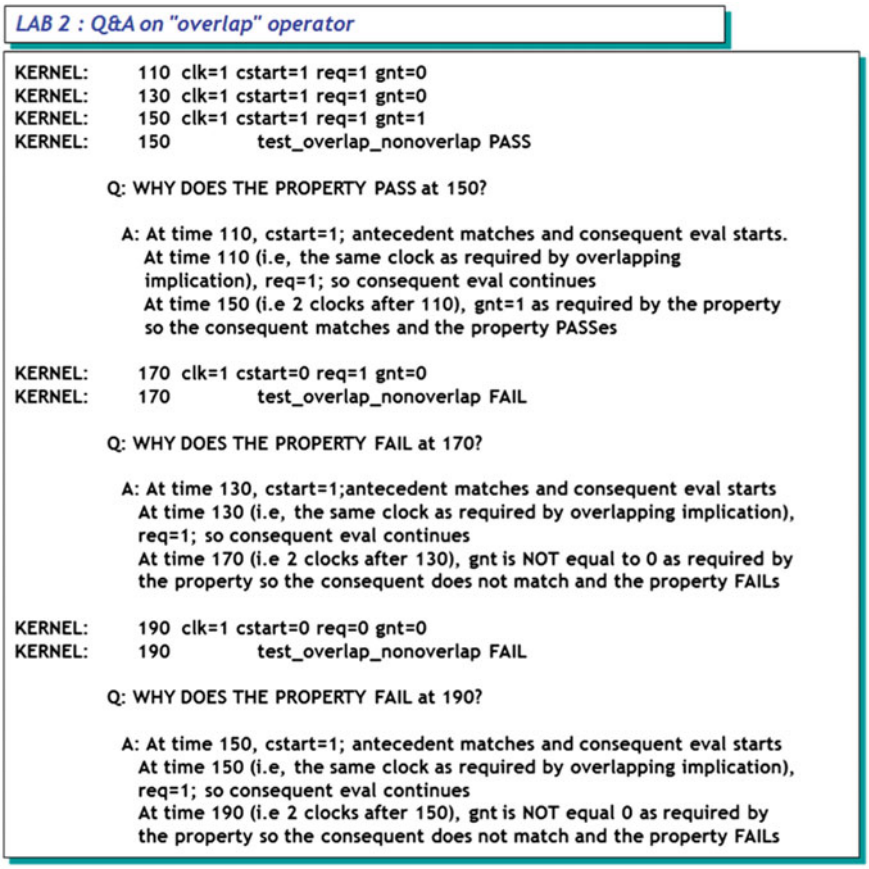


Fig. 18.5 LAB1: Q&A on 'non-overlap' operator (answers)

LAB 2 : Code snippet with "non-overlap" operator

```
sequence sr1;
  req ##2 gnt;
endsequence

property pr1;
  @(posedge clk) cstart |=> sr1;
endproperty

property pr1_for_cover;
  @(posedge clk) cstart ##1 sr1;
endproperty
```

LAB 2 : Q&A on "non-overlap" operator

```
KERNEL:      10  clk=1 cstart=0 req=0 gnt=0
KERNEL:      30  clk=1 cstart=1 req=0 gnt=0
KERNEL:      50  clk=1 cstart=1 req=1 gnt=0
KERNEL:      70  clk=1 cstart=0 req=0 gnt=0
KERNEL:      70            test_overlap_nonoverlap FAIL

          Q: WHY DOES THE PROPERTY FAIL at 70?

            A: This failure is for the thread that started at time 50 (and not 30).
              At time 50, cstart=1; so antecedent matches and consequent eval starts
              At time 70 (i.e, one clock later as required by nonoverlapping
              implication), req is NOT EQUAL to 1; so consequent does not match and
              the property FAILs

KERNEL:      90  clk=1 cstart=0 req=0 gnt=1
KERNEL:      90            test_overlap_nonoverlap PASS

          Q: WHY DOES THE PROPERTY PASS at 90?

            A: This pass is for the thread that started at time 30 (and not 50).
              At time 30, cstart=1; so antecedent matches and consequent eval starts
              At time 50 (i.e, one clock later as required by nonoverlapping
              implication), req == 1; so consequent eval continues
              At time 90 (i.e, two clocks later as required by the property),
              gnt == 1; so consequent matches and the property PASSes.
```

Fig. 18.5 (continued)

LAB 2 : Q&A on "non-overlap" operator

```
KERNEL:     110  clk=1 cstart=1 req=1 gnt=0
KERNEL:     130  clk=1 cstart=1 req=1 gnt=0
KERNEL:     150  clk=1 cstart=1 req=1 gnt=1
KERNEL:     170  clk=1 cstart=0 req=1 gnt=0
KERNEL:     170           test_overlap_nonoverlap FAIL

          Q: WHY DOES THE PROPERTY FAIL at 170?

            A: This failure is for the thread that started at time 110
              At time 110, cstart=1; antecedent matches and consequent eval starts
              At time 130 (i.e, one clock later as required by nonoverlapping
              implication), req is EQUAL to 1; so consequent eval continues.
              At time 170 (i.e, two clocks later as required by the property),
              gnt is NOT EQUAL to 1; so consequent does not match and
              the property FAILs.

KERNEL:     190  clk=1 cstart=0 req=0 gnt=0
KERNEL:     190           test_overlap_nonoverlap FAIL

          Q: WHY DOES THE PROPERTY FAIL at 190?

            A: This failure is for the thread that started at time 130
              At time 110, cstart=1; so antecedent matches and consequent eval starts
              At time 150 (i.e, one clock later as required by nonoverlapping
              implication), req is EQUAL to 1; so consequent eval continues.
              At time 190 (i.e, two clocks later as required by the property),
              gnt is NOT EQUAL to 1; so consequent does NOT match and
              the property FAILs.

KERNEL:     210  clk=1 cstart=0 req=0 gnt=1
KERNEL:     210           test_overlap_nonoverlap PASS

          Q: WHY DOES THE PROPERTY PASS at 210?

            A: This pass is for the thread that started at time 150
              At time 150, cstart=1; antecedent matches and consequent eval starts
              At time 170 (i.e, one clock later as required by nonoverlapping
              implication), req is EQUAL to 1; so consequent eval continues.
              At time 210 (i.e, two clocks later as required by the property),
              gnt is EQUAL to 1; so consequent matches and the property PASSes.
```

18.3 LAB3: Answers: Synchronous FIFO

See Fig. 18.6.

LAB 3 : fifo_property.sv

```
//     ------------------------------------------
//       1. Check that on reset,
//                rd_ptr=0; wr_ptr=0; cnt=0; fifo_empty=1 and fifo_full=0
//     ------------------------------------------
`ifdef check1
property check_reset;
  @(posedge clk)
    (!rst_ |-> (`rd_ptr==0 && `wr_ptr==0 && fifo_empty==1 && fifo_full==0));
endproperty
check_resetP: assert property (check_reset) else $display($stime,"\t\t
FAIL::check_reset\n");
`endif

//     ------------------------------------------
//       2. Check that fifo_empty is asserted the same clock that fifo 'cnt' is 0.
//          Disable this property 'iff (!rst)'
//     ------------------------------------------
`ifdef check2
property fifoempty;
  @(posedge clk) disable iff (!rst_)
    (`cnt==0 |-> fifo_empty);
endproperty
fifoemptyP: assert property (fifoempty) else $display($stime,"\t\t
FAIL::fifo_empty condition\n");
`endif

//     ------------------------------------------
//       3. Check that fifo_full is asserted any time fifo 'cnt' is greater than 7.
//          Disable this property 'iff (!rst)'
//     ------------------------------------------
`ifdef check3
property fifofull;
  @(posedge clk) disable iff (!rst_)
    (`cnt>(fifo_depth-1)  |-> fifo_full);
endproperty
fifofullP: assert property (fifofull) else $display($stime,"\t\t FAIL::fifo_full
condition\n");
`endif
```

Fig. 18.6 LAB3: FIFO: answers

LAB 3 : fifo_property.sv

```
//     ------------------------------------------
//       4. Check that if fifo is full and you attempt to write (but not read) that
//          the wr_ptr does not change.
//     ------------------------------------------
`ifdef check4
property fifo_full_write_stable_wrptr;
  @(posedge clk) disable iff (!rst_)
    (fifo_full && fifo_write && !fifo_read |=> $stable(`wr_ptr));
endproperty
fifo_full_write_stable_wrptrP: assert property (fifo_full_write_stable_wrptr)
     else $display($stime,"\t\t FAIL::fifo_full_write_stable_wrptr condition\n");
`endif

`ifdef check5
//     ------------------------------------------
//       5. Check that if fifo is empty and you attempt to read (but not write) that
//          the rd_ptr does not change.
//     ------------------------------------------
property fifo_empty_read_stable_rdptr;
  @(posedge clk) disable iff (!rst_)
    (fifo_empty && fifo_read && !fifo_write |=> $stable(`rd_ptr));
endproperty
fifo_empty_read_stable_rdptrP: assert property (fifo_empty_read_stable_rdptr)
               else $display($stime,"\t\t FAIL::fifo_empty_read_stable_rdptr
condition\n");
`endif

//     ------------------------------------------
//       6. Write a property to Warn on write to a full fifo
//             This property will give Warning with all simulations
//     ------------------------------------------
`ifdef check6
property write_on_full_fifo;
  @(posedge clk) disable iff (!rst_)
    fifo_full |-> !fifo_write;
endproperty
write_on_full_fifoP: assert property (write_on_full_fifo)
               else $display($stime,"\t\t WARNING::write_on_full_fifo\n");
`endif
```

LAB 3 : fifo_property.sv

```
//     ------------------------------------------
//       7. Write a property to Warn on read from an empty fifo
//             This property will give Warning with all simulations
//     ------------------------------------------
`ifdef check7
property read_on_empty_fifo;
  @(posedge clk) disable iff (!rst_)
    fifo_empty |-> !fifo_read;
endproperty
read_on_empty_fifoP: assert property (read_on_empty_fifo)
          else $display($stime,"\t\t  WARNING::read_on_empty_fifo  condition\n");
`endif
```

18.4 LAB4: Answers: Counter

See Fig. 18.7.

LAB 4 : counter_property.sv

```
//-------------------------------------
//       CHECK # 1. Check that when 'rst_' is asserted (==0) that data_out == 8'b0
//-------------------------------------
`ifdef check1
property counter_reset;
  @(clk) disable iff (rst_) !rst_ |=> (data_out == 8'b0);
endproperty

counter_reset_check: assert property(counter_reset)
  else $display($stime,,,"\t\tCOUNTER  RESET CHECK FAIL:: rst_=%b data_out=%0d \n",
                                                    rst_,data_out);
`endif

//-------------------------------------
//       CHECK # 2. Check that if ld_cnt_ is deasserted (==1) and count_enb is not enabled
//                (==0) that data_out HOLDS it's previous value.
//                Disable this property 'iff (!rst)'
//-------------------------------------
`ifdef check2
property counter_hold;
  @(posedge clk) disable iff (!rst_) (ld_cnt_ & !count_enb) |=> data_out ===
$past(data_out);
endproperty

counter_hold_check: assert property(counter_hold)
  else $display($stime,,,"\t\tCOUNTER  HOLD CHECK FAIL:: counter HOLD \n");
`endif
```

Fig. 18.7 LAB4: counter: answers

LAB 4 : counter_property.sv

```
//-------------------------------------
// CHECK # 3. Check that if ld_cnt_ is deasserted (==1) and count_enb is
// enabled (==1) that if updn_cnt==1 the count goes UP and if updn_cnt==0 the
// count goes DOWN.
//-------------------------------------

`ifdef check3
property counter_count;
  @(posedge clk) disable iff (!rst_) (ld_cnt_ & count_enb) |->
    if (updn_cnt) ##1 (data_out-8'h01) == $past(data_out)
    else          ##1 (data_out+8'h01) == $past(data_out);
endproperty

counter_count_check: assert property(counter_count)
  else $display($stime,,,"\t\tCOUNTER COUNT CHECK FAIL:: UPDOWN COUNT using
$past \n");
`endif

//-------------------------------------
// Alternate way of writing assertion for CHECK # 3
// Check for count using local variable
//-------------------------------------
/*
`ifdef check3
property counter_count_local;
logic[7:0] local_data;
  @(posedge clk) disable iff (!rst_) (ld_cnt_ & count_enb, local_data = data_out)
|->
    if (updn_cnt) ##1 (data_out == (local_data+8'h01))
    else          ##1 (data_out == (local_data-8'h01));
endproperty

counter_count_check: assert property(counter_count)
  else $display($stime,,,"\t\tCOUNTER COUNT CHECK FAIL:: UPDOWN COUNT using
$past \n");

`endif
*/
```

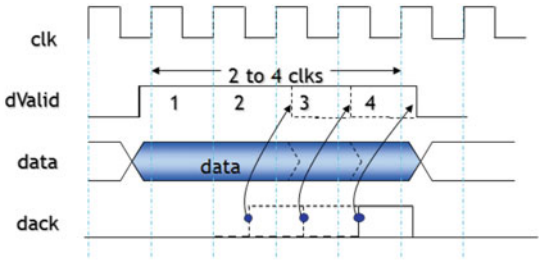
18.5 LAB5: Answers: Data Transfer Protocol

See Fig. 18.8.

LAB Overview

Specification for a simple data transfer protocol.

- dValid must remain asserted for minimum of 2 clocks but no more than 4 clocks.
- 'data' must be known when 'dValid' is High.
- 'dack' going high signifies that target have accepted data and that master must de-assert 'dValid' the clock after 'dack' goes high.
  - *Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and that it must not remain low for more than 3 clocks (because data must trasnfer in max 4 clocks).*



LAB 5 : bus_protocol_property.sv

```
/*-------------------------------------------------
      CHECK # 1. Check that once dValid goes high that it is consecutively
        asserted (high) for minimum 2 and maximum 4 clocks.
              Check also that once dValid is asserted (high) for 2 to 4 clocks that
              it does de-assert (low) the very next clock.
        -------------------------------------------------*/
`ifdef check1
      property checkValid;
       @(posedge clk) disable iff (reset) $rose(dValid) |=> (dValid) [*2:4] ##1 $fell(dValid);
      endproperty
      assert property (checkValid) else $display($stime,,,"checkValid  FAIL");
`endif

      /*-------------------------------------------------
        CHECK # 2. Check that data is not unknow and remains stable after dValid goes
                   high and until dAck goes high.
        -------------------------------------------------*/
`ifdef check2
      property checkdataValid;
       @(posedge clk) disable iff (reset)
          $rose(dValid) |=> (!$isunknown(data) && $stable(data)) [*1:$] ##0 $rose(dAck);
endproperty
      assert property (checkdataValid) else $display($stime,,,"checkdataValid  FAIL");
`endif
```

Fig. 18.8 LAB5: data transfer bus protocol: answers

LAB 5 : bus_protocol_property.sv

```
/*-----------------------------------------------------
      CHECK # 3. Check that 'dAck' and 'dValid' relationship is maintained to complete the
data transfer. In other words,

     'dack' going high signifies that target have accepted data and that master must de-
assert 'dValid' the clock after 'dack' goes high.

Note that since data must be valid for minimum 2 cycles, that 'dack' cannot go High
for at least 1 clock after the transfer starts (i.e. after the rising edge of 'dValid') and
that it must not remain low for more than 3 clocks (because data must trasnfer in max 4
clocks).
      -----------------------------------------------------*/
`ifdef check3
            property checkdAck;
             @(posedge clk) disable iff (reset)
               $rose(dValid) |=> (dValid && !dAck) [*1:3] ##1 $rose (dAck) ##1 $fell (dValid);
            endproperty
            assert property (checkdAck) else $display($stime,,,"checkdAck  FAIL");
`endif
```

18.6 LAB6: Answers: PCI Read Protocol

See Fig. 18.9.

LAB 6 : pci_protocol_property.sv

```
/*-----------------------------------------------------
      CHECK # 1. On falling edge of FRAME_, AD or C_BE_ cannot be unknown.
       -----------------------------------------------------*/
`ifdef check1
      property checkPCI_AD_CBE;
        @(posedge clk) disable iff (!reset_) $fell(FRAME_) |->
                                  !($isunknown(AD) || $isunknown(C_BE_)) ;
      endproperty
      assert property (checkPCI_AD_CBE) else
$display($stime,,,"CHECK1:checkPCI_AD_CBE  FAIL\n");
`endif

      /*-----------------------------------------------------
      CHECK # 2. When IRDY_ and TRDY_ are asserted (low) AD or C_BE_ cannot be
unknown.
       -----------------------------------------------------*/
`ifdef check2
      property checkPCI_DataPhase;
        @(posedge clk) disable iff (!reset_) (!IRDY_ && !TRDY_) |->
                                  !($isunknown(AD) || $isunknown(C_BE_)) ;
      endproperty
      assert property (checkPCI_DataPhase) else
$display($stime,,,"CHECK2:checkPCI_DataPhase  FAIL\n");
`endif

      /*-----------------------------------------------------
      CHECK # 3. FRAME_ can go High only if IRDY_ is asserted.
                In other words, master can signify end of cycle only if IRDY_ is
asserted.
       -----------------------------------------------------*/
`ifdef check3
      property checkPCI_Frame_Irdy;
        @(posedge clk) disable iff (!reset_) $rose(FRAME_) |-> !IRDY_;
      endproperty
      assert property (checkPCI_Frame_Irdy) else
$display($stime,,,"CHECK3:checkPCI_frmIrdy  FAIL\n");
`endif
```

Fig. 18.9 LAB6: PCI protocol: answers

LAB 6 : pci_protocol_property.sv

```
  /*-----------------------------------------------------
      CHECK # 4. TRDY_ can be asserted (low) only if DEVSEL_ is asserted (low)
       -----------------------------------------------------*/
`ifdef check4
      property checkPCI_trdyDevsel;
        @(posedge clk) disable iff (!reset_) !TRDY_ |-> !DEVSEL_;
      endproperty
      assert property (checkPCI_trdyDevsel) else
$display($stime,,,"CHECK4:checkPCI_trdyDevsel  FAIL\n");
`endif

      /*-----------------------------------------------------
      CHECK # 5. Once the cycle starts (i.e. at FRAME_ assertion)
                    C_BE_ should not float until FRAME_ is de-asserted
       -----------------------------------------------------*/
`ifdef check5
      property checkPCI_CBE_during_trx;
        @(posedge clk) disable iff (!reset_)
                $fell(FRAME_) |-> !($isunknown(C_BE_)) [*0:$] ##0 $rose(FRAME_);
      endproperty
      assert property (checkPCI_CBE_during_trx) else
$display($stime,,,"CHECK5:checkPCI_CBE_during_trx  FAIL\n");
`endif
```

Chapter 19
Functional Coverage

Ah, so you have done everything to check the design. But what have you done to check your testbench? How do you know that your testbench has indeed covered everything that needs to be covered? That's where Functional Coverage comes into picture. But first let us make sure we understand difference between the good old Code Coverage and the new Functional Coverage methodology.

19.1 Difference Between Code Coverage and Functional Coverage

- Code Coverage
  - Derived directly from the design code; not user specified.
  - Evaluates to see if design *structure* has been covered (i.e., assign, branch, expression, state transition, etc.)
  - But does not evaluate the *intent* of the design

    If the user specified busGnt = busReq && (idle || !(reset));
    instead of the real *intent* busGnt = busReq && (idle **&&** !(reset));
    Code coverage won't catch it. For intent, you need both a robust testbench to weed out functional bugs and a way to objectively predict how robust the testbench is.
- Functional Coverage
  - User specified
  - Based on design specification (as we have already seen with 'cover' of an assertion).
  - Measures coverage of design *intent*
  - Control-oriented coverage

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_19

Have I exercised all possible protocols that read cycle supports (burst, non-burst, etc.)?
Transition coverage

- Did we issue transactions that access Byte followed by Qword followed by multiple Qwords. (use SystemVerilog *transition* coverage).
- A Write to L2 is followed by a Read from the same address (and vice-versa). Again, the *transition* coverage will help you determine if you have exercised this condition.

Cross coverage

- Tag and Data Errors must be injected at the same time (use SystemVerilog *cross* coverage).

– Data-oriented coverage

Have we accessed cache lines at all granularity levels (odd bytes, even bytes, word, quad-word, full cache line, etc.)?

19.2 Assertion Based Verification (ABV) and Functional Coverage (FC) Based Methodology

First let us examine the components of SystemVerilog language that contribute to Functional Coverage.

First component is the 'cover' statement associated with an assertion. This 'cover' statement allows us to measure temporal domain functional coverage. Recall that 'assert' checks for failures in your design and 'cover' sees if the property did get exercised (i.e. got covered). Pure combinatorial coverage is not sufficient. What I call 'low level' temporal domain conditions such as every req should be followed by a gnt. If this assertion does not fail, it could be because the logic is correct or *because you never really asserted 'req' to start with*. 'cover' completes this story. We 'cover' exactly the same property that we 'assert'. In the req/gnt example, if 'cover' passes we know that the property did get exercised by the testbench and it did not fail (if 'assert' did not fail).

Second component is the Functional Coverage language which is the gist of this entire section. Functional coverage allows you to specify the 'function' you want to cover via the so-called *coverpoints* and *covergroups*. More importantly, it also allows you to measure *transition* as well as *cross* coverage to see that we have indeed covered finer details of our design. This section will clarify all this.

Figure 19.1 clearly shows the different components of SystemVerilog as well as code coverage that all ties together to determine if a design have indeed been completely verified.

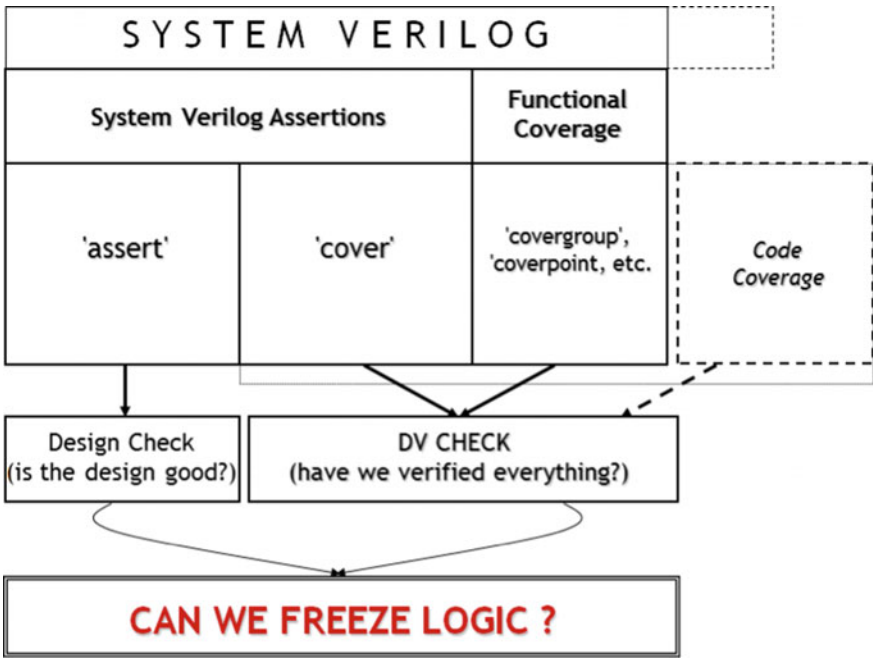


Fig. 19.1 Assertion based verification (ABV) and functional coverage (FC) based methodology

Here are some more points from project methodology point of view.

- Your test plan is (obviously) based on what functions you want to test (i.e. cover).
- So, create a Functional Cover Matrix based on your test plan that includes each of the functions (control and data) that you want to test.
  - Identify in this matrix all your functional covergroups/coverpoints (more on that coming soon)
  - Measure their coverage during verification/simulation process
  - You may even automate updating the matrix directly from the coverage reports. That methodology is depicted in Fig. 19.2.
- Measure effectiveness of your tests from the coverage reports. To reiterate what we just discussed above since the following points are indeed the gist of what functional coverage allows you to accomplish.
  - For example, if your tests are accessing mostly 32 byte granules in your cache line, you will see byte, word, quadword coverage low or not covered. Change or add new tests to hit bytes/words, etc. Use constrained random methodology to narrow down the target of your tests. Constrained random is a very powerful methodology and goes hand in hand with Functional Coverage. Constrained random is beyond the scope of this book.

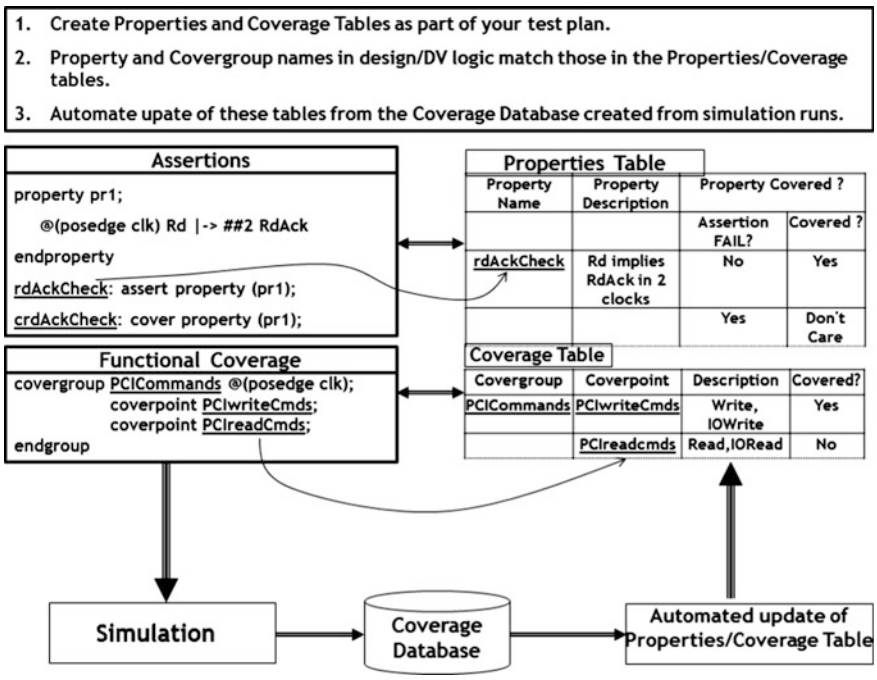


Fig. 19.2 Assertions and coverage closed loop verification methodology—I

  - Or that the tests do not fire transactions that access Byte followed by Qword followed by multiple Qwords. Check this with *transition* coverage.
  - Or that Tag and Data Errors must be injected at the same time (*cross* coverage between Tag and Data Errors)

- 'cover' temporal domain assertions.
- And add more *coverpoints* for critical functional paths through design.

  - For example, a Write to L2 is followed by a Read from the same address and that this happens from both processors in all possible write/read combinations.

- Remember to update your Functional Cover plan as verification progresses.

  - Just because you created a plan in the beginning of the project does not mean it's an end in itself.
  - As your knowledge of the design and its corner cases increase, so should the exhaustiveness of your test plan and the functional cover plan.
  - Continue to add *coverpoints* for any function that you didn't think of at the onset.

Figures 19.2 and 19.3 show an assertion and coverage driven methodology.

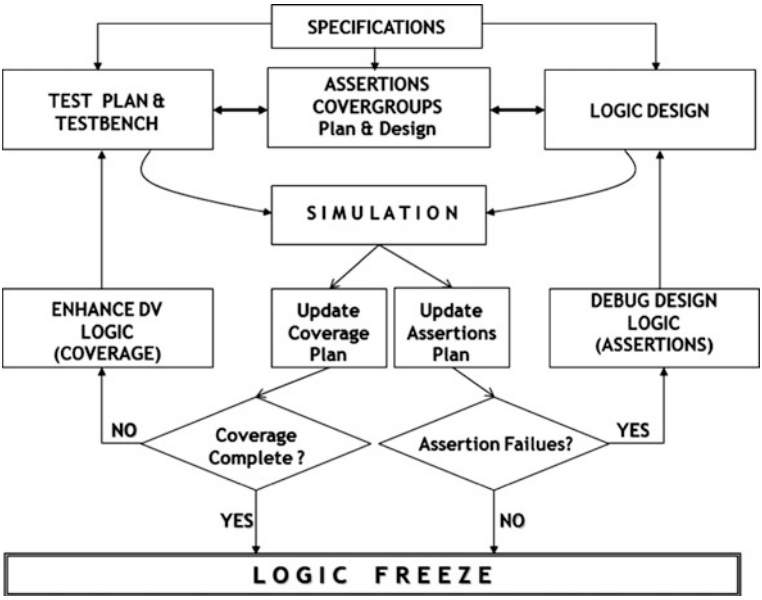


Fig. 19.3 Assertion and functional Coverage closed loop verification methodology—II

1. For every 'assert' in a property, have an associated 'cover'. Give meaningful names to the property and assert Labels.
2. Create a Properties Table which automatically reads in your assertions and creates a FAIL/Covered matrix. If the assertion FAILs, well, fill in the FAIL column. If not and if gets covered, fill in the Covered column. How do we fill in this matrix? Read on …
3. Create a Functional Coverage plan with *covergroup* and *coverpoint*. Again give meaningful names to covergroup and coverpoint(s).
4. Create a Coverage Table that automatically derives the covergroup/coverpoint names from step 3 and creates a matrix for "Covered" results. This matrix is for those functions that are not covered by assertion 'cover' nor are they covered by code coverage. So, you need to carefully design your covergroups and coverpoints.
5. Simulate your design with assertions and functional cover groups.
6. Simulation will create a "coverage database". This database has all the information about failed assertions and 'cover'ed properties and covered covergroups and coverpoints
7. Using EDA vendor provided API, shift through this database and update the Properties Table and Coverage Table
8. Loop back.

Advantage of such methodology is that you continually know if you are spinning the wheel without increasing coverage. Without such continual measure you may keep simulating; bugs don't get reported; you start feeling comfortable only to realize later that the functional coverage was really inadequate. You were basically running the tests that target the same logic over and over again. If you have a methodology as described above, you will have a correct notion of what functional logic to target to increase bug rate.

19.2.1 Follow the Bugs!!

- So, when do you *start* collecting coverage?
  - Code and Functional Coverage add to simulation overhead.
  - So, don't turn on code/functional coverage at the very 'beginning' of the project.
  - But what does 'beginning' of the project mean? When does the 'beginning' end?
- That's where the bugs come into picture!
  - Create Bug Report charts
  - During the 'beginning' time, bug rate will (should) be high. All low hanging fruits are being picked ☺
  - When the Bug Rate starts to drop; the 'beginning' has come to an 'end'
  - That's when your existing test strategy is running out of steam ☹
  - That's when you start code and functional coverage to determine

    If new tests are simply exercising the same logic repeatedly
    And which part of logic is not yet covered

  - Develop tests for the uncovered functionality. Use constrained random methodology.
  - Your Bug Rate will again go up (guaranteed! ☺).

Chapter 20
Functional Coverage—Language Features

Introduction: This chapter covers the entire language "Functional Coverage". We will cover the following features in the upcoming sections.

1. covergroups and coverpoints for variables and expressions
2. automatic as well as user-defined coverage bins
3. 'bins' for transition coverage,
4. 'wildcard bins', 'illegal_bins', 'ignore_bins'
5. Cross Coverage
6. Coverage Options
7. Flexible coverage sample—events, sequences, procedural
8. Directives to control and query coverage
9. Application: Coverage Methods and procedural activation of coverage methods.

20.1 Covergroup/Coverpoint

What is a covergroup?

I am taking the definition directly from the LRM since it is indeed well worded.

- 'covergroup' is a user defined type that allows you to collectively sample all those variables/transitions/cross that are sampled at the same clock (sampling) edge.
- "The 'covergroup' construct encapsulates the specification of a coverage model."
- A 'covergroup' can be defined in a 'package', 'module', a 'program', an 'interface' or a 'class'.

What Is a Coverpoint?

- A coverpoint is a variable or an expression that functionally covers design parameters (reg, logic, enum, etc.)

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_20

- Each coverpoint includes a set of bins associated with its sampled value or its value transition.
- The so-called 'bins' can be defined by the user or created automatically by an EDA tool. A bin tells you the actual coverage measure.

OK, that's all fundamentals from the LRM. [The LRM authors did do a good job in writing some of this in English that hardware types can understand (like myself) …;)].

Figure 20.1 makes it plenty simpler to explain covergroup and coverpoint.

20.2 System Verilog 'Covergroup'—Basics …

Figure 20.1 is self-explanatory with its annotations. Key syntax of the covergroup and coverpoint is pointed out. A few points to reiterate are as follows.

1. *covergroup* without a coverpoint is useless *and* the compiler won't give an Error (at least the simulators that the author has tried)
2. *covergroup*, as the name suggests, is a group of coverpoints, meaning you can have multiple coverpoints in a covergroup.
3. You have to instantiate the covergroup.
4. Actual arguments are evaluated when the new operator is executed
5. You may provide (not mandatory) a sampling edge to determine when the *coverpoints* in a *covergroup* get sampled. If the clocking event is omitted, you must procedurally trigger the coverage sample window using a built-in method called sample(). We will discuss sample() later in the chapter.
6. A '*covergroup'* can be declared in
   a. package
   b. interface
   c. module
   d. program
   e. class (we'll see an example soon).

Other points are annotated in Fig. 20.1. Carefully study them so that the rest of the chapter is easier to digest.

20.3 SystemVerilog Coverpoint Basics

Syntax:
cover_point ::=
[[data_type_or_implicit] cover_point_identifier:] coverpoint expression [iff (expression)] bins_or_empty

Coverpoint and bins associated with the coverpoint do all the work. The syntax for coverpoint is as shown in Fig. 20.2. 'covergroup g1' is sampled at (posedge clk).

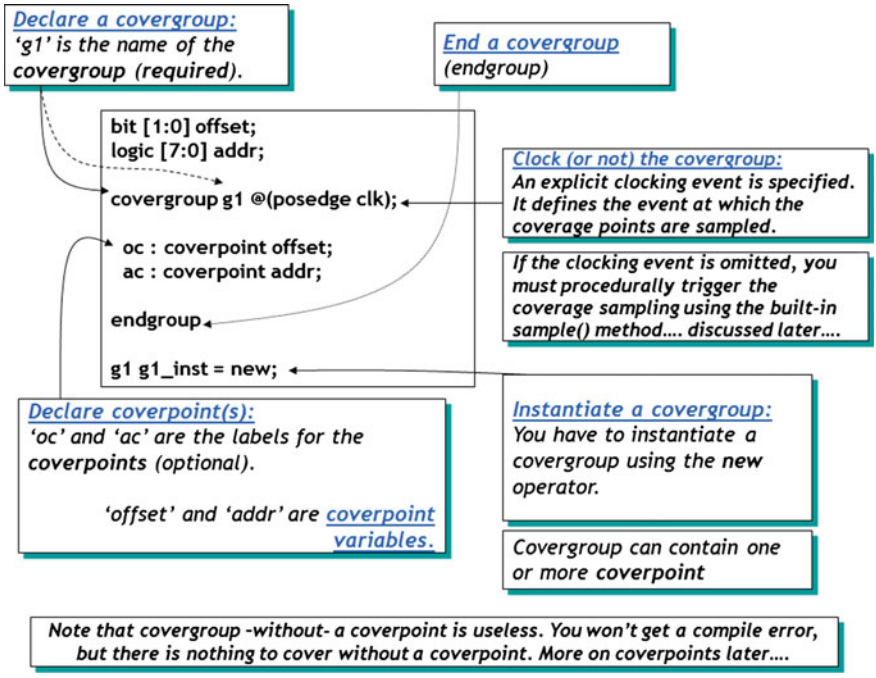


Fig. 20.1 'covergroup' and 'coverpoint'—basics

'oc' is the coverpoint name (or label). This is the name by which simulation log refers to this coverpoint. 'oc' covers the 2-bit variable 'offset'.

We haven't yet covered 'bins', so please hang on with the following description for a while. We will cover plenty of 'bins' in the upcoming sections. So, in this example, you do not see any 'bins' associated with the coverpoint 'oc' for variable 'offset'. Since there are no bins to hold coverage results, simulator will create those for you. In this example, the simulator will create 4 bins because 'offset' is a 2 bit variable. If 'offset' were a 3 bit vector, there would be 8 bins and so on. We will discuss a lot more on 'bins' in upcoming sections and hence I am showing only the so-called auto bins created by the simulator.

I haven't shown the entire testbench but the simulation log (at the bottom of Fig. 20.2) shows that there are 4 auto generated bins called 'bin auto[0]' … 'bin auto[3]'. Each of these bins covers 1 value of 'offset'. For example, auto[0] bin covers 'offset == 0'. In other words, if 'offset == 0' has been simulated, then auto[0] will be considered covered. Again, this will become clearer when we go through basics of 'bins'. Since all 4 bins of coverpoint 'oc' have been covered, the coverpoint 'oc' is considered 100 % covered, as shown in the simulation log. Now let us look at a real life example of covergroup/coverpoint.

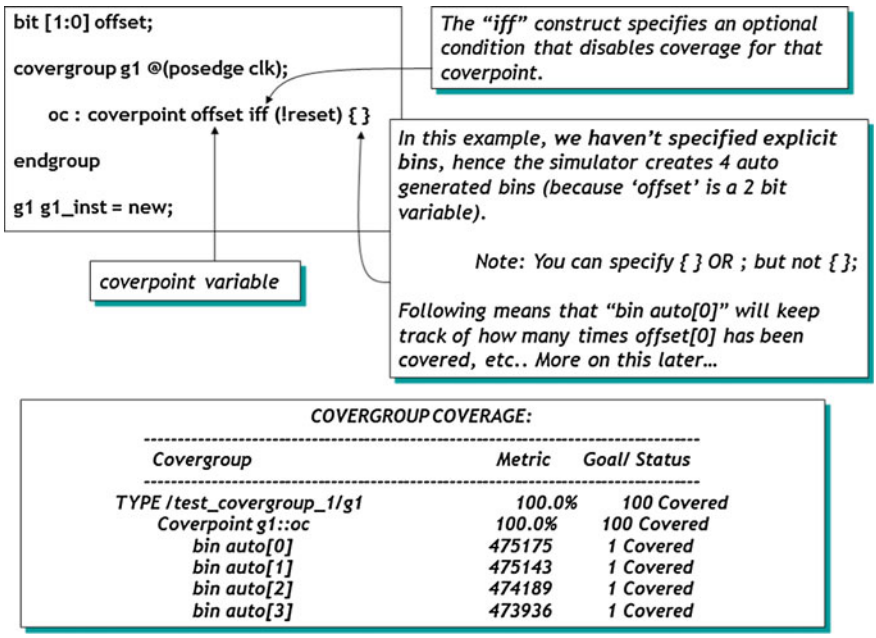


Fig. 20.2 'coverpoint'—basics

A data type for the coverpoint may also be specified explicitly or implicitly in *data_type_or_implicit*. In either case, it is understood that a data type is specified for the coverpoint. The data type must be an integral type. If a data type is specified, then a *cover_point_identifier* must also be specified.

If a data type is specified, then the coverpoint expression must be assignment compatible with the data type.

Values for the coverpoint will be of the specified data type and will be determined as though the coverpoint expression were assigned to a variable of the specified data type.

If no data type is specified, then the inferred data type for the coverpoint will be the self-determined type of the coverpoint expression.

The expression within the *iff* construct specifies an optional condition that disables coverage for that coverpoint.

If the guard expression evaluates to false at a sampling point, the coverage point is ignored.

```
covergroup AR;
        coverpoint s0 iff(!reset);
endgroup
```

In the preceding example, coverage point s0 is covered only if the value of 'reset' is low.

20.3.1 Covergroup/Coverpoint Example …

PCI protocol consists of many different types of bus cycles. We want to make sure that we have covered each type of cycle. The enum type pciCommands (Fig. 20.3) describes the cycle types. The covergroup specifies the *correct* sampling edge (that being the 'negedge FRAME_') for the PCI Commands. In other words, the sampling edge is quite important for performance reasons. If you sampled the same covergroup @ (posedge clk), there will be a lot of overhead because FRAME_ will fall only when a PCI cycle is to start. Sampling unnecessarily will indeed affect simulation performance.

In this example we have not specified any bins. So the simulator creates 12 auto bins for the 12 bus cycles types in the enum. Every time FRAME_ falls that the simulator will see if any of the cycle types in 'enum pciCommands' is simulated. Each of the 12 auto bin corresponds to each of the enum type. When a cycle type is exercised, the auto bin corresponding to that cycle (i.e. that 'enum') will be considered covered. Now, you may ask why wouldn't code coverage cover this. Code coverage will indeed do the job. But I am building a small story around this example. We'll see how this covergroup will be then reused for transition coverage which *cannot* be covered by code coverage. Ok. now on to 'bins'.

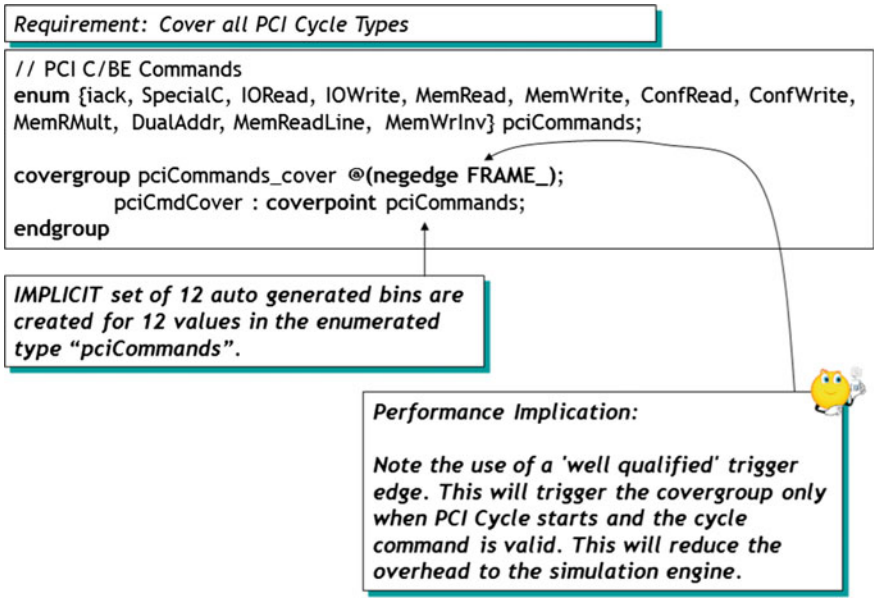


Fig. 20.3 'covergroup'/'coverpoint' example

20.4 System Verilog 'Bins'—Basics …

What's a 'bin'? A 'bin' is something that collects coverage information (collect in a 'bin'). bins are created for coverpoints. A coverpoint that is covering a variable (let's say the 8 bit 'adr' as shown in Fig. 20.4) and would like to have different values of that variable be collected in different collecting entities, the 'bins' will be those entities. 'bins' allows you to organize the coverpoints sample (or transition) values.

You can declare bins many different ways for a coverpoint. Recall that bins collect coverage. From that point of view, you have to carefully choose the declaration of your bins.

OK, here's the most important point *very* easy to misunderstand. In the following statement, how many bins will be created? 16 or 4 or 1 and what will it cover?

bins adrbin1 = {[0:3]};
Answer: 1 bin will be created to cover 'adr' values equal to '0' or '1' or '2' or '3'.

Note that 'bins adrbin1' is without the [] brackets. In other words, 'bins adrbin1' will *not* auto-create 4 bins for 'adr' values {[0:3]}, it will rather create only 1 bin to cover 'adr' values '0','1', '2', '3'.

Very important point: Do not confuse {[0:3]} *to mean that you are asking the bin to collect coverage for adr0 to adr15.* {[0:3]} *literally means 'adr' value =0, =1, =2, =3.*

Another important point. What 'bins adrbin1 = {[0:3]};' also says is that if we hit *either* of the 'adr' value ('0','1','2' or '3') that the single bin will be considered *completely* covered. Not very intuitive, I agree. But that's what the language semantics dictate. Again, you don't have to cover all four values to have "bins adrbin1" considered covered. You hit any one of those 4 values and the "adrbin1" will be considered 100 % covered.

But what if you want each value of the variable 'adr' be collected in separate bins so that you can indeed see if each value of 'adr' is covered explicitly. That's where 'bins adrbin2[] = {[4:5]};' comes into picture. Here '[]' tells the simulator to create two explicit bins called adrbin2[1] and adrbin2[2] each covering the 2 'adr' values = 4 and =5. adrbin2[1] will be considered covered if you exercised adr == 4 and adrbin2[2] will be considered covered if adr == 5 is exercised.

Other ways of creating bins are described in Fig. 20.4 with annotation to describe the nuances. Note that you can have 'less' or 'more' # of bins than the 'adr' values on the RHS of a bins assignment. How will 'bins' be allocated in such cases is explained in the figure. Note also the case {[31:$]} called 'bins heretoend'. What does '$' mean in this case? It means [32:255] since 'adr' is an eight bit variable.

Rest of the semantics is well described with annotation in the figure. Do study them carefully, since they will be very helpful when you start designing your strategy to create 'bins'.

Here's an example of how you may end up making mistakes and get Warnings from a simulator (courtesy LRM).

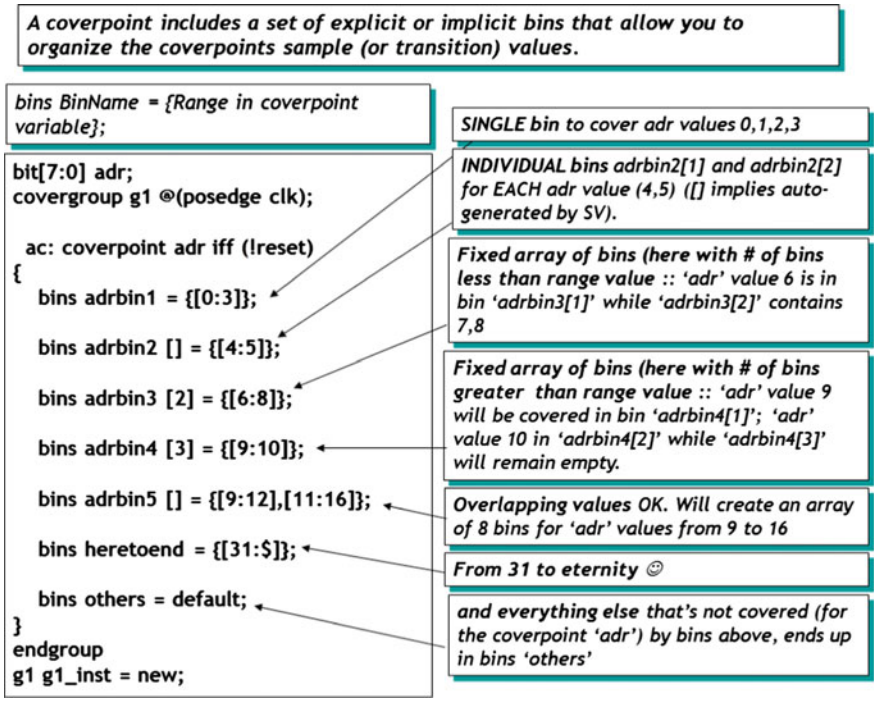


Fig. 20.4 'bins'—basics

```
bit [2:0] p1; //values 0 to 7
bit signed [2:0] p2; //values– 4 to 3
covergroup g1 @(posedge clk);
        coverpoint p1 {
                bins b1 = { 1, [2:5], [6:10] };
                bins b2 = { -1, [1:10], 15 };
        }
        coverpoint p2 {
                bins b3 = {1, [2:5], [6:10] };
                bins b4 = { -1, [1:10], 15 };
        }
endgroup
```

Exercise: See if you can figure out why these Warnings are issued? Note that p2 is 'signed'.

- For b1, a warning is issued for the range [6:10]. b1 is treated as though it had the specification {1, [2:5], [6:7]}.
- For b2, a warning is issued for the range [1:10] and for the values −1 and 15. b2 is treated as though it had the specification {[1:7]}.
- For b3, a warning is issued for the ranges [2:5] and [6:10]. b3 is treated as though it had the specification {1, [2:3]}.

– For b4, a warning is issued for the range [1:10] and for the value 15. b2 is treated as though it had the specification {−1, [1:3]}.

Following shows how the 'with' clause comes in handy to further restrict the creation of bins.

The 'with' clause specifies that only those values that satisfy the given expression are included in the bin. In the expression, the name '*item*' is used to represent the candidate value. The candidate value is of the same type as the coverpoint.
Consider the following example:

```
a: coverpoint x
        {
                bins mod3[] = {[0:255]} with (item % 3 == 0);
        }
```

This bin definition selects all values from 0 to 255 that are evenly divisible by 3.

The 'with' clause is actually a SystemVerilog construct (e.g. its usage in constraint random verification). Please refer to SystemVerilog LRM to understand its further nuances.

20.4.1 Covergroup/Coverpoint with Bins—Example …

Recall the example on PCI that we started in previous section. Its story continues here.

In Fig. 20.5 we are assigning different groups of PCI commands to different bins.

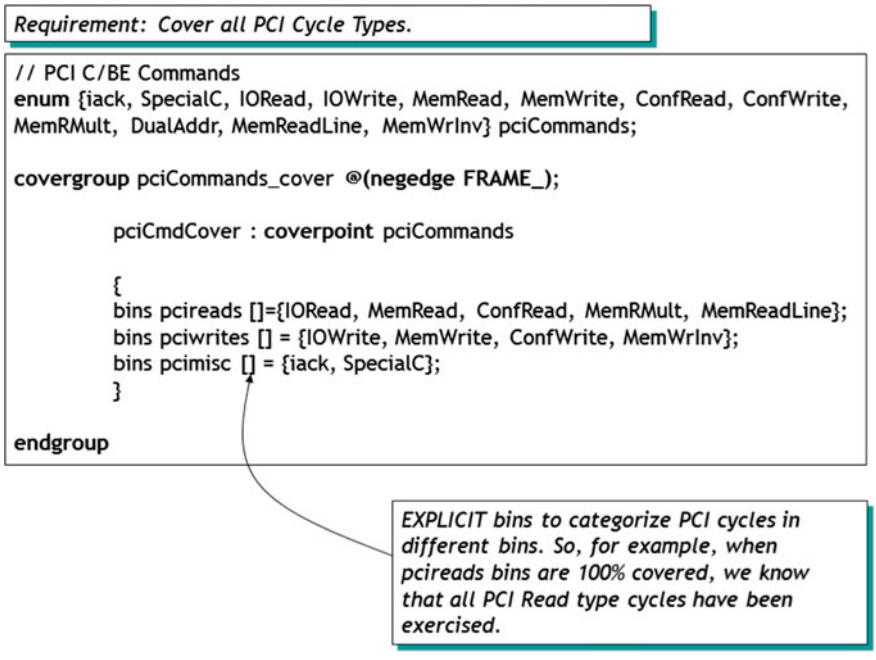


Fig. 20.5 'covergroup'/'coverpoint' example with 'bins'

Recall that [] means auto generated bins. Hence, for example, since 'bins pcireads[]' in Fig. 20.5 has 5 variables (enum type in this example), 5 bins will be created—one for each enum type. This way of creating bins is very useful because in the complex maze of features to cover, it is sure nice to have a clear distinction among different functional groups. Here pcireads[] (5 bins) covers all PCI Read Cycles; pciwrites[] (4 bins) covers all PCI Write Cycles and for the special cycles, there is the pcimisc[] (2 bins).

20.4.2 System Verilog 'covergroup'—Formal and Actual Arguments

Figure 20.6 outlines the following points.

1. Covergroup can be parameterized for reuse.
2. You have to use 'ref' type since covergroup 'gc' has been instantiated twice. 'Ref' type is required when you pass variables as actual to a formal. In other words, if you were passing a constant you would *not* need a 'Ref' type as shown in Fig. 20.6.
3. Actual arguments are passed when you instantiate a covergroup.
4. This is an example of reusability. Instead of creating two covergroups; 1 for 'adr1' and another for 'adr2', we have created only 1 covergroup called 'gc' which has a formal called 'address'. We pass 'adr1' to 'address' in the instance gcadr1 and pass 'adr2' to 'address' In the instance gcadr2. We also pass the

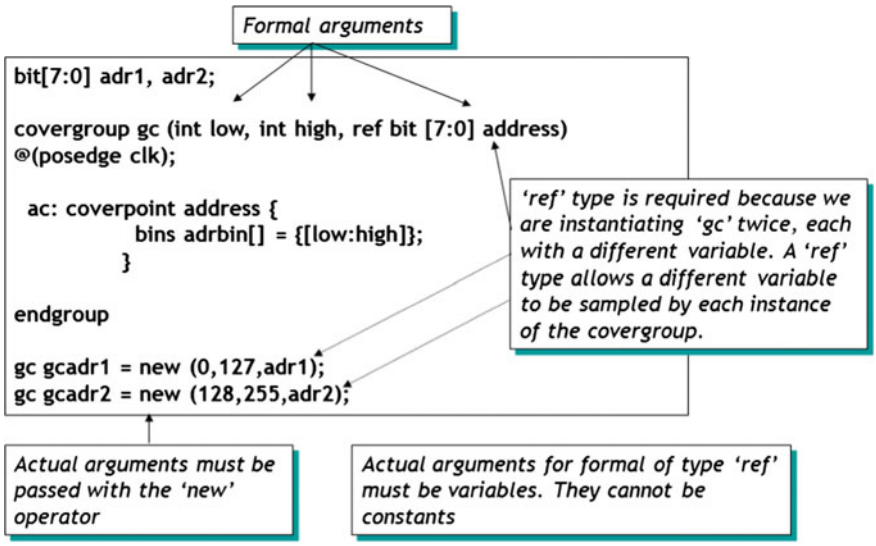


Fig. 20.6 'covergroup'—formal and actual arguments

range of adr1 and adr2 to be covered with each instance. In short, it is a good idea to create parameterizable covergroups, as the situation permits. They can be useful not only within a project but also across projects.
5. An output or inout is illegal as a formal argument.
6. Since a covergroup cannot modify any argument to the new operator, a ref argument will be treated the same as a read-only const ref argument.
7. The formal arguments of a covergroup cannot be accessed using a hierarchical name (the formals cannot be accessed outside the covergroup declaration).
8. If an automatic variable is passed by reference, behavior is undefined.

Exercise: How many bins will be created for 'bins adrbin[]' for each instance ('gcadr1' and 'gcadr2') of covergroup 'gc'?

20.4.3 'covergroup' in a 'class'

So where do you use or declare this 'covergroup'? One of the best places to embed a coverage group is within a 'class'. Why a class? Here are some reasons. (Note—discussion of 'class' is beyond the scope of this book. The author is assuming familiarity with SystemVerilog 'class').

- An embedded covergroup defines a coverage model for protected and local properties.
- Class members can be used in coverpoint expressions, coverage constructs, option initialization (we'll see option initialization in Chap. 22), etc.
- By embedding a coverage group within a class definition, the covergroup provides a simple way to cover a subset of the class properties.
- This style of declaring covergroups allow for modular verification environment development.
- An embedded covergroup can define a coverage model for protected and local class properties without any changes to the class data encapsulation.
- Class members can be used in coverpoint expressions or can be used in other coverage constructs, such as option initialization.

OK, let us see what Fig. 20.7 depicts.

'covergroup xyzCover' is sampled on any change on variable 'm_z'. This covergroup contains two coverpoints, namely 'm_x' and 'm_y'. Note that there are no explicit bins specified for the coverpoints. How many bins for each coverpoint will be created? As an exercise, please refer to previous sections to figure out.

Note that covergroup is instantiated within the 'class'. That makes sense since the covergroup is embedded within the class. Obviously, if you do not instantiate a covergroup in the 'class', it will not be created and there will not be any sampling of data.

Finally, a 'class' can indeed have more than one 'covergroup' as shown Fig. 20.8.

```
class xyz;
  bit [3:0] m_x;
  int m_y;
  bit m_z;

  covergroup xyzCover @(m_z);
            coverpoint m_x;
            coverpoint m_y;
  endgroup

  function new();
     xyzCover xyzCovInst = new;
  endfunction
endclass
```

- *By embedding a 'covergroup' within a class definition, the 'covergroup' provides a simple way to cover as part of class definition the class properties (modular development)*
- *A 'class' can have more than one 'covergroup'*

Fig. 20.7 'covergroup' in a SystemVerilog class (courtesy LRM 1800-2005)

```
class xyz;
  bit [3:0] m_x;
  int m_y;
  bit m_z, m_a;

  covergroup  m_xCover @(m_z) coverpoint m_x;
   covergroup  m_yCover @(m_a) coverpoint m_y;

  endgroup

  function new();
     m_xCover m_x_CovInst = new;
  endfunction

 function new();
     m_yCover m_y_CovInst = new;
  endfunction
endclass
```

Fig. 20.8 Multiple 'covergroup' in a SystemVerilog class

Following is an example of hierarchical accessibility (courtesy LRM).

```
Following is an example of hierarchical accessibility (courtesy LRM).

class Helper;
        int m_ev;
endclass

class MyClass;
        Helper m_obj;
        int m_a;
        covergroup Cov @(m_obj.m_ev);
                coverpoint m_a;
        endgroup

function new();
        m_obj = new; //coverage group Cov uses m_obj, m_obj must be instantiated before Cov
        Cov = new; // Create embedded covergroup after creating m_obj
endfunction

endclass
```

In this example, covergroup Cov is embedded within class MyClass, which contains an object of type Helper class, called m_obj. The clocking event for the embedded coverage group refers to data member m_ev of m_obj. Because the coverage group Cov uses m_obj, m_obj must be instantiated before Cov.

20.5 'cross' Coverage

'cross' is a very important feature of functional coverage. This is where code coverage completely fails. Figure 20.9 describes the syntax and semantics.

Syntax:
cover_cross :: = [cross_identifier:] **cross** list_of_cross_items [**iff** (expression)] cross_body

Two variables 'offset' and 'adr' are declared. Coverpoint for 'offset' creates four bins called ofsbin[0] … ofsbin[3] for the four values of 'offset' namely, 0,1,2,3. Coverpoint 'adr' also follows the same logic and creates adrbin[0] … adrbin[3] for the four values of 'adr' namely, 0,1,2,3.

adr_ofst is the label given to the 'cross' of ar, ofst. First of all, the 'cross' of 'ar' (label for coverpoint adr) and 'ofst' (label for coverpoint offset) will create another set of 16 bins (four bins of 'adr' * four bins of 'offset'). These 'cross' bins will keep track of the result of 'cross'. However, what does 'cross' mean?

Four values of 'adr' need to be covered (0,1,2,3). Let us assume adr == 2 has been covered (i.e. adrbin[2] is covered). Similarly, there are four values of 'offset' that need to be covered (0,1,2,3) and that offset == 0 has also been covered (i.e. ofsbin[0] has been covered). However, have we covered 'cross' of adr = 2 (adrbin

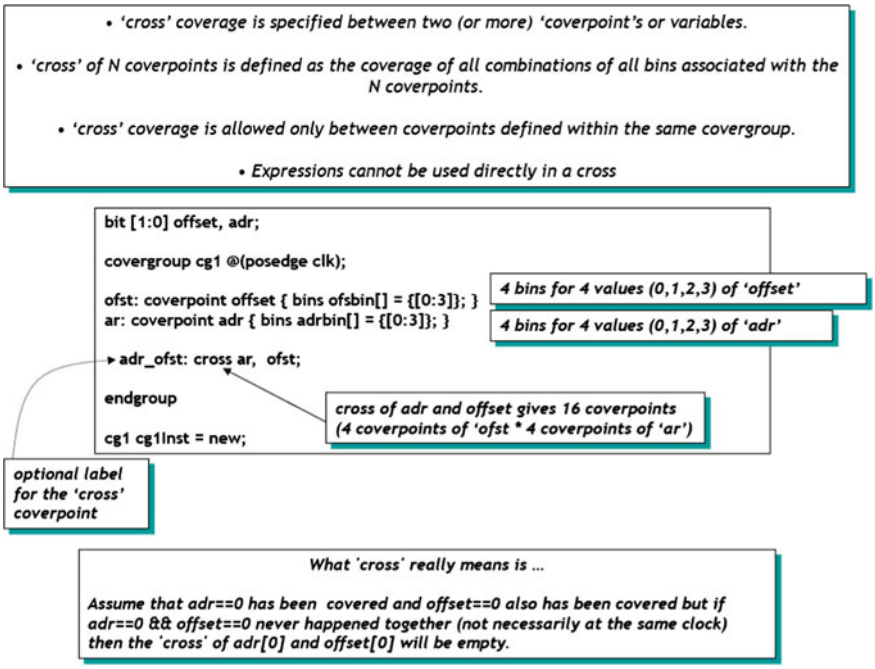


Fig. 20.9 'cross' coverage—basics

[2]) and offset = 0 (ofsbin[0])? Not necessarily. 'cross' means that adr = 2 and offset = 0 must be true 'together' at some point in time. This does *not* mean that they need to be 'covered' at the *same* time. It simply means that (e.g.) if adr = 2 that it should remain at that value until offset = 0 (or vice versa). This will make both of them true 'together'. If that is the case, then the 'cross' of adrbin[2] and ofsbin[0] will be considered 'covered'.

In the simulation log in Fig. 20.10, we see that both adrbin[2] and ofsbin[0] have been individually covered 100 %. However, their 'cross' has not been covered.

Let us look at the simulation log further.

First, you will see the 4 bins (ofsbin[0] to ofsbin[3]) of coverpoint cg1::ofst. All 4 bins are covered and hence coverpoint cg1::ofst is 100 % covered. Next, you will see the 4 bins (adrbin[0] to adrbin[3]) of coverpoint cg1::ar. All bins are covered here as well and so is the coverpoint cg1::ar.

Now let us look at the 'cross' of 4 * 4 bins = 16 bins coverage. Both 'ofst' and 'ar' are 100 % covered—but—the 3 cases that follow (among many others) are not covered because whatever values the testbench drove, these bins never had the same value at any given point in time (e.g., adrbin[2] is '2' at time t, then ofsbin[0] should be '0' either at time t or any time after that, as long as adrbin[2] = '2').

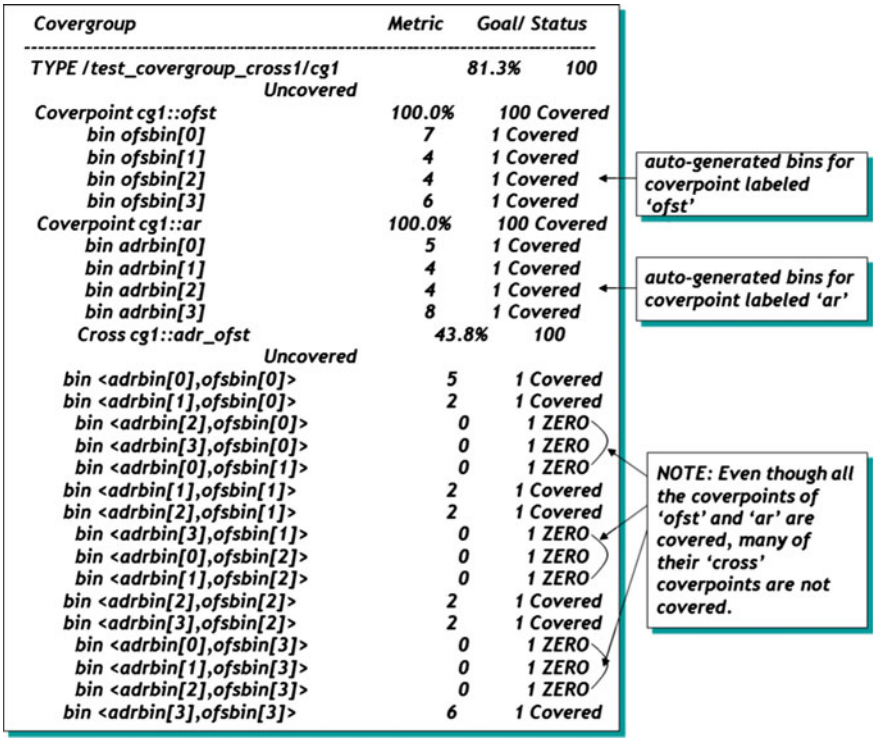


Fig. 20.10 'cross' coverage—simulation log

Hence,

$$\text{bin} < \text{adrbin}[2], \text{ofsbin}[0] > \quad 0 \quad 1 \text{ ZERO}$$

Similarly, there are other cases of 'cross' that are not covered as shown in the simulation log. Such a log will clearly identify the need to enhance your testbench. To reiterate, such 'cross' cannot be derived from code coverage.

Figure 20.11 shows how cross is achieved between an enum type and a bit type. Idea is to show how cross coverpoint/bins are calculated. The figure describes different ways 'cross' bins are calculated.

Recall that a 'cross' can be defined only between N coverpoints, meaning you must have an explicit coverpoint for a variable in order to cross with another coverpoint. That is where there is an anomaly when it comes to enum type. The enum type 'color' has no coverpoint defined for it. Yet, we are able to use it in 'cross'. That is because the language semantics implicitly creates a coverpoint for the enum type 'color' and track its cross coverage. Other than an enum type, you must create a coverpoint of a variable in the covergroup, if you need to 'cross' it with another variable coverpoint. Note also that you use the 'label' of each coverpoint in the 'cross' statement, *not* the name of the variable. Again, the exception

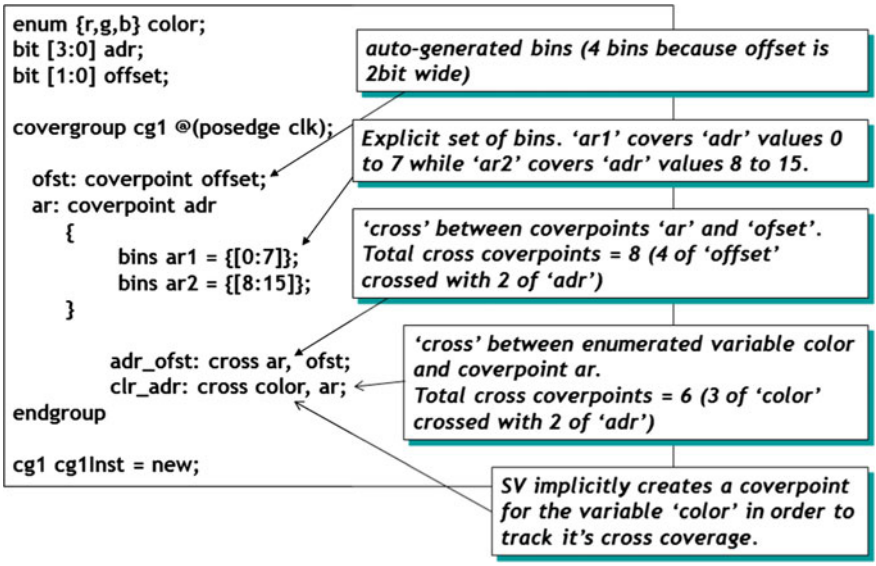


Fig. 20.11 'cross'—example (further nuances)

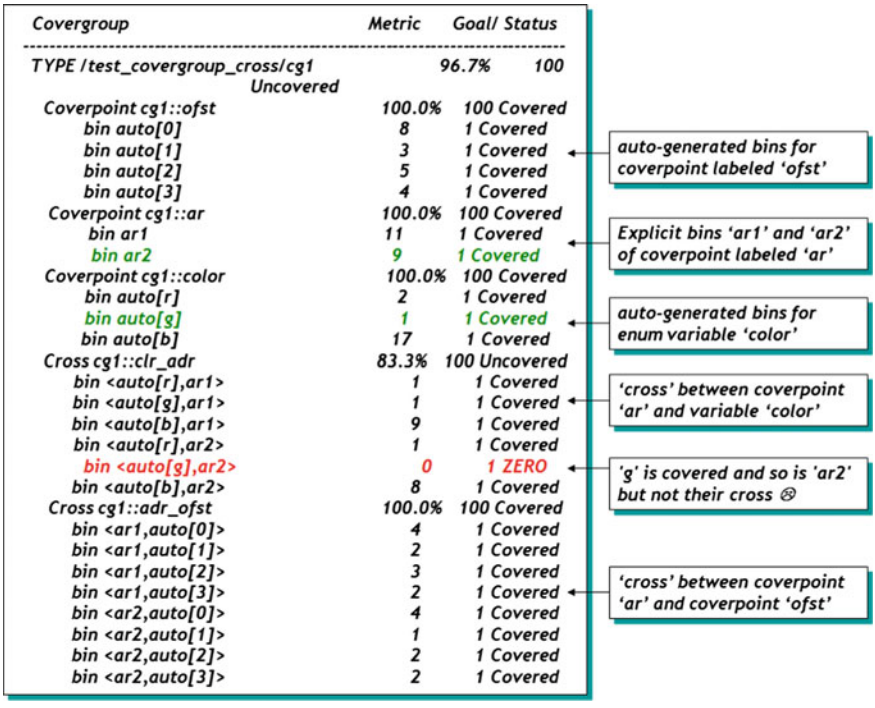


Fig. 20.12 'cross' example—simulation log

here is the enum type for which we use the enum type name, as in 'color' (because we didn't have to create a coverpoint for it—so there is no coverpoint name).

Please study this and other figures carefully since they will act as guideline for your projects. A simulation log of this example is presented in Fig. 20.12. The annotations describe what's going on.

20.6 More 'Bins'

20.6.1 'Bins' for Transition Coverage

As noted in Fig. 20.13, this is by far the most useful feature of functional coverage. Transaction level transitions are very important to cover. For example, did CPU issue a read followed by write-invalid? Did you issue a D$miss followed by a D$hit cycle? Such transitions are where the bugs occur and we want to make sure that we have indeed exercised such transitions.

Figure 20.13 explains how the semantics work. Note that we are addressing both the 'transition' as well as the 'cross' of 'transition' coverage.

There are two transitions in the example.

bins ar1 = (8'h00 => 8'hff); which means that adr1 should transition from '0' to 'ff' on two consecutive posedge of clk. In other words, the testbench must exercise this condition for it to be covered.

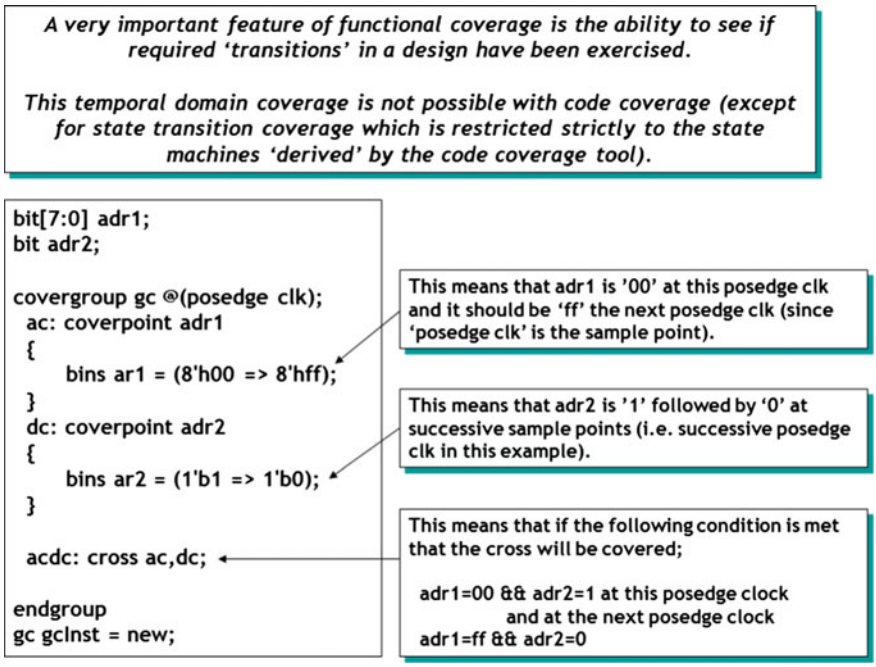


Fig. 20.13 'bins' for transition coverage

Similarly, there is the 'bins ar2' that specifies the transition for adr2 (1 => 0).

The cross of transitions is shown at the bottom of the figure. Very interesting how this works. Take the first values of each transition (namely, adr1 = 0 && adr2 = 1). This will be the start points of cross transition at the posedge clk. If at the next (posedge clk) values are adr1 = 'ff' && adr2 = 0, the cross transition is covered.

More on the 'bins' of transition is shown in the Fig. 20.14. In the figure, different styles of transitions have been shown. 'bins adrb2' requires that 'adr1' should transition from 1=>2=>3 on successive posedge clk. Of course, this transition sequence can be of arbitrary length. 'bins adrb3[]' shows another way to specify multiple transitions. The annotation in the figure explains how we get 4 transitions.

'bins adrb5' is (in some sense) analogous to the consecutive operator of assertions. Here 'hf [*3] means that adr1 = 'hf should repeat 3 times at successive posedge clk.

Similarly, the non-consecutive transition ('ha [->3]) means that adr1 should be equal to 'ha, 3 times and not necessarily at consecutive posedge clk. Note that just as in non-consecutive operator, here also 'ha need to arrive 3 times with arbitrary number of clocks in-between their arrival and that 'adr1' should *not* have any other value in-between these 3 transitions. The simulation log shows the result of a testbench that exercises all the transition conditions.

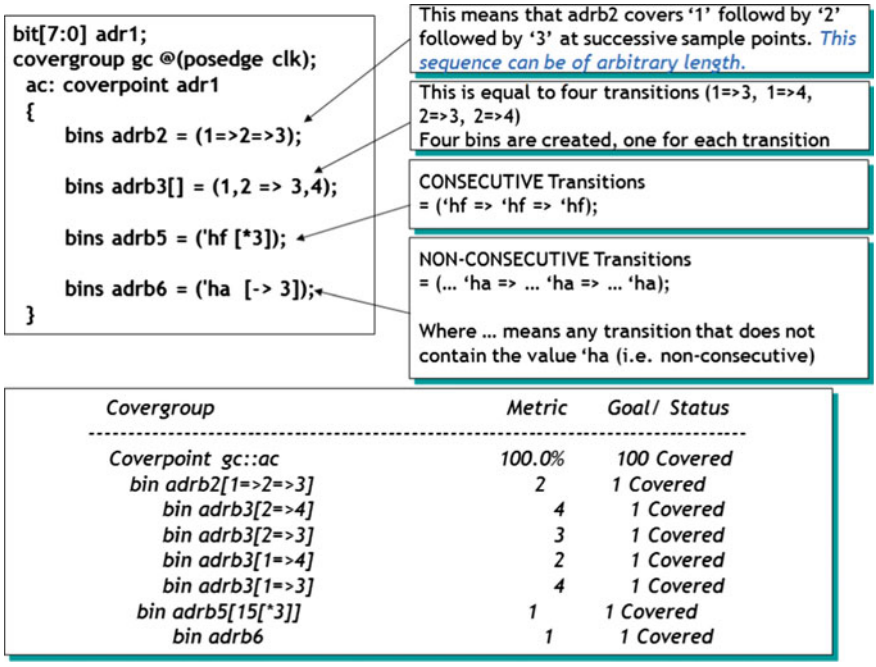


Fig. 20.14 'bins'—transition coverage further features

One more example of transition coverage in Fig. 20.15.

This figure shows a very interesting semantic feature of transition. Note that 'bins adrb3[] = (1,2 => 3,4)' is *not* the same as 'bins adrb4[] = (1=>3, 1=>4, 2=>3, 2=>4);'.

'bins adrb3[] = (1,2 => 3,4)' means transitions ***(1=>3, 1=>4, 2=>3, 2=>4)***. BUT 'bins adrb4[] = (1=>3, 1=>4, 2=>3, 2=>4) means

[1=>3=>4=>3=>4]
[1=>3=>4=>2=>4]
[1=>3=>2=>3=>4]
[1=>3=>2=>2=>4]
[1=>1=>4=>3=>4]
[1=>1=>4=>3=>4]
[1=>1=>2=>3=>4]
[1=>1=>2=>2=>4]

Is that intuitive? I don't think so. However, the following will explain.

'bins adrb4[] = (1=>3, 1=>4, 2=>3, 2=>4)' is *equivalent* to
'bins adrb4[] = (1=>(3, 1)=>(4, 2)=>(3, 2)=>4)'

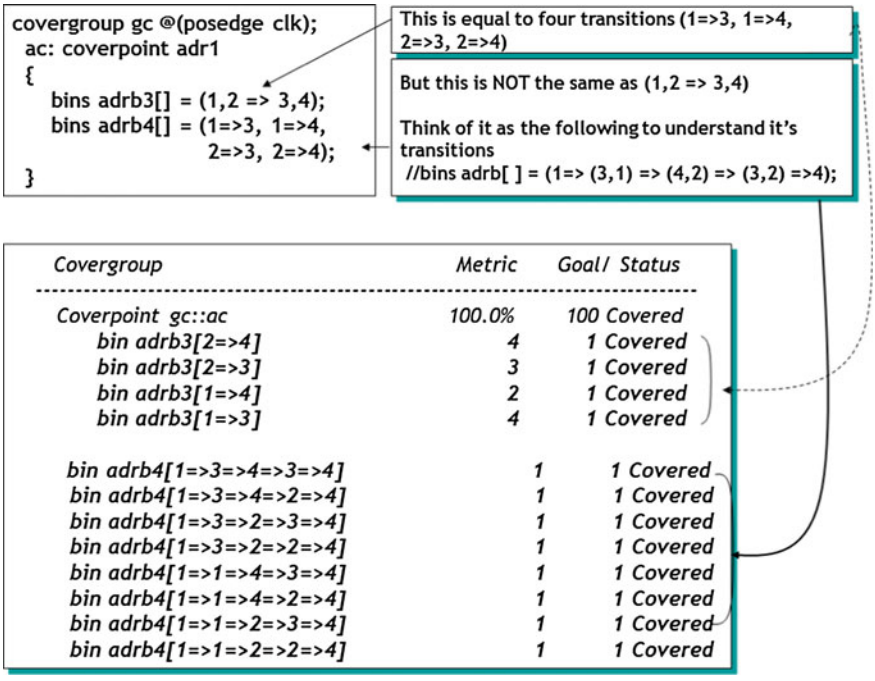


Fig. 20.15 'bins' for transition—example with simulation log

If you see the equivalent definition, you will be able to understand the transition. Study them carefully, you will figure out why the transitions look the way they do.

In short, you need to be careful how you specify transition coverpoints.

Now let us turn back to our favorite PCI example that we have been following. We started with simple coverage, moved to 'bins' coverage and now we will see the 'transition' coverage. This is the reason we were building on the same example showing how such a coverage model can be written starting with a simple model.

Figure 20.16 shows the same enum type and same bins. But in addition, it now defines two transition bins named R2W (for Read to Write) and W2R (for Write to Read).

bins R2W means that all possible Read cycles types are followed by all possible Write cycles. In this example that means we must cover the following transactions

IORead => IOWrite; IORead => MemWrite; IORead => ConfWrite;
MemRead => IOWrite; MemRead => MemWrite; MemRead => ConfWrite;
ConfRead => IOWrite; ConfRead => MemWrite; ConfRead => ConfWrite;
MemReadLine => IOWrite; MemReadLine => MemWrite; MemReadLine => ConfWrite;

Same type of transitions for bins W2R[].

As you notice, this is quite powerful. Many bugs occur when there is a transition from one transaction type to the next. You have to make sure that your testbench indeed covers such transitions.

Requirement: Cover all PCI Cycle Types and transitions among Read and Write cycles.

```
// PCI C/BE Commands
enum {iack, SpecialC, IORead, IOWrite, MemRead, MemWrite, ConfRead, ConfWrite,
MemRMult, DualAddr, MemReadLine, MemWrInv} pciCommands;

covergroup pciCommands_cover @(posedge clk);
        pciCmdCover : coverpoint pciCommands
        {
        bins pcireads [] = {IORead, MemRead, ConfRead, MemRMult, MemReadLine};
        bins pciwrites [] = {IOWrite, MemWrite, ConfWrite, MemWrInv};
        bins pcimisc [] = {iack, SpecialC};

        bins R2W [] = (IORead, MemRead, ConfRead, MemRMult, MemReadLine =>
IOWrite, MemWrite, ConfWrite, MemWrInv);
        bins W2R [] = (IOWrite, MemWrite, ConfWrite, MemWrInv => IORead,
MemRead, ConfRead, MemRMult, MemReadLine);
        }
endgroup
```

Fig. 20.16 Example of PCI cycles transition coverage

20.6.2 'wildcard bins'

Since no one likes to type a sequence repeatedly, we create don't care (or as the functional coverage lingo calls it 'wildcard' bins). Self-explanatory. As shown in Fig. 20.17, you can use either an 'x' or a 'z' or '?' (doesn't this look familiar to Verilog?) to declare 'wildcard' bins. Note that such bins must precede with the keyword 'wildcard'. 'wildcard bins ainc' specifies that adr1 values 1100, 1101, 1110, 1111 need to be covered.

Note also 'wildcard bins adrb1[]' and 'wildcard bins adrb2'. One creates implicit ([]) 4 bins while the other creates only 1 bin. The one that creates 4 explicit bins will check to see that—*each of the*—4 transitions take place. While adrb2 that creates only 1 bin will be considered covered if—*any*—of the 4 transitions take place.

20.6.3 'ignore_bins'

'ignore_bins' is very useful when you have a large set of bins to be defined. Instead of defining every one of those, if you can identify the ones that are not of interest,

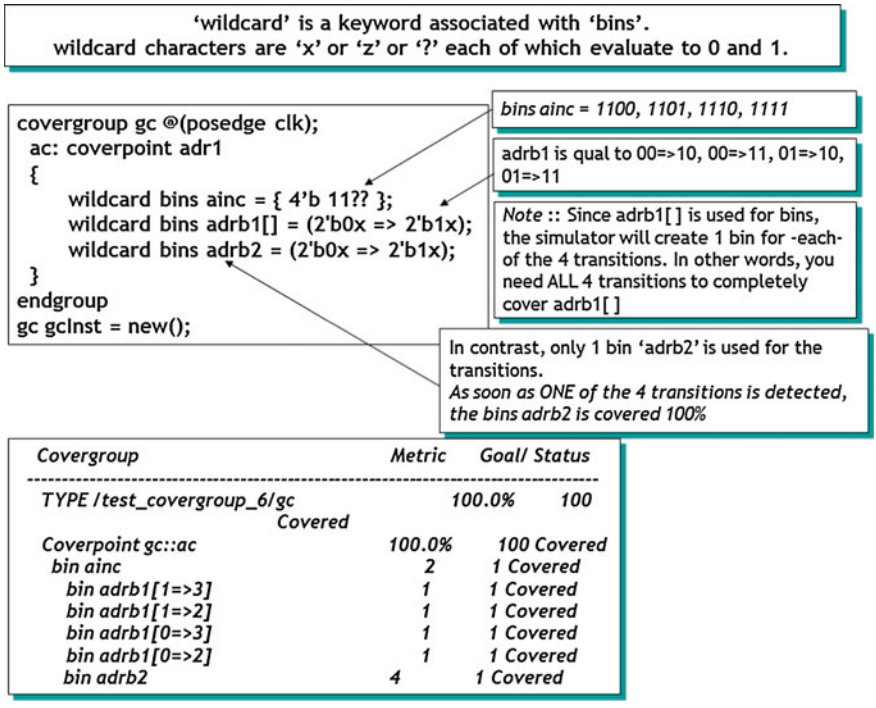


Fig. 20.17 wildcard 'bins'

then you can simply define 'ignore_bins'. The ones that are specified to be ignored, will indeed be ignored and rest will be covered.

As we noted at the onset of this chapter, when there is no explicit 'bins' defined for a coverpoint, the simulator creates all possible bins that the 'covered' variable requires. In Fig. 20.18, only 'ignore_bins' are specified in the coverpoint adr1 but no 'bins'. This means that the simulator will first create all 16 bins (adr1 is 4 bit wide) for adr1. Then it will ignore value 4,5 and 6,7,8,9,10,11,12,13,14,15 and cover only adr1 = 0,1,2,3. This is reflected in the simulation log at the bottom of Fig. 20.18.

20.6.4 'illegal_bins'

'illegal_bins' is interesting in that it will complain, if you *do* cover a given scenario. In Fig. 20.19, we refer to the same old 'adr1'. 16 auto bins are created for coverpoint 'adr1' since we don't explicitly declare any bins for it. And we say that if the testbench ever hits adr1 == 0, that it should be considered illegal. Coverage of adr1 == 0 should not occur. As seen from the simulation log, coverage of adr1 = 0 results in an Error.

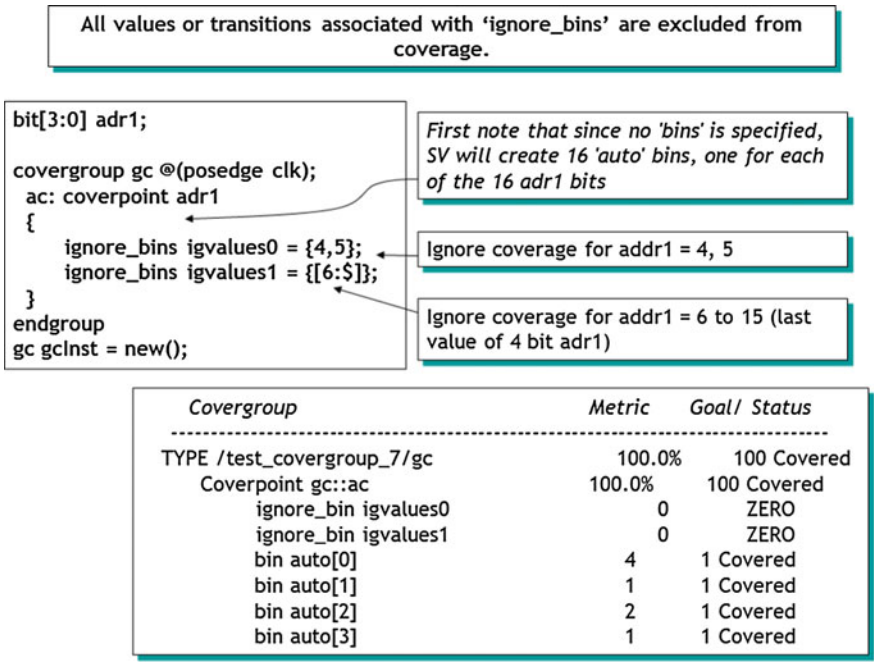


Fig. 20.18 'ignore_bins'—basics

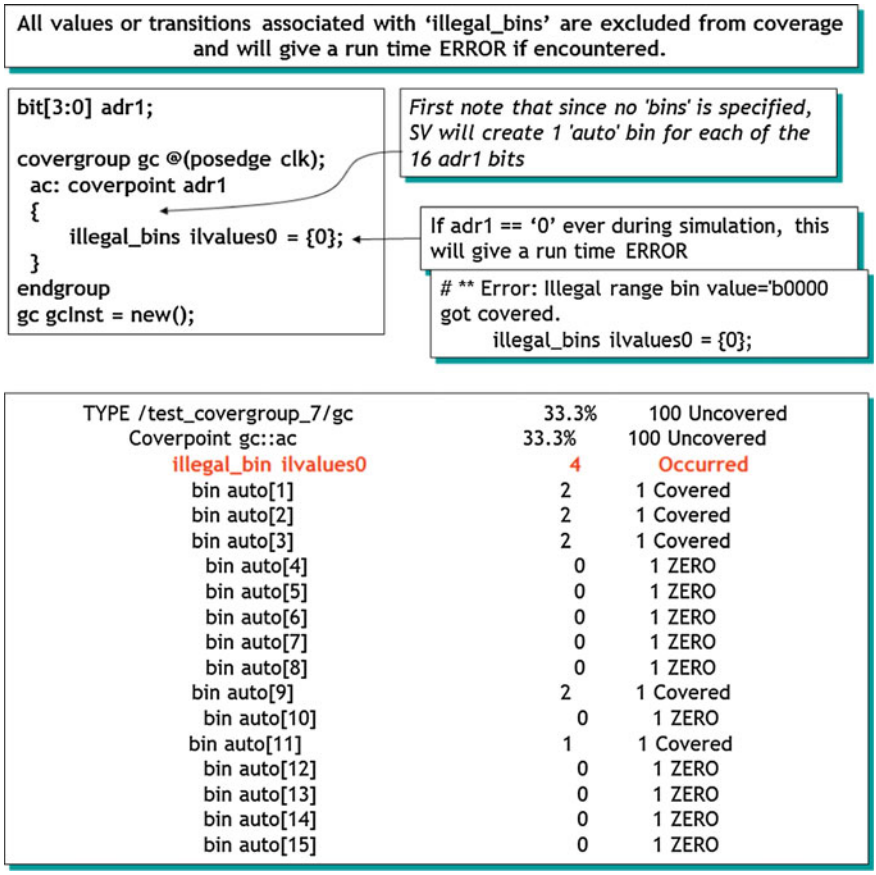


Fig. 20.19 'illegal_bins'

Illegal bins take precedence over any other bins, that is, they will result in a run-time error even if they are also included in another bin. Specifying an illegal value has no effect on a transition that includes the value. Illegal transition bins cannot specify a sequence of unbounded or undetermined varying length.

20.6.5 'binsof' and 'intersect'

Ok, now we are in some seriously esoteric territory! Figure 20.20 shows that 'coverpoint b' has two bins; one is called 'bb' and has value from 0 to 12 of 'b'. bins cc on the other hand has values 13,14,15,16 of 'b' that need to be covered. So far so good. Then we declare a 'cross' of a, bc. So, let us first see what this cross looks like.

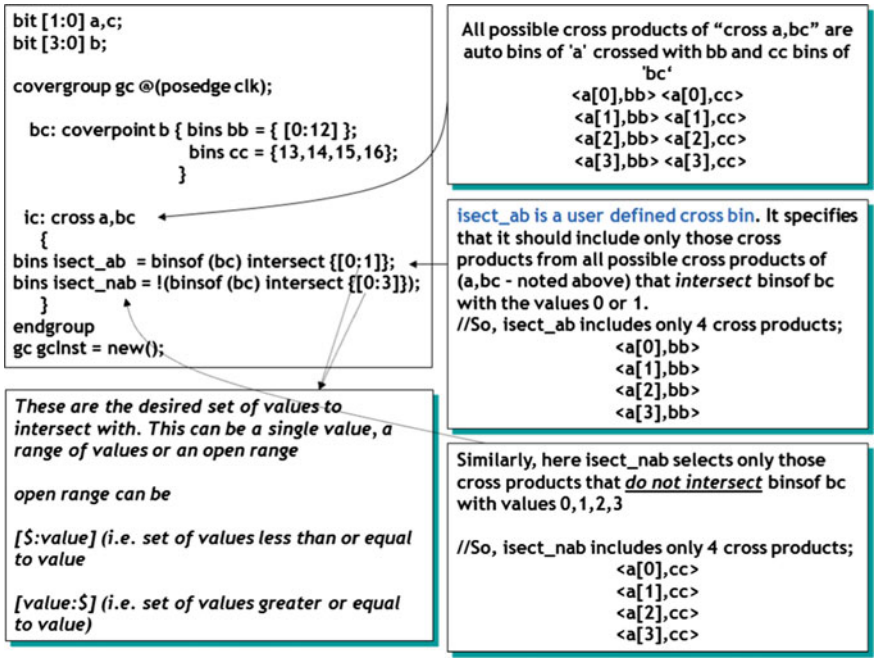


Fig. 20.20 'binsof' and 'intersect'

'a' has 4 implicit bins a[0], a[1], a[2], a[3] for four values 0,1,2,3 and 'bc' has two bins 'bb' and 'cc. So the 'cross a,bc' produces

<a[0],bb> <a[0],cc>
<a[1],bb> <a[1],cc>
<a[2],bb> <a[2],cc>
<a[3],bb> <a[3],cc>

We are moving along just fine—right? But now it gets interesting. We don't want to cover all 'cross' bins into our 'cross' of 'a' and 'bc'. We want to 'intersect' them and derive a new subset of the total 'cross' set of bins.

```
bins isect_ab = binsof (bc) intersect {[0:1]};
```

says that take all the bins of 'bc' (which are 'bb' and 'cc') and intersect them with the 'values' 0 and 1. Now note carefully that 'bb' carries values {[0:12]} which includes the values 0 and 1, while 'cc' does not cover 0 or 1. Hence, only those cross products of 'bc' that cover 'bb' are included in this intersect. Note that the below mentioned subset is selected from the 'cross' product shown above.

<a[0],bb>, <a[1],bb>, <a[2],bb>, <a[3],bb>

Now onto the next intersect.

```
bins isect_nab = !(binsof (bc) intersect {[0:3]})
```

Similarly, first, note that here we are using negation of binsof. So this'bins' statement says, take the intersect of binsof (bc) and {[0:3]} and discard them from the 'cross' of a, bc. Keep only those that do not intersect. Note that {[0:3]} again fall into the bins 'bb' and would have resulted in exactly the same set that we saw for the non-negated intersect. Since this one is negated it will ignore cross with 'bb' and only pick the 'bins cc' from the original 'cross' of a, bc.

```
<a[0],cc>, <a[1],cc>, <a[2],cc>, <a[3],cc>
```

Chapter 21
Performance Implications of Coverage Methodology

Introduction: This chapter describes the methodology components of Functional Verification. What you should cover, when you should cover, performance implications and applications on how to write properties that combine the power of assertions with power of Functional Coverage.

21.1 Know *What* You Should Cover

- Don't cover the entire 32-bit address bus.
  - Cover only the addresses of interest (e.g., Byte/word/dword aligned; start/end address; bank crossing address, etc.)
- Don't cover the entire counter range
  - Cover only the rollover counter values (transition from all 1's to all 0's)
- No need to cover the entire 2K Fifo
  - Cover only fifo full, fifo empty, fifo full crossed with fifo_push, fifo empty crossed with fifo read, etc.
- Auto generated bins are both a convenience and a nuisance. They may create a lot of clutter with auto-generated bins that may not be relevant. Be judicious in usage of auto generated 'bins'
- Use 'cross' and 'intersect' to weed out unwanted 'bins'. Also, 'illegal_bins' and 'ignore_bins'.

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_21

21.2 Know *When* You Should Cover

- Enable your cover points only when they are meaningful
  - Disable coverage during 'reset'
  - Cover 'test mode' signals only when in test mode (for example, JTAG TAP Controller TMS asserted)
  - Make effective use of coverage methods such as 'start', 'stop', 'sample' (more on this later…)
  - Do not repeat with covergroups what you have covered with SVA 'cover'
  - Make effective use of covergroup 'trigger' condition
  - Make effective use of the 'action' block associated with 'cover' to activate a covergroup

If some of these points (e.g. 'trigger') don't quite make sense, please hold on. We will be covering such features in upcoming sections.

21.3 When to 'Cover' (Performance Implication)

Functional coverage should be carefully collected as discussed above. The language does allow tasks that allow you to control when to start collecting coverage and when to stop. These tasks can be associated with an instance of a covergroup and invoked from procedural block.

Figure 21.1 shows the covergroup 'rg' with two coverpoints 'pc' and 'gc'. 'pc' covers all the pending requests and 'gc' covers the number of masters on the bus when those requests are made. 'my_rg' is the instance of this covergroup.

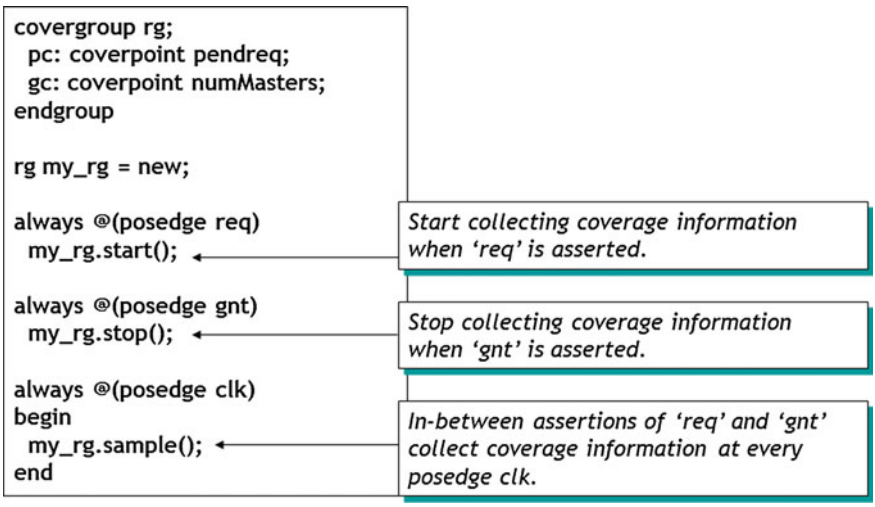


Fig. 21.1 Functional coverage—performance implication

Since we want to start collecting pending requests at the assertion of req. When the requests are granted, we don't want to cover pending requests and number of masters any more. 'gnt' related cover can be another covergroup.

Simple control but very good performance improvement. Use it wisely to speed up your simulation and a more meaningful coverage log.

Lastly, there is the sampling edge task sample() which derives its sampling edge from "always @ (posedge clk)" and applies it to 'my_rg' as its sampling edge. This also tells us that we can have covergroup specific sampling edges. Very good feature. Note that my_rg.sample() will 'start' when my_rg.start() is executed and will stop when my_rg.stop() is executed. This is about as easy as it gets when it comes to controlling collection of coverage information.

Note that optionally, there is also a 'strobe' option (see Section 22) that can be used to modify the sampling behavior. When the strobe option is not set (the default), a coverage point is sampled the *instant* the clocking event takes place, as if the process triggering the event were to call the built-in sample() method. If the clocking event occurs multiple times in a time step, the coverage point will also be sampled multiple times. The 'strobe' option can be used to specify that coverage points are sampled in the Postponed region, thereby filtering multiple clocking events so that only one sample per time slot is taken. The strobe option only applies to the scheduling of samples triggered by a clocking event.

21.4 Application: Have You Transmitted All Different Lengths of a Frame?

This good application combines local variables, subroutine calls, covergroups and interaction with procedural code outside of the assertion. Here's how it works (Fig. 21.2).

Read this example bottom up.

Property frameLength says that when the rising edge of TX_EN is sampled that we should check the length of the transmitted frame using sequence frmLength.

Sequence frmLength declares a local variable 'cnt' and at TX_EN==1, initializes cnt=1. One clock later (##1) it increments cnt forever ((TX_EN, cnt++)[*0:$]) until TX_EN deasserts (falls). At that time, we call a task (i.e. a subroutine) called store_Frame_Lngth(cnt) and provide it the final count as a parameter. This final count is the length of the Frame that started with TX_EN assertion.

The task store_Frame_Lngth takes the 'cnt' as input and assigns it to 'logic' type FrameLngth and triggers a named event called measureFrameLength.

Now the covergroup length_cg triggers at 'measureFrameLength' edge, which we just triggered explicitly from task store_Frame_Lngth. The coverpoint covers FrameLngth.

In short, we measure the Frame Length starting assertion of TX_EN until deassertion of it. We measure the frame length between assertion and deassertion of

This application exemplifies the use of
- *local variables*
- *subroutine call associated with an expression to update a variable*
- *covergroup triggered from an explicit event.*

```
logic [7:0] FrameLngth = 0;
event measureFrameLength;

covergroup length_cg @(measureFrameLength );
        coverpoint FrameLngth;
endgroup

task store_Frame_Lngth;
input [7:0] x;
        FrameLngth = x;
        -> measureFrameLength;
endtask

sequence frmLength;
int cnt;
        (TX_EN, cnt=1)  ##1  ((TX_EN, cnt++)[*0:$])
        ##1 (!TX_EN, store_Frame_Lngth(cnt))
endsequence

property frameLength;
        @(posedge TX_CLK) $rose(TX_EN) |-> frmLength;
endproperty

fLength: assert property (frameLength);
```

Fig. 21.2 Application—have you transmitted all different lengths of a frame?

TX_EN and cover it. With every new assertion of TX_EN, we measure the length of a new frame.

Note that "coverpoint FrameLngth" does not specify any explicit bins. That will create 256 explicit bins each containing a frame length. This way we make sure that we have covered all (i.e. 256) different frame lengths.

Chapter 22
Coverage Options

Introduction: This chapter describes the Coverage Options offered by the language. Options for 'covergroup' type (both instance specific and instance specific per-syntactic level) are described. Practical project methodology based examples are provided that you can directly deploy in your project (Fig. 22.1).

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4_22

The syntax for specifying these options in the covergroup definition is

option.option_name = expression;

Option Name	Default	Description
weight = *number*	1	If set at the **covergroup** syntactic level, it specifies the weight of this **covergroup** instance for computing the overall instance coverage of the simulation. If set at the **coverpoint** (or **cross**) syntactic level, it specifies the weight of a **coverpoint** (or **cross**) for computing the instance coverage of the enclosing **covergroup**.
goal = *number*	90	Specifies the target goal for a **covergroup** instance, or a **coverpoint** or a cross of an instance.
name = *string*	unique name	Specify a name of the **covergroup** instance. If unspecified, a unique name for each instance is automatically generated by the tool.
comment = *string*	" "	A comment that appears with the instance of a **covergroup**, or a **coverpoint** or **cross** of the **covergroup** instance. The comment is saved in the coverage database and included in the coverage report.
at_least = *number*	1	Minimum number of hits for each bin. A bin with a hit count that is less than *number* is not considered covered.
detect_overlap = *boolean*	0	When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a **coverpoint**
auto_bin_max = *number*	64	Maximum number of automatically created bins when no bins are explicitly defined for a **coverpoint**
cross_auto_bin_max = *number*	unbounded	Maximum number of automatically created cross product bins for a **cross**.
cross_num_print_missing = *number*	0	Number of missing (not covered) cross product bins that must be saved to the coverage database and printed in the coverage report.
per_instance = *boolean*	0	Each instance contributes to the overall coverage information for the **covergroup** type. When true, coverage information for this **covergroup** instance is tracked as well.

LRM: SystemVerilog 3.1a, Table 20-1

Fig. 22.1 Coverage options—reference material

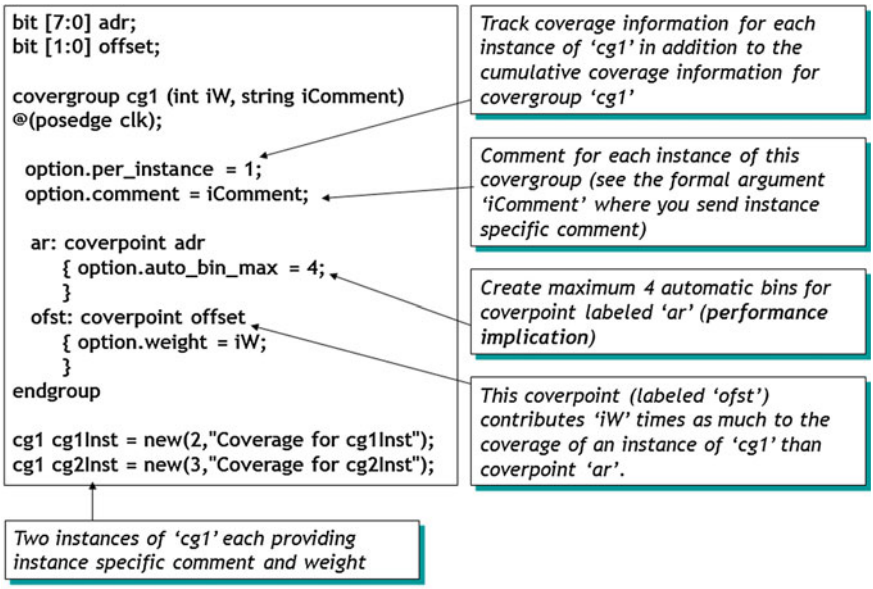


Fig. 22.2 Coverage options—instance specific—example

22.1 Coverage Options—Instance Specific—Example

See Fig. 22.2.

22.2 Coverage Options—Instance Specific Per-Syntactic Level

See Figs. 22.3 and 22.4.

*The following table summarizes the syntactical level (**covergroup**, **coverpoint**, or **cross**) at which instance options can be specified. All instance options can be specified at the covergroup level. Except for the **weight**, **goal**, **comment**, and **per_instance** options, all other options set at the covergroup syntactic level act as a default value for the corresponding option of all coverpoints and crosses in the covergroup. Individual coverpoint or crosses can overwrite these default values. When set at the covergroup level, the **weight**, **goal**, **comment**, and **per_instance** options do not act as default values to the lower syntactic levels.*

Option Name	Allowed in Syntactic Level		
	covergroup	coverpoint	cross
name	Yes	No	No
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
at_least	Yes (default for coverpoints & crosses)	Yes	Yes
detect_overlap	Yes (default for coverpoints)	Yes	No
auto_bin_max	Yes (default for coverpoints)	Yes	No
cross_auto_bin_max	Yes (default for crosses)	No	Yes
cross_num_print_missing	Yes (default for crosses)	No	Yes
per_instance	Yes	No	No

LRM: SystemVerilog 3.1a, Table 20-2

Fig. 22.3 Coverage options—instance specific per-syntactic level

*The following table lists options that **describe a feature of the 'covergroup' type as a whole.***

type_option.option_name = expression;

Option Name	Default	Description
weight = *constant_number*	1	If set at the **covergroup** syntactic level, it specifies the weight of this **covergroup** for computing the overall cumulative (or type) coverage of the saved database. If set at the **coverpoint** (or **cross**) syntactic level, it specifies the weight of a **coverpoint** (or **cross**) for computing the cumulative (or type) coverage of the enclosing covergroup.
goal = *constant_number*	90	Specifies the target goal for a **covergroup** type, or a **coverpoint** or cross of a **covergroup** type.
comment = *string_literal*	" "	A comment that appears with the **covergroup** type, or a **coverpoint** or cross of the **covergroup** type. The comment is saved in the coverage database and included in the coverage report
strobe = *constant_number*	0	If set to 1, all samples happen at the end of the time slot, like the $strobe system task.

LRM: SystemVerilog 3.1a, Table 20-3

Coverage type-options per Syntactic Level

LRM: SystemVerilog 3.1a, Table 20-4

Option Name	Allowed Syntactic Level		
	covergroup	coverpoint	cross
weight	Yes	Yes	Yes
goal	Yes	Yes	Yes
comment	Yes	Yes	Yes
strobe	Yes	No	No

Fig. 22.4 Coverage options type specific per syntactic level

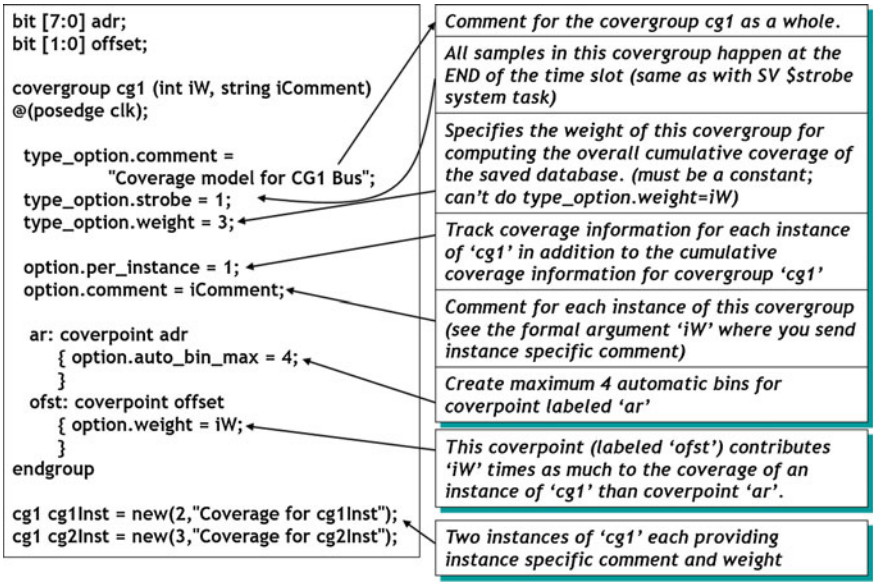


Fig. 22.5 Coverage options for 'covergroup' type specific—comprehensive example

22.3 Coverage Options for 'Covergroup' Type—Example

See Figs. 22.5 and 22.6.

Predefined coverage system tasks and functions	
$set_coverage_db_name (<name>);	Sets the filename of the coverage database into which coverage info. is saved at the *end* of a simulation run
$load_coverage_db (<name>);	Load from a given filename the cumulative coverage information for all the coverage group types.
$get_coverage ();	Returns a real number in the range of 0 to 100 the overall coverage of *all* covergroups.

Pre-defined coverage methods used in procedural code

LRM: SystemVerilog 3.1a, Table 20-5

Method	Can be called on			Description
	covergroup	coverpoint	cross	
void sample()	Yes	No	No	Triggers sampling of the covergroup
real get_coverage()	Yes	Yes	Yes	Calculates type coverage number (0...100)
real get_inst_coverage()	Yes	Yes	Yes	Calculates coverage number (0...100)
void set_inst_name (string)	Yes	No	No	Sets the instance name to the given string
void start()	Yes	Yes	Yes	Starts collecting coverage information
void stop()	Yes	Yes	Yes	Stops collecting coverage information
real query()	Yes	Yes	Yes	Returns the cumulative coverage information (for the coverage group type as a whole)
real inst_query()	Yes	Yes	Yes	Returns the per-instance coverage information for this instance.

Fig. 22.6 Predefined coverage system tasks and functions

Index

© Springer International Publishing Switzerland 2016

A.B. Mehta, *SystemVerilog Assertions and Functional Coverage*,
DOI 10.1007/978-3-319-30539-4