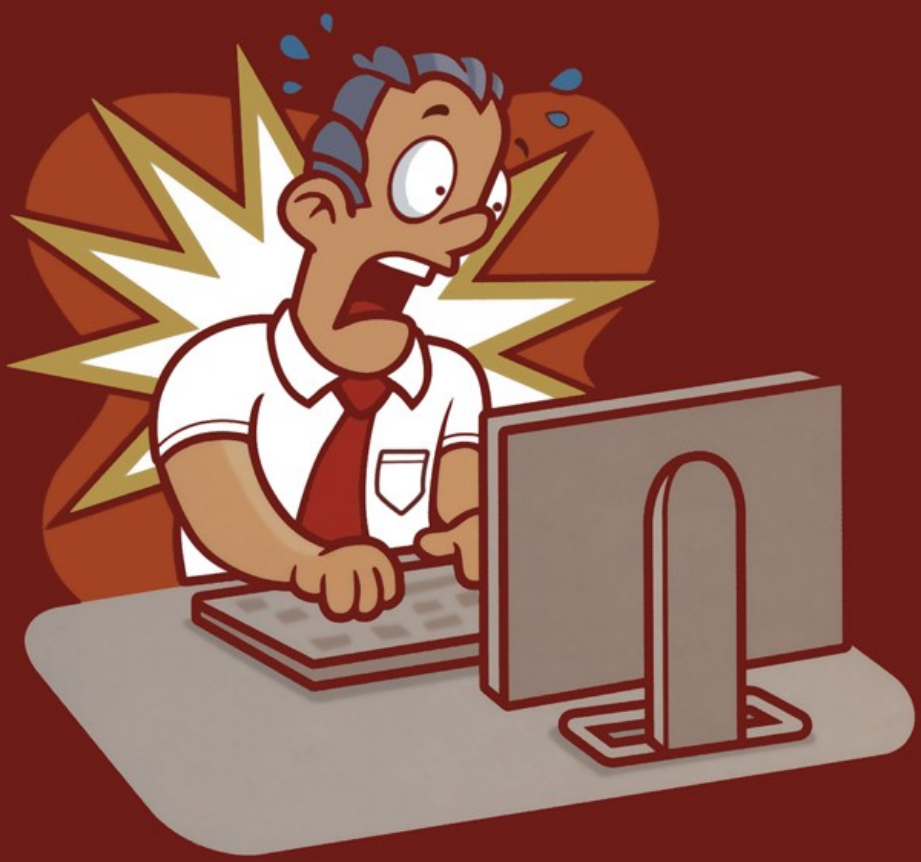
Stuart Sutherland and Don Mills

Verilog and SystemVerilog Gotchas

101 Common Coding Errors and How to Avoid Them



Springer

Verilog and SystemVerilog Gotchas

101 Common Coding Errors and How to Avoid Them

Stuart Sutherland
Don Mills

Verilog and SystemVerilog Gotchas

101 Common Coding Errors and How to Avoid Them

Springer

Stuart Sutherland
Sutherland HDL, Inc.
Tualatin, OR
USA

Don Mills
LCDM Engineering
Chandler, AZ
USA

Library of Congress Control Number: 2007926706

ISBN 978-0-387-71714-2 e-ISBN 978-0-387-71715-9

Printed on acid-free paper.

9 8 7 6 5 4 3 2 1

springer.com

Dedication

To my wonderful wife, LeeAnn, and my children, Ammon, Tamara, Hannah, Seth and Samuel — thank you for your patience during the many long hours and late nights you tolerated while this book was being written.

Stu Sutherland
Portland, Oregon

To my wife and sweetheart Geri Jean, and my children, Sara, Kirsten, Adam, Alex, Dillan, Donnelle, Grant and Gina — thanks to each of you for the patience you have had with me as I have dealt with debugging many of these gotchas on designs over the years.

Don Mills
Chandler, Arizona

About the Authors

Mr. Stuart Sutherland is a member of the IEEE 1800 working group that oversees both the Verilog and SystemVerilog standards. He has been involved with the definition of the Verilog standard since its inception in 1993, and the SystemVerilog standard since work began in 2001. In addition, Stuart is the technical editor of the official IEEE Verilog and SystemVerilog Language Reference Manuals (LRMs). Stuart is an independent Verilog consultant, specializing in providing comprehensive expert training on the Verilog HDL, SystemVerilog and PLI. Stuart is a co-author of the books "*SystemVerilog for Design*", "*Verilog-2001: A Guide to the New Features in the Verilog Hardware Description Language*" and is the author of "*The Verilog PLI Handbook*", as well as the popular "*Verilog HDL Quick Reference Guide*" and "*Verilog PLI Quick Reference Guide*". He has also authored a number of technical papers on Verilog and SystemVerilog, which are available at www.sutherland-hdl.com/papers. You can contact Stuart at stuart@sutherland-hdl.com.

visit the author's web page at www.sutherland-hdl.com

Mr. Don Mills has been involved in ASIC design since 1986. During that time, he has worked on more than 30 ASIC projects. Don started using top-down design methodology in 1991 (Synopsys Design Compiler 1.2). Don has developed and implemented top-down ASIC design flows at several companies. His specialty is integrating tools and automating the flow. Don works for Microchip Technology Inc. as an internal SystemVerilog and Verilog consultant. Don is a member of the IEEE Verilog and System Verilog committees that are working on language issues and enhancements. Don has authored and co-authored numerous papers, such as "SystemVerilog Assertions are for Design Engineers Too!" and "RTL Coding Styles that Yield Simulation and Synthesis Mismatches". Copies of these papers can be found at www.lcdm-eng.com. Mr. Mills can be reached at mills@lcdm-eng.com or don.mills@microchip.com.

visit the author's web page at www.lcdm-eng.com

Acknowledgments

The authors express their sincere appreciation to the contributions of several Verilog and SystemVerilog experts.

Chris Spear of Synopsys, Inc. suggested several of the verification related gotchas, provided the general descriptions of these gotchas, and ran countless tests for us.

Shalom Bresticker of Intel also suggested several gotchas.

Jonathan Bromley of Doulos, Ltd., ***Clifford Cummings*** of Sunburst Design, ***Tom Fitzpatrick*** of Mentor Graphics, ***Steve Golson*** of Trilobyte Systems, ***Gregg Lahti*** of Microchip Technology, Inc. and ***Chris Spear*** of Synopsys, Inc. provided thorough technical reviews of this book, and offered invaluable comments on how to improve the gotcha descriptions.

Steve Golson of Trilobyte Systems provided a wonderful foreword to this book

Lastly, we acknowledge and express our gratitude to our wives, ***LeeAnn Sutherland*** and ***Geri Jean Mills***, for meticulously reviewing this book for grammar and punctuation. If any such errata remain in the book, it could only be due to changes we made after their reviews.

Table of Contents

List of Gotchas

List of Gotchas

Foreword by Steve Golson

Some people collect baseball cards, old car magazines, or maybe rubber duckies.

I collect Verilog books.

It started back in 1989 with a looseleaf copy of "Gateway VERILOG-XL Reference Manual Version 1.5a" in a three-ring binder. Verilog was a bit simpler back then—it's hard to believe we actually designed chips using only one type of procedural assignment (nonblocking assigns were not part of the language yet). And we ran our simulations on a VAX, or maybe a fancy Apollo workstation.

Since then I've bought pretty much every Verilog book that came along. I've got a few synthesis books, and I'll pick up an occasional VHDL reference or maybe a text on the history of hardware description languages, but mostly it's Verilog. Dozens and dozens of books about Verilog.

There's a funny thing about most of these books though. After I leaf through them a few times, they sit on the shelf. I admit that it looks pretty impressive once you have an entire bookcase filled with Verilog books, but the discerning visitor will notice how fresh and new they all are. Unused. Unread. Useless.

I'm often disappointed to find very little information which is useful for the practicing engineer. What I'm looking for is a book I can use every day, a book that will help me get my chip out the door, on time and working.

Stu and Don have written such a book. I've known these guys for many years, and they have probably forgotten more Verilog than I've ever known. They have distilled their collective knowledge into this helpful and extremely useful book. Read it and you won't be disappointed.

If you are an old hand at Verilog try to pick out all the Gotchas that you have found the hard way. Smile and say to yourself "Oh yeah, I remember getting caught by that one!"

Those of you who are new to Verilog and SystemVerilog, welcome aboard! Here's your chance to learn from two of the leading experts in the field. And if you ever have a chance to take a training class from either of these gentlemen, don't hesitate to sign up. I guarantee you won't regret it.

Oh by the way, my favorite Gotcha is "Gotcha 65: Infinite for loops". Why? Well, I built a chip with that bug in it. Believe me, when a modeling error causes you to have broken silicon, you never forget why it happened. Back then I didn't have this book to help me, but you do! Keep this book close at hand, refer to it often, and may all your models compile and all your loops terminate.

Steve Golson
Trilobyte Systems
http://www.trilobyte.com

Chapter 1

Introduction, What Is A Gotcha?

This chapter defines what a "*gotcha*" is, and why programming languages allow gotchas. For the curious, the chapter also provides a brief history of the Verilog and SystemVerilog standards. The topics presented in this chapter include:

- What are Verilog and SystemVerilog
- The definition of a gotcha
- A brief description of the Verilog and SystemVerilog standards

What are Verilog and SystemVerilog?

The terms "Verilog" and "SystemVerilog" are sometimes a source of confusion because the terms are not used consistently in the industry. For the purposes of this book, "Verilog" and SystemVerilog are used as follows:

Verilog is a Hardware Description Language (HDL). It is a specialized programming language used to model digital hardware designs and, to a limited extent, to write test programs to exercise these models.

SystemVerilog is a substantial set of extensions to the Verilog HDL. A primary goal of these extensions is to enable modeling and verifying larger designs with more compact code. By itself, SystemVerilog is not a complete language; it is just a set of additions to the base Verilog language.

What is a Gotcha?

A programming "gotcha" is a language feature, which, if misused, causes unexpected—and, in hardware design, potentially disastrous—behavior. The classic example in the C language is having an assignment within a conditional expression, such as:

```
if (day=15)       /* GOTCHA! assigns value of 15 to day, then */
  do_mid_month_payroll;  /* if day is non-zero, do a payroll */
```

Most likely, what the programmer intended to code is `if (a==b)` instead of `if (a=b)`. The results are very different! This classic C programming Gotcha is not a syntax error; the code is perfectly legal. However, the code probably does not produce the intended results. If the coding error is not detected before a product is shipped, a simple bug like this could lead to serious ramifications in a product.

Just like any programming language, Verilog, and the SystemVerilog extensions to Verilog, have gotchas. There are constructs in Verilog and SystemVerilog that can be used in ways that are syntactically correct, but yield unexpected or undesirable results. Some of the primary reasons Verilog and SystemVerilog have gotchas are:

- Inheritance of C and C++ gotchas

  Verilog and SystemVerilog leverage the general syntax and semantics of the C and C++ languages. Verilog and SystemVerilog inherit the strengths of these powerful programming languages, but they also inherit many of the gotchas of C and C++. (Which raises the question, can the common C coding error such as `if (day=15)` be made in Verilog/SystemVerilog? The answer can be found in Gotcha 44 on page 99.)

- Loosely typed operations

  Verilog and SystemVerilog are *loosely typed* languages. As such, operations can be performed on any data type, and underlying language rules take care of how operations should be performed. If a design or verification engineer does not understand these underlying language rules, then unexpected results can occur.

- Allowance to model good and bad designs

  An underlying philosophy of Verilog and SystemVerilog is that engineers should be allowed to model and prove both what works correctly in hardware, and what will not work in hardware. In order to legally model hardware that does not work, the language must also permit unintentional modeling errors when the intent is to model designs that work correctly.

The Verilog and SystemVerilog standards

Verilog is an international standard Hardware Description Language. The official standard is **IEEE Std 1364-2005 Verilog Language Reference Manual** (LRM), commonly referred to as ***"Verilog-2005"***. The Verilog standard defines a rich set of programming and modeling constructs specific to representing the behavior of digital logic. The Verilog Hardware Description Language was first created in 1984. Verilog was designed to meet the needs of engineering in the mid 1980s, when a typical design was under 50,000 gates and ICs were based on 3 micron technology. As digital design size and technologies changed, Verilog evolved to meet new design requirements. Verilog was first standardized by the IEEE in 1995 (IEEE Std 1364-1995). In 2001, The IEEE released the Verilog-2001 standard (IEEE Std 1364-2001) which enhanced Verilog in several ways, such as synthesizable signed arithmetic on any vector size and re-entrant tasks and functions. The IEEE updated the Verilog standard in 2005, but no major modeling enhancements were added in this version. Instead, all enhancements to Verilog were documented in a separate standard, SystemVerilog.

SystemVerilog is a standard set of extensions to the Verilog-2005 Standard. These extensions are documented in a separate standard, **IEEE Std 1800-2005 SystemVerilog Language Reference Manual**, commonly referred to as ***"SystemVerilog-2005"***. The SystemVerilog extensions enable writing synthesizable models that are continuously increasing in size and complexity, as well as verifying these multi-million gate designs. SystemVerilog adds to Verilog features from the SUPERLOG, VERA C, C++, and VHDL languages, along with OVA and PSL assertions. SystemVerilog was first developed by Accellera, a consortium of companies that do electronic design and companies that provide Electronic Design Automation (EDA) tools. Accellera released a preliminary version of the extensions to Verilog in 2002, called ***SystemVerilog 3.0*** (3.0 to show that SystemVerilog was the next generation of Verilog, where Verilog-1995 was the first generation and Verilog 2001 was the second generation). In 2003, Accellera released ***SystemVerilog 3.1*** and in 2004 ***SystemVerilog 3.1a***. This latter Accellera standard was then submitted to the IEEE for full standardization.

The original intent was for the IEEE to fold the Accellera SystemVerilog extensions into the Verilog standard. At the insistence of EDA companies, however, the IEEE made the decision to temporarily keep the SystemVerilog extensions in a separate document to make it easier for EDA companies to implement the extensive set of new features in their Verilog tools.

The official standards for Verilog and SystemVerilog are:

- *"IEEE 1364-2005 standard for the Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey, 2002. ISBN 978-1-4020-7089-1.
- *"IEEE 1800-2005 standard for the SystemVerilog Hardware Description and Verification Language"*, IEEE, Pascataway, New Jersey, 2001. ISBN 0- 7381-4811-3.

For more details on the Verilog and SystemVerilog languages, refer to the books:

- *"The Verilog Hardware Description Language, 5th edition", by* Donald Thomas and Philip Moorby. Published by Springer, Boston, MA, 2002, ISBN 978-1-4020-7089-1.
- *"Verilog-2001: A Guide to the New Features in the Verilog Hardware Description Language", by* Stuart Sutherland. Published by Springer, Boston, MA, 2002, ISBN 978-0-7923-7568-5.
- *"SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling, Second Edition", by* Stuart Sutherland, Simon Davidmann and Peter Flake. Published by Springer, Boston, MA, 2006, ISBN 978-0-387-33399-1.
- *"SystemVerilog for Verification: A Guide to Learning the Testbench Language Features", by* Chris Spear. Published by Springer, Boston, MA, 2006, ISBN 978-0-387-27036-4.

There are many more excellent books on using Verilog and SystemVerilog.

Note that at the time this book was written, the IEEE had begun the process of merging the base Verilog language into the SystemVerilog Language Reference Manual, to create a single language and standard. In time, "Verilog" will disappear, and "SystemVerilog" will be a single, unified *"Hardware Design and Verification Language"*.

Chapter 2

Declaration and Literal Number Gotchas

Gotcha 1: Case sensitivity

Gotcha: The names in my code look correct and worked in my VHDL models, but Verilog/SystemVerilog gets errors about "undeclared identifiers".

Synopsis: Verilog and SystemVerilog are case-sensitive languages, whereas VHDL is a case-insensitive language.

An *identifier* in Verilog and SystemVerilog is the user-specified name of some object, such as the name of a module, wire, variable, or function. Verilog and SystemVerilog are case-sensitive languages, meaning that lowercase letters and uppercase letters are perceived as different in identifiers and in keywords. Keywords are always in all lowercase letters. User-created identifiers can use a mix of lowercase and uppercase letters, as well as numbers and the special characters _, $, and \ (the latter is an escape character).

This case sensitivity is often a gotcha to engineers learning Verilog/ SystemVerilog, especially those migrating from a case insensitive language such as VHDL. Even experienced engineers occasionally get caught making a case sensitivity error. Generally, this case sensitivity gotcha occurs when what is intended to be the same identifier is sometimes spelled using lowercase characters, and at other times using uppercase characters.

Note: the code examples in this chapter are contrived in order to illustrate each gotcha using small examples. In real design and verification code, these gotchas might not be as obvious or easy to debug.

The following example has three case sensitivity gotchas.

```
module FSM (...);

  enum logic [1:0] {WAIT, LOAD, READY} State, nState;

  ...

  always_comb begin
    case (state)                    // GOTCHA!
      WAIT:  nState = LOAD;         // GOTCHA!
      LOAD:  nState = READY;
      READY: nState = wait;         // GOTCHA!
    endcase
  end
endmodule: FSM
```

One gotcha in the preceding example is that the enumerated variable `State` is declared using a mix of uppercase and lowercase characters. Later in the code—possibly hundreds of lines after the declaration—a signal called `state` is referenced. These identifiers read the same in English, but, to a Verilog or SystemVerilog tool, they are very different names. *Gotcha!*

A second gotcha is that the enumerated label `LOAD` is in all uppercase letters. But later in the code an identifier called `LOAD` is referenced. Visually, these identifiers may appear to be the same, but to a Verilog/SystemVerilog tool they are very different names. The difference is that the enumerated label contains an uppercase letter "`O`" (pronounced "oh") in the name, whereas the reference in the code body contains the number "`0`" (or "zero") in the name. *Gotcha, again!*

The third gotcha in the example above is the enumerated label `WAIT`. While syntactically correct, this is a poor choice for an identifier name because there is a Verilog/SystemVerilog keyword `wait`. Later in the model, the `nState` variable is assigned the value of `wait`. This time, a Verilog/SystemVerilog tool won't report an error about an undeclared identifier. It will issue an error about a procedural **`wait`** statement appearing where an expression was expected. Identifier names with capital letters that are the same name as a keyword in all lowercase letters can make code harder to read, and can lead to coding errors that are difficult to see, and can cause obscure syntax error messages. *Gotcha, a third time!*

In the example above, the misspelled names will all result in compilation errors, which may be frustrating, but at least do not cause a gotcha that runs but does not behave as intended. Case sensitivity errors can also result in gotchas that will compile without an error, and will cause functional problems in the design. Gotcha 2 on page 10 shows an example of this.

How to avoid this Gotcha

The way to avoid this case-sensitivity gotcha is to adopt good naming conventions within a company, and then strictly enforce these conventions. The naming conventions used in this book are:

- Variable and net names are in lowercase.
  - User-defined type names end with `_t`.
  - Enumerated variable names end with `_e`.
  - Active-low signal names end with `_n`.
- Constant names and enumerated labels are in all uppercase (e.g. `LOAD`).
- Class definition names begin with a capital letter, followed by all lowercase letters (e.g. `class Packet;`).
- Names are chosen that would not conflict with a keyword if the language were case insensitive (e.g. `HOLD` instead of `WAIT`).

A corrected version of the previous example is:

```
module fsm (...);

  enum logic [1:0] {HOLD, LOAD, READY} state_e, nstate_e;

  ...

  always_comb begin
    case (state_e)                // OK, names match declarations
      HOLD:  nstate_e = LOAD;     // OK, names match declarations
      LOAD:  nstate_e = READY;
      READY: nstate_e = HOLD;     // OK, names match declarations
    endcase
  end
endmodule: fsm
```

Gotcha 2: Implicit net declarations

Gotcha: A typo in my design connections was not caught by the compiler, and only showed up as a functional problem in simulation.

Synopsis: Mis-typed identifiers may infer an implicit net instead of a syntax error.

What does a Verilog/SystemVerilog tool do when it encounters an undeclared identifier? The answer to this question depends on the context in which the undeclared identifier is used.

- If an undeclared identifier is used on the right- or left-hand side of a procedural assignment statement, then a compilation error occurs. For example (also see Gotcha 1 on page 7):

```
logic [7:0] foo;
initial foo = bar;   // ERROR: bar not declared
```

- If an undeclared identifier is used on the right-hand side of a continuous assignment statement, then a compilation error occurs.

```
logic [7:0] foo;
assign foo = bar;   // ERROR: bar not declared
```

- If an undeclared identifier is used on the left-hand side of a continuous assignment, then *an implicit net declaration is inferred, and no error or warning is reported.*

```
logic [7:0] foo;
assign bar = foo;   // GOTCHA: bar not declared, but no error
```

- If an undeclared identifier is used as a connection to an instance of a module, interface, program, or primitive, *then an implicit net is inferred, and no error or warning is reported.*

The last rule above can cause hard-to-find functional errors in a design, as shown in the following example of a netlist connecting two 1-bit adders together.

```
module adder (input  logic a, b, ci,
              output logic sum, co);
  ...
endmodule

module top;
  wire a, b, ci, s1, s2, c1;
  adder i1 (.a(a), .b(b), .ci(c1), .sum(s1), .co(c1) ); // GOTCHA!
  adder i2 (.a(a), .b(b), .ci(c),  .sum(s2), .co(co) ); // GOTCHA!
endmodule
```

One gotcha in this example is the declaration of `c1` ("see-one"), but the reference to `c1` ("see-ell") in the `i1` adder instance. Another gotcha is an undeclared identifier, `c`, in the `i2` adder instance. These typos are not syntax errors. Instead, they infer implicit nets in the design, causing functional errors that must be detected and debugged. *Gotcha!*

Why does Verilog/SystemVerilog allow this gotcha? Because the ability to have implicit data types automatically inferred can be useful, when used correctly. One of the benefits of implicit data types is that in a large, multi-million gate design that has thousands of interconnecting wires, it is not necessary to explicitly declare every wire.

How to avoid this Gotcha using Verilog

Some language-aware editors, such as Emacs with a Verilog mode[1], can auto-complete the connections in a Verilog netlist. This is a simple and practical way to avoid the gotcha of a typographical error.

Most engineers, including the authors, find that implicit nets is a strength of the Verilog language, despite the potential gotcha of a typographical error in a netlist showing up as a functional bug instead of a compilation error. The Verilog language does provide a mechanism to disable implicit data types. The way it is done, however, is controversial,. Some engineers feel it can introduce more gotchas then it might help avoid. This Verilog-based control is not discussed in this book.

How to avoid this Gotcha using SystemVerilog

SystemVerilog provides two convenient short cuts, *dot-name* and *dot-star*, for connecting nets to instances of modules, programs, and interfaces. These shortcuts remove the repetition in named port connections.

- The dot-name shortcut explicitly names the port to which a connection is being made, but infers that a net of the same name is connected.
- The dot-star shortcut automatically infers that ports and signals with the same name are connected.

The following example illustrates all three connections styles: fully explicit, dot-name partially inferred, and dot-star fully inferred connections.

1. One popular Verilog mode for Emacs is available at www.verilog.com.

```
module adder (input  logic a, b, ci,
              output logic sum, co);
  ...
endmodule

module top;
  wire a, b, ci, s1, s2, c1;
  adder i1 (.a, .b, .ci, .sum(s1), .co(c1) );  // .name connections
  adder i2 (.sum(s2), .ci(c1), .* );           // .* connections
endmodule
```

By reducing the number of times a signal name must be typed, the possibility of typographical errors is also reduced. The dot-name and dot-star shortcuts also require that all nets be explicitly declared. A typo in the netlist or in the net declarations will not infer an implicit wire when using the shortcuts. Such typos will be compilation errors instead of functional problems in simulation.

Programmers who use the Emacs editor can take advantage of a nice feature in some Verilog/SystemVerilog modes for Emacs which will automatically expand dot-star inferred port connections to its inferred explicit connections. This feature requires that the dot-star be the last item in the port connection list, as shown in the example above.

Gotcha 3: Default of 1-bit internal nets

Gotcha: In my netlist, only bit zero of my vector ports get connected.

Synopsis: Undeclared internal connections within a netlist infer 1-bit wires, even if the port to which the net is connected is a vector.

Verilog and SystemVerilog have a convenient shortcut when modeling netlists, in that it is not necessary to declare all of the interconnecting nets. Undeclared identifiers used as port connections default to a **`wire`** net type (see Gotcha 2 on page 10). In a netlist with hundreds or thousands of connections, implicit wires can significantly simplify the Verilog/SystemVerilog source code.

The vector size of implicit nets is determined from local context. If the undeclared signal is also a port of the module containing the signal, then the implicit net will be the same size as the containing module's port. If the undeclared signal is only used internally in the containing module, then a 1-bit net is inferred. Verilog and SystemVerilog do not look at the port sizes of what the signal is connected to in order to determine the implicit net type size.

The following top-level netlist connects signals to a 4-to-1 multiplexer. The data inputs and outputs of the mux are 8 bits wide. The `select` input is 2 bits wide. No data types are declared in the top-level netlist. Therefore, implicit wires will be inferred for all connections.

```
module top_level
(output [7:0] out,          // 8-bit port, no data type declared
 input  [7:0] a, b, c, d    // 8-bit ports, no data type declared
);

  mux4 m1 (.y(out),         // out infers an 8-bit wire type
           .a(a),           // a, b, c, d infer 8-bit wires
           .b(b),
           .c(c),
           .d(d),
           .sel(select) );  // GOTCHA! select infers 1-bit wire
  ...
endmodule

module mux4
(input  logic [1:0] sel,          // 2-bit input port
 input  logic [7:0] a, b, c, d,   // 8-bit input ports
 output logic [7:0] y             // 8-bit output port
);
  ...
endmodule
```

In the example above, the top-level netlist connects a signal called `select` to the `sel` port of `mux4`. Within `mux4`, the `sel` port is 2 bits wide. When inferring undeclared nets in the `top_level` module, however, Verilog and SystemVerilog only look within the local context of `top_level`. There is nothing within `top_level` from which to infer the size of `select`. Therefore, `select` defaults to a 1-bit wire. *Gotcha!*

Most Verilog/SystemVerilog tools will generate compilation warning messages for this gotcha, reporting size mismatches in port connections. *Engineers should not ignore these warnings!* Almost without exception, warnings about size mismatches in port connections indicate unintentional errors in a netlist.

How to avoid this Gotcha

To avoid this gotcha, all internal nets and variables that are more than 1-bit wide must be explicitly declared. An internal signal is one that is only used within the module, program or interface, and is not a port of that module, program or interface through a port.

The SystemVerilog dot-name and dot-star port connection shortcuts can help in avoiding this gotcha. These shortcuts will not infer undeclared nets. Further, these shortcuts will not infer connections that do not match in size. Gotcha 2 on page 10 explains the dot-name and dot-star port connection shortcuts.

The following example, which uses dot-star implicit port connections, will cause a compilation error instead of a functional gotcha.

```
module top_level
(output logic [7:0] out,          // 8-bit port
 input  logic [7:0] a, b, c, d    // 8-bit ports
);

  mux4 m1 (.y(out), .* );  // ERROR: no net declared for sel port

endmodule

module mux4
(input  logic [1:0] sel,           // 2-bit input port
 input  logic [7:0] a, b, c, d,    // 8-bit input ports
 output logic [7:0] y              // 8-bit output port
);
```

Gotcha 4: Single file versus multi-file compilation of $unit declarations

Gotcha: My models compile OK, and the models from another group compile OK; but when compiled together, I get errors about multiple declarations.

Synopsis: Separate file compilation has separate $unit declaration name spaces. Multi-file compilation has a single $unit compilation name space.

`$unit` is a declaration space that is visible to all design units that are compiled together. The purpose of `$unit` is to provide a place where design and verification engineers can place shared definitions and declarations. Any user-defined type definition, task defintion, function definition, parameter declaration or variable declaration that is not placed inside a module, interface, test program, or package is automatically placed in `$unit`. For all practical purposes, `$unit` can be considered to be a predefined package name that is automatically wildcard imported into all modeling blocks. All declarations in `$unit` are visible without having to specifically reference `$unit`. Declarations in `$unit` can also be explicitly referenced using the package scope resolution operator. This can be necessary if an identifier exists in multiple packages. An example of an explicit reference to `$unit` is:

```
// following declaration is in $unit
typedef enum logic [1:0] {RESET, HOLD, LOAD, READY} states_t;

module chip (...);
  ...
  $unit::states_t state_e, nstate_e; // OK: definition in $unit
```

A gotcha with `$unit` is that these shared definitions and declarations can be scattered throughout multiple source code files, and can be at the beginning or end of a file. At best, this is an unstructured, spaghetti-code modeling style, that can lead to design and verification code that is difficult to debug, difficult to maintain, and nearly impossible to reuse. Worse, `$unit` definitions and declarations scattered across multiple files can result in name resolution conflicts. Say, for example, that a design has a `$unit` definition of an enumerated type containing the label `RESET`. By itself, the design may compile just fine. But then, what happens if an IP model is added to the design that also contains a `$unit` definition of an enumerated type containing a label called `RESET`? The IP model also compiles just fine by itself, but when compiled along with the design's `$unit` declarations, there is a name conflict. There are now two definitions in the same name space trying to reserve the label `RESET`. *Gotcha!*

How to avoid this Gotcha

Use packages for shared declarations, instead of `$unit`. Packages serve as containers for shared definitions and declarations, preventing inadvertent spaghetti code. Packages also have their own name space, which will not collide with definitions in other packages. There can still be name collision problems if two packages are wildcard imported into the same name space. This can be prevented by using explicit package imports and/or explicit package references, instead of wildcard imports (see Gotcha 9 on page 27 for examples of wildcard and explicit imports).

Gotcha 5: Local variable declarations

Gotcha: I get compilation errors on my local variable declarations, but the declaration syntax is correct.

Synopsis: Verilog and SystemVerilog allow local variables to be declared within a statement group, but require all declarations to come before any procedural code.

Verilog and SystemVerilog allow local variables to be declared within a **begin...end** or **fork...join** statement group. However, just as in C, variables must be declared before any programming statements. This is different from the context of a module, interface or program block, where variables can be declared anywhere, so long as they are declared before they are referenced. This subtle difference can be a gotcha.

```
package Transaction;
  ...
endpackage

package Extented_trans extends Transaction;
  ...
endpackage

initial begin
  Transaction tr = new;       // declaration with constructor
  bit status;                 // declaration
  status = tr.randomize;      // procedural statement
  Extended_trans etr = new;   // GOTCHA! declaration after statement
  ...
end
```

How to avoid this Gotcha

Two modeling styles can be used to fix this local variable gotcha. One style is to declare all variables at the top of the block, before any procedural statements. For example:

```
initial begin  // move all declarations to top of the block
  Transaction tr = new;       // declaration with constructor
  Extended_trans etr = new;   // OK, declaration before statement
  bit status;                 // declaration
  status = tr.randomize;      // procedural statement
  ...
end
```

The second style is to create a new **begin...end** block within the procedure to localize the scope of a variable, as follows:

```
initial begin
  Transaction tr = new;          // declaration with constructor
  bit status;                    // declaration
  status = tr.randomize;         // procedural statement
  begin
    Extended_trans etr = new;    // OK, local declaration
    ...
  end
  ...
end
```

Gotcha 6: Escaped names in hierarchical paths

Gotcha: I get weird compiler errors when I try to reference a design signal with an escaped name from my testbench.

Synopsis: Escaped identifiers in a hierarchical path require embedded spaces in the path.

An *identifier* in Verilog and SystemVerilog is the name of some object, such as the name of a module, the name of a wire, the name of a variable, or the name of a function. The legal characters in an identifier are alphabetic characters, numbers, underscore, or dollar sign. All other characters, such as +, -, (,), [, and], are illegal in an identifier name.

Verilog and SystemVerilog allow these illegal characters to be used in a name by escaping the identifier. A name is escaped by preceding the name with a back slash (\ ) and terminating the name with a whitespace character. A whitespace character is a space, tab, carriage return, line feed, form feed, or an end-of-file. Some examples of escaped identifiers are:

```
module \d-flop (output logic q, \q- ,
                input  logic \d[0] ,ck, \rst- );
  ...
endmodule
```

Observe in the above example that a whitespace character must be used before the commas that follow `\q-` and `\d[0]`. The whitespace terminates the escaped name, so that the comma is a separator between port names. If there were no whitespace before the comma, then the comma would become part of the escaped name. A whitespace is also required between the last escaped name, `\rst-`, and the closing parenthesis.

Potential gotcha: An escaped name with square brackets, such as `\d[0]`, can be confusing. It is a name for 1-bit wire. `\d[0]` is not a bit select of vector called `\d`. This difference is illustrated with the following declarations:

```
logic [7:0] \a ;        // 8-bit vector called \a
logic       \b[0] ;     // 1-bit signal called \a[0]

buf b1 (\b[0] , \a[0] );   // GOTCHA: infers a net called \a[0]
buf b2 (\b[0] , \a [0] );  // CORRECT: bit select of vector \a
```

Observe that a whitespace was required after `\a` and before the `[0]` in order to do a bit select of a vector that has an escaped name. The use of `\a[0]` in buffer `b1` will not be a syntax error. An implicit net called `\a[0]` will be declared. *Gotcha!* (see Gotcha 2 on page 10 for a description of implicit net gotchas).

Another *gotcha* can occur when an escaped identifier is used as part of a hierarchical path. For example:

```
module test (output [7:0] q, input [7:0] d, input ck, rst_n);
  \d-flop \d-0 (.q(q[0]),.\q~(), .\d[0](d[0]),     // GOTCHA!
                .ck(ck), .\rst-(rst_n));           // GOTCHA!

  initial begin
    $display("d = %b", test.\d-0.\d[0] );          // GOTCHA!
  end
endmodule
```

This example will get compilation errors, but the errors will probably not indicate the real problem. An escaped name must be terminated by a whitespace. In the code above, `\d-flop` and `\d-0` are correctly followed by a whitespace. The escaped name `\q~` is not followed by a whitespace, making all text following it part of the escaped name. A Verilog/SystemVerilog tool will see the name, "`\q~(),`", which is not what was intended. Since the parenthesis and comma were also escaped, the tool will report a compilation error due to those missing tokens.

The next line also has a gotcha. The escaped name `\rst-` is not terminated by a whitespace. The first whitespace comes after the semicolon at the end of the module instance. Therefore, the name a Verilog/SystemVerilog tool will see is "`\rst-(rst_n));`". The tool will not find a closing parenthesis and semicolon to end the module instance, and will probably report an obscure error message about the **`initial`** keyword two lines further down in the code.

The `$display` statement will also get a compilation error that might not be obvious. The display statement uses a hierarchical path to print the value of `\d[0]` inside the flip-flop model. However, the instance name is an escaped name, which must be terminated by a whitespace. What the tool sees is a single name of "`\d-0.\d[0]`".

How to avoid this Gotcha

Escaped names must be terminated by a whitespace, even when part of an explicit port name or a hierarchical path. The correct code for the example above is:

```
module test (output [7:0] q, input [7:0] d, input ck, rst_n);
  \d-flop \d-0 (.q(q[0]), .\q~ (), .\d[0] (d[0]),  // CORRECT
                .ck(ck),  .\rst- (rst_n));          // CORRECT

  initial begin
    $display("d = %b", test.\d-0 .\d[0] );  // CORRECT
  end
endmodule
```

The whitespace in the middle of a hierarchical path, as in the `$display` statement above, might look like it breaks the hierarchical path into two identifiers, but the terminating whitespace is ignored, which, in effect, concatenates the two names into one name.

The authors recommend avoiding purposely using escaped names in a design, especially an escaped name with square brackets in the name, as in the contrived example above. Unfortunately, life is not that simple. Not all identifiers are user-defined. Some software tools, especially synthesis tools, often create tool-generated identifier names in the Verilog or SystemVerilog code. And, as ugly as these tool-generated identifiers looks to users, these tools often put square brackets in escaped identifiers.

Gotcha 7: Hierarchical references to automatic variables

Gotcha: I get compilation errors when my testbench tries to print out some signals in my design, but other signals can be printed without a problem.

Synopsis: Automatic variables cannot be referenced using hierarchical paths. They are also not dumped to VCD files.

Verilog has automatic tasks and functions, which dynamically allocate storage each time they are called, and automatically free that storage when they exit. SystemVerilog adds many more types of automatic storage, including classes for object-oriented programming, dynamically sized arrays, queues, and strings. These dynamically allocated types are intended for—and are very important in—modeling test programs using modern verification methodologies.

The following two examples are nearly identical. Both examples use a task to calculate a result. The task contains an intermediate variable, `s1`. The task is called twice, and the intermediate `s1` variable can have different values for each call.

The only difference between the two models is that `math_static` uses static storage for the local variable, whereas `math_auto` uses automatic storage. Automatic storage would avoid possible conflicts between the two task calls, because each call to the task is allocated a unique copy of the `s1` variable.

```
module math_static ( ... );                    // use static storage
  task math ( ... );
    logic [15:0] s1;                           // local variable
    ...
  endtask
  always_comb begin
    math(a1, b1, c1, o1);
    math(a2, b2, c2, o2);
  end
endmodule

module automatic math_auto (...);              // use automatic storage
  task math ( ... );
    logic [15:0] s1;                           // local variable
    ...
  endtask
  always_comb begin
    math(a1, b1, c1, d1, o1a, o1b);
    math(a2, b2, c2, d2, o2a, o2b);
  end
endmodule
```

The following testbench is used to verify the two modules. As part of the verification, coverage, and possible debug of the design, the intent in this testbench is to use a hierarchical path to look at the values of the `s1` variable in each module's `math` task.

```
module top;
  ...
  test        t1 (.*);    // connect testbench to design
  math_static m1 (.*);
  math_auto   m2 (.*);
endmodule

program automatic test;
  ...
  initial begin
    ... // apply stimulus
    $display (" math_static s1 = %h", top.m1.math.s1  // OK
    $display (" math_auto   s1 = %h", top.m2.math.s1  // GOTCHA!
  end
endprogram
```

Everything in the testbench excerpt looks correct, and accessing internal data in the design, in order to do verification and debug, is certainly reasonable. Hierarchical references allow the verification code to evaluate, and possibly stimulate, logic deep down in the design, without having to pull those internal signals up to the testbench through multiple layers of extra, verification only, module ports.

Unfortunately, the testbench above will not compile, due to the hierarchical references to s1 in module instance `m2`. *Gotcha!*

The gotcha happens because it is illegal to hierarchically reference automatic variables. The reason is that hierarchical paths are static in nature, whereas automatic variables come and go during simulation. This limitation places a burden on the verification engineer. Before using a hierarchical path to reference a variable, the verification engineer must first examine the source code, to determine whether the variable is static or automatic. *Gotcha!*

A closely related gotcha is that the Verilog and SystemVerilog value changes on automatic variables are not dumped to a *Value Change Dump* (VCD) file. VCD files are used as an input to waveform displays and other design analysis tools. However, only static nets and variables are dumped to VCD files. When simulation results are analyzed, important information might be missing. *Gotcha, again!*

How to avoid this Gotcha

This gotcha cannot be completely avoided, but it can be minimized. A coding guideline is to only use automatic storage in the testbench, and for tasks and functions declared in `$unit`, packages and interfaces. In this way, test programs will be able to hierarchically access most design data. It is also helpful to use naming conventions that make automatic variables obvious in the source code.

There is no workaround for this VCD limitation. Proprietary waveform tools that do not use VCD files, however, might not have this limitation.

Coding guidelines to help avoid static versus automatic storage gotchas

Programs should be declared as **`program automatic`**. This makes the default storage automatic, which is more like the C and C++ programming languages. Note, however, that this only changes the default for variables declared within tasks, functions, and procedural blocks. Variables declared at the program block level will still be static by default. The following parallel examples illustrate the effects of declaring a program as **`automatic`**.

```
program test;
 int a;              //a is static

 task t (int b);     //b is static
  int c;             //c is static
  ...
 endtask

 initial begin
  logic d;           //d is static
  ...
 end
endprogram
```

```
program automatic test;
 int a;              //a is static

 task t (int b);     //b is automatic
  int c;             //c is automatic
  ...
 endtask

 initial begin
  logic d;           //d is automatic
  ...
 end
endprogram
```

Packages should be declared as **`package automatic`**. Task and function definitions in a package are often shared between several programs, modules and interfaces, and therefore should be automatic to avoid problems of sharing storage between multiple callers. Gotcha 43 on page 96 describes RTL design problems that can occur when static storage in a package is shared with multiple callers.

Modules should *not* be declared as automatic. Tasks, functions and local variables in a module that are automatic will not be accessible hierarchically, which limits the ability to debug design problems. If automatic storage is needed in a module, such as for a recursively called function, the local variable, task or function should be explicitly declared as automatic. This both documents the exception, and permits debug access to the rest of the module.

Gotcha 8: Hierarchical references to variables in unnamed blocks

Gotcha: With Verilog, my testbench could print out local variables in a begin...end block, but with SystemVerilog I get compilation errors.

Synopsis: Variables declared in an unnamed scope have no hierarchical paths.

Verilog allows local variables to be declared in named **begin**...**end** and **fork**...**join** blocks. These variables can be referenced hierarchically, using the block name as part of the hierarchical path.

The following example defines a **begin**...**end** block called `loop`, with a local variable called `temp`. A testbench module hierarchically references `temp` to print out its value.

```
module chip_vlog_style (...);
  ...
  always @(posedge clk)
    for (i=0; i<=15; i=i+1) begin: loop      // named block
      integer temp;                          // local variable
      ...
    end
endmodule

module test;
  ...
  chip_vlog_style dut (...);
  initial $display ("temp = %0d", test.dut.loop.temp); // OK
endmodule
```

An second advantage of local variables is that they can prevent the inadvertent gotcha of having multiple **initial** or **always** procedural blocks write to the same variable (see Gotcha 66 on page 145).

SystemVerilog simplifies Verilog by allowing local variables to be declared in unnamed **begin**...**end** and **fork**...**join** blocks. This simplification comes with a potential *gotcha*!

```
module chip_sv_style (...);
  ...
  always_ff @(posedge clk)
    for (int i=0; i<=15; i++) begin      // unnamed block
      integer temp;                      // local variable
      ...
    end
endmodule
```

```
program automatic test;
  ...
  chip_sv_style dut (...);
  initial $display ("temp = %0d", test.dut.temp);   // GOTCHA!
endmodule
```

Local variables in unnamed blocks cannot be accessed from the testbench for verification or debugging. Binding assertions to a design to verify functionality or coverage involving the local variable is also not allowed. Nor will local variables in unnamed blocks show up in Value Changed Dump (VCD) files used by many waveform displays. *Gotcha!*

The gotcha is that variables declared in an unnamed block cannot be referenced hierarchically, because there is no named scope to reference in the hierarchical path. This is the same limitation that exists for automatic variables, as discussed in Gotcha 7 on page 22.

A similar gotcha exists with **`for`** loop control variables defined as part of the **`for`** loop, which is shown in Gotcha 67 on page 147.

How to avoid this Gotcha

If a local variable needs to be referenced hierarchically, or dumped to a waveform file, declare local variables in named **`begin`**...**`end`** or **`fork`**...**`join`** blocks as shown in the first example in this gotcha description. Use unnamed blocks if there is a reason to protect the local variable from code outside of the block, including the testbench and VCD waveform files.

Gotcha 9: Hierarchical references to imported package items

Gotcha: My design can use imported package items just fine, but my testbench cannot access the items for verification.

Synopsis: Imported identifiers must be referenced using the scope resolution operators instead of hierarchically.

SystemVerilog packages allow type definitions and other information to be declared in a shared space. Package definitions can be imported into other design blocks to give access to those shared definitions. Verification of imported items is handled differently than verification of data declared locally within a design block. The following example has a gotcha in the testbench program.

```
package chip_types;
  typedef enum logic [1:0] {RESET, HOLD, LOAD, READY} states_t;
endpackage: chip_types

module chip (...);
  import chip_types::*;  // wildcard import definitions in package
endmodule: chip

module top;
  chip chip (...);  // instance of design that uses the package
  test test (...);  // instance of test program
endmodule: top

program automatic test (...);
  ...
  initial begin
    $display ("RESET is %b", top.chip.RESET);  // GOTCHA! illegal
endprogram: test
```

In the example above, the `test` program uses a hierarchical path to access the value of `RESET` in the scope in which `RESET` is imported and used. This results in an error. Hierarchical paths are used to access signals where they are declared. The enumerated label `RESET` in the example above is not declared in module `chip`. It is only imported into `chip`. When package items are imported into a module, interface, or test program, these items are *not* locally defined within that scope. These imported items cannot be referenced hierarchically.

How to avoid this Gotcha

External references to package items are done using the package name followed by the scope resolution operator (`::`) instead of hierarchical paths. For example:

```
  $display ("RESET is %b", chip_types::RESET);  // OK
```

Gotcha 10: Importing enumerated types from packages

Gotcha: I imported an enumerated type from a package, but I cannot access the labels defined by the enumerated type.

Synopsis: Importing an enumerated type definition does not import its enumerated labels.

Enumerated type definitions defined in a package can be explicitly imported into a design or verification block. For example:

```
package chip_types;
  typedef enum logic [1:0] {RESET, HOLD, LOAD, READY} states_t;
endpackage

module chip (...);
  import chip_types::states_t;  // explicit import of states_t type
  states_t state_e, nstate_e;

  always_ff @(posedge clock, negedge reset_n)
    if (!reset_n) state_e <= RESET;  // GOTCHA: RESET not imported
    else          state_e <= nstate_e;
  ...
endmodule
```

The package contains a user-defined enumerated type called `states_t`, which has the value labels `RESET`, `HOLD`, `LOAD` and `READY`. The `chip` module imports `states_t` type from the package. When this example is read in by a software tool, a compilation error will result, stating that `RESET` has not been defined. The reason is that the import of `states_t` only imports the name `states_t`. It does not import the labels that `states_t` uses. *Gotcha!*

How to avoid this Gotcha

One way to avoid this gotcha is to explicitly import each enumerated label along with the enumerated type definition.

```
import chip_types::states_t;  // explicit import of states_t type
import chip_types::RESET;     // and its labels
import chip_types::HOLD;
import chip_types::LOAD;
import chip_types::READY;
```

A second way to avoid this gotcha with a wildcard import of the package, which will make both the enumerated type definition and its enumerated labels visible.

```
import chip_types::*; // wildcard import of package declarations
```

Wildcard imports have a gotcha if multiple packages are used in a design block, as discussed in Gotcha 11 on page 29.

Gotcha 11: Importing from multiple packages

Gotcha: I get errors when I try to wildcard import multiple packages, but I can wildcard import each package separately without any errors.

Synopsis: Wildcard imports from multiple packages can cause name collisions.

Large designs, and designs that use IP models, will likely have multiple packages. For convenience, a package can be wildcard imported into a design, which can save having to explicitly import each item from each package. However, wildcard imports of multiple packages can lead to a gotcha, as illustrated in the following example.

```
package chip_types;
  typedef enum logic [1:0] {HOLD, LOAD, READY} states_t;
endpackage

package bus_types;
  localparam HOLD = 32;
  ...
endpackage

module chip (...);
  import chip_types::*;  // wildcard import of a package
  import bus_types::*;   // wildcard import of another package

  states_t state_e, nstate_e;

  always_ff @(posedge clock, negedge reset_n)
    if (!reset_n) state_e <= HOLD;   // GOTCHA: HOLD has multiple
    else          state_e <= nstate_e;  // definitions
  ...
```

The gotcha in the example above is that both packages contain an identifier named HOLD. Wildcard importing both packages will result in a compilation error due to the name conflict.

How to avoid this Gotcha

The gotcha with wildcard package imports occurs when there are identifiers common to more than one package. To avoid this gotcha, explicitly import any duplicate identifiers from the desired package. Wildcard imports of other packages will not import identifiers that have been explicitly declared or explicitly imported in the local scope.

```
  import chip_types::*;  // wildcard import of a package
  import bus_types::*;   // wildcard import of another package
  import chip_types::HOLD;  // explicit import of HOLD
```

Gotcha 12: Default base of literal integers

Gotcha: Some branches of my case statement are never selected, even with the correct input values.

Synopsis: Literal integers have a default base that might not be what is intended.

Literal integers in Verilog and SystemVerilog can be specified as a *simple decimal integer* (e.g. `5`) or as a *based integer* (e.g. `'h5`). A based literal integer is specified using the following syntax:

```
<size>'s<base><value>
```

Where:

- `<size>` is optional. If given, it specifies the total number of bits represented by the literal integer. If not given, the default size, per the Verilog/SystemVerilog standard is "at least" 32 bits.
- `s` is optional. If given, it specifies that the literal integer should be treated as a signed value in operations. If not given, the default is unsigned. (The signed specifier was added to Verilog as part of the Verilog-2001 standard.)
- `<base>` is required, and specifies whether the value is in binary, octal, decimal, or hex.
- `<value>` is required, and specifies the literal integer value.

A simple literal integer (e.g. `5`) defaults to a decimal base. To use a binary, octal, or hex value, a based-literal integer must be specified (e.g. `'h5` or `2'b10`). The base options are represented using `b`, `o`, `d`, or `h` for binary, octal, decimal and hex, respectively. The base specifier can be either lowercase or uppercase (i.e. `'h5` and `'H5` are the same).

The following example of a 4-to-1 multiplexer illustrates a common gotcha when an engineer forgets that a simple integer number is a decimal value.

```
logic [1:0] select;     // 2-bit vector

always_comb begin
  case (select)         // intent is for a 4-to-1 MUX behavior
    00: y = a;
    01: y = b;
    10: y = c;          // GOTCHA! This branch is never selected
    11: y = d;          // GOTCHA! This branch is never selected
  endcase
end
```

This gotcha fits nicely with the joke that only engineers laugh at: *"There are 10 types of people in the world, those that know binary, and those that don't".*

The previous example may look reasonable, and it is syntactically correct. Since the default base of a simple integer is decimal, however, the case select values "10" and "11" are *ten* and *eleven*. The 2-bit `select` signal can only contain the values 0, 1, 2 and 3. The `select` values of 2 and 3 will not match any of the case items, and the branches for the case items of decimal "10" and "11" will never be executed. This is not a syntax error. The problem shows up in simulation as a functional failure which can be difficult to detect and debug. *Gotcha!*

How to avoid this Gotcha using Verilog

With Verilog, the easiest coding style for detecting that there is a design problem is to add a default branch to the case statement that detects when none of the expected branches evaluate as true. For example:

```
case (select)          // intent is for a 4-to-1 MUX behavior
  00: y = a;
  01: y = b;
  10: y = c;           // GOTCHA! This branch is never selected
  11: y = d;           // GOTCHA! This branch is never selected
  default: $display("select value of %b not decoded", select);
endcase
```

How to avoid this Gotcha using SystemVerilog

SystemVerilog adds a **unique** modifier for **case** statements. This extension to Verilog provides an easy way to detect this gotcha.

```
unique case (select)   // intent is for a 4-to-1 MUX behavior
  00: y = a;
  01: y = b;
  10: y = c;           // GOTCHA! This branch is never selected
  11: y = d;           // GOTCHA! This branch is never selected
endcase
```

The `unique` modifier reports an error if two or more case select items are true at the same time, or if no case select items are true. The example above becomes an error, rather than a functional bug in the code.

The following example codes the case statement correctly.

```
unique case (select)   // intent is for a 4-to-1 MUX behavior
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;        // OK, this branch can be selected
  2'b11: y = d;        // OK, this branch can be selected
endcase
```

Caution! The **unique** modifier is not appropriate for every case statement. See Gotcha 35 on page 79 for additional discussion on using **unique case**.

Gotcha 13: Signedness of literal integers

Gotcha: My incrementor model sometimes gets incorrect values when I increment using a literal 1'b1.

Synopsis: Unbased literal integers default to signed. Based literal integers default to unsigned.

Literal integers in Verilog and SystemVerilog can be specified as a *simple decimal integer* (e.g. `5`) or as a *based integer* (e.g. `'h5`). A simple literal integer (e.g. `5`) defaults to a *signed* value, and cannot be specified as unsigned. A based literal integer (e.g. `'h5`) defaults to an *unsigned* value, unless explicitly specified as signed (e.g. `'sh5`). A based literal integer is specified as follows:

```
<size>'s<base><value>
```

- `<size>` is optional. If given, it specifies the total number of bits represented by the literal integer. If not given, the default size, per the Verilog/SystemVerilog standard is "at least" 32 bits.
- `s` is optional. If given, it specifies that the literal integer should be treated as a signed value in operations. If not given, the default is unsigned. (The signed specifier was added to Verilog as part of the Verilog-2001 standard.)
- `<base>` is required, and specifies whether the value is in binary, octal, decimal, or hex.
- `<value>` is required, and specifies the literal integer value.

The signedness of a literal value affects several types of operations. Unexpected operation results will occur if an engineer forgets—or is not aware of—the different signedness of a simple literal integer versus a based literal integer.

Are the following two signed counter statements the same?

```
byte in;          // signed 8-bit variables
int out1, out2;   // signed 32-bit variables

initial begin
  in = -5;
  out1 = in + 1;      // OK:      -5 + 1   = -4 (literal 1 is signed)
  out2 = in + 1'b1;   // GOTCHA: -5 +1'b1 = 252 (1'b1 is unsigned)
end
```

How to avoid this Gotcha

To avoid this gotcha, engineers need to know, properly use, and take advantage of Verilog's rich set of signed and unsigned literal values. Signed arithmetic is discussed in more detail in Gotcha 47 on page 108.

Gotcha 14: Signed literal integers zero extend to their specified size

Gotcha: When I specify a signed, sized literal integer with a negative value, it does not sign extend.

Synopsis: Too small a size truncates the most-significant bits of a value. Too large a size left-extends a value with 0, x, or z, but does not sign extend.

Literal integers can be specified as *unsized integers* (e.g. `'h5`) or as *sized integers* (e.g. `16'h5`). The syntax for specifying literal integers is:

```
<size>'s<base><value>
```

Where:

- `<size>` is optional. If given, it specifies the total number of bits represented by the literal integer. If not given, the default size, per the Verilog/SystemVerilog standard is "at least" 32 bits.
- `s` is optional. If given, it specifies that the literal integer should be treated as a signed value in operations. If not given, the default is unsigned. (The signed specifier was added to Verilog as part of the Verilog-2005 standard.)
- `<base>` is required, and specifies whether the value is in binary, octal, decimal, or hex.
- `<value>` is required, and specifies the literal integer value.

It is legal to specify a mismatch between the size and the number of bits represented by the value. Verilog/SystemVerilog has built-in rules for how to handle a mismatch. If these rules are not understood, engineers might be caught by a gotcha. The following example specifies a value that appears to be a negative value, but is zero extended as if an unsigned value.

```
logic signed [11:0] a;
a = 12'shFF; // GOTCHA! signed value hex FF does not represent -1
             // 8-bit FF value extends to 12-bit 000011111111
             // WHY?
```

The Verilog/SystemVerilog rules for a mismatch between the size and the value are explained in the following paragraphs.

Size smaller than value rule. If the bit size specified is fewer bits than the value, then the left-most bits of the value are truncated. This can be a gotcha when the value is signed, because the sign bit will be truncated as well. In the following example, a negative `15` (8-bit `F1` hex) is truncated to 4-bits wide, becoming a positive `1`. *Gotcha!*

```
logic signed [7:0] b;
b = -4'sd15;   // GOTCHA! 11110001 (-15) is truncated to 0001 (+1)
```

Size greater than value rule. When the bit size specified is more bits than the value, the value will be expanded to the size by left-extending. The fill value used to left extend is based on the most-significant bit of the value, as follows:

- If the most-significant bit of the specified value is a 0 or a 1, then the value is left extended with zeros.
- If the most-significant bit is an X, then the value is left extended Xs.
- If the most-significant bit is a Z, then the value is left extended Zs.

This left extension can be useful. For example, it is not necessary to specify the value of each and every bit of a vector to reset the vector to zero or to set the vector to high impedance.

```
64'h0;   // fills all 64 bits with 0
64'bZ;   // fills all 64 bits with Z
```

When the bit-size is larger than the value, and the value is signed, the expansion rules above do *not* sign-extend a signed value. Even if the number is specified to be signed, and the most-significant bit is a 1, the value will still be extended by 0's. For example:

```
logic signed [11:0] a, b;

initial begin
  a = 12'sh3c; // OK,      signed 00111100 expands to 000000111100
  b = 12'so74; // GOTCHA! signed   111100 expands to 000000111100
end
```

The hex value `3c` is an 8-bit value that does not set its most-significant bit. When the value is expanded to the 12-bit size, the expansion zero-extends, regardless of whether the literal integer is signed or unsigned. This is as expected.

The octal value `74` is the same bit pattern as a hex `3c`, but is a 6-bit value with its most-significant bit set. The expansion to the 12 bit size still zero-extends, rather than sign-extends. *Gotcha!*.

The subtlety in the preceding example is that the sign bit is not the most-significant bit of the value. It is the most-significant bit of the specified size. Thus, to specify a negative value in the examples above, the value must explicitly set bit 12 of the literal integer. Negating a positive value will set its sign bit.

```
12'shFFB     // expands to 111111111011, which is -5 decimal
-12'sh5      // expands to 111111111011, which is -5 decimal
```

How to avoid this Gotcha

The way to avoid these gotchas is to be sure that the bit size specified is the same size as the value, especially when using signed values. Some tools, such as lint tools (coding style checkers), will detect a size versus value mismatch.

Gotcha 15: Literal integer size mismatch in assignments

Gotcha: When I assign a 4-bit negative value to an 8-bit signed variable, it is not sign extended.

Synopsis: A size mismatch in an assignment might zero extend or might sign extend, depending on the types of expressions on the right-hand side of the assignment.

When a literal integer is assigned to a variable, two expansion/truncation rules are applied:

- First, the value is expanded or truncated to the specified size of the literal integer, via the rules discussed in Gotcha 14 on page 33.
- Second, the assignment operator rules are then applied, as discussed in this gotcha.

Note: This gotcha discusses assignment of literal integers (e.g. `a = 5;`). Gotcha 46 on page 105 discusses assignment rules and gotchas when there are operators on the right-hand side of the assignment.

The assignment operation rules for assigning literal integers are:

- When the left-hand side expression of an assignment statement is fewer bits than the right-hand side literal integer, then the most-significant bits of the right-hand side value are truncated.
- When the left-hand side expression of an assignment statement is more bits than the right-hand side literal integer, then:
  - If the right-hand side literal integer is unsigned, it will be left extended with zeros.
  - If the right-hand side literal integer is signed, it will be left extended using sign extension.
  - Exceptions: If the right-hand side is an unsized high-impedance literal integer (e.g.: `'bz`) or unsized unknown (e.g.: `'bx`), the value will left-extend with `z` or `x`, respectively, regardless of whether signed or unsigned.

These rules might seem, at first glance, to be the same rules discussed in Gotcha 14 on page 33 for when a literal integer size does not match the literal integer value. There is a subtle, but important, difference in the rules, however. The difference is that literal integer expansion will never sign-extend, but an assignment statement *might* sign-extend.

And now the gotcha. Sign extension of the right-hand side only occurs if the expression on the right-hand side of the assignment statement is signed. The

signedness of the left-hand side expression does not affect whether or not sign extension will occur.

Consider the following examples:

```
logic [3:0]        a;  // unsigned 4-bit variables
logic signed [3:0] b;  // signed 4-bit variables
logic [7:0]        u;  // unsigned 8-bit variables
logic signed [7:0] s;  // signed 8-bit variables

u = 4'hC;  // OK, 1100 (hex C) is zero-extended to 00001100

s = 4'hC;  // GOTCHA! 1100 is zero-extended to 00001100,
           // even though s is a signed variable

u = 4'shC; // GOTCHA! 1100 is sign-extended to 11111100,
           // even though u is an unsigned variable

s = 4'shC; // OK, 1100 is sign-extended to 11111100

a = 4'hC;  // assign 4-bit literal to 4-bit unsigned variable
u = a;     // OK, 1100 is zero-extended to 00001100

s = a;     // GOTCHA! 1100 is zero-extended to 00001100,
           // even though s is a signed variable

b = 4'hC;  // assign 4-bit literal to 4-bit signed variable
u = b;     // GOTCHA! 1100 is sign-extended to 11111100,
           // even though u is an unsigned variable

s = b;     // OK, 1100 is sign-extended to 11111100
```

These simple examples illustrate two types of gotchas:

- Assigning to a *signed variable* does *not* cause sign extension. Sign extension only occurs if the right-hand side expression is signed.
- Assigning to an *unsigned variable can* have sign extension. Sign extension occurs if the right-hand side expression is signed.

In other words, it is the right-hand side of an assignment that determines if sign extension will occur. The signedness of the left-hand side has no bearing on sign extension.

These same assignment expansion rules apply when operations are performed on the right-hand side of an assignment. However, whether zero extension or sign extension will occur also depends on the type of operation. These operation rules are covered in Gotcha 45 on page 101.

Gotcha 16: Filling vectors with all ones

Gotcha: I can use a literal integer to set all bits to Z on a vector of any size, but when I use the same syntax to set all bits to 1, I get a decimal 1 instead.

Synopsis: Verilog does not have a literal integer value that fills all bits of a vector with ones. SystemVerilog does.

In Verilog, assigning `'bx`, `'bz`, or 0 will fill a vector of any size with all bits set to X, Z, or zero, respectively. However, assigning `'b1` is not orthogonal. It does not fill a vector with all bits set to one.

```
parameter WIDTH = 64;
reg [WIDTH-1:0] data;

data = 'b0;       // fills with 64'h0000000000000000
data = 'bz;       // fills with 64'hzzzzzzzzzzzzzzzz
data = 'bx;       // fills with 64'hxxxxxxxxxxxxxxxx
data = 'b1;       // fills with 64'h0000000000000001   GOTCHA!
```

Note: Prior to the Verilog-2001 standard, using `'bz` or `'bx` would only fill up to 32 bits with `z` or `x`. Any additional bits to the left of these 32 bits were zero filled.

How to avoid this Gotcha using Verilog

In order to assign a vector of any size with all bits set to one, designers must learn clever coding tricks involving various Verilog operators, such as:

```
data = {64{1'b1}};  // replicate operation (must hard code size)
data = -1;          // negate operation (must use signed literal)
data = ~0;          // invert operation
```

None of these coding tricks for filling a vector of any size with all bits set to 1 is obvious, self-documenting code. SystemVerilog has a better way to avoid this gotcha, as shown below.

How to avoid this Gotcha using SystemVerilog

SystemVerilog provides a simple and consistent syntax for filling any size of variable with all ones, all zeros, all Xs or all Zs. This is done by just assigning `'<value>`, as shown below:

```
parameter WIDTH = 64;
logic [WIDTH-1:0] data;

data = '1;        // fills with 64'hffffffffffffffff
data = '0;        // fills with 64'h0000000000000000
data = 'z;        // fills with 64'hzzzzzzzzzzzzzzzz
data = 'x;        // fills with 64'hxxxxxxxxxxxxxxxx
```

Gotcha 17: Array literals versus concatenations

Gotcha: The wrong values are stored when I assign a list of values to a packed array or structure.

Synopsis: Packed arrays and structures can be assigned either a concatenation or an assignment pattern with a list of values.

The Verilog concatenation operator joins one or more values and signals into a single vector. Array and structure literals (also known as an *assignment patterns*) are lists of one or more individual values. To make the difference between these constructs obvious to both engineers and software tools, the syntax for an array or structure literal is an apostrophe and { } surrounding a list of values, which is different from concatenation, which uses just { }.

```
logic [1:0] [31:0] A; // two-dimensional packed array
A = {1'b1, 1'b1};       // GOTCHA? assign concatenation to A
A = '{1'b1, 1'b1};      // GOTCHA? assign list of values to A
```

The gotcha in this example is that it is not clear whether the intent was to assign a list of values or to assign a concatenation. Since A is a packed array, both types of assignments are legal. If a list of values was intended in the example above, but the apostrophe is inadvertently omitted, it is not a syntax error. The values assigned to A, however, are very different. As a list of values, the assignment is equivalent to:

```
A[1] = 1'b1;    // decimal 1
A[0] = 1'b1;    // decimal 1
```

As a concatenation, the assignment is equivalent to:

```
A[1] = 1'b0    // decimal 0
A[0] = 2'b11;  // decimal 3
```

How to avoid this Gotcha

One way to avoid this gotcha is to only use unsized values in a list of values. The concatenation operator requires sized values, which would make inadvertently leaving off the apostrophe a syntax error.

```
logic [1:0] [31:0] A;  // two-dimensional packed array
A = {1, 1};            // ERROR, illegal concatenation
A = '{1, 1};           // OK, assign list of values to A
```

Another way to detect this coding error is with tools that look for assignment size mismatches, such as a lint tools (coding style checkers).

A closely related gotcha is described in Gotcha 100 on page 206.

Gotcha 18: Port connection rules

Gotcha: My design doesn't work correctly when I connect all the modules together, but each module works correctly by itself.

Synopsis: The size of a port and the size of the net or variable connected to it can be different.

The Verilog standard states that module ports are treated as continuous assign statements that continuously transfer values into, and out of, modules. This is not a gotcha, but rather a rule that helps explain some gotchas relating to port connections.

In Verilog, the receiving side of an `input` or `inout` port can only be a net type, such as `wire`. The transmitting side of an `output` port can be either a net or a variable. When considering ports as continuous assignment statements, it becomes easier to understand why the receiving sides of ports are required to be net data types, and why the driver (or source) side of ports could be either nets or variables. The receiving side of a port is the same as the left-hand side of a continuous assign statement, and the driver or source side of a port is the same as the right-hand side of a continuous assign statement. In other words, the left-hand side/right-hand side rules for continuous assignments apply directly to port assignments.

With this in mind, consider the four scenarios regarding port size connections (three of which can be gotchas if not well understood):

- The port size and the size of the signal driving the port are the same size
- The signal driving the port has more bits than the port (a gotcha)
- The signal driving the port has fewer bits than the port (a gotcha)
- An input port is unconnected; there is no signal driving the port (a gotcha)

Applying Verilog/SystemVerilog assignment size rules to ports gives the following effects for these four scenarios:

1. If the size of the port and the size of the signal driving the port match, then the value passes through the port with no change.
2. If the signal driving the port (the right-hand side of a continuous assignment) has more bits than the port's receiving net (the left-hand side of an assignment), then the upper bits of the driving signal are truncated, including any sign bit.

3. If the signal driving the port (the right-hand side of a continuous assignment) has fewer bits than the port's receiving net (the left-hand side of an assignment), then the upper bits are extended, following Verilog's assignment rules:
   - If the driving signal (right-hand side of an assignment) is unsigned, the upper bits are zero-extended to the size of the receiving signal.
   - If the driving signal (right-hand side of an assignment) is signed, then the upper bits will be sign-extended.
4. If an input port is unconnected, the value for the receiving signal of the port will be the default uninitialized value for its given data type. For the **`wire`** net type, the uninitialized value is Z. For **`tri0`** and **`tri1`** net types, the uninitialized values are 0 and 1, respectively, with a pull-up strength.

In Verilog/SystemVerilog, an incorrect size declaration is an easy design mistake to make, especially when the modules that make up a design are written by several different engineers, (and possibly even come from outside sources). A simple typo in a netlist, or an incorrect parameter redefinition, can also lead to port size mismatches. Typographical errors in a netlist can also result in some ports of a module instance unintentionally left unconnected.

Most often, any mismatch in port connection sizes is a design error. It is not a syntax error, however. Because it is not an error, understanding the rules of how Verilog/SystemVerilog handles a mismatch in connection size helps avoid unexpected simulation or synthesis results (gotchas). Trying to trace back to why some bits disappeared from a vector, or additional bits suddenly appeared, can be difficult. And that assumes that verification detected that there is a problem! The following example illustrates how values are extended or truncated when passed through a port of a different size than the value.

```
module top;
  wire [3:0] data = 4'b1111;       // decimal 15
  wire [7:0] addr = 8'b11111111;   // decimal 255

  block1 b1 (.data(data),    // 4-bit net connected to 8-bit port
             .addr(addr));   // 8-bit net connected to 4-bit port
                             // third port left unconnected
endmodule: top

module block1 (input [7:0] data,
               input [3:0] addr,
               input [3:0] byte_en);

  initial
    #1 $display(" data = %b \n address = %b \n byte_en = %b\n",
                data, address, byte_en);
endmodule: block1
```

The output from simulating this example is:

```
data = 00001111    GOTCHA!
address = 1111
byte_en = zzzz
```

The example above shows how confusing unconnected or partially connected ports can be. The value of `data` has mysteriously changed, gaining an extra four bits. The value of `address` changed from 255 to 15 (decimal), and `byte_en` is high-impedance instead of a valid logic value.

How to avoid this Gotcha using Verilog

Most simulators, synthesis tools and other software tools generate warnings when there are port connection mismatches, but such warnings are not required by the Verilog or SystemVerilog standards, and engineers are notorious for ignoring these warnings. In Verilog, the only way to avoid this type of gotcha is to pay attention to warning messages.

How to avoid this Gotcha using SystemVerilog

SystemVerilog provides a great solution to this gotcha: implicit port connections using either dot-name or dot-star module instantiations (see Gotcha 2 on page 10). When using the dot-name or the dot-star module instantiation syntax, the driver and receiver port signals are required to be the same size. If, for some reason, a driver/receiver signal pair size mismatch is desired, the port must be explicitly connected. This makes it very obvious in the code that the mismatch was intended.

```
module top;
  wire [7:0] data;
  wire [7:0] addr;

  block1 b1 (.data,       // implicit port connections
             .addr);
endmodule: top

module block1 (input [7:0] data,
               input [3:0] addr,
               input [3:0] byte_en);
  ...
endmodule: block1
```

In this example, the size of the wire called `data` has been corrected to be 8-bits wide, which is the same as the size of the `data` port in `block1`. The dot-name shortcut will infer that the wire called `data` is connected to the port called `data`. However, there is still a typo in the declaration of the wire called `addr`. Instead of a port connection mismatch (a gotcha), a compilation error will occur because the

signal at the top level is a different size than the port in `block1`. The dot-name will not infer connections that do not match in size.

The dot-name method will allow unconnected ports, such as the `byte_en` port in the example above. Was this port left unconnected on purpose, or is it another typo in the netlist? To catch all the size mismatches and unconnected ports, using the dot-star shortcut is the best solution. The dot-star shortcut requires explicitly listing all unconnected ports and all the signals with different sizes.

```
module top;
  wire [7:0] data;
  wire [7:0] addr;

  block1 b1 (.*,                // implicit port connection
             .addr(addr[7:4]), // explicitly shows size mismatch
             .byte_en() );      // explicitly shows unconnected
endmodule: top

module block1 (input [7:0] data,
               input [3:0] addr,
               input [3:0] byte_en);
  ...
endmodule: block1
```

Gotcha 19: Back-driven ports

Gotcha: I declared my port as an input, and software tools let me accidentally use the port as an output, without any errors or warnings.

Synopsis: Software tools can ignore the declared direction of a module port, based on how the port is used.

One of the surprising gotchas in Verilog and SystemVerilog is that a module **input** port can be used as an **output**. If a designer mistakenly assigns a value to a signal declared as an input port, there will not be any warnings or errors. Instead, Verilog/SystemVerilog simply treats the input port as if it were a bidirectional **inout** port. Similarly, a higher level module can drive values back into a module's output port. The output port is simply treated as if it were a bidirectional **inout** port, and no errors or warnings are generated.

The Verilog/SystemVerilog standard refers to driving a value back onto a port declared the opposite direction as *port coercion.* Port coercion can only occur when net data types (such as **wire**) are used on both sides of a port. This is because net types allow multi-driver functionality. Since back-driven ports are coerced to **inout** ports, they become multi-driver ports, which require net types.

Port coercion can be useful. A port is really just a transparent connection between an external and internal signal. Port coercion allows software tools to connect modules in the way they are used. It also accurately represents physical hardware, where "ports" don't exist; the external and internal signals are the same single net, and there is no connection of one net to another net. However, port coercion can also allow unexpected design behavior (gotchas), as illustrated in the following example.

```
module top
(output wire [7:0] out,           // net data type
 input  wire [7:0] in             // net data type
);

  buffer8 b1 (.y(out), .a(in));
endmodule

module buffer8
(output wire [7:0] y,       // net data type
 input  wire [7:0] a        // net data type
);
  assign a = y;             // GOTCHA! this should have been y = a;
endmodule
```

In this example, there is a coding error in module `buffer8`. Instead of assigning the input value to the output (`y = a`), the model assigns the output to the input (`a = y`). Instead of being a syntax error, software tools can coerce the module's ports to be **`inout`** ports. *Gotcha!*

How to avoid this Gotcha using Verilog

Port coercion cannot occur if a variable type (e.g. **`reg`** or **`logic`**) is used as a port. Verilog allows output ports to be declared as a variable type, but input ports must be a net type. Prior to the Verilog-2005 standard, the gotcha in the previous example could not be completely avoided. Designers had be careful not to inadvertently assign values to input ports. Some tools, such as lint tools (coding style checkers) may issue a warning when this occurs, which could help detect that the gotcha is present.

The Verilog-2005 standard adds a **`uwire`** (unresolved wire) net data type. The **`uwire`** type only allows a single driver on a net. Thus, when `buffer8` is connected within module top, a compilation error occurs because the `buffer8` input port (`a`) has multiple drivers.

```
module buffer8
(output uwire [7:0] y,  // variable data type
 input  uwire [7:0] a   // variable data type
);
  assign a = y;         // ERROR! multiple drivers for a;
endmodule
```

The functional gotcha has become a compilation error because both the input port and the continuous assignment write to the **`uwire`** `a`.

How to avoid this Gotcha using SystemVerilog

SystemVerilog allows variables to be used on both input and output ports. (Bidirectional inout ports must still be a net type, as in Verilog). SystemVerilog also allows continuous assignments to assign to variables. In addition, SystemVerilog restricts variables to having a single source, which can be a single port, a single continuous assignment, or any number of procedural assignments (which are treated as one source).

In SystemVerilog, module `buffer8`, can be coded as follows.

```
module buffer8
(output logic [7:0] y,  // variable data type
 input  logic [7:0] a   // variable data type
);
  assign a = y;         // ERROR! multiple sources for a;
endmodule
```

The functional gotcha has become a compilation error because both the input port and the continuous assignment write to the variable `a`.

Coding guidelines

When using SystemVerilog, the authors recommend that all module inputs and outputs be declared as **`logic`** variable types, unless it is intended to have multiple drivers on the port (e.g. a bidirectional data bus). By using variables, port coercion cannot occur, and the coding mistake becomes an error that is detected at compilation time.

Gotcha 20: Passing real (floating point) numbers through ports

Gotcha: I cannot find a way to pass real values from one module to another using either Verilog or SystemVerilog.

Synopsis: Verilog does not allow real numbers to be passed directly through ports. SystemVerilog does, but requires a special port declaration.

Verilog has a **real** variable type, which stores a double-precision floating point value. The following example attempts to pass real values from one module to another module through module ports and a top-level netlist.

```
module real_out (output real ro, ...);   // ERROR: output is real
  ...
endmodule

module real_in (input real ri, ...);     // ERROR: input is real
  ...
endmodule

module top;
  real real_connect;
  real_out r1 (.ro(real_connect)); // ERROR: real connected to port
  real_in  r2 (.ri(real_connect)); // ERROR: real connected to port
endmodule
```

The gotcha is that it is illegal in Verilog to pass real numbers through ports.

How to avoid this Gotcha using Verilog

In Verilog, the only way to pass floating point values through ports is by using a pair of built-in system functions to convert real numbers to a format that can be passed through ports. The numbers are then converted back to real in the receiving module. These functions are **$realtobits** and **$bitstoreal**.

```
module real_out (output wire [63:0] net_real_out); // vector output
  real r;
  assign net_real_out = $realtobits(r);  // convert real to vector
  ...
endmodule

module real_in (input wire [63:0] net_real_in);  // vector input
  real r;
  always @(net_real_in)
    r = $bitstoreal(net_real_in);  // convert vector to real
  ...
endmodule
```

```
module top;
  wire [63:0] net_real;       // net types used in netlist
  real_out ro (.net_real_out(net_real));
  real_in  ri (.net_real_in(net_real));
endmodule
```

How to avoid this Gotcha using SystemVerilog

SystemVerilog allows floating point values to be passed directly through ports without having to convert the real values to and from bit vectors. However, the SystemVerilog syntax is not intuitive. An **output** port of a module can be declared as a **real** (double precision) or **shortreal** (single precision) type, but **input** ports must be declared with a keyword pair, **var real** or **var shortreal**.

```
module real_out (output real r);    // output is real variable type
  ...
endmodule

module real_in (input var real r); // input is real variable type
  ...
endmodule

module top;
  real r;               // variable types can be used in netlist
  real_out ro (.r);
  real_in  ri (.r);
endmodule
```

A closely related gotcha involving how tools implement real ports is covered in Gotcha 101 on page 208.

Chapter 3

RTL Modeling Gotchas

Gotcha 21: Combinational logic sensitivity lists with function calls

Gotcha: My combinational logic seemed to simulate OK, but after synthesis, the gate-level simulation does not match the RTL simulation.

Synopsis: If combinational logic calls a function, then the combinational sensitivity list must include signals that the function reads. @ does not infer sensitivity to values read by functions called from combinational logic.*

Synthesizable RTL modeling style requires that Verilog **always** procedural blocks have an edge sensitive timing control (the @ token) following the **always** keyword. This time control is referred to as the block's *sensitivity list.*

```
always @(a, b) begin       // OK, sensitivity list complete
  sum = a + b;
end

always @(a, b) begin       // OK, sensitivity list complete
  prod = mult(a, b);       // call function that reads a, b
end

always @(a, b) begin       // GOTCHA! sensitivity list not complete
  out = sel? sum: prod ;   //   missing sel
end

function [15:0] mult (input [7:0] m, n);
  mult = m * n;
endfunction
```

Note: the code examples in this chapter are contrived in order to illustrate each gotcha using small examples. In real design and verification code, these gotchas might not be as obvious or easy to debug.

When modeling combinational logic, if the sensitivity list is not complete, then the outputs of the block will not be updated for all possible input changes. This behavior models a latch in simulation. However, synthesis will assume a complete sensitivity list and build combinational logic instead of a latch. The simulation results of the RTL model and the synthesized gate-level model will not match. *Gotcha!*

In the simple examples above, it is easy to manually code a complete sensitivity list, and to see if something is missing. Real designs are not always that simple. A complex decoder, for example, could read several dozen signals, each and every one of which must be listed in the sensitivity list. A very common coding gotcha occurs when a designer, in the process of implementing a design, adds another statement to the complex decode logic that reads an additional variable, and forgets to add that additional signal to the sensitivity list. The functional problem that results can be very difficult to detect and debug. *Gotcha!*

How to avoid this Gotcha using Verilog

Verilog has an `@*` wildcard sensitivity list that infers a complete sensitivity list for both simulation and synthesis—most of the time. The `@*` wildcard will automatically be sensitive to any nets or variables that are read in the **`always`** procedural block, including any nets or variables that are passed to a function input. Using `@*` will fix the gotcha in the example above (either `@*` or `@(*)` can be used; they are equivalent).

```
always @* begin              // OK, infers @(a, b)
  sum = a + b;
end

always @* begin              // OK, infers @(a, b)
  prod = mult(a, b);         // call function that reads a, b
end

always @* begin              // OK, infers @(sel, sum, prod)
  out = sel? sum: prod ;
end

function [15:0] mult (input [7:0] m, n);
  mult = m * n;
endfunction
```

However, `@*` has a subtle gotcha that is not widely known. It only infers sensitivity to signals directly referenced in the **`always`** block. It will not infer sensitivity to signals that are externally referenced in a function that is called from the **`always`** block. That is, the `@*` will only be sensitive to the signals passed into the function or task. The following example illustrates this gotcha:

```
always @* begin              // GOTCHA! infers @(a, b)
  prod = mult(a, b);         // call function that reads a, b, max_rtn
end

function [15:0] mult (input [7:0] m, n);
  mult = m * n;
  if (mult > max_rtn)        // reference to external variable
    mult = max_rtn;
endfunction
```

In the preceding example, the sensitivity list inferred by **@*** will not be complete, and therefore will not correctly represent combinational logic in RTL simulations. Synthesis will assume a complete sensitivity list, leading to a mismatch in RTL simulation versus the gate-level simulation. *Gotcha!*

How to avoid this Gotcha using SystemVerilog

SystemVerilog has two specialized procedural blocks that infer a complete sensitivity list, **always_comb** and **always_latch** (there is also an **always_ff** procedural block for sequential logic). The **always_comb** and **always_latch** procedural blocks will descend into function calls to infer the sensitivity list.

```
always_comb begin            // OK, infers @(a, b)
  sum = a + b;
end

always_comb begin            // OK, infers @(a, b, max_rtn)
  prod = mult(a, b);         // call function that reads a, b, max_rtn
end

always_comb begin            // OK, infers @(sel, sum, prod)
  out = sel? sum: prod ;
end

function [15:0] mult (input [7:0] m, n);
  mult = m * n;
  if (mult > max_rtn)        // reference external variable
    mult = max_rtn;
endfunction
```

Note that **always_comb** and **always_latch** do not descend into task calls. If a synthesizable task-like subroutine is required, a SystemVerilog **void** function should be used.

Gotcha 22: Arrays in sensitivity lists

Gotcha: I need my combinational logic block to be sensitive to all elements of a RAM array, but the sensitivity list won't trigger at the correct times.

Synopsis: It is not straightforward to explicitly specify a combinational logic sensitivity list when the combinational logic reads values from an array.

A subtlety that is not well understood is combinational logic sensitivity when the combinational block reads a value from an array. For example:

```
logic [31:0] mem_array [0:1023];   // array of vectors
always @( /* WHAT GOES HERE? */ )  // want combinational logic
  data = mem_array[addr];
```

In order to accurately model true hardware combinational logic behavior, what should the sensitivity include? Should the logic only be sensitive to changes in `addr`, or should it also be sensitive to changes in the contents of `mem_array` being selected by `addr`? If sensitive to changes in the contents of `mem_array`, which address of the array?

The answer, in actual hardware, is that `data` will continually reflect the value that is currently being selected from the array. If the address changes, `data` will reflect that change. If the contents of the array location currently being indexed change, `data` will also reflect that change.

The problem, and gotcha, is that this behavior is not so easy to model at the RTL level, using an explicit sensitivity list. In essence, the sensitivity list only needs to be sensitive to changes on two things: `addr`, and the location in `mem_array` currently selected by `addr`. But, an explicit sensitivity list needs to be hard-coded before simulation is run, which means the value of `addr` is not known at the time the model is written. Therefore, the explicit sensitivity list needs to be sensitive to changes on any and all locations of `mem_array`, rather than just the current location.

To be sensitive to the entire array, it would seem reasonable to write:

```
always @( addr or mem_array )    // ERROR! illegal reference to array
  data = mem_array[addr];
```

Unfortunately, the example above is a syntax error. Neither Verilog nor SystemVerilog allow explicitly naming an entire array in a sensitivity list. Only explicit selects from an array can be listed. For example:

```
always @( addr, mem_array[0], mem_array[1], mem_array[2], ... )
  data = mem_array[addr];
```

This example will work, but it is not practical to explicitly list every array location. Even the relatively small one-dimensional array used in this example, which has 1024 addresses, would be tedious to code.

What about the following example? Will it be sensitive to both `addr` and the value of the `mem_array` location currently selected by `addr`?

```
always @( mem_array[addr] )    // GOTCHA! not sensitive addr
  data = mem_array[addr];
```

The answer is...*It almost works*. The example above is sensitive to a change in value of `mem_array` at the location currently indexed by `addr`. However, it is not sensitive to changes on `addr`. If `addr` changes, `data` will not be re-evaluated to reflect the change. *Gotcha!*

How to avoid this Gotcha

There are three ways to properly model combinational logic sensitivity when reading from an array. The best way is to use Verilog's **`always`** `@*` or SystemVerilog's **`always_comb`** to infer the sensitivity list. Both constructs will infer a correct sensitivity list. Using **`always_comb`** has an added advantage of triggering once at simulation time zero, even if nothing in the sensitivity list changed. This ensures that the outputs of the combinational logic match the inputs at the beginning of simulation.

```
always @*       // OK, infers @(addr, mem_array[addr])
  data = mem_array[addr];

always_comb     // OK, infers @(addr, mem_array[addr])
  data = mem_array[addr];
```

The Verilog-1995 solution to this gotcha is to explicitly specify a sensitivity list that includes the select address and an array select with that address. For example:

```
always @(addr, mem_array[addr]) // OK, this works correctly
  data = mem_array[addr];
```

Another way to avoid this gotcha is to use a continuous assignment instead of a procedural block to model the combinational logic. This will work correctly, but has the limitation that continuous assignments cannot directly use programming statements.

```
assign data = mem_array[addr];  // This works correctly
```

Gotcha 23: Vectors in sequential logic sensitivity lists

Gotcha: My always block is supposed to trigger on any positive edge in a vector, but it misses most edges.

Synopsis: A sequential logic sensitivity list triggers on changes to the least significant bit of the vector.

A sensitivity list can trigger on changes to a vector, which, in the right context, is useful and important.

```
logic [15:0] address, data;

always @(address, data)  // OK: trigger on change to
                         // any bit of vectors
  ...
```

There is a gotcha if the sensitivity list contains a **posedge** or **negedge** edge qualifier on a vector. In this case, the edge event will only trigger on a change to the least significant bit of the vector.

```
always @(posedge address) // GOTCHA! triggering on
                          // specific edge of vector
  ...
```

How to avoid this Gotcha

The **posedge** and **negedge** event qualifiers serve as filters. Consider the following example:

```
always @(posedge clock or negedge reset_n)
```

The sensitivity list triggers on a rising transition of `clock` and filters out all falling transitions on `clock`. The sensitivity list also triggers whenever `reset_n` has a falling transition, but any rising transition of `reset_n` is filtered out.

When used with 1-bit signals such as a clock or reset, the behavior of the **posedge** and **negedge** event qualifiers accurately represents hardware. Testing for a positive or negative edge of a vector, however, does not make sense in hardware. Consider the following:

```
logic [3:0] data;  // 4-bit vector
initial begin
  #1 data = 5;  // data changed from 4'bxxxx to 4'b0101
  #1 data = 3;  // data changed from 4'b0101 to 4'b0011
end
```

On a 1-bit signal, a change from x to 1 is a positive edge, and from x to 0 is a negative edge. In the code above, when data changes from 4'bxxxx to 4'b0101, both scenarios occur. Should always @(posedge data) trigger because a rising transition occurred, or not trigger because a falling transition occurred? Or should the block not trigger at all because **posedge** filtered out the falling transitions and **negedge** filtered out the rising transitions? Similarly, when data changes from 4'b0101 to 4'b0011, both rising and falling transitions occur on the same signal.

Verilog's language rule is well-defined for this situation. By only evaluating the least significant bit of the vector, there is no ambiguity on how simulation will behave when multiple bits change at the same time. This rule, however, is not obvious, and if different behavior was intended or expected, then a *gotcha* has occurred.

To avoid this gotcha, only single-bit expressions should be used with the **posedge** and **negedge** edge event qualifiers. When transitions on a vector need to be evaluated, there are some coding tricks that avoid the gotcha of trying to sample the entire vector. Some examples follow.

To trigger if *any* bit of vector has a rising transition:

```
always @(posedge data[0],      // rising edge of each bit
         posedge data[1],
         posedge data[2],
         posedge data[3])
```

To trigger if *a specific* bit of vector has a rising transition, and ignores transitions on all other bits:

```
always @(posedge data[3])  // rising edge of specific bit only
```

To trigger if a vector transitions from zero to any non-zero value:

```
always @(posedge |data)  // rising edge of unary OR of all bits
```

To trigger if a vector transitions from any non-zero value to zero:

```
always @(negedge |data)  // falling edge of unary OR of all bits
```

Observe that each of these examples is triggering on a 1-bit expression or a list of 1-bit expressions. This concept can be extended to other types of operations, so long as the final expression monitored by posedge or negedge is a 1-bit expression.

A related gotcha involving operations in sensitivity lists is Gotcha 24 on page 56.

Gotcha 24: Operations in sensitivity lists

Gotcha: My sensitivity list should trigger on any edge of a or b, but it misses some changes.

Synopsis: Operations in sensitivity lists only trigger on changes to the operation result.

Occasionally, an engineer might mistakenly use the vertical bar (|) OR operator instead of the **or** keyword as a delimiter in a sensitivity list. The code compiles without any errors, but does not function as expected. *Gotcha!*

The @ symbol is typically used to monitor a list of identifiers used as event triggers for a procedural block sensitivity list. The Verilog standard also allows @ to monitor an event expression.

```
always @(a or b)    // "or" is separator, not operator
  sum = a + b;

always @(a | b)     // GOTCHA! "|" is operator
  sum = a + b;

always @(a && b)    // GOTCHA!
  sum = a + b;

always @(a == b)    // GOTCHA!
  sum = a + b;
```

When an operation is used in a sensitivity list, the @ token will trigger on a change to the result of the operation. It will not trigger on changes to the operands. In the **always** @(a | b) example above, if a is 1, and b changes, the result of the OR operation will not change, and the procedural block will not trigger.

Why does Verilog allow this gotcha? Using expressions in the sensitivity list can be useful for modeling concise verification monitors or high-level bus-functional models. An example usage might be to trigger on a change to a true/false test, such as **always** @(address1 != address2). The procedural block sensitivity list will trigger if the expression changes from false to true (0 to 1), or vice versa.

How to avoid this Gotcha

When modeling combinational logic, the best way to avoid this gotcha is to use the SystemVerilog **always_comb** procedural block, which automatically infers a correct sensitivity list. This eliminates any possibility of typos or mistakes in combinational sensitivity lists. The Verilog @* can also be used, but this has its own gotcha (see Gotcha 21 on page 49). When modeling sequential logic, engineers need to be careful to avoid using operations within a sensitivity list.

Gotcha 25: Sequential logic blocks with begin...end groups

Gotcha: The clocked logic in my sequential block gets updated, even when no clock occurred.

Synopsis: Resettable sequential procedural blocks with a begin..end block can contain statements that execute asynchronous to the clock.

A common modeling style is to place a **begin**...**end** block around the code in **initial** and **always** procedural blocks, even when the procedural block contains just one statement. Some companies even mandate this modeling style. For example:

```
always @(state_e) begin
  nstate_e = HOLD;           // first statement in block
  case (state_e)             // second statement in block
    HOLD: if (ready) nstate_e = LOAD;
    LOAD:  if (done)  nstate_e = HOLD;
  endcase
end
```

This modeling style has a gotcha when modeling resettable sequential logic such as flip-flops. A synthesis requirement is that a resettable sequential procedural block should only contain a single **if**...**else** statement, though each branch of the **if**...**else** might contain multiple statements. An example of a correct sequential procedural block is:

```
always @(posedge clock or negedge reset_n)  // good code
  if (!reset_n) state_e <= RESET;
  else          state_e <= nstate_e;
```

The purpose of **begin**...**end** is to group multiple statements together so that they are semantically a single statement. If there is only one statement in the procedural block, then the **begin**...**end** is not required. In a combinational logic procedural block, specifying **begin**...**end** when it is not needed is extra typing, but does not cause any gotchas.

When modeling resettable sequential logic, however, adding **begin**...**end** can lead to functional gotchas in the model. A resettable sequential block should only contain a single **if**...**else** statement. Adding **begin**...**end** allows additional statements in the block that are functionally incorrect. For example:

```
always @(posedge clock or negedge reset_n) begin
  if (!reset_n) state_e <= RESET;       // first statement
  else          state_e <= nstate_e;
  fsm_out <= decode_func(nstate_e);     // GOTCHA! second statement
end
```

This is a gotcha where Verilog/SystemVerilog allows engineers to prove what won't work in hardware. If the simulation results are not analyzed carefully, it may appear that `fsm_out` behaves as a flip-flop that is set on a positive edge of `clock`. *Gotcha!*

In the example above, `fsm_out` is not part of the **`if...else`** decision for the reset logic. This means:

1. The `fsm_out` sequential block output does not get reset by the reset logic.
2. When reset goes active, the `fsm_out` assignment will be executed asynchronously to the clock.

Both 1 and 2 above are not flip-flop behavior. Because of this, synthesis tools will not allow statements outside of an **`if...else`** statement in resettable sequential procedural blocks. The example above can be simulated and proven to not work correctly, but it cannot be synthesized.

How to avoid this Gotcha

Some engineers prefer to automatically add **`begin...end`** to every procedural block, even when there is only one statement in the block. This style should be discouraged! Using **`begin...end`** is not appropriate for resettable sequential procedural blocks, and leads to the gotcha described above.

A better modeling guideline is to mandate that **`begin...end`** *not* be used in sequential procedural blocks that have reset logic. The following example will report a compilation error, instead of allowing the incorrect code to simulate.

```
always @(posedge clock or negedge reset_n) // no begin
  if (!reset_n) state_e <= RESET;       // first statement
  else          state_e <= nstate_e;
  fsm_out <= decode_func(nstate_e);     // ERROR instead of gotcha!
```

The only time **`begin...end`** should be used in resettable sequential procedural blocks is within the **`if...else`** branches, as follows:

```
always @(posedge clock or negedge reset_n) // no begin
  if (!reset_n) begin    // multiple statements in if branch
     q1 <= 1'b0;
     q2 <= 1'b0;
  end
  else begin             // multiple statements in else branch
     q1 <= d1;
     q2 <= d2;
  end
```

A related potential gotcha is when **`begin...end`** is used in the **`if...else`** branches of a resettable sequential device, as described in Gotcha 26 on page 59.

Gotcha 26: Sequential logic blocks with resets

Gotcha: Some of the outputs of my sequential logic do not get reset.

Synopsis: Resettable sequential procedural blocks can incorrectly reset only some of the outputs.

A syntactically legal, but functionally incorrect, flip-flop model is illustrated below:

```
always @(posedge clock or negedge reset_n)
  if (!reset_n) begin
    q1 <= 0;
    q2 <= 0;
    q3 <= 0;       // GOTCHA! q4 is missing from this branch
  end
  else begin
    q1 <= ~q4;
    q2 <= q1;
    q3 <= q2;
    q4 <= q3;
  end
```

The problem with the example above is that `q4` is not part of the reset logic, but is part of the clocked logic. Because `q4` is not reset, it is not the same type of flip-flop as `q1`, `q2` and `q3`.

In the RTL model above, what will happen to q4 on a reset in simulation? The answer is that `q4` will retain its old value. If a clock occurs while reset is active, `q4` is neither reset nor clocked. This behavior represents a gated or disabled clock on `q4` during reset. Synthesis will most likely create an ugly gate-level implementation of this disabled clock, which is probably not what the designer intended. *Gotcha!*

A closely related gotcha is if the reset branch assigns to some variables that are not assigned in the clock branch. This will also not behave as correct sequential logic, and is probably not what the designer intended.

How to avoid this Gotcha

To avoid this gotcha requires careful modeling. Designers need to make sure that the same variables are assigned values in both branches of the **`if`**...**`else`** decision. SystemVerilog cross coverage can be used to verify that all variables are assigned values when reset occurs and when clock occurs. Software tools such as a lint checkers (coding style checkers) and synthesis tools might warn that the two branches do not assign to the same variables.

Gotcha 27: Asynchronous set/reset flip-flop for simulation and synthesis

Gotcha: When I code an asynchronous set/reset D-type flip-flop following synthesis coding rules, my simulation results are sometimes wrong.

Synopsis: The coding style required by synthesis to model an asynchronous set/reset D-type flip-flop has a simulation race condition. When the race condition is fixed, the code will not synthesize.

Synthesis tools have very specific coding rules for modeling sequential logic devices such as flip-flops, so that, when the code is read into the tool, the correct flip-flop type can be selected. One of the rules for an **always** block sensitivity list is that, when one item in the list has an edge qualifier (**posedge** or **negedge**), all items in the sensitivity list must have an edge qualifier. In the case of an asynchronous set/reset D-type flip-flop, the following sensitivity list is required by synthesis:

```
always_ff @(posedge clk, negedge rst_n, negedge set_n)
```

The modeling within the always block necessitates certain coding styles to ensure the engineer's intent is captured. For a set/reset D-type flip-flop, the set/reset functionality is modeled using an **if**...**else**...**if**...**else** priority encoding style to prioritize the set and the reset. The model of a synthesizable set/reset set/reset D-type flip-flop model is:

```
always_ff @(posedge clk, negedge rst_n, negedge set_n)
  if (!rst_n)           // reset has priority over set
    q_out <= 1'b0;      // reset assignments
  else if (!set_n)
    q_out <= 1'b1;      // set assignments
  else
    q_out <= data_in;   // d input assignment
```

This model synthesizes as intended. However, the model does not work correctly for all simulation conditions. Consider `rst_n` going low. While `rst_n` is low, `set_n` goes low. With both `rst_n` and `set_n` low at the same time, the flip-flop will be held in reset, because of the priority coding of reset and set in the model. Everything is OK so far. Next, `rst_n` goes high and `set_n` stays low. Since the sensitivity list is only sensitive to edges, and is monitoring only the negedge of `rst_n`, the release of `rst_n` will not trigger the sensitivity list. This means that the flip-flop will be held in reset while only the set signal is active. *Gotcha!*

This gotcha only exists with synthesizable asynchronous set/reset flip-flops, and will only be evident until the next clock. The next clock will cause the **if**...**else** decisions to be re-evaluated and transition the flip-flop to its set state.

The problem is that actual asynchronous set/reset inputs are level sensitive, so when the reset is removed, the active set takes over and drives the flip-flop to its set level. In the model, however, synthesis rules require the sensitivity list trigger on the leading edges of the set/reset inputs, causing the simulation gotcha.

How to avoid this Gotcha

This gotcha is a result of the synthesis-imposed coding style for a set/reset d-type flip-flop. In order to model accurate simulation behavior, a level-sensitive, combinational logic block must be added that overrides the synthesizable set/reset logic in simulation. The override is done using the force and release statements that are normally reserved for verification tests. This simulation-specific additional code must be hidden from synthesis by using conditional compilation.

```
`ifndef SYNTHESIS  // start non-synthesizable simulation code
always @*
   if (rst_n && !set_n)  force   q_out = 1'b1;
   else                  release q_out;
`endif              // start synthesizable and simulatable code
always_ff @(posedge clk, negedge rst_n, negedge set_n)
  if (!rst_n)           // reset has priority over set
    q_out <= 1'b0;      // reset assignments
  else if (!set_n)
    q_out <= 1'b1;      // set assignments
  else
    q_out <= data_in;   // d input assignment
```

Gotcha 28: Blocking assignments in sequential procedural blocks

Gotcha: My shift register sometimes does a double shift in one clock cycle.

Synopsis: Blocking assignments in sequential logic is syntactically legal, but usually the wrong functionality.

Verilog has two types of assignments: *Blocking assignments* (e.g. `a = b`) have the simulation behavior of hardware combinational logic. *Nonblocking assignments* (e.g. `q <= d`) have the behavior of hardware sequential logic with a clock-to-Q propagation.

The following example illustrates a very common Verilog coding gotcha. The example uses a blocking assignment where a nonblocking assignment would normally be used. The use of blocking assignments in a clocked procedural block is not a syntax error. The example proves that a shift register will not work if a flip-flop does not have a clock-to-Q delay.

```
always @(posedge clock) begin  // NOT a shift register
  q1 = d;   // GOTCHA! load d into q1 without clock-to-Q delay
  q2 = q1;  // load q1 into q2
end
```

Why does Verilog allow blocking assignments in sequential procedural blocks if they result in simulation race conditions? For two reasons. One reason is that if the sequential logic block uses a temporary variable that is assigned and read within the block, that assignment needs to be made with a blocking assignment. A second reason is the underlying philosophy of Verilog that a hardware description and verification language needs to be able to prove what will work correctly—and what won't work correctly—in hardware.

In the example above, if `q1` and `q2` were positive edge triggered flip-flops, then this example would represent a shift register, where `d` is loaded into flip-flop `q1` on a positive edge of `clock`, and then shifted into `q2` on the next positive edge of `clock`. Using simulation, however, it can be proven that this example does not behave as a shift register. Verilog's blocking assignment to `q1` "blocks" the evaluation of the statement that follows it, until the value of `q1` has been updated. This means that the value of `d` passes directly to `q2` on the first clock edge, rather than being shifted through a flip-flop with a clock-to-Q delay. In other words, the example has proven that a flip-flop without a clock-to-Q propagation behavior will not function properly in hardware.

As an aside, the synthesis tool will recognize that `q1` behaves like a buffer, rather than a flip-flop. If the value of `q1` is not used outside of the procedural block, then synthesis will remove `q1` from the design, and `d` will be directly loaded into `q2`.

How to avoid this Gotcha

As a general rule, engineers should adopt a modeling style that requires the use of nonblocking assignments in sequential procedural blocks. Lint tools (coding style checkers) can help enforce this coding style.

```
always @(posedge clock) begin  // NOT a shift register
  q1 <= d;   // OK, load d into q1 with clock-to-Q delay
  q2 <= q1;  // OK, load previous q1 into q2
end
```

NOTE: There are exceptions to this rule, where a blocking assignment is needed within a sequential procedural block. Only by understanding how blocking and nonblocking assignments work, will engineers know when to correctly make an exception to the rule. One such exception is shown in Gotcha 29 on page 64.

Gotcha 29: Sequential logic that requires blocking assignments

Gotcha: I'm following the recommendations for using nonblocking assignments in sequential logic, but I still have race conditions in simulation.

Synopsis: When modeling clock dividers, the RTL synthesis design guidelines don't always apply.

RTL modeling guidelines recommend that nonblocking assignments should be used for modeling sequential assignments. In a zero-delay RTL model these guidelines help prevent simulation race conditions. A race condition occurs when a value is read at the same moment in time in which it is changing.

These RTL coding guidelines are intended for modeling data flow and data manipulation. These RTL guidelines for using nonblocking assignments do not apply to non-RTL models. When the guidelines are applied to clock generators such as clock dividers and PLLs, the guidelines may actually cause race conditions in the generated clocks. The following code illustrates this problem:

```
always @(posedge clk)
  if (!rstn) clk_divided2 <= 0;
  else       clk_divided2 <= ~clk_divided2;  // GOTCHA!
                                             // delay update to after delta

always @(posedge clk)
  if (!rstn) out1 <= 0;
  else       out1 <= in1;                    // delay update to after delta

always @(posedge clk_divided2)
  if (!rstn) out2 <= 0;
  else out2 <= out1;                         // race condition with out1
```

Nonblocking assignments represent the behavior of a flip-flop clock-to-Q delay, but with zero time. To do this, a nonblocking assignment breaks the assignment into two steps: first, evaluate the right-hand side expression, and second, after a delta, update the left-hand side. During the delta, other statements scheduled for the current simulation time are executed. This two-step process is critical for preventing read/write race conditions in zero-delay RTL models.

In the clock divider example above, however, the nonblocking assignments cause a race condition, instead of preventing it. The basic sequence of events that occur are:

1. Advance the simulator clock. If a positive edge of clock, then:
   - Evaluate `~clk_divided2` and schedule `clk_divided2` to change after a zero-delay delta
   - Evaluate `in1` and schedule `out1` to change after a zero-delay delta

2. After a zero-delay delta, *in any order*:
   - Update `clk_divided2` with its new value. If a positive edge of `clk_divided2` occurred:
     - Evaluate `out1` and schedule `out2` to change after a second delta.
   - Update `out1` with its new value.
3. If a positive edge of `clk_divided2` occurred:
   - Evaluate `out1` and schedule `out2` to change after a second delta.
4. After the second zero-delay delta:
   - Update `out2` with its new value.
5. Advance the simulation time clock.

The race condition in this sequence of events is subtle, but real. It occurs in step 2, when `clk_divided2` is updated. The update could cause a positive edge, which then triggers step 3, to sample `out1`. *But, has `out1` been updated yet?*

The answer is maybe, and maybe not. *This is the race condition!* The value of `out1` is being sampled at the same time, and in the same delta, in which the value is being updated. Simulators are permitted to execute this in a read-then-write or a write-then-read event order. *Gotcha!*

How to avoid this Gotcha

To avoid this gotcha, it is necessary to code the clock divider so that `out1` will always be evaluated before the delta in which `out1` will change. This can be done a few ways, but perhaps the easiest is to use a blocking assignment for modeling the clock divider.

```
always @(posedge clk)
  if (!rstn) clk_divided2 = 0;
  else       clk_divided2 = ~clk_divided2;  // OK, immediate update
```

Blocking assignments update the left-hand side immediately, without a clock-to-Q delta. Using a blocking assignment will cause an immediate event on `clk_divided2`, which, if a positive edge, will immediately trigger the sensitivity list of the `always @(clk_divided2)` block. Thus, the sampling of the right-hand side of `out1` is guaranteed to occur before the delta in which `out1` will change.

This modeling style will synthesize correctly because the clock divider does not receive any data input from other sources, nor does it source data to other **`always`** blocks.

Gotcha 30: Nonblocking assignments in combinational logic

Gotcha: My RTL simulation locks up and time stops advancing.

Synopsis: Nonblocking assignments in a combinational logic block can cause infinite loops that lock up simulation.

Verilog's nonblocking assignment is intended to model the behavior of sequential logic clock-to-Q delay. A nonblocking assignment evaluates its right-hand side expression immediately, and schedules a change on the left-hand side variable after a clock-to-Q delta within the current moment in time. Any statements following the nonblocking assignment statement are "not blocked", and will be executed prior to the delta in the current time. This delta between evaluation and change behaves as a clock-to-Q delay, even in a zero-delay RTL model.

The following example uses nonblocking assignments incorrectly, by placing them in a combinational logic procedural block. The example can potentially lock up simulation in the time step in which m or n changes value.

```
always @(m, n)  // combinational sensitivity list (no clock edge)
  m <= m + n;   // GOTCHA! schedules change after clock-to-Q delta
```

The **always** block triggers when either m or n changes value. The result of m + n is calculated, and the left-hand side of the nonblocking assignment, m, is scheduled to be updated after a zero-delay clock-to-Q delta. During this delta, the nonblocking assignment does not block the execution flow of the procedural block, and so the block returns to its sensitivity list to wait for the next change on m or n. After the clock-to-Q delta, the value of m is updated. This change will once again trigger the sensitivity list, repeating the evaluation and update of m after a clock-to-Q delta. As long as the result of m + n results in a new value of m, simulation will be stuck in the current simulation time, continually scheduling changes to m, and then triggering on the change. *Gotcha!*

There are actually two gotchas in the preceding example. One is that simulation locks up as soon as m or n changes value the first time (assuming n is not 0). The second gotcha is that this is actually a bad design, that would likely cause instability when implemented in gates. This second gotcha is an example of the underlying philosophy of Verilog, which is that engineers should be permitted to model designs that won't work correctly, in order to analyze the behavior of the incorrect hardware. In this case, the model represents combinational logic with a zero-delay feedback path.

This example is not realistic because of the combinational feedback loop with the output, m, also in the input sensitivity list. The example was contrived to show the gotcha of nonblocking assignments in combinational logic in a small circuit.

The following example is based on code from a real design that can also lock up simulation—and did—due to the use of nonblocking assignments within combinational logic blocks.

```
module chip (...)
  ...
  always @( a or ...) begin   // a combinational process
    ...
    b <= 1'b0;                  // nonblocking assignment to b
    ...
    case (state1)
      STATE_G: begin
                 b <= c ;       // nonblocking assignment to b
                 ...
               end
      ...
    endcase
  end
  // many lines of code later...
  always @( b or ...) begin   // another combinational process
    ...
    a <= 1'b0;                  // nonblocking assignment to a
    ...
    case (state2)
      STATE_H: begin
                 ...
                 a <= d;        // nonblocking assignment to a
               end
      ...
    endcase
  end
endmodule: chip
```

There are three Verilog/SystemVerilog event scheduling rules that are important to understand in the example above. This book summarizes those rules, and leaves it to the reader to ponder how this can lead to locking up simulation.

- Events scheduled from parallel procedural blocks (the two **always** blocks in the example above) can be scheduled in any order, which means the parallel events are allowed to be interleaved.
- Events scheduled between **begin**...**end** are executed in the order listed in the source code, including nonblocking assignments.
- In a zero-delay **always** procedural block, nonblocking assignments do not update their left-hand side variables until the procedural block has returned to its sensitivity list at the beginning of the block.

How to avoid these Gotchas using Verilog

The simulation lock-up problem can be fixed by changing the assignment statement from nonblocking to blocking, blocking the execution of the rest of the procedural block until the left-hand side variable has its new value. In the example below, a blocking assignment ensures that `m` will have a new, stable value before the procedural block returns to its sensitivity list, and thus will not re-trigger the procedural block.

```
always @(m, n) // combinational sensitivity list (no clock edge)
  m = m + n;   // OK, immediate update to m with no clock-to-Q delta
```

This change only fixes the lock-up in simulation. It does not fix the second gotcha of an RTL model that does not represent good combinational logic design. There are two ways to fix this design problem, depending on whether the intent is to model a simple combinational logic adder or an accumulator (an adder that stores its output, allowing that output to feedback to the adder input).

How to avoid these gotchas using SystemVerilog

SystemVerilog comes to the rescue with specialized always procedural blocks. The **`always_comb`** and **`always_ff`** constructs can be used to help avoid this coding error gotcha. These constructs do more than just document what type of logic is intended. The **`always_comb`** procedural block infers a proper combinational logic sensitivity list and also enforces some coding rules that help ensure proper combinational logic is modeled. One of these rules is that only one source can write to a variable. In the code `m <= m + n;`, `m` is being used as both an input and an output of the adder. If any other part of the design also writes a value to `m` (as an input to the adder), it is a syntax error. In the context of a full design, the following code causes a syntax error, instead of locking up simulation.

```
always_comb    // inferred combinational logic sensitivity list
  m <= m + n;  // PROBABLE SYNTAX ERROR: no other process can
               // write to m
```

Following is an example error message generated by one simulator when **`always_comb`** is used, and some other source also generates values for the adder inputs.

```
Variable "m" driven by invalid combination of procedural drivers.
   Variables written on left-hand of "always_comb" cannot be
   written to by any other processes.
```

If the intent is to model a simple adder, then a blocking assignment should be used, and the output of the adder should be assigned to a different variable, to prevent the combinational logic feedback loop. For example:

```
always_comb    // inferred combinational logic sensitivity list
  y = m + n;   // immediate update to y with no clock-to-Q delay
```

If the intent is to model an accumulator with a registered output, then a clock needs to be specified in the procedural block sensitivity list. The clock edge controls when the feedback path can change the adder input. The SystemVerilog **always_ff** procedural block helps document that the intent is to have clocked sequential logic.

```
always_ff @(posedge clk) // sequential sensitivity list with clock
  m <= m + n;  // scheduled change to m after clock-to-Q delta
```

Gotcha 31: Combinational logic assignments in the wrong order

Gotcha: Simulation of my gate-level combinational logic does not match RTL simulation.

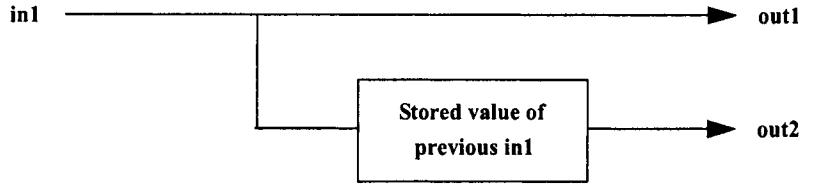
Synopsis: Synthesis might optimize away inferred storage in combinational logic.

Verilog and SystemVerilog require that the left-hand side of procedural assignments be variable types. In simulation, variables have storage, and preserve values between assignments. In hardware, combinational logic devices do not have storage. If the designer's intent is to model combinational logic, then the RTL model should not rely on the storage of the simulation variables. That is, when the combinational block is entered, all outputs of the combinational logic must be assigned a value. If a value is not assigned, then the output is relying on the variable's storage from a previous assignment.

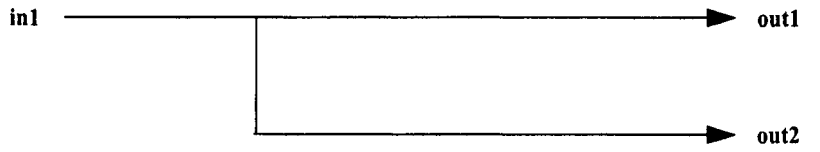
Generally, a synthesis tool is very good at detecting if a combinational logic procedural block is relying on simulation storage. When variable storage is used, the synthesis tool will add latches to the gate-level implementation to preserve that storage. In the following example, however, synthesis tools do not detect that the RTL model is using the storage of the variables.

```
module bad_comb_logic (input  wire in1,
                       output reg  out1, out2
                      );
  always @(in1) begin
    out2 = out1;  // GOTCHA: out2 is first stores last out1 value
    out1 = in1;   // second, out1 is updated to new value of in1
  end
endmodule
```

In simulation, variable `out2` is assigned the current value of variable `out1`, which is the value of `in1` stored the previous time the procedural block was evaluated. After `out2` has saved the current value of `out1`, variable `out1` is updated to reflect the new value of input `in1`. The functionality represented by this RTL model is:



When this example is synthesized, the following gate-level functionality is created:



Simulation of the post-synthesis functionality might not match the RTL simulation functionality. Some synthesis tools will fail to detect that `out2` is reflecting the stored value of `out1` (which is the previous value of `in1`), and do not implement the RTL functionality. *Gotcha!*

How to avoid this Gotcha

This coding example is a bad model. The RTL functionality does not match combinational logic, latched logic, or sequential logic. The problem is that the model assigns to the two combinational logic outputs in the wrong order, and therefore creates a dependency on the variable storage. To correct the problem, the model should be coded as:

```
always @(in1) begin
  out1 = in1;   // out1 is first updated to new value of in1
  out2 = out1;  // OK, second out2 gets new value of out1
end
```

Gotcha 32: Casez/casex masks in case expressions

Gotcha: My casex statement is taking the wrong branch when there is an error in the case expression.

Synopsis: Masked bits can be specified on either side of a casez or casex statement comparison.

Verilog's **casez** and **casex** statements allow bits to be masked out from the case comparisons. With **casez**, any bits set to Z or ? are masked out (Z and ? are equivalent). With **casex**, any bits set to X, Z or ? are masked out. These constructs concisely model many types of hardware, as well as in verification code. An example of using the wildcard **casex** statements is:

```
always_comb begin
  casex (instruction)                    // potential GOTCHA!
    4'b0???: opcode = instruction[2:0];  // only test upper bit
    4'b1000: opcode = 3'b001;
    ...  // decode other valid instructions
    default: begin
               $display ("ERROR: invalid instruction!");
               opcode = 3'bxxx;
             end
  endcase
end
```

In the preceding example, the mask bits are set in the first case item, using `4'b0???`. The intent is that, if the left-most bit of instruction is 0, the other bits do not need to be evaluated. After all possible valid instructions have been decoded, a default branch is used to trap a design problem, should an invalid instruction occur.

What case branch will be taken if there is a design problem, and `instruction` *has the value* `4'bxxxx`*?* The intuitive answer is that the default branch will be taken, and an invalid instruction will be reported. *Gotcha!*

The **casex** and **casez** statements allow the mask bit to be set on either side of the comparison. In the preceding example, if instruction has a value of `4'bxxxx` or `4'bzzzz`, all bits are masked from the comparison, which means the first branch of the case statement will be executed.

How to avoid this Gotcha using Verilog

A partial solution is to use **casez** instead of **casex**. In the example above, if a **casez** were used, a design problem that causes an instruction of `4'bxxxx` (or even just an X in the left-most bit) will not be masked, and an invalid instruction

will be reported by the default branch. However, a design problem that causes an instruction of `4'bzzzz` (or just a Z in the left-most bit) will still be masked, and an invalid instruction will not be trapped.

How to avoid this Gotcha using SystemVerilog

SystemVerilog offers two solutions to this gotcha. The first solution is a special one-sided, wildcard comparison operator, **==?**, which returns true if its two operands match in value and false if its two operands do not match. There is also a **!=?** operator, which negates the true/false test result. This wildcard operator works similar to **casex**, in that bits can be masked from the comparison using X, Z, or ?. However, the mask bits can only be set in the left-hand side of the comparison. In the following example, any X or Z bits in `instruction` will not be masked, and the invalid instruction will be trapped.

```
if (instruction ==? 4'b0???) opcode = instruction[2:0];
else if ... // decode other valid instructions
else begin
       $display ("ERROR: invalid instruction!");
       opcode = 3'bxxx;
     end
```

A second solution to the gotcha is the SystemVerilog **case() inside** statement. This statement allows mask bits to be used in the case items using X, Z or ?, as with **casex**. But, **case() inside** uses a one-way, asymmetric masking for the comparison. Any X or Z bits in the case expression are not masked. In the following example, any X or Z bits in `instruction` will not be masked, and the invalid instruction will be trapped:

```
always_comb begin
  case (instruction) inside
    4'b0???: opcode = instruction[2:0]; // only test upper bit
    4'b1000: opcode = 3'b001;
    ...  // decode other valid instructions
    default: begin
               $display ("ERROR: invalid instruction!");
               opcode = 3'bxxx;
             end
  endcase
end
```

Gotcha 33: Incomplete decision statements

Gotcha: My full_case, parallel_case decision statement simulated as I expected, but the chip does not work.

Synopsis: Incomplete case statements or if...else decision statements can result in hard-to-detect design errors.

Verilog's **if...else** and **case** statements (including **casez** and **casex**) have some potential gotchas that can result in design problems:

- It is legal to have incomplete decision statements, where not all possible selection values have a corresponding decision branch.
- It is legal to have redundant selection values, where the selection values for two or more decision branches are true at the same time.
- Simulation evaluates multi-branch decisions in source code order (priority decoding), but synthesis might implement the evaluation as parallel decoding.

Verilog/SystemVerilog synthesis tools can use *pragmas*, which are commands hidden in comments, to instruct the synthesis tool on how to handle incomplete or redundant case statements. The synthesis **full_case** pragma instructs synthesis to ignore any unspecified decision selection values. The **parallel_case** pragma instructs synthesis to ignore the possibility of redundant selection values.

The following example illustrates the RTL logic to decode the next state of a finite state machine. The state variable is 3 bits wide, which can have 8 possible values. Only three of these values are used by the state machine encoding. The synthesis **full_case** and **parallel_case** pragmas are used to instruct synthesis tools that the other 5 values are not used by the design.

```
logic [2:0] state, nstate;  // 3-bit variables (8 possible values)

always @(state) begin       // next state decoder
  case (state)   //synthesis full_case -- GOTCHA!
    3'b001: nstate = 3'b010;
    3'b010: nstate = 3'b100;
    3'b100: nstate = 3'b001;
  endcase
end
```

Note that this example hard codes the state values to make discussion of the potential gotchas more obvious. The preferred coding style is to define the state values as local parameter (if using Verilog) or enumerated types (if using SystemVerilog). The same gotchas can still exist, but won't be as obvious.

The example above is an incomplete case statement. There is a decision branch for only three of the eight possible values of `state`. The designer has assumed that since the state encoding is one-hot, the five possible values that were not specified will never happen. This design assumption has been documented for the synthesis compiler using a `//synthesis full_case` pragma.

What happens if a state value of 2'b000 occurs? The synthesis pragma has instructed synthesis that none of the unspecified values will ever happen. Therefore the gate-level implementation will most likely not decode the unspecified values. The logic gates will do something, but the tool, instead of the designer, has selected what will happen. In RTL simulation, if a selection value occurs that does not select a branch, then no assignment statements are executed. The value of the `nstate` variable in this example is not updated, and just retains its previous value. Thus RTL simulation will exhibit one behavior if `state` is `3'b000`, and the gate-level implementation will do something different. *Gotcha!*

A second problem in RTL simulation is that since `nstate` does not change when an unspecified selection value occurs, it might not be obvious that an unexpected `state` value occurred. The design could appear to be working fine in RTL simulation, when there is actually a problem in the design. *Gotcha, again!*

How to avoid this Gotcha in Verilog

With Verilog the only way to avoid this gotcha is to not use the **`full_case`** pragma, and instead explicitly specify what should happen with unexpected selection values. This can be done using a **`default`** branch in the case statement. In the preceding example, if the default branch assigns a known value to `nstate`, the gate-level behavior is well-defined for any unexpected values of `state`.

```
case (state)
  3'b001:  nstate = 3'b010;
  3'b010:  nstate = 3'b100;
  3'b100:  nstate = 3'b001;
  default: nstate = 3'b001;  // on error, go back to first state
endcase
```

Many Verilog engineers like to use the default branch to assign a value of X.

```
  default: nstate = 3'bxxx;  // on error, set nstate to unknown
```

What will happen in the synthesized gate-level implementation if the default branch assigns a value of X to nstate? Synthesis treats this assignment the same as a **`full_case`** pragma, meaning the gate-level implementation will do something, but the designer has given up control of what should happen for unexpected selection values. The gotcha of a difference between RTL and gate-level is still there, but the gotcha of not knowing that unexpected state values occurred in RTL simulation has been avoided.

How to avoid this Gotcha in SystemVerilog

SystemVerilog provides the synthesis optimization advantages of the **full_case** and **parallel_case** pragmas, and, at the same time, avoids the gotcha of undetected functional problems in RTL simulation. This is done using two decision modifiers, **unique** and **priority**, that can be specified on either **case** statements or **if...else...if** statements.

For synthesis, **unique case** is the same as specifying both the **full_case** and **parallel_case** pragmas. A **priority case** is the same as specifying the **full_case** pragma. The gotcha with the synthesis pragmas is that they only affect synthesis. They are ignored by simulation. The **unique** and **priority** modifiers are both synthesizable and simulatable, enabling verification that the instructions to the synthesis compiler are correct.

In simulation, the behavior of **unique case** is:

- A warning is issued if the case statement is entered and no branch is taken.
- A warning is issued if the case statement is entered and more than one case select expression is true. That is, if more than one branch could be taken.

In simulation, the behavior of **priority case** is:

- A warning is issued if the case statement is entered and no branch is taken.

For example:

```
always @(state) begin       // next state decoder
  unique case (state)
    3'b001: nstate = 3'b010;
    3'b010: nstate = 3'b100;
    3'b100: nstate = 3'b001;
  endcase
end
```

Using **unique case** allows the designer to verify that the instructions on how to synthesize the case statement are correct. The run-time checking can be combined with SystemVerilog's constrained random test generation and functional coverage to prove that unexpected values truly cannot occur, or to warn if they do occur. Formal verification can also use the **unique** decision modifier to prove, or disprove, the designer's assumptions about the decision statement.

WARNING! All design guidelines for the proper use of **full_case** and **parallel_case** still apply with **unique case** and **priority case**. That is, these decision modifiers can be abused, just as the pragmas can be abused.

Additional examples of using **unique case** and **priority case** are shown in Gotcha 34 on page 77 and Gotcha 35 on page 79.

Gotcha 34: Overlapped decision statements

Gotcha: One of my decision branches never gets executed.

Synopsis: Redundant decision selection values can go undetected in simulation.

Verilog evaluates a series of **if**...**else**...**if**...**else** decisions in the order in which the decisions are listed. If a coding error is made, such that two decisions could both evaluate as true, then only the first branch is executed.

```
always @* begin
       if (sel == 2'b00) y = a;
  else if (sel == 2'b01) y = b;
  else if (sel == 2'b01) y = c;  // GOTCHA! same sel value
  else if (sel == 2'b11) y = d;
end
```

The coding error in this example—probably a cut-and-paste error—is not a syntax error. The code will compile and simulate, but the third branch will never execute. Since it is not a syntax error, the coding error can go undetected in simulation. *Gotcha!*

A similar cut-and-paste error can be made in **case** statements. An overlap in case decisions is not an error. Instead, only the first matching case branch is executed.

```
always @* begin
  case (sel)
    2'b00: y = a;
    2'b01: y = b;
    2'b01: y = c;  // GOTCHA! same sel value as previous line
    2'b11: y = d;
  endcase
end
```

How to avoid this Gotcha using Verilog

Both of the above examples are easy errors to make, and can be difficult to detect in Verilog simulation. Software tools such as lint tools (coding style checkers) and synthesis tools will warn about the overlap in decisions in the preceding examples. However, since it is only a warning message, it may go unnoticed.

How to avoid this Gotcha using SystemVerilog

SystemVerilog adds a **unique** modifier that can be used with both **if**...**else**...**if** and **case** decision statements.

```
always_comb begin
  unique if (sel == 2'b00) y = a;
    else if (sel == 2'b01) y = b; // will get simulation warning
    else if (sel == 2'b01) y = c; // will get simulation warning
    else if (sel == 2'b11) y = d;
end

always_comb begin
  unique case (sel)
    2'b00: y = a;
    2'b01: y = b;   // will get simulation warning
    2'b01: y = c;   // will get simulation warning
    2'b11: y = d;
  endcase
end
```

The **unique** decision modifier requires that simulators report a warning if two or more decision selection expressions are true at the same time. The **unique** modifier also requires that simulation generate a warning message if no decision branch is taken. *Do not ignore the simulation warnings generated by using unique—the warnings indicate there is a coding problem!*

Additional examples of using **unique case** are shown in Gotcha 33 on page 74 and Gotcha 35 on page 79.

Gotcha 35: Inappropriate use of unique case statements

Gotcha: I am using unique case everywhere to help trap design bugs but my synthesis results are not what I expected.

Synopsis: SystemVerilog's unique case synthesizes the same as a case statement tagged with full_case and priority_case.

SystemVerilog extends the Verilog language with a **unique** decision modifier keyword. This modifier can be specified on either **case** statements or **if...else...if** statements. The **unique** modifier specifies that a decision statement should be considered complete, and that there is no overlap in the decision selection values.

Many coding methodologies today are recommending that **unique** be specified for all case statements, just as the synthesis **full_case** pragma was recommended for all case statements a few years ago. The reason **unique case** is recommended is that it comes with built-in assertions that provide visibility during RTL simulation, indicating when a case statement did not execute as expected. Design problems can potentially become visible earlier in the design cycle.

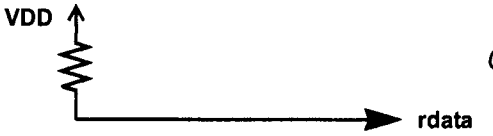
This coding guideline of using **unique case** for all case statements is a *Gotcha!*

The intent in the following example is to have a simple decoder that sets the `rdata` flag if `address` is zero.

```
module address_decocde (input  logic [1:0] address,
                        output logic       rdata);

  always_comb begin
    rdata = 1'b0;              // default value for rdata
    unique case (address)      // decode address -- GOTCHA!
      2'b00 : rdata = 1'b1;
    endcase
  end
endmodule
```

The example may look overly simple, but it comes from a real design, and is a real gotcha! The simulation results were as expected; when `address` is 0, `rdata` is 1; for any other value of `address`, `rdata` is 0. Here's what a synthesis tool sees from this model:



Gotcha!

By specifying **unique case**, the designer has informed synthesis that all unspecified case selection values should be considered a "don't care", and can be ignored. Since the designer has indicated that only the selection values listed are real, and that no other values can occur, the default assignments for these other selection values do not have meaning. They are assigning a default output for conditions that **unique case** says can never happen. Therefore, synthesis ignores the default assignments! *Gotcha!*

How to avoid this Gotcha

To avoid this gotcha *do not ignore simulation warnings!* The **unique case** statement will issue a run-time warning anytime no branch is taken. In the example above, however, it was easy to look at the code and come to the conclusion that the warnings were bogus and could be ignored. Yes, there are times when no case branch would be taken, but the default assignment before the case statement takes care of those situations. In truth, the simulation warnings also indicated that the case statement was not evaluating the way synthesis had been instructed to interpret it. The warnings from **unique case** should not be ignored!

The real problem in the example above is not understanding the purpose of the **unique** decision qualifier. The purpose of **unique** is to inform tools that not all of the possible selection values of a decision are being used, because the values are not used in the design. In the example above, the decoder actually does decode every value of `address`. An address of zero sets `rdata`, and all other addresses clear `rdata`. It is incorrect to use **unique case** in this design, as evidenced in the synthesis results.

The correction for the example above is simple. Do not use the **unique** decision qualifier.

```
module address_decocde (input  logic [1:0] address,
                        output logic       rdata);

  always_comb begin
    rdata = 1'b0;             // default value for rdata
    case (address)            // OK, incomplete case statement
      2'b00 : rdata = 1'b1;     // decodes exception to default
    endcase
  end
endmodule
```

This is not to say **unique case** should never be used. It is to say that **unique case** should be used correctly, just like the synthesis **full_case** pragma needs to be used wisely and correctly.

A few years ago, many companies followed a coding guideline that all case statements should be specified with the synthesis **full_case** pragma, so that they would synthesize more optimally. In the past few years, there have been several conference papers showing why blanket usage of the synthesis **full_case** pragma can be bad for a design. Experience has proven that specifying **full_case** should be the exception, rather than the general rule. The indiscriminate use of the **full_case** pragma is now strongly discouraged.

SystemVerilog's **unique case** does not change this guideline. Synthesis tools treat **unique case** as if the case statement has both the **full_case** and **parallel_case** pragmas specified. The only difference between the synthesis pragmas and **unique case** is that **unique case** can also be simulated. That is, however, an important difference, as illustrated in the next paragraph.

When **unique case** is used correctly, it has significant advantages over the synthesis **full_case** pragma. An example of an appropriate place to use **unique case** is in a one-hot state machine decoder, where only certain state values are valid. The other values should never occur. In that situation, **unique case** is appropriate. It instructs synthesis that the unused values of the state variable are "don't cares", and need not be implemented in the gate-level design. At the same time, the **unique case** instructs simulation to assert that the unused state values never occur. SystemVerilog's constrained random test generation, coupled with SystemVerilog's functional coverage, can be used to thoroughly exercise the design to verify that the unused state values truly cannot occur. Formal verification tools can also use the **unique** decision modifier to guide what needs to be formally proven.

Gotcha 33 on page 74 describes the simulation semantics of **unique case**. Gotcha 34 on page 77 illustrates another appropriate use **unique case**.

Note: The SystemVerilog **priority case** statement has the same gotcha. Synthesis tools treat **priority case** the same as if the synthesis **full_case** pragma had been specified. Care needs to be taken to only use **priority case** where it is appropriate, which is when not all decision selection values are used, and it is permissible to have an overlap in the selection values for each decision branch.

Gotcha 36: Resetting 2-state models

Gotcha: My design fails to reset the first time in RTL simulation.

Synopsis: Two-state data types begin with a known value, and might not cause simulation events the first time a value is assigned.

One of the features of SystemVerilog is 2-state data types, which, in theory, can be advantageous in simulation. However, 2-state types also have some simulation gotchas. One of these gotchas is that, at the beginning of simulation (time zero), the value of each variable is a default uninitialized value, which is X for 4-state variables and zero for 2-state variables. The uninitialized 2-state value of zero can lead to a reset gotcha. Consider the following code:

```
module chip_tb;
  logic clk;      // 4-state type
  bit   rst_n;    // GOTCHA! 2-state type for reset

  initial begin       // clock oscillator
    clk <= 0;
    forever #5ns clk = ~clk;
  end

  initial begin          // reset stimulus (active low reset)
    rst_n <= 0;          // turn on reset at time zero
    #3ns rst_n = 1;      // turn off reset after 3 nanoseconds
  end

  chip u1(.rst_n, .clk, ...);  // instance of design under test

endmodule: chip_tb

module chip (input bit rst_n, clk, ...);  // GOTCHA! 2-state types

  enum {HOLD, LOAD, STORE} state_e, nstate_e;  // GOTCHA! 2-state

  always_ff @(posedge clk, negedge rst_n)  // asynchronous reset
    if (!rst_n) state_e <= HOLD;
    else        state_e <= nstate_e;
  ...
endmodule: chip
```

In the example above, the **always_ff** flip-flop in module `chip` is supposed to reset on a negative edge of `rst_n`. The testbench sets `rst_n` to zero at the beginning of simulation, and holds it low for 3 nanoseconds. However, in the testbench, `rst_n` is a 2-state type, which begins simulation with a value of zero. Setting `rst_n` to zero does not change its value, and therefore does not cause a

negative edge on rst_n. Since the testbench does not cause a negative edge on rst_n, the **always_ff** sensitivity list for the flip-flop in module chip does not trigger, and the flip-flop does not reset asynchronously. If rst_n were held low at least one clock cycle, the flip-flop would reset synchronously when clock occurred. In this example, though, the test stimulus does not hold rst_n low a full clock cycle, and therefore the reset is completely missed. *Gotcha!*

How to avoid this Gotcha

This gotcha can be avoided in a number of ways. One way is to initialize the 2-state reset signal to the non-reset value with a blocking assignment, and then to the reset value with a nonblocking assignment. This will trigger the **always_ff** blocks waiting for a negative edge of reset. Additionally, the nonblocking assignment will ensure that all the **always_ff** blocks are active before the transition to zero occurs.

```
initial begin
  rst_n  = 1; // initialize to inactive value
  rst_n <= 0; // set to active value using nonblocking assign
  #3ns rst_n = 1;
  ...
```

Note, however, that this solution potentially creates a new *gotcha!* Changing any signal at time zero using a blocking assignment can potentially cause a race condition with any procedural blocks that trigger on that signal. This is because the order in which procedural blocks become active at time zero is not defined, so the change in value could occur before other procedural blocks have activated their sensitivity lists. Since rst_n is an active low reset, there is little or no risk of a race condition by setting it to a logic 1, the inactive state, at time zero.

A second way to avoid this gotcha is to use in-line variable initialization to assign rst_n to its inactive value at the same time rst_n is declared. Then, when the test program assigns rst_n to 0, a change will occur which will trigger the **always_ff** sensitivity list.

```
bit rst_n = 1  // initialize to inactive value

initial begin
  rst_n <= 0; // set to active value using nonblocking assign
  #3ns rst_n = 1;
  ...
```

A third way to fix this gotcha is to use 4-state types instead of 2-state types for active-low signals. 4-state variable types will begin simulation with a value of X. Assigning a 4-state type a value of zero, even at simulation time zero, will cause an X-to-0 transition, which is a negative edge.

Gotcha 37: Locked state machines modeled with enumerated types

Gotcha: My state machine model locks up in its start-up state.

Synopsis: Two-state enumerated variables begin with a known value, and might not cause simulation events the first time a value is assigned.

By default, enumerated types are 2-state types. Also by default, the value of the first label in an enumerated list is zero. Functional logic based on 2-state enumerated data types can have gotchas. Consider the following code:

```
module controller (output logic   read, write,
                   input  instr_t instruction,
                   input  logic   clock, reset_n);

  enum {HOLD, LOAD, STORE} state_e, nstate_e;  // 2-state types

  always @(posedge clock, negedge reset_n)  // state sequencer
    if (!reset_n) state_e <= HOLD;
    else state_e <= nstate_e;

  always @(state_e) begin        // GOTCHA! next state decoder
    unique case (state_e)
      HOLD:  nstate_e = LOAD;
      LOAD:  nstate_e = STORE;
      STORE: nstate_e = HOLD;
    endcase
  end
  ...
endmodule
```

In simulation, this example will lock up in the `HOLD` state. Applying reset, whether 2-state or 4-state, will not get the state machine out of this lock up. This is because `state_e` and `nstate_e` are 2-state enumerated variables. 2-state types begin simulation with a value of zero, which is the value of `HOLD` in the enumerated list. When the **`always_ff`** state sequencer is reset, it will assign `state_e` the value of `HOLD`, which is the same value as the current value of `state_e`, and thus does not cause a transition on `state_e`. Since `state_e` does not change, the **`always`** `@(state_e)` combinational procedural block does not trigger. Since the combinational block is not entered, `nstate_e` is not updated to a new value, and retains its initial value of `HOLD`. On a positive edge of clock, `state_e` is assigned the value of `nstate_e`, but, since the two variables have the same value of `HOLD`, `state_e` does not change,. Once again, the **`always`** `@(state_e)` combinational block is not triggered and `nstate_e` is not updated. The simulation is stuck in the start-up state, no matter how many clock cycles are run, and no matter how many times the state machine is reset. *Gotcha!*

How to avoid this Gotcha

The best way to avoid this gotcha is to use the SystemVerilog **always_comb** for the combinational block in this code. Unlike the Verilog **always** procedural block, an **always_comb** procedural block will automatically execute once at time zero, even if the sensitivity list was not triggered. When the **always_comb** block executes, nstate_e will be assigned the correct value of LOAD. Then, after reset is removed, the state machine will function correctly, and not be locked in a HOLD state.

A second method to avoid this gotcha is to declare the state_e and nstate_e enumerated variables as 4-state types, as follows:

```
// 4-state types
enum logic [1:0] {HOLD, LOAD, STORE} state_e, nstate_e;
```

By doing this, state_e and nstate_e will begin simulation with the value of X. When state_e is assigned HOLD during reset, the **always** @(state_e) will trigger, setting nstate_e to LOAD.

A third way to fix this 2-state lock-up gotcha is to explicitly assign values to the HOLD, LOAD and READY labels that are different from the uninitialized value of the enumerated variables. For example:

```
enum bit [2:0] {HOLD  = 3'b001,
                LOAD  = 3'b010,
                STORE = 3'b100} state_e, nstate_e;
```

In this example, state_e and nstate_e are 2-state types, which begin simulation with an uninitialized value of zero. This value does not match any of the values in the enumerated list. When reset is applied, state_e will be assigned HOLD. The change on state_e will trigger the **always** @(state_e) combinational block, which will update nstate_e to LOAD, preventing the lock-up gotcha.

Gotcha 38: Hidden design problems with 4-state logic

Gotcha: There was a problem deep inside the logic of my design, but it never propagated to module boundaries.

Synopsis: Some programming statements do not propagate logic X values.

In 4-state simulation, a logic value of X can occur. Logic X is not a real hardware logic value. Nor is it a "don't care", the way it used in some data books. Logic X is the simulator's way of saying that simulation algorithms cannot predict what actual hardware would do with a given set of circumstances. While no engineer likes to see X values in the simulation log files or waveform displays, savvy engineers have come to know that X is their friend. When an X value does show up, it is a clear indication of a problem in a design.

But there is a gotcha. A number of Verilog programming statements can swallow an X value, and generate a seemingly good value. These statements hide design problems, which can be disastrous. Two of the most common X hiding constructs are decisions statements and optimistic operators. An example of a decision statement that will hide design errors is:

```
always_comb begin
  if ( sel) y = a;   // 2-to-1 MUX
  else      y = b;
end
```

In this example, should a design bug cause `sel` to have a logic X, the **`else`** branch will be taken, and a valid value assigned to `y`. If the verification code is primarily verifying the functional results of the design, it will only see known values on this multiplexer output. The design bug on the `sel` signal has been hidden. *Gotcha!*

How to avoid this Gotcha using Verilog

The ideal would be if each model had internal code to trap errors within the model. Functional verification of the design could focus on verifying the overall functionality, and each design block would take care of detecting unexpected values within that block.

In Verilog, adding self-checking logic within the model can be awkward and require many extra lines of code. This extra code must be hidden from synthesis tools, as it is not really part of the hardware. There is also a risk that the extra code could inadvertently change the intended behavior of the design. Because of the extra coding and associated risks, design engineers are often hesitant to add error-monitoring code within RTL models. Consider the following:

```
always_comb begin
  if (sel)
    y = a;   // do true statements
  else
//synthesis translate_off
  if (!sel)  // opposite of if condition)
//synthesis translate_on
    y = b;   // do the not true statements
//synthesis translate_off
  else begin
    y = 'bx;
    $display("if condition tested either an X or Z");
  end
//synthesis translate_on
end
```

This additional code illustrates the awkwardness of embedding error handling in RTL models using Verilog. Few, if any design engineers are willing to do this.

How to avoid this Gotcha using SystemVerilog

A better way to avoid this gotcha is to use SystemVerilog Assertions (SVA). Assertions are more concise, and do not need to be hidden from synthesis tools. Assertions can be turned on and off as needed. They can also provide verification coverage information. An assertion for the example above can be written as:

```
always_comb begin
  assert ($isunknown(sel)) else $error("sel = X");
  if ( sel) y = a;   // 2-to-1 MUX
  else      y = b;
end
```

This example uses an immediate assertion, which will execute every time the always block is entered. False assertion failures could be reported if `sel` glitches, but becomes stable before the MUX output is used. To avoid executing the assertion on glitches, synchronous concurrent assertion can be used.

For more details on X hiding gotchas and using assertions, refer to two papers from the authors, "*Being Assertive With Your X*"[1], and "*SystemVerilog Assertions are for Design Engineers, too*"[2].

---

1. *Being Assertive With Your X*, by Don Mills. Published in the proceedings of SNUG San Jose, 2004. Also available from the author's web site, http://www.lcdm-eng.com/assertiveX.pdf.
2. *SystemVerilog Assertions are for Design Engineers, Too*, by Don Mills and Stuart Sutherland. Published in the proceedings of SNUG San Jose, 2006. Also available from the author's web site, http://www.sutherland.com/papers.html.

Gotcha 39: Hidden design problems using 2-state types

Gotcha: Some major functional bugs in my design did not show up until after synthesis, when I ran gate-level simulations.

Synopsis: Design errors might not propagate through 2-state logic.

An important gotcha to be aware of when modeling with 2-state data types, whether at the RTL level or at the verification level, is the fact that 2-state types begin simulation with a value of 0 instead of X. It is common for a value of 0 to also be the reset value of registers within a design. Consider the following example:

```
bit [31:0] data_reg;  // 2-state variable

always_ff @(posedge clock, negedge reset_n)   // data register
  if (!reset_n) data_reg <= 0;                // reset to zero
  else          data_reg <= data_in;
```

The initial value of `data_reg` is zero. This is also the value to which `data_reg` is reset. This means that, if for some reason the design fails to generate a reset, it will not be obvious by looking at the value of `data_reg` that there was a failure in the design logic.

Another way in which 2-state logic can hide design errors is when an operation returns a logic X, as illustrated below:

```
module comparator (output bit eq,        // 2-state output
                   input  bit a, b);     // 2-state inputs

  assign eq = (a == b);

endmodule
```

In the example above, the gotcha is the 2-state inputs. What will happen if there is a design error, and either the `a` or `b` input is left unconnected? With 4-state values, the unconnected input would float at high-impedance, and the `(a == b)` operation will return a logic X—an obvious design failure. With 2-state inputs, however, there is no high-impedance to represent a floating input. The design error will result in zero on the input, and an output of one or zero. The design failure has been hidden, and did not propagate to an obvious incorrect result. *Gotcha!*

What if the inputs and outputs in the preceding example were 4-state, but the output was connected to another design block, perhaps an IP model written by a third party provider, that was modeled using 2-state types? In this case, the

comparator module would output a logic X, due to the unconnected input design failure, but that X would be converted to a zero as it propagates into the 2-state model, once again hiding the design problem. *Gotcha, again!*

How to avoid this Gotcha

The best way to avoid this gotcha is to use 4-state types in all design blocks. 4-state variables begin simulation with a value of X, making it very obvious if reset did not occur. Should an operation or programming statement produce a logic X, the use of 4-state types will propagate the design error instead of hiding it. In addition to using 4-state types, SystemVerilog assertions can be used to verify that inputs to each design block are valid. SystemVerilog functional coverage can also be used to verify that reset occurs during simulation.

CAUTION! 4-state types can also hide design problems, but in different ways. See Gotcha 38 on page 86 for more details.

Gotcha 40: Hidden problems with out-of-bounds array access

Gotcha: A design bug caused references to nonexistent memory addresses, but there was no indication of a problem in RTL simulation.

Synopsis: Out-of-bounds errors might not propagate through 2-state logic.

A type of failure that can be hidden by 2-state types is when an out-of-bounds address is read from an array. An example where this can occur follows:

```
module RAM #(parameter SIZE = 1024, A_WIDTH = 16, D_WIDTH = 31)
            (output logic [D_WIDTH-1:0] data_out,
             input  logic [D_WIDTH-1:0] data_in,
             input  logic [A_WIDTH-1:0] addr,     // 16 bit bus
             input  logic               read, write);

  bit [D_WIDTH-1:0] mem_array [0:SIZE-1];  // 2-state array
                              // GOTCHA! only need 10 bit index

  assign data_out = read? mem_array[addr] : 'z; // read from array
  ...
endmodule
```

In this example, the address bus is wider than is required to access all addresses of `mem_array`. If a 4-state array is accessed using an address that does not exist, a logic X is returned. But, when a 2-state array is accessed using an address that does not exist, a value of zero is returned. Since a value of zero could be a valid value, the out-of-bounds read error has been hidden. *Gotcha!*

The example above is an obvious design error, but is also one that could easily be inadvertently coded. The same error is less obvious when the defaults of the memory size and address bus parameters are correct, but an error is made when redefining the parameter values for an instance of the RAM. *Gotcha, again!*

How to avoid this Gotcha

There are a few ways to avoid this gotcha. One way is to use 4-state types for arrays. An out-of-bounds reference to a 4-state array will return a logic X, indicating that a design problem occurred. However, a 4-state array requires twice the amount of simulation storage as a 2-state array. It can be advantageous to use 2-state arrays to model large memories.

Another way to avoid this gotcha is to use SystemVerilog assertions to verify that the redefined values of parameters cannot result in an out-of-bounds access. The assertions only need to execute once at time zero.

A third way, and a preferred modeling style, can be used when the values of constants are related, such as the SIZE, D_WIDTH and A_WIDTH constants in the preceding example. In this case, the value of one constant can be calculated based on the value of another constant.

```
module RAM #(parameter A_WIDTH = 16,
                       SIZE    = 1<<A_WIDTH,
                       D_WIDTH = $clog2(SIZE)

             (output logic [D_WIDTH-1:0] data_out,
              input  logic [D_WIDTH-1:0] data_in,
              input  logic [A_WIDTH-1:0] addr,
              input  logic               read, write);
```

This solution reduces, but does not completely avoid the gotcha of incorrect parameter sizes. The calculated constant values will be correct, but, since the constants are a **parameter** type, they could be overridden using parameter redefinition and end up with incorrect values.

To completely avoid the gotcha of incorrect parameter sizes, the calculated constant should be declared as a **localparam**. A **localparam** constant cannot be redefined, ensuring that the calculated value cannot be overridden. It is not legal to declare **localparam** constants in the module declaration parameter list, however. To use **localparam** values in port declarations, the older Verilog-1995 style of module declarations must be used. For example:

```
module RAM (data_out, data_in, addr, read, write);
  parameter  A_WIDTH = 16;
  localparam SIZE    = 1<<A_WIDTH,
             D_WIDTH = $clog2(SIZE)

  output logic [D_WIDTH-1:0] data_out;
  input  logic [D_WIDTH-1:0] data_in;
  input  logic [A_WIDTH-1:0] addr;
  input  logic               read, write);
```

(Note: the **$clog2** function used in the example above was added in the Verilog-2005 standard. Prior to 2005, this function had to be written by the designer, using either a recursive Verilog function or using the Verilog PLI.)

Gotcha 41: Out-of-bounds assignments to enumerated types

Gotcha: My enumerated state machine variables have values that don't exist in the enumerated definition.

Synopsis: Enumerated types are strongly typed, but can still have values other than those in their enumerated list.

Verilog is a loosely typed language. Any data type can be assigned to a variable of a different type without an error or warning. Unlike Verilog, the SystemVerilog enumerated type is, in theory, a strongly typed variable. Part of the definition of an enumerated type variable is the legal set of values for that variable. For example:

```
typedef enum bit [2:0] {HOLD  = 3'b001,          // 2-state type
                        LOAD  = 3'b010,
                        STORE = 3'b100} states_t;

states_t state_e, nstate_e; // two enumerated variables
```

A surprising gotcha is that an enumerated type variable can have values that are outside of the defined set of values.

There are two parts to this gotcha of out-of-bounds enumerated values, which are explained in more detail, below.

Part One: Unitialized enumerated variables

As with all static Verilog and SystemVerilog variables, enumerated variables begin simulation with a default value. For enumerated variables, this default is the uninitialized value of its base data type. In the preceding example, the base data type of `state_e` is a 2-state **`bit`** type, which begins simulation with an uninitialized value of zero. This value is not in the variable's enumerated list, and is, therefore, out-of-bounds. *Gotcha!*

How to avoid this Gotcha

In actuality, this gotcha can be a desirable feature of the language. If the uninitialized enumerated variable value is out-of-bounds, it is a clear indication that the design has not been properly reset. This is even more obvious if the base data type is a 4-state type, which has an uninitialized value of X.

Part Two: Using casting with enumerated variables

SystemVerilog requires that any procedural assignment to an enumerated variable be in the enumerated list, or from another variable of the same enumerated type.

The following examples illustrate legal and illegal assignments to state_e.

```
nstate_e = LOAD;          // legal assignment
nstate_e = state_e;       // legal assignment
nstate_e = 5;             // illegal (not an enum label)
nstate_e = 3'b001;        // illegal (not an enum label)
nstate_e = state_e + 1;   // illegal (not an enum label)
```

SystemVerilog allows a normally illegal assignment to be made to an enumerated variable using casting. For example:

```
nstate_e = states_t'(state_e + 1);  // legal, but GOTCHA!
```

When a value is cast to an enumerated type, the value is forced into the variable without any type checking. In the example above, if state_e had the value of HOLD (3'b001), then state_e + 1 would result in the value of 3'b010. This can be forced into the nstate_e variable using casting. As it happens, this value matches the value of LOAD. If, however, state_e had the value of LOAD, then state_e + 1 would result in the value of 3'b011. When this value is forced into the enumerated variable nstate_e, it does not match any of the enumerated labels. The nstate_e variable now has an out-of-bounds value. *Gotcha!*

How to avoid this Gotcha

There are two ways to avoid this gotcha. Instead of using the static cast operator, the SystemVerilog dynamic **$cast** function can be used. Dynamic casting performs run-time error checking, and will not assign an out-of-bounds value to an enumerated variable. The general syntax of the **$cast** function is:

```
success_flag = $cast(target_variable, expression)
```

The **$cast** function converts the expression to the type of the target variable. If the expression is a legal value for the target variable, **$cast** returns 1 and makes the assignment. If the value of the expression is not legal, **$cast** returns 0 and leaves the target variable unchanged. The return value of **$cast** can be tested with an assertion.

```
assert(nstate_e, state_e+1); // increment to next label in list
else nstate_e = LOAD;
```

SystemVerilog enumerated types have several built-in methods which can manipulate the values of enumerated variables, and, at the same time, ensure the variable never goes out-of-bounds. For example, the **.next()** method will increment an enumerated variable to the next label in the enumerated list, rather than incrementing by the value of 1. If the enumerated variable is at the last label in the enumerated list, **.next()** will wrap around to the first label in the list. An example of using the **.next()** method is:

```
nstate_e = state_e.next(1); // increment to next label in list
```

Gotcha 42: Undetected shared variables in modules

Gotcha: My RTL model output changes values when it shouldn't, and to unexpected values.

Synopsis: Variables written to by multiple processes create shared resource conflicts.

Syntactically, Verilog and SystemVerilog variables declared at the module level can be read or written by any number of **initial** or **always** procedural blocks within the module. Reading a variable from multiple procedural blocks is fine, and provides a way for parallel processes to pass values between themselves. But, there is a gotcha when two or more procedural blocks write to the same variable. The effect is that the same piece of storage is shared by all the procedural blocks. Since these procedural blocks run concurrently, it is possible, and likely, that the code within the blocks will collide, and interfere with each other's functionality.

The following example shows a common—and perhaps not obvious in large models—Verilog/SystemVerilog gotcha, where the variable `result` is shared by two concurrent **always** procedural blocks.

```
module chip (output logic [31:0] result,    // local variable
             input  logic [31:0] a, b, c, d);

  always @(a or b)
    result = a & b;  // this process writes to result

  // dozens of lines of code later...

  always @(c or d)
    result = c | d;  // GOTCHA: this process also writes to result
endmodule
```

How to avoid this Gotcha using Verilog

Verilog does not restrict how variables are used, which provides versatility in writing test programs and abstract bus functional models. In an RTL model that is intended to be synthesized, such as the example above, however, this versatility becomes a *gotcha*. When using Verilog without the SystemVerilog extensions, the only way to avoid this gotcha is to use software tools such as lint tools (coding style checkers) to check for multiple processes writing to the same variable.

A better way to avoid this gotcha is to use SystemVerilog, which enforces proper RTL coding rules, as shown in the following explanation.

How to avoid this Gotcha using SystemVerilog

For RTL models, a simple way to avoid this gotcha is to use SystemVerilog's **always_comb**, **always_ff**, **always_latch**, and continuous **assign** to assign values to a variable. These processes make it illegal for a variable to be written to by multiple processes. If the code is for verification or an abstract bus functional model, the way to avoid this gotcha is to use SystemVerilog's inter-process synchronization (event triggers, semaphores or mailboxes) to prevent concurrent processes from writing to the same variable at the same time.

Gotcha 43 on page 96 and discusses similar problems with shared variables in interfaces, packages, tasks and functions.

Gotcha 66 on page 145 illustrates another common gotcha with shared variables used in **for** loops.

Gotcha 76 on page 160 shows some gotchas with shared variables in verification code.

Gotcha 43: Undetected shared variables in interfaces and packages

Gotcha: Variables in my package keep changing at unexpected times and to unexpected values.

Synopsis: Interface, package and global variables written to by multiple design and/or verification blocks create shared resource conflicts.

SystemVerilog compounds the Verilog shared variable gotcha described in Gotcha 42 on page 94 by providing more places where shared variables can be declared (or obfuscated). In SystemVerilog, variables can be declared in external spaces outside of a module. These external declaration spaces are user-defined packages, `$unit` (a type of built-in package), and interfaces. These externally declared variables can then be referenced by multiple modules, creating a shared variable.

Multiple **`initial`** and **`always`** procedural blocks that write to shared variables will likely interfere with each other. These procedural blocks can be in different design and verification blocks, which are generally in different files. This can make it very difficult to find and debug shared variable conflicts. *Gotcha!*

```
package sig_defs;
  logic [31:0] result, pipe;
endpackage

module blk1 (output logic [31:0] d_out,
             input  logic [31:0] a, b,
             input  logic        clk, rstn);

  import sig_defs::*;

  always @*
    result = a & b;          // GOTCHA! shared variable

  always @(posedge clk or negedge rstn)
    if (!rstn) begin
      pipe  <=0;
      d_out <=0;
    end
    else begin
      pipe  <= result;       // GOTCHA! shared variables
      d_out <= pipe;
    end
endmodule
```

```
module blk2 (output logic [31:0] d_out,
             input  logic [31:0] c, d,
             input  logic        clk, rstn);

  import sig_defs::*;

  always @*
    result = a | b;          // GOTCHA! shared variable

  always @(posedge clk or negedge rstn)
    if (!rstn) begin
      pipe  <=0;
      d_out <=0;
    end
    else begin
      pipe  <= result;       // GOTCHA! shared variables
      d_out <= pipe;
    end
endmodule
```

How to avoid this Gotcha

Shared variables are generally not synthesizable, and should not be used in RTL models. They can easily be avoided by using SystemVerilog's **always_comb**, **always_ff**, **always_latch**, and continuous **assign** to assign values to variables. With these processes, it is illegal for a variable to be written to by more than one process, even when these processes are in different modules, interfaces or test programs.

```
module blk1 (...);

  import sig_defs::*;

  always_comb
    result = a & b;     // ERROR! multiple processes write to result
  ...
endmodule

module blk2 (...);

  import sig_defs::*;

  always_comb
    result = a | b;     // ERROR! multiple processes write to result

  ...
endmodule
```

Shared variables can be useful in verification code, but care must be taken to avoid conflicts between processes sharing the same storage. This can be accomplished through the use of process synchronization, such as event triggers, semaphores or mailboxes.

Chapter 4

Operator Gotchas

Gotcha 44: Assignments in expressions

Gotcha: I need to do an assignment as part of an if condition, but cannot get my code to compile.

Synopsis: SystemVerilog allows assignments within expressions, with the same gotchas as C, but SystemVerilog's syntax is different from C, confusing programmers familiar with C.

In Verilog, assignments are not allowed within an expression. Therefore, the common C gotcha of `if (a=b)` is illegal. Unfortunately, this also means the useful application of an assignment within an expression is also illegal, such as: `while (data = fscanf(...) ....`

SystemVerilog extends Verilog, and adds the ability to make an assignment within an expression. Thus, with SystemVerilog, the intentional usage of this capability, such as to exit a loop on zero, is legal. SystemVerilog requires that the assignment be enclosed in parentheses to prevent unintentional uses of this capability, such as `if (a=b)`. Thus:

```
if (a=b) ...       // illegal in SystemVerilog
if ( (a=b) ) ...   // legal in SystemVerilog; probably not useful
while ((a=b)) ... // legal in SystemVerilog, and can be useful
```

Ironically, in an effort to prevent the common C gotcha of `if (a=b)`, the SystemVerilog syntax becomes a gotcha. Speaking from the personal experience of one of the authors, programmers familiar with C will attempt, more than once,

Note: the code examples in this chapter are contrived in order to illustrate each gotcha using small examples. In real design and verification code, these gotchas might not be as obvious or easy to debug.

to use the C-like syntax, and then wonder why the tool is reporting a syntax error. Is it an error because, like Verilog, assignments in an expression are not allowed? Is the error because the tool has not implemented the capability? No, it is an error because SystemVerilog's syntax is different from C's. *Gotcha!*

How to avoid this Gotcha

The SystemVerilog syntax can help prevent the infamous C gotcha of `if (a=b)`. The gotcha of a different syntax cannot be avoided, however. Engineers must learn, and remember, that C and SystemVerilog use a different syntax to make an assignment within an expression.

Gotcha 45: Self-determined versus context-determined operators

Gotcha: In some operations, my data is sign extended and in other operations it is not sign extended, and in yet other operations it is not extended at all.

Synopsis: Some Verilog and SystemVerilog operators are context-determined, other operators are self-determined. The type of operation determines if and how vectors are expanded.

What should happen if a 4-bit vector is ANDed with a 6-bit vector, and the result is assigned to an 8-bit vector? Will the results be different if one or both of the AND operands are signed or unsigned? Does the result change if the vector to which the operation is assigned is signed or unsigned?

Verilog and SystemVerilog are "loosely typed" languages. Loosely typed does not mean there are no data type rules. Rather, loosely typed means that the language has built-in rules for performing operations on various data types, and for assigning one data type to another data type. The most subtle of these rules is whether an operator is "*self-determined*" or "*context-determined*". If an engineer does not understand the difference between these two operation types, he or she may find the result of the operation to be different from expected. *Gotcha!*

A ***context-determined operator*** looks at the size and data types of the complete statement before performing its operation. All operands in the statement are expanded to the largest vector size of any operand before the operations are performed. Consider the following example:

```
logic [5:0] a = 6'b010101;    // 6-bit vector
logic [3:0] b = 4'b1111;      // 4-bit vector
logic [7:0] c;                // 8'bit vector

c = a & b;                    // results in 8-bit 00000101
```

In this example, the ***context*** of the bitwise AND operation includes the vector sizes of `a`, `b` and `c`. The largest vector size is 8 bits. Therefore, before doing the operation, the 4-bit vector and the 6-bit vector are expanded to 8-bit vectors.

Why were `a` and `b` left extended with zeros? This question is answered in Gotcha 46 on page 105, which discusses zero-extension and sign-extension in Verilog.

A ***self-determined operator*** is only affected by the data types of its operands. The operation is not affected by the context in which it is performed. For example, a unary AND operation will AND all the bits of its operand together, without changing the size of the operand.

For example:

```
logic [5:0] a = 6'b101010;    // 6-bit vector
logic [3:0] b = 4'b1111;      // 4-bit vector
logic [7:0] c;                // 8'bit vector

c = a | &b;                   // results in 8-bit 00101011
```

In this example, the unary AND of b (&b) is self-determined. The vector sizes of a and c have no bearing on the unary AND of b. The result of ANDing the bits of 4'b1111 together is a 1'b1.

If the self-determined operator is part of a compound expression, as in the example above, then the result of the self-determined operator becomes part of the context for the rest of the statement.

What if &b had been context-determined? In context, b would first be expanded to 8 bits wide, becoming 00001111. The unary AND of this value is 1'b0, instead of 1'b1. The result of a | &b would be 00101010, which would be the wrong answer. But this is not a gotcha, because the unary AND operator is self-determined, and therefore gets the correct answer.

How to avoid this Gotcha

Verilog generally does the right thing. Verilog's rules of self-determined and context-determined operators behave the way hardware behaves (at least most of the time). The gotcha is in not understanding how Verilog and SystemVerilog operators are evaluated, and therefore expecting a different result. The only way to avoid the gotcha is proper education on Verilog and SystemVerilog. Table 4-1, below, should help. This table lists the Verilog and SystemVerilog operators, and whether they are self-determined or context-determined.

Table 4-1: Determination of Operand Size and Sign Extension[1]

Operator	Operand Extension Determined By	Notes
Assignment statements = <=	context	Both sides of assignment affect size extension. Only right-hand side affects sign extension[2].
Assignment operations += -= *= /= %= &= |= ^=	context	Both sides of assignment affect size extension. Left operand is part of the right-hand side assignment context (e.g. a += b expands to a = a + b).

Table 4-1: Determination of Operand Size and Sign Extension[1] (continued)

Operator	Operand Extension Determined By	Notes
Assignment operations <<= >>= <<<= >>>=	see notes	Left operand is context-determined. Right operand is self-determined. Left operand is part of the right-hand side assignment context. (e.g. a <<= b expands to a = a << b)
Conditional ?:	see notes	First operand (the condition) is self determined. Second and third operands are context-determined.
Arithmetic + - * / %	context	
Arithmetic Power **	see notes	Left operand (base) is context-determined. Right operand (exponent) is self-determined.
Increment and Decrement ++ --	self	
Unary Reduction ~ & ~& | ~| ^ ~^ ^~	self	Result is a self-determined, unsigned, 1-bit value.
Bitwise ~ & | ^ ~^ ^~	context	
Shift << <<< >> >>>	see notes	Left operand is context-determined. Right operand (shift factor) is self-determined.
Unary Logical !	self	Result is a self-determined, unsigned, 1-bit value.
Binary Logical && ||	self	Result is a self-determined, unsigned, 1-bit value.
Equality == != === !== ==? !=?	context	Result is a self-determined, unsigned, 1-bit value.

Table 4-1: Determination of Operand Size and Sign Extension[1] (continued)

Operator	Operand Extension Determined By	Notes
Relational `< <= > >=`	context	Result is a self-determined, unsigned, 1-bit value.
Concatenation `{} {{}}`	self	Result is unsigned.

Bit and Part Select
`[] [:]`
`[+:] [-:]`
	self	Result is unsigned.

[1] This table only reflects operations where the operands are vectors. There are also rules for when operands are real (floating point) numbers, unpacked structures, and unpacked arrays, which are not covered in this book.

[2] An assignment in an expression can be on the right-hand side of another assignment (e.g. `d = (a = b + 5) + c;`). In this case, the left-hand side expression of the assignment-in-an-expression is part of the context of the right-hand side of the assignment statement (i.e. `a` in the example does not affect the sign context of `b + 5`, but does affect the sign context of the `+ c` operation).

Additional note: If a context-determined operation is an operand to a self-determined operation, the context of the context-determined operation is limited to its operands, instead of the full statement. E.g., in `d = a >> (b + c);`, the context of the ADD operation is only `b` and `c`.

Self-determined and context-determined operations affect the gotchas described in Gotchas 46, 47 and 48, which follow (with some interesting examples).

Gotcha 46: Operation size and sign extension in assignment statements

Gotcha: I declared my outputs as signed types, but my design is still doing unsigned operations.

Synopsis: In an assignment statement, sign extension context is only dependent on the right-hand side of the assignment.

Operation sign extension is controlled by the operands of the operator, and possibly the context in which the operation is performed. A *self-determined operator* is only affected by the data types of its operands. A *context-determined operator* is affected by the size and data types of all operands in the full expression. Gotcha 45 on page 101, Table 4-1, lists which operators are self-determined and which are context-determined.

Before a context-determined operation is evaluated, its operands are first expanded to the largest vector width in the operation context. There are three steps in this operand expansion, and *these steps use different context rules!*

Step 1. Evaluate the size and sign that will result from all self-determined operations on the right-hand and left-hand sides of the assignment. This information will be used in the subsequent steps.

Step 2. Determine the largest vector size in the context. The context is the largest vector on both the right-hand and left-hand sides of assignment statements.

Step 3. Expand all context-determined operands to the largest vector size by left-extending each operand. The expansion will either zero-extend or sign-extend, based on the operation context, as follows:

- If *any* operand or self-determined operation result on the right-hand side of the assignment is unsigned, then all operands and self-determined operation results on the right-hand side are treated as unsigned, and the smaller vectors are left extended with zeros.
- If *all* operands and self-determined operation results on the right-hand side of the assignment are signed, then all operands and self-determined operation results on the right-hand side are left extended using sign extension.

Note the difference in steps 2 and 3! The context for largest vector size is both sides of an assignment statement, whereas the context for sign extension is just the right-hand side of the assignment containing the operation.

Verilog's rules for operand expansion reflect how hardware works. The following examples illustrate cases where Verilog's rules work as one would expect (no gotchas).

```
logic        [3:0] u1, u2;    // unsigned 4-bit vectors
logic signed [3:0] s1, s2;    // signed 4-bit vectors

logic        [7:0] u3;        // unsigned 8-bit vector
logic signed [7:0] s3;        // signed 8-bit vector
logic              o;         // unsigned 1-bit vector

u3 = u1 + u2;  // zero extension (unsigned = unsigned + unsigned)

s3 = s1 + s2;  // sign extension (signed = signed + signed)

s3 = s1 + 1;   // sign extension (signed = signed + signed)

s3++;          // sign extension (expands to s3 = s3 + 1,
               // which is signed = signed + signed)

u3 += 2'b11;   // zero extension (expands to u3 = u3 + 2'b11,
               // which is unsigned = unsigned + unsigned)

s3 += 2'sb11;  // sign extension (expands to s3 = s3 + 2'sb11,
               // which is signed = signed + signed)
```

A gotcha can occur when an engineer doesn't understand the operand expansion rules. The next examples show some operation results that might be different from expected. These examples use the same declarations as the examples above.

```
s3 = u1 + u2;  // GOTCHA? zero extension, even though S3 is signed
               // Rule: left-hand side does not affect sign
               // extension context of operands on right-hand side

u3 = s1 + s2;  // GOTCHA? sign extension, even though U3 is unsigned
               // Rule: left-hand side does not affect sign
               // extension context of operands on right-hand side

s3 = s1 + u2;  // GOTCHA? zero extension, even though s1 and S3
               // are signed
               // Rule: unsigned type on right-hand side means the
               // entire right-hand side context is unsigned

s3 = s1 + 1'b1;// GOTCHA? zero extension, even though s1 and S3
               // are signed
               // Rule: unsigned type on right-hand side means the
               // entire right-hand side context is unsigned

s3 += 2'b11;   // GOTCHA? zero extension, even though s3 is signed
               // (operation is same as: s3 = s3 +2'b11)
               // Rule: unsigned type on right-hand side means
               // entire right-hand side context is unsigned

u3 += 2'sb11;  // GOTCHA? zero extension, even though the 2'sb11
               // is signed (operation is same as: u3 = u3 +2'sb11)
               // Rule: unsigned type on right-hand side means the
               // entire right-hand side context is unsigned
```

A compound expression can contain a mix of self-determined operations and context-determined operations. In this case, the resultant type of the self-determined operation is used to determine the types that will be used by the context-determined operations. The following examples use the same declarations as the previous examples.

```
{o,u3} = u1 + u2; // First evaluate the self-determined
                  // concatenation on the left-hand side.
                  // This affects the size context of operations
                  // on the right-hand side (which are expanded
                  // to 9-bit size of the concatenation result)

u3 = u1 + |u2;    // First do unary OR of 8-bit vector u3
                  // (self-determined) then zero-extend the 1-bit
                  // unary OR result to 8 bits before doing the
                  // context-determined math operation

s3 = s1 + |s2;    // GOTCHA? First do unary OR of 4-bit vector s2
                  // (self-determined), then zero-extend s1 and the
                  // 1-bit unary OR result to 8 bits (even though s1
                  // is a signed type, the |s2 result is unsigned,
                  // and therefore the right-hand side context
                  // is unsigned)
```

The gotcha of zero extension versus sign extension is, in reality, a useful feature of the Verilog and SystemVerilog languages. A single operator token, such as +, can model an adder with or without overflow, depending on the largest vector size in the context of the operation. The same + operator can model either a signed adder or an unsigned adder, again depending on the context of the operation.

How to avoid this Gotcha

The gotcha of operand expansion comes from not understanding when vector expansion will occur, and whether the vector will be zero-extended or sign-extended. To avoid this gotcha, engineers must know the underlying loosely typed rules of Verilog and SystemVerilog. Once the rules are understood, engineers must use the correct sizes and data types for the intended type of operation. Verilog-2001 provides control over the signedness of an operand with the `$signed()` and `$unsigned()` functions. SystemVerilog gives engineers more control over the application of these expansion rules through the use of type casting, size casting, and signedness casting. For example (assuming the same declarations as in the examples above):

```
s3 = s1 + u2;    // GOTCHA? zero extension (u2 is unsigned)

s3 = 8'(s1) + signed'(u2);  // OK, cast s1 to 8 bits wide (self-
                            // determined) cast u2 to signed and
                            // do sign extension
```

Gotcha 47: Signed arithmetic rules

Gotcha: My signed adder model worked perfectly until I added a carry-in input, and now it only does unsigned addition.

Synopsis: The entire right-hand side context of an assignment must be signed, in order to have signed arithmetic operations.

Gotcha 13 on page 32 discusses some of the gotchas with literal integer sign extension rules, and Gotcha 46 on page 105 covers gotchas with sign extension in operations. This gotcha covers important gotchas when performing arithmetic operations on signed data.

Verilog and SystemVerilog overload the math operators so that they can represent several types of hardware. For example, the + operator can represent:

- An adder of any bit width with no carry-in or carry-out
- An adder of any bit width with no carry-in but with carry-out
- An adder of any bit width with carry-in and with carry-out
- An unsigned adder
- A signed adder
- A single-precision floating point adder
- A double-precision adder

The type of arithmetic performed is controlled by the types of the operands and the context of the operation. In order to perform signed operations, all operands must be signed. Arithmetic operators are context-determined. Not only must the operands to the arithmetic operator be signed, all other operands on the right-hand side of an assignment must also be signed.

The example below is a signed adder with no gotchas, that simulates and synthesizes correctly.

```
module signed_adder_no_carry_in
(input  logic signed [3:0] a, b,    // signed 4-bit inputs
 output logic signed [3:0] sum,     // signed 4-bit output
 output logic              co);     // unsigned 1-bit output

  assign {co,sum} = a + b;          // signed 5-bit adder
endmodule
```

In the example above, the left-hand side concatenation is a self-determined expression that defines a 5-bit unsigned vector. The size of the left-hand side affects the right-hand side ADD operation, but the signedness of the left-hand side has no bearing on operations. All operands on the right-hand side of the

assignment are signed, which does affect the add operation. In this context, the ADD operator performs a 5-bit signed operation.

Using an unsigned carry-in. The next example is almost the same, but adds a 1-bit carry-in input. This example has a gotcha! It does not simulate or synthesize as a signed adder.

```
module signed_adder_with_carry_in
(input  logic signed [3:0] a, b, // signed 4-bit inputs
 input  logic              ci,   // unsigned 1-bit input
 output logic signed [3:0] sum,  // signed 4-bit output
 output logic              co);  // unsigned 1-bit output

  assign {co,sum} = a + b + ci;  // GOTCHA! unsigned 5-bit adder
endmodule
```

In simulation, the only indication that there is a problem is in the value of the result when either `a` or `b` is negative. Synthesis tools will issue a warning message to the effect that `a` and `b` were coerced to unsigned types. The reason for this coercion is that Verilog's arithmetic operators are context-determined. Even though `a` and `b` are signed, one of the operands in the compound expression, `ci`, is unsigned. Therefore, all operands are converted to unsigned values before any context determined operation is performed. *Gotcha!*

Using a signed carry-in. Declaring the 1-bit carry-in input as a signed type seems like it would solve the problem. This change is illustrated below.

```
module signed_adder_with_carry_in
(input  logic signed [3:0] a, b,   // signed 4-bit inputs
 input  logic signed       ci,     // signed 1-bit input
 output logic signed [3:0] sum,    // signed 4-bit output
 output logic              co);    // unsigned 1-bit output

  assign {co,sum} = a + b + ci;    // GOTCHA! ci is subtracted
endmodule
```

Now all operands on the right-hand side are signed, so a signed operation will be performed, right? No. *Gotcha!*

The example above does signed arithmetic, but uses incorrect sign extension—at least incorrect for the intended functionality. The gotcha again relates to the ADD operator being context-determined. As such, all operands are first expanded to the vector size of the largest operand, which is the 5-bit self-determined concatenate operator on the left-hand side of the assignment. Before the addition operations are performed, `a`, `b` and `ci` are sign-extended to be 5-bits wide. This is correct for `a` and `b`, but is the wrong thing to do for `ci`. If `ci` has a value of 0, sign-extending it to 5 bits will be `5'b00000`, which is still zero. However, if `ci` is 1, sign-extending it to 5 bits will be `5'b11111`, which is negative 1, instead of positive 1. The result of the ADD operation is `a + b + -1`. *Gotcha!*

Using sign casting. Verilog-2001 introduced the `$signed` and `$unsigned` conversion functions, and SystemVerilog adds sign casting. Sign casting allows changing the signedness of an operand. The following example uses sign casting to try to fix the signed adder problem.

```
module signed_adder_with_carry_in
(input  logic signed [3:0] a, b,   // signed 4-bit inputs
 input  logic              ci,     // unsigned 1-bit input
 output logic signed [3:0] sum,    // signed 4-bit output
 output logic              co);    // unsigned 1-bit output

  assign {co,sum} = a + b + signed'(ci); // GOTCHA! ci is subtracted
endmodule
```

Casting the sign of the carry-in introduces the same gotcha as declaring carry-in as signed. When carry-in is set, it is sign-extended to 5 bits, making the carry-in a negative 1. *Gotcha!*

How to avoid this Gotcha

The real problem is that a signed 1-bit value cannot represent both a value and a sign bit. Declaring or casting a 1-bit value to signed creates a value where the value and the sign bit are the same bit, which does not represent true hardware.

The correct way to avoid this signed arithmetic gotcha is to cast the 1-bit carry-in input to a 2-bit signed expression, as follows:

```
  assign {co,sum} = a + b + signed'({1'b0,ci});  // signed 5-bit
                                                 // adder
```

The `signed'({1'b0,ci})` operation creates a 2-bit signed operand, with the sign bit always zero. When the 2-bit signed value is sign-extended to the size of the largest vector in the expression context, the sign extension will zero-extend, maintaining the positive value of the carry-in bit.

Gotcha 48: Bit-select and part-select operations

Gotcha: All my data types are declared as signed, and I am referencing the entire signed vectors in my operations, yet I still get unsigned results.

Synopsis: The result of a part-select operation is always unsigned, even when the entire vector is selected.

Selecting a bit of a vector, or a part of a vector, is an operation. The bit-select and part-select operators always return an unsigned value, even if the vector itself is signed. This change in signedness can be unexpected, and is another source for signed arithmetic gotchas.

```
parameter MSB = 31;

logic signed [MSB:0] a, b;         // signed vectors
logic signed [MSB:0] sum1, sum2;   // signed vectors
logic signed [  7:0] sum3;         // 8-bit signed vector

  assign sum1 = a + b;                    // OK, signed adder
  assign sum2 = a[MSB:0] + b[MSB:0];   // GOTCHA! unsigned adder
  assign sum3 = {a[MSB],a[6:0]} + {b[MSB],b[6:0]};  // GOTCHA!
                                                    // unsigned adder
```

The two gotchas above occur because the result of a part-select operation is always unsigned, and bit-select and part-select operations are self-determined, and therefore evaluated before the context-determined ADD operation. The context for the ADD operation is unsigned.

How to avoid this Gotcha

Since the assignment to `sum2` is selecting the full vectors of `a` and `b`, one easy way to avoid this gotcha is to just not do a part-select, as in the assignment to `sum1`. However, code is often generated by software tools, which may automatically use part-selects, even when the full vector is being selected. Part selects are also commonly used in heavily-parameterized models, where vector sizes can be redefined. For the `sum3` example above, there is no choice but to do a part-select, since only part of the `a` and `b` vectors are being used. When a part-select of a signed vector must be used, the correct modeling style is to cast the result of the part-select to a signed value. Either the Verilog-2001 `$signed` function or SystemVerilog sign casting can be used. For example:

```
assign sum2 = $signed(a[MSB:0]) + $signed(b[MSB:0]);
assign sum3 = signed'({a[MSB],a[6:0]}) + signed'({b[MSB],b[6:0]});
```

Gotcha 49: Increment, decrement and assignment operators

Gotcha: I'm using the ++ operator for my counter; the counter value is correct, but other code that reads the counter sees the wrong value.

Synopsis: Increment, decrement, and assignment operations are blocking assignments.

SystemVerilog provides the C-like ++ and -- increment/decrement operators, and the C-like assignment operators such as +=, -=, *= and /=. These are intuitive and useful in C programming, and that usage carries over to modeling verification testbenches in SystemVerilog. But there is a gotcha when using these operators in RTL models of hardware. All of these operators behave as blocking assignments when updating their target variable. In RTL models, blocking assignments are only appropriate for representing combinational logic. If these operators are used to model sequential logic, then a simulation race condition is likely to occur. The following example illustrates such a race condition.

```
always_ff @(posedge clock, negedge reset_n)
  if (reset_n)              fifo_write_ptr = 0;
  else if (!fifo_full)      fifo_write_ptr++;

always_ff @(posedge clock)
  if (fifo_write_ptr == 15) fifo_full <= 1;       // GOTCHA!
  else                      fifo_full <= 0;
```

The first procedural block in this example modifies the value of `fifo_write_ptr` on a clock edge. In parallel, and possibly in a very different location in the source code, the second procedural block is reading the value of `fifo_write_ptr` on the same clock edge. Because the ++ operator performs a blocking assignment update to `fifo_write_ptr`, the update can occur before or after the second block has sampled the value. Both event orders are legal. It is very likely that two different simulators will function differently for this example. *Gotcha!*

How to avoid this Gotcha

The SystemVerilog increment/decrement operators and the assignment operators should not be used in sequential logic blocks. These operators should only be used in combinational logic blocks, as a **for** loop increment, and in contexts where the increment/decrement operand is not being read by a concurrent process. The correct way to model the fifo incrementer is:

```
always_ff @(posedge clock, negedge reset_n)
  if (reset_n)              fifo_write_ptr = 0;
  else if (!fifo_full)      fifo_write_ptr <= fifo_write_ptr + 1;
```

Gotcha 50: Pre-increment versus post-increment operations

Gotcha: My while loop is supposed to execute 16 times, but it exits after 15 times, even though the loop control variable has a value of 16.

Synopsis: Pre-increment versus post-increment can affect the result of some expressions.

Pop Quiz: The following two lines of code do the same thing, right?

```
sum = i++;
sum = i+1;
```

Answer: **No!** (*Gotcha!*)

Like the C language, the SystemVerilog ++ increment operator, or -- decrement operator, can be placed before a variable name (e.g. `++i`) or after a variable name (e.g. `i++`). These two usages are referred to as a *pre-increment* or a *post-increment*, respectively. The result of the operation is the same. The variable is incremented by 1. In many contexts, pre-increment and post-increment can be used interchangeably. In a **`for`** loop step assignment, for example, either pre- or post-increment can be used, with the same results.

```
for (int i=0; i<=255; ++i) ... ;
for (int i=0; i<255; i++) ... ;
```

The two examples are functionally the same in the **`for`** loop examples above because `++i` and `i++` are used as stand-alone statements. No other expression is reading the value of `i` in the same statement in which it is incremented. The statement which follows (the `i<=255` test in the examples above) will see the new value of `i`, regardless of whether it is a pre-increment or a post-increment.

The gotcha, which comes straight from the C language, is when the value of the variable is used within the same statement in which it is being incremented. If the increment operator is before the variable name, the variable is incremented before the value is used in that same statement (*pre-increment*). If the increment operator is placed after the variable, then the value of the variable is used first in the same statement, and then incremented (*post-increment*).

```
i = 10;
j = i++;     // assign i to j, then increment i; j gets 10
j = ++i;     // increment i, then assign result to j; j gets 11
```

Either of the examples above could be correct, depending on what the design or verification engineer intends to do. If, however, the engineer is expecting one functionality but uses the wrong operator, then *Gotcha!*

The effects of pre- and post-increment are less obvious in some contexts. For example:

```
i = 16;

while (i--) ... ;    // test i, then decrement; loop will
                     // execute 16 times

while (--i) ... ;    // decrement i, then test; loop will
                     // execute 15 times
```

Either of the examples above could be correct, depending on what one wants to do. If, however, an engineer is expecting a loop to run a certain number of times, and the wrong operator is used, then *Gotcha!*

How to avoid this Gotcha

The way to avoid this gotcha is to fully understand how pre- and post-increment/ decrement work. Both types of operations are useful, but need to be used with prudence.

Gotcha 51: Modifying a variable multiple times in one statement

Gotcha: When I have multiple operations on a variable in a single statement, I get different results from different simulators.

Synopsis: The evaluation order is undefined when a compound expression modifies the same variable multiple times on the right-hand side of an assignment statement.

SystemVerilog has assignment operators (such as += and -=), and increment/decrement operators (++ and --). These operators both read and modify the value of their operand. Two examples are:

```
j = ++i;          // OK, increment i, then assign result to j
j = (i += 1);     // OK, increment i, then assign result to j
```

Both of these examples modify a variable on the right-hand side of the assignment statement before making the assignment. There is a gotcha, however, if the same variable is modified multiple times in the same expression. For example:

```
i = 10;
j = --i + ++i;  // GOTCHA! multiple operations on same variable
```

In this example, the value of i is both read and modified multiple times on the right-hand side of the assignment statement. The gotcha is that the SystemVerilog standard does not guarantee the order of evaluation and execution of these multiple read/writes to the same variable in the same expression. After execution, the value of j in this example could be 19, 20 or 21 (and perhaps even other values), depending upon the relative ordering of the increment operation and the decrement operation.

How to avoid this Gotcha

This gotcha can be avoided by not using operators which make multiple reads and writes to a variable within the same statement. Synthesis tools do not permit these types of operations, because of the indeterminate results. The correct way to model the example above depends on what order the designer intended the operations on i to occur. One possibility is:

```
always_comb begin
  temp = --i;         // pre-decrement i, save result in temp
  j = temp1 + i++;    // add temp1 and i, then increment i
end
```

Gotcha 52: Operator evaluation short circuiting

Gotcha: I am calling a function twice in a statement, but sometimes only one of the calls is executed.

Synopsis: Simulation might not evaluate all operands in some circumstances.

Software simulation does not always evaluate statements in the same way as hardware. Consider the following example:

```
always @(posedge clock)
  if (mem_en && write) mem[addr] <= data_in; // OK, no side effects
```

In this example, the logical AND operator (`&&`) checks if both `mem_en` and `write` are true. In hardware, this operation is an AND gate. The two inputs are continuously evaluated, and affect the output of the AND gate. In simulation, however, the logical operation is performed from left-to-right. If `mem_en` is false, then the result of the logical AND operation is known, without having to evaluate `write`. Exiting an operation when the answer is known, but before all operands have been evaluated, is referred to as *operation short circuiting*. The Verilog/ SystemVerilog standards explicitly allow, but do not require, tools to short circuit logical AND, logical OR and the `?:` conditional operations. The standards are not clear as to whether other operators can short circuit. It is neither expressly permitted nor expressly prohibited.

Does short circuiting matter? Not in the preceding example. The simulation results of the logical AND operation will match the behavior of actual hardware.

Now consider a slightly different example:

```
always @(posedge clock)
  if ( f(i1,o1) && f(i2,o2) )  // GOTCHA! possible side effects
    mem[addr] <= data_in;

function f(input [7:0] d_in, output [7:0] d_out);
  d_out = d_in + 1;
  if (d_out == 255) return 0;
  else              return 1;
endfunction
```

The function above modifies the value passed into it and passes that result back as a function output argument. In addition, the function returns a status flag. The function is called twice, on the right-side and the left-side of the `&&` operator.

In hardware, the logical AND operator can be implemented as an AND gate, and the function replicated as combinational logic to each input of the gate. As combinational logic, both `o1` and `o2` are continuously updated to reflect their input values.

In software, however, the logical AND operation is evaluated from left-to-right. If the return of the first function call is 0, the result of the AND operation is known without having to evaluate the second operand. A simulator or other tool is permitted to short circuit the operation. If this occurs, the function is not called the second time, and `o2` is not updated to reflect the value of `i2`. *Gotcha!*

How to avoid this Gotcha

The only way to avoid this gotcha is to avoid operands with side effects. A side effect occurs when the operand modifies a value when the operand is evaluated. If the operands do not have side effects, then the behavior of short circuiting are the same, and simulation will correctly match hardware behavior.

To avoid the short circuiting gotcha in the example above, it must be re-coded to ensure that both calls to the function always execute. One way to do this is:

```
always @(posedge clock) begin
  temp1 = f(i1,o1);          // o1 will be updated every time
  temp2 = f(i2,o2);          // o2 will be updated every time
  if ( temp1 && temp2 )      // OK! no side effects
    mem[addr] <= data_in;
end
```

Gotcha 53: The not operator (!) versus the invert operator (~)

Gotcha: My if statement with a not-true condition did not execute when I was expecting it to.

Synopsis: The logical NOT operator and the bitwise invert operator perform different operations and can be used incorrectly.

Engineers new to Verilog, and even a few veterans, sometimes misuse the Verilog *logical NOT* operator (!) and the *bitwise invert* operator (~). In some operations, the results of these operations happen to be the same, but, in other operations, they yield very different results. Consider the following example:

```
logic a;        // 1-bit 4-state variable
logic [1:0] b;  // 2-bit 4-state variable

initial begin
  a = 1;
  b = 1;

  if (!a) ... // evaluates as FALSE
  if (~a) ... // evaluates as FALSE

  if (!b) ... // evaluates as FALSE
  if (~b) ... // evaluates as TRUE -- GOTCHA!
end
```

The gotcha is that the logical NOT operator (!) performs a true/false test first and then inverts the 1-bit test result. On the other hand, the bitwise invert operator (~) just inverts the value of each bit of a vector. If the bitwise invert operation is used in the context of a true/false test, the bit inversion occurs first, and the true/false evaluation is performed second, possibly on a multi-bit value.

Inverting the bits of a vector, and then testing to see whether it is true or false is not the same as testing whether the vector is true or false, and then inverting the result of that test. *Gotcha!*

How to avoid this Gotcha

The bitwise invert operator should never be used to negate logical true/false tests. Logical test negation should use the logical NOT operator.

```
if (!b) ... // OK, logical operator used for true/false test
```

Conversely, the logical NOT operator should never be used to invert a value. The result of logical NOT and bitwise inversion is the same for 1-bit values, but very different for vectors. Even when inverting a 1-bit signal, such as a clock, gotchas will be avoided by using the invert operator for invert operations.

```
always #5 clk = ~clk;  // OK, invert operator used to invert clock
```

Gotcha 54: Array method operations

Gotcha: I get the wrong result when I sum all the values of an array using the built-in .sum method.

Synopsis: Some of the SystemVerilog array methods are context-determined.

SystemVerilog adds several built-in functions for working with arrays, called *array methods*. The gotcha illustrated here uses the **.sum** array method, but similar gotchas exist with several of the array methods.

The **.sum** method returns the total of adding all the values in all elements of the array. In the following example, the `fifties` array holds four 8-bit vectors, which are initialized with several integer values. The values in the array are summed using the **.sum** method, and printed using a **$display** statement.

```
logic [7:0] fifties [0:3] = '{50,100,150,200};  // 8-bit array
$display("fifties.sum is %0d", fifties.sum);    // GOTCHA!
```

This example sums up **50 + 100 + 150 + 200** and returns the clearly incorrect total of **244**. *Gotcha!*

The reason for this result is that the **.sum** method, and several other array methods, are context-determined. With context-determined operations, the vector size of the operation is based on the context in which the operation is used. Gotcha 45 on page 101 explains the rules for context-determined operations.

In this example, each member of the array is 8-bits wide. There are no other values involved in the **.sum** operation, so the context of the operation is 8-bit values. As the values of the array are summed, the most-significant bits of any result that overflows the 8-bit size are truncated. That is, in context, the array is summed with an 8-bit adder with no carry.

How to avoid this Gotcha

This gotcha can be avoided by using the array method in a context with a vector size that can hold the maximum value the method could return. Two of many ways the operation context can be changed are shown below.

One simple way to change the size context is to add a literal value of 0 to the method return. The following example adds a 16-bit literal 0 to the **.sum** return. This makes the context of the operation 16 bits, which is sufficient to hold the maximum value. This example will return the correct sum of 500.

```
$display("fifties.sum is %0d", (fifties.sum + 16'd0));  // OK
```

A second way to change the context vector size is to assign the method return to a larger vector. The size context for operations is based on both the right-hand and left-hand sides of assignment statements. The following example also returns the correct sum of 500.

```
int total;  // 32-bit integer variable

total = fifties.sum
$display("total is %0d", total);  // OK
```

Gotcha 55: Array method operations on an array subset

Gotcha: I get the wrong answer when I sum specific array elements in an array.

Synopsis: Using sum with(), returns a sum of the with() expressions, not a sum of a subset of array element values.

SystemVerilog provides a number of array methods to search and manipulate data within arrays. These methods operate only on unpacked arrays, and include array searching, array ordering and array reduction.

The **.sum** method, an array reduction method, returns the sum of the values stored in all elements of an array. An optional **with()** clause can be used to filter out some array values. But, when using **.sum with()**, there is a subtle gotcha.

In the following example, the intent is to sum up all values in the array that are greater than 7.

```
program automatic test;
  initial begin
    int count, a[] = '{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    count = a.sum with (item > 7);   // GOTCHA: expect 17, get 2
    $display("\"a.sum with(item > 7)\" returns %0d", count);
  end
endprogram
```

When the optional **with()** clause is used, the **.sum** method adds up the return values of the expression inside the **with()** clause, instead of summing the values of the array elements. In the example above, the `(item > 7)` is a true/false expression, which is represented with the values 1'b1 or 1'b0. If the array contains the values `{9,8,7,3,2,1}`, then the true/false test for each array element returns the set of values `{1'b1,1'b1,1'b0,1'b0,1'b0,1'b0}`. The sum of these true/false values is 2. *Gotcha!*

Other array methods that have a with() clause can have a similar gotcha.

How to avoid this Gotcha

The true/false result of the relational expression can be used to select just the element values where the test is true. Two simple ways to do this are:

```
count = a.sum with( (item > 7) ? item : 0 );
count = a.sum with( (item > 7) * item );
```

Chapter 5

General Programming Gotchas

Gotcha 56: Verifying asynchronous and synchronous reset at time zero

Gotcha: Sometimes my design resets correctly at time zero, and sometimes it fails to reset.

Synopsis: Initial procedural blocks can activate in any order relative to always procedural blocks.

A common verification gotcha is not understanding the event scheduling of **`initial`** and **`always`** procedural blocks. Because of the name "initial", some engineers assume that initial blocks are executed before **`always`** blocks. Other engineers believe just the opposite is true, that **`initial`** blocks are guaranteed to execute after all **`always`** blocks are active.

In which order should the initial and always blocks be started? If the verification goal is to test resetting the design at time zero, the answer to this question is not straightforward. It depends if the design is modeled with asynchronous resets or synchronous resets.

The following example illustrates a testbench where the goal is to reset the design at time zero, and the design uses asynchronous resets.

Note: the code examples in this chapter are contrived in order to illustrate each gotcha using small examples. In real design and verification code, these gotchas might not be as obvious or easy to debug.

```
module chip (input logic clock, input logic reset_n, ...);
  always_ff @(posedge clock, negedge reset_n)  // asynchronous rest
    if (!reset_n) q <= 0;
    else          q <= d;
endmodule

module test (input logic clock, output logic reset_n, ...);
  initial begin
    reset_n = 0;            // GOTCHA! activating reset at time zero
    #10 reset_n = 1;
    ...
  end
endmodule

module top;
  logic clock, reset_n, ...;
  test test (.*);           // connect testbench to design
  chip dut  (.*);
  initial begin
    clock = 0;              // OK! first rising clock edge at time 5
    forever #5 clock = ~clock;
  end
endmodule
```

In this example, if a simulator activates the **always_ff** procedural block first, it will encounter the @ timing control in the sensitivity list and suspend execution, while waiting for a negative edge of reset_n. Then, when the **initial** procedural block in module test activates and changes reset_n to 0, the **always_ff** block will sense the change and the flip-flop will reset.

But, the activation order of procedural blocks is not guaranteed. The Verilog and SystemVerilog standards state that **initial** and **always** procedural blocks become active at time zero, *in any order.* The **initial** block in the module test, above, could activate and assign reset_n to 0 before the **always_ff** block activates. In this case, the flip-flop misses the change at time zero on reset_n, and fails to reset asynchronously at time zero. *Gotcha!*

Instead of depending on activation order, specific programming constructs must be used to control when verification events occur. There are ways to avoid this gotcha, and ensure that the **always_ff** block activates first, before reset_n changes to 0 at time zero. Before looking at the solution, however, let's consider what is needed to reset the design at time zero, if the design uses synchronous reset flip-flops, as shown in the following example:

```
module chip (input logic clock, input logic reset_n, ...);
  always_ff @(posedge clock)       // synchronous rest
    if (!reset_n) q <= 0;
    else          q <= d;
endmodule

module test (input logic clock, output logic reset_n, ...);
  initial begin
    reset_n = 0;           // GOTCHA! activating reset at time zero
    #10 reset_n = 1;
    ...
  end
endmodule

module top;
  logic clock, reset_n, ...;
  test test (.*);          // connect testbench to design
  chip dut  (.*);
  initial begin
    clock = 1;          // GOTCHA! first rising clock edge at time
zero
    forever #5 clock = ~clock;
  end
endmodule
```

There are only two changes between this example and the first example. The **always_ff** has been changed to model synchronous resets, and the clock generator in module `top` has been changed to generate a positive edge of clock at time zero, in order to reset the design at time zero.

The first example with asynchronous resets required the **always_ff** block to activate before reset was initialized. Thus, there was a race condition between the design and the activation of reset. With synchronous resets, the needed behavior is that both the **always_ff** and the **initial** block that activates reset must occur before the initialization of clock at time zero. Since the activation of the three procedural blocks could be in any order, there is a three-way race condition. *Gotcha!*

How to avoid this Gotcha using Verilog

This gotcha is avoided by proper education and understanding of the Verilog event scheduling of concurrent statements. It is also necessary to understand the type of hardware being verified, as different types of hardware require different event ordering.

The simplest way to avoid this gotcha, and one that is highly recommended, is to avoid clocking and resetting a design at time zero. If clock and reset occur any

time after time zero, all procedural blocks will be active, and the order of events is far less critical.

If resetting the design at time zero is needed, the correct mix of blocking and nonblocking assignments must be used. When executed, blocking assignments immediately update the variable on the left-hand side. Nonblocking assignments update variables after a delta in the current simulation time. At simulation time zero, all procedural blocks are guaranteed to be active before this delta.

If the design uses asynchronous resets, both clock and reset should be initialized using nonblocking assignments.

```
module test (input logic clock, output logic reset_n, ...);
  initial begin
    reset_n <= 0;           // OK, reset activated after delta
    #10 reset_n = 1;
    ...
  end
endmodule

module top;
  ...
  initial begin
    clock <= 1;             // OK, first rising clock after delta
    forever #5 clock = ~clock;
  end
endmodule
```

With asynchronous resets, it does not matter what order clock and reset occur, so long as the value changes occur after the time zero delta. Both are listed in the sequential logic sensitivity list, so, no matter what order clock and reset occur, if the sequential logic procedure has been activated, it will reset at time zero.

If the design uses synchronous resets, the order in which the activation of clock and reset occur is important. The reset signal needs to have its active value before the first edge of clock occurs. To guarantee this order of events, only the clock should be initialized using nonblocking assignment. The reset should be assigned with a blocking assignment, so that it is updated to its value before the delta delay.

```
module test (input logic clock, output logic reset_n, ...);
  initial begin
    reset_n = 0;            // OK, reset activated before delta
    #10 reset_n = 1;
    ...
  end
endmodule
```

```
module top;
  ...
  initial begin
    clock <= 1;              // OK, first rising clock after delta
    forever #5 clock = ~clock;
  end
endmodule
```

The scheduling of nonblocking assignments guarantees that all procedural blocks, whether **initial** or **always**, have been activated, in any order, before the assignment takes place.

How to avoid this Gotcha using SystemVerilog

With SystemVerilog, the testbench can be, and should be, modeled using a **program** instead of a **module**.

```
program automatic test (input logic clock, output logic reset_n);
  initial begin
    reset_n <= 0;       // OK, reset activated as verification event
    #10 reset_n <= 1;
    ...
  end
endmodule

module top;
  ...
  initial begin
    clock <= 1;              // OK, first rising clock after delta
    forever #5 clock = ~clock;
  end
endmodule
```

A program block has special event scheduling semantics that help avoid *most* types of test-to-design race conditions. In brief, value changes generated from a program block are scheduled to take place after both blocking and nonblocking assignments that are in modules and interfaces. This scheduling eliminates the gotcha with scheduling asynchronous resets at simulation time zero in the correct order. Note, however, that the program block does not solve the gotcha described above for scheduling synchronous resets at time zero.

See Gotcha 7 on page 22 for coding guidelines with program blocks.

Gotcha 57: Nested if...else blocks

Gotcha: My else branch is pairing up with the wrong if statement.

Synopsis: An else construct pairs with the nearest if statement that does not have an else; begin...end can override this default pairing.

The **else** branch of a Verilog/SystemVerilog **if...else** statement is optional. This can lead to confusion when **if...else** statements are nested within other **if...else** statements, and some of the optional **else** statements are not specified. Which **else** goes to which **if**? The following example is a gotcha...

```
if (a >= 5)
  if (a <= 10)
    $display (" 'a' is between 5 and 10");
else
  $display (" 'a' is less than 5");   // GOTCHA! pairs with wrong if
```

The indentation of the code above implies that the **else** statement goes with the first **if** statement, but that is not how Verilog/SystemVerilog works. The language rules state that an **else** statement is automatically associated with the nearest previous **if** statement that does not have an **else**. Indentation does not change the language pairing rules.

Therefore, the example above, with correct indentation and $display statements, is actually:

```
if (a >= 5)
  if (a <= 10)
    $display (" 'a' is between 5 and 10");
  else
    $display (" 'a' is greater than 10");  // CORRECT!
```

How to avoid this Gotcha

The automatic **if...else** association can be overridden using **begin...end** to explicitly show which statements belong within an **if** branch. The first example, above, can be correctly coded as follows:

```
if (a >= 5) begin
  if (a <= 10)
    $display (" 'a' is between 5 and 10");
end
else
  $display (" 'a' is less than 5");          // CORRECT!
```

A language-aware text editor can also help avoid this gotcha. The editor tool can properly indent nested **if...else** statements, lining up the **else** statement with its corresponding **if** statement.

Gotcha 58: Evaluation of equality with 4-state values

Gotcha: My testbench completely misses problems on design outputs, even though it is testing the outputs.

Synopsis: The equality operators have three answers, true, false and unknown, but if...else decision statements only have two branches.

A common task in verification is to compare data from the design to expected results. Verilog has an equivalence operator ==, that appears to be similar to the C language operator. *They are not the same!* Verilog also has a not-equivalent operator, `!=`, that uses the same token as C, but works differently.

The following excerpt from a testbench might seem reasonable in most programming languages, but with 4-state logic, the code does not work as intended.

```
always @(posedge test_clock)
  if (data != expected)   // GOTCHA!
    $display("Error on data: saw %h, expected %h", data, expected);
```

In this example, if data always has known values, then the verification code will work as intended. But, if data has any bits that are X or Z, the **`if`** branch will not be taken. Most likely, an X or Z value on any bit of data is a design error, but the verification code misses the problem. *Gotcha!*

The gotcha comes from not understanding how the equality/inequality operators handle 4-state values. In brief, the rules are:

- If the two operands are numerically equivalent, the operation result is *true*.
- If the two operands are numerically different, the operation result is *false*.
- If the either operand is not a known number, the operation result is *unknown*.

The true, false, and unknown results are represented with the 1-bit values, `1'b1`, `1'b0`, and `1'bx`, respectively.

The **`if`**...**`else`** decision only has two branches. If the expression tested is true, the **`if`** branch is executed. If the expression is evaluates as either false or unknown, the **`else`** branch is taken. The verification example above traps errors using the **`if`** branch of the decision, and does nothing on the **`else`** branch. Should there be a problem with the design that results in a logic z or x on any bit of the output, the not-equal (`!=`) operator will return unknown, which will not take the **`if`** branch of the verification code. The design error will go undetected.

How to avoid this Gotcha

This gotcha can only be avoided by understanding how 4-state values are evaluated as true or false. Both design and verification engineers need to know that, in order for a 4-state equivalence test to be true or false, no bits in the expression can have a Z or X value.

The verification example above can be corrected by using special verification operators in Verilog, the *identity operators* (=== and !==), instead of the usual programming *equality operators* (== and !=). For example:

```
always @(posedge test_clock)
  if (data !== expected)    // OK, will detect bits with X or Z
    $display("Error on data: saw %h, expected %h", data, expected);
```

The identity operators perform a bit-by-bit comparison of its two operands. The values of each bit are compared for all four logic values. If both bits have identical values, the comparison of those bits evaluates as true. If all bits are identical, the operation returns true. If any bits are different in any way, the operation evaluates as false. The not-identical operator, !==, negates the true/false results.

Gotcha 59: Event trigger race conditions

Gotcha: I'm using the event data type to synchronize processes, but sometimes when I trigger an event, the sensing process does not activate.

Synopsis: An event that is triggered in the same time step in which a process begins looking for the event may not be sensed.

Verilog provides a basic inter-process synchronization mechanism via the **event** data type, the -> event trigger operator, and the @ event timing control. Many engineers don't know that the feature even exists in the language, and are unaware of how to use it. An engineer who had been using Verilog for a number of years recently attended a Verilog training class with his team. When the section on event data types was presented, the engineer asked if this was something new with SystemVerilog. The answer was no, that it has been in the Verilog language since its beginning in 1984. To this, the veteran Verilog engineer replied, "Why hasn't anyone told me about this before?"

There is gotcha, however, in that there can easily be simulation race conditions with Verilog's event triggering. The following code demonstrates this potential race condition.

```
module event_example1;
  event get_data, send_data;    // handshaking flags
  initial -> get_data;          // GOTCHA! trigger at time zero
  always @(get_data) begin      // wait for a get_data event
    ... // code to get data
    ... // when done, trigger send_data
    -> send_data;               // sync with send_data process
  end
  always @(send_data) begin     // wait for a send_data event
    ... // code to send data
    ... // when done, trigger get_data
    -> get_data;                // sync with get_data process
  end
endmodule
```

The two **always** blocks above model simple behavioral handshaking. The **event** data type is used to signal the completion of one block and enabling the other. The **initial** block is used to start the handshaking sequence.

The gotcha lies in the fact that, at simulation time zero, each of the procedural blocks must be activated. If the **initial** block activates and executes before the **always** @(get_data) block activates, then the sequence will never start.

How to avoid this Gotcha using Verilog

In Verilog, the only way to solve this issue is to delay the trigger in the **initial** block from occurring until all the procedure blocks have been activated. This is done by preceding the statement with an explicit zero delay, as shown in the code below.

```
initial #0 -> get_data;          // OK! delayed trigger
always @(get_data) begin         // wait for a get_data event
  ... // code to get data
```

Using the #0 delay will hold off triggering the get_data event until all the procedure blocks have been activated. This ensures that the **always** @(get_data) block will sense the start of a handshake sequence at time zero.

The Verilog #0 construct can be abused, however, and cause other hard-to-debug race conditions. Some Verilog trainers have recommended never using #0, because of its inherent dangers.

How to avoid this Gotcha using SystemVerilog

SystemVerilog comes to the rescue, with two solutions that will remove the event trigger race condition without using #0.

SystemVerilog solution 1:

SystemVerilog defines a nonblocking event trigger, **->>**, that will schedule the event to trigger after a zero delay delta, in the same way as nonblocking assignments. For the example in this section, this eliminates the race condition at time zero, and eliminates the need for a #0 delay. Triggering the get_data after the nonblocking delta allows for the **always** procedure blocks to become active before the event is triggered.

```
initial ->> get_data;            // OK! trigger after delta
always @(get_data) begin         // wait for a get_data event
  ... // code to get data
```

SystemVerilog solution 2:

SystemVerilog provides a second approach that will provide a solution to many more situations than the simple example shown in this book. This second solution uses a trigger persistence property that makes the trigger visible through the entire time step, and not just in the instantaneous moment that the event was triggered.

```
module event_example2 ( ... );

  event get_data, send_data;    // handshaking flags

  initial -> get_data;          // OK, trigger get_data at time zero

  always begin
    wait(get_data.triggered)    // wait for a get_data event
    ... // do code to get data
    ... // when done, trigger send_data
    -> send_data;               // sync with send_data process
  end

  always begin
    wait(send_data.triggered)   // wait for a send_data event
    ... // do code to send data
    ... // when done, trigger get_data
    -> get_data;                // sync with get_data process
  end
endmodule
```

The **wait** (get_data.triggered) returns true in the time step in which get_data is triggered. It does not matter if, within the current simulation time, the trigger event occurs before or after the **wait** statement is activated. So, in the above example, if the **initial** block activates and executes before the first **always** block, the trigger persistence will still be visible when the first **always** block becomes active and executes the **wait** (get_data.triggered) statement.

Gotcha 60: Using semaphores for synchronization

Gotcha: My processes are not synchronizing the way I expected using semaphores. Even when there are waiting processes, some other process gets to run ahead of them.

Synopsis: Semaphore keys can be added to a bucket without having first obtained those keys. Keys can be obtained without waiting for prior requests to be serviced.

The Verilog **event** data types provide a means to synchronize procedural blocks. But, this method of procedural handshaking and communication is too limiting for modern, object-oriented verification methodologies. SystemVerilog provides two additional inter-process synchronization mechanisms that provide more flexibility and versatility than simple event triggering provides. These mechanisms are *semaphores* and *mailboxes*. Both of these new synchronization methods have subtle behaviors that must be considered. This section describes the gotchas involving semaphores. Gotcha 61 on page 137 describes the gotchas involving mailboxes.

Semaphores are like a bucket that can hold a number of keys or tokens. Methods are available to put any number of keys into the bucket and to get any number of keys out of the bucket.

The `put()` method is straight forward. The number specified as an argument to `put()` is the number of keys placed in the bucket. Any number of keys can be placed into the bucket, regardless of how many were retrieved from the bucket. A process can even add keys to the bucket without having retrieved any keys at all. (A potential gotcha not addressed in this book is that incorrect code could keep adding more keys to the bucket than were retrieved from the bucket.)

The `get()` method is used to retrieve keys from the bucket. Any number of keys can be requested. If the number of keys requested is not available, the calling process is blocked from continuing execution until the requested number of keys becomes available.

The `get()` method has a subtle, non-intuitive gotcha. If more than one key is requested, and that number of keys is not available, the request is put into a FIFO (First In, First Out), and will wait until the requested number of keys becomes available. If more than one process requests keys that are not available, the requests are added to the FIFO in the order received. When keys become available, the requests in the queue are serviced in the order in which the requests were received.

The gotcha is that, each time `get()` is called, an attempt is made to retrieve the requested keys, without first putting the request into the FIFO. If the requested number of keys is available, the get() is serviced, even if other requests are waiting in the FIFO. The following example demonstrates this gotcha.

```
module sema4_example ( ... );

  semaphore s_test = new;     // create a semaphore bucket

  initial begin: Block1       // At simulation time zero...
    s_test.put(5);               // bucket has 5 keys added to it
    s_test.get(3);               // bucket has 2 keys left
    s_test.get(4);               // get(4) cannot be serviced
                                 //   because the bucket only has 2
                                 //   keys; the request is put in
                                 //   the request FIFO
    $display("Block1 completed at time %0d", $time);
  end: Block1

  initial begin: Block2 #10   // At simulation time 10...
    s_test.get(2);               // GOTCHA! Even though get(4)
                                 //   came first, and is waiting
                                 //   in the FIFO, get(2) will be
                                 //   serviced first
    s_test.get(1);               // this request will be put on
                                 //   the FIFO, because the bucket
                                 //   is empty; it will not be
                                 //   serviced until the get(4)
                                 //   is serviced
    $display("Block2 completed at time %0d", $time);
  end: Block2

  initial begin: Block3 #20   // At simulation time 20...
    s_test.put(3);               // nothing is run from the FIFO,
                                 //   since get(4)is first in the
                                 //   FIFO
    #10                       // At simulation time 30...
    s_test.put(2);               // get(4) and get(1) can now be
                                 //   serviced, in the order in
                                 // which they were placed in
                                 // the FIFO
    $display("Block3 completed at time %0d", $time);
  end: Block3
endmodule
```

When a `get()` method is called, and there are enough keys in the bucket to fill the request, it retrieves the requested keys immediately, even if there are previous `get()` requests waiting in the FIFO for keys. In the example above, the `Block1` process starts at simulation time zero. It executes until `get(4)` is called. At that time, there are only 2 keys available. Since the request could not be filled, it is put

in the request FIFO. The execution of `Block1` is then suspended until 4 keys are retrieved.

Next, a separate process, `Block2` requests 2 keys at simulation time 10. The `get(2)` executes and retrieves the 2 remaining keys from the bucket immediately, even though there is the `get(4)` in the request FIFO waiting to be serviced. The process then executes a `get(1)`. This request cannot be serviced, because the bucket is now empty, and therefore is put on the request FIFO.

At simulation time 30, the `Block3` process puts three keys back in the semaphore bucket. The `get(4)` request sitting in the FIFO still cannot be serviced, because there are not enough keys available. There is also a `get(1)` request in the queue, but is not serviced because that request was received after the `get(4)` request. Once placed on the FIFO, the `get()` requests are serviced in the order which they were received. The `get(4)` must be serviced first, then the `get(1)`.

How to avoid this Gotcha

The gotcha is having a `get()` request serviced immediately, even when there are other `get()` requests waiting in the request FIFO. This gotcha can be avoided if the `get()` requests are restricted to getting just one key at a time. In this way, any requests in the FIFO will never be waiting for more than one key. As soon as a key becomes available, the first request in the FIFO will be serviced. A new `get()` request will not be serviced ahead of the FIFO.

If a process needs more than one key, instead of calling `get()` once for the multiple keys, the process should call `get(1)` multiple times. A repeat loop is a convenient way to request multiple keys, one at a time. For example:

```
repeat(3) s_test.get(1);   // request 3 keys, one at a time
```

When the process is done, it can return multiple keys with a single `put()`. It is not necessary to put the keys back one at a time.

Gotcha 61: Using mailboxes for synchronization

Gotcha: My mailbox works at first, and then starts getting errors during simulation.

Synopsis: Run-time errors occur if an attempt is made to read the wrong data type from a mailbox.

Mailboxes provide a mechanism for both inter-process synchronization and the passage of information between processes. By default, mailboxes are typeless, which means that messages of any data type can be put into the mailbox. The gotcha is that, when messages are retrieved from the mailbox with the `get()` method, the receiving variable must be the same data type as the value placed in the mailbox. If the receiving variable is a different type, then a run time error will be generated.

```
module mbox_example1 ( ... );
  logic [15:0] a, b;
  int i, j, s;
  struct packed {int u, v, w;} d_in, d_out;

  mailbox mbox1 = new;     // typeless mailbox

  initial begin
    mbox1.put(a);     // OK: can put message of any data type
    mbox1.put(i);     // OK: can put message of any data type
    mbox1.put(d_in); // OK: can put message of any data type

    mbox1.get(b);     // OK: data type of b matches data type of
                      // first message in mbox1
    mbox1.get(b);     // ERROR: b is wrong type for next message
                      // in mbox1
  end
endmodule
```

How to avoid this Gotcha

There are three ways of avoiding this gotcha. First is the brute force method of managing the data types manually. The manual approach could be error prone. It places a burden on the verification engineers to track what type of data was put in the mailbox, and in what order, so that the correct types are retrieved from the mailbox.

The second approach is to use the `try_get()` method, instead of the `get()` method. The `try_get()` method retrieves the message via an argument passed to `try_get()`, and returns a status flag. One of three status flags is returned:

- Returns 0 if there is no message in the mailbox to retrieve.
- Returns 1 if the message and the receiving variable are type-compatible, and the message is retrieved.
- Returns a negative value if the message and the receiving variable are type-incompatible, in which case the message is not retrieved.

The return value of `try_get()` can be processed by conditional statements, to determine the next verification action. The following example illustrates using a typeless mailbox and the `try_get()` method.

```
module mbox_example1 ( ... );
  logic [15:0] a, b;
  int i, j, s;
  struct packed {int u, v, w;} d_in, d_out;

  mailbox mbox1 = new;     // typeless mailbox

  initial begin
    mbox1.put(a);     // OK: can put message of any data type
    mbox1.put(i);     // OK: can put message of any data type
    mbox1.put(d_in); // OK: can put message of any data type

    s = mbox1.try_get(d_out);  // must check status to see if OK
    case (s)
       1: $display("try_get() succeeded");
       0: $display("try_get() failed, no message in mailbox");
      default: $display("try_get() failed due to type error");
    endcase
  end
endmodule
```

The third approach to avoiding a mailbox run-time error gotcha is to use *typed mailboxes*. These mailboxes have a fixed storage type. The tool compiler will give a compilation error if the code attempts to place any messages with incompatible data types into the mailbox. The `get()` method can be safely used, because it is known beforehand what data type will be in the mailbox. An example of declaring a typed mailbox is.

```
typedef struct {int a, b} data_packet_t;
mailbox #(data_packet_t) mbox2 = new;     // typed mailbox
```

With this typed mailbox example, only messages of data type `data_packet_t` can be put into `mbox2`. If an argument to the `put()` method is any other type, a compilation error will occur.

Gotcha 62: Triggering on clocking blocks

Gotcha: I cannot get my test program to wait for a clocking block edge.

Synopsis: When a test waits for a clocking block edge to occur, the posedge or negedge keyword should not be used.

Test code that uses the @ event control to delay until a clocking block clock occurs should not specify **posedge** or **negedge** of the clocking block name. The following example causes a compilation error:

```
program automatic test (input  logic clk,
                        input  logic grant,
                        output logic request
                       );
  clocking cb @(posedge clk);
    output request;
    input  grant;
  endclocking

  initial begin
    @(posedge cb) // GOTCHA: cannot select edge of clocking block
      $display("At %0d: clocking block triggered", $time);
    ...
  end
endprogram
```

How to avoid this Gotcha

When test code needs to delay for a clocking block clock using the @ event control, only the clocking block name should be used. This is because clocking block definitions already specify which edge of the clock is being used. For example:

```
  initial begin
    @(cb)          // OK: delay until clocking block event occurs
      $display("At %0d: clocking block triggered", $time);
    ...
  end
endprogram
```

Using a clocking block name for an event control can make test code more robust and easier to maintain, especially when the clocking block is defined in an interface. The test program does not need to know if the interface uses a positive edge, negative edge, or both edges (double data rate) of the clock. All the test program needs to reference is the clocking block name.

Gotcha 63: Misplaced semicolons after decision statements

Gotcha: Statements in my if() decision execute, even when the condition is not true.

Synopsis: A semicolon after the closing parenthesis of a decision statement is legal, and causes the statements that should be within the if() to be outside the if().

A semicolon (;) by itself is a complete programming statement, representing a null-operation statement. A misplaced semicolon after **if** is legal. However, the misplaced semicolon can cause the statement or **begin...end** group after the misplaced semicolon to execute at times that were not intended.

```
module foo;
  reg a;
  initial begin
    a = 1;
    if (a);     // semicolon is wrong, but NOT syntax error
      $display("'a' is true"); // GOTCHA! also prints when 'a'
                               // is false
  end
endmodule
```

In the example above, there is no syntax error. The semicolon is a legal statement, and is the only statement associated with the **if** condition. The **$display** statement, though nicely indented, is not part of the **if** statement. The **$display** message prints every time, regardless of whether the variable a is true or false. *Gotcha!*

The next example illustrates how a misplaced semicolon can lead to a syntax error on a subsequent line of code.

```
module bar;
  reg a;
  initial begin
    a = 1;
    if (a);                    // semicolon is NOT an error
      $display("'a' is true");
    else                       // SYNTAX ERROR! 'else' does not
                               // follow 'if'
      $display("'a' is false");
  end
endmodule
```

The **else** line in the example above appears to be paired with the **if** statement. However, the only statement in the **if** branch is the misplaced semicolon, which

is a null statement. Therefore, the **$display** statement that follows is not part of the **if** statement, which means the **else** statement is not paired with the **if** statement. The compiler will report an error on the line with **else**, which is actually two lines after the real problem. *Gotcha!*

How to avoid this Gotcha

This is an example of a gotcha that is inherited from the C language, from which Verilog and SystemVerilog have their syntax and semantic roots. The same coding mistakes illustrated above can be made in C. The way to prevent this coding gotcha is to know Verilog syntax, and to correctly use semicolons.

A language-aware text editor, such as Emacs with a Verilog mode, can help to avoid this gotcha. A good language-aware editor for Verilog can add auto indentation. The examples above would have obvious indentation errors with such an editor. The first example, above, might be indented as follows:

```
initial begin
  a = 1;
  if (a);
  $display("'a' is true");  // statement is not auto-indented
end
```

Gotcha 64: Misplaced semicolons in for loops

Gotcha: My for loop only executes one time.

Synopsis: A semicolon at the end of a for loop declaration effectively makes the loop always execute just one time.

A semicolon (;) by itself is a complete programming statement, representing a null-operation A misplaced semicolon after **for** is syntactically legal. However, the misplaced semicolon has the effect of making the loop appear to execute only one time.

```
module foo;
  integer i;
  initial begin
    for (i=0; i<=15; i=i+1);        // semicolon is NOT an error
      begin
        $display("Loop pass executing");  // GOTCHA! only executes
                                          // once
      end
  end
endmodule
```

In the example above, there is no syntax error. The semicolon is a legal statement, and is the only statement within the **for** loop. The **begin**...**end** group with the **$display** statement is not part of the **for** loop. The loop will execute 16 times, executing a null statement. After the loop has completed, the group of statements that appear to be inside the loop—but which are not—will execute one time. *Gotcha!*

This gotcha can also occur with **while**, **repeat**, **forever** and **foreach** loops.

Looping multiple times executing a null statement is not necessarily a coding error. A common verification coding style is to use a null statement in a **repeat** loop, in order to advance multiple clock cycles. For example:

```
initial begin
  reset_n <= 0;
  repeat (8) @(posedge clock) ;  // loop 8 clock cycles doing no-op
  reset_n = 1;
end
```

How to avoid this Gotcha

This gotcha is inherited from the C programming language, where the same coding error is syntactically legal. A language-aware editor with auto-indenting can help to avoid this gotcha. A good Verilog editor will show the indentation to be wrong for this code, which will indicate a misplaced semicolon.

There is another gotcha with the **for** loop example above. Even though a null statement in a **for** loop is legal code, some tools make it a syntax error. The intent in making this an error is to help engineers avoid a common C programming gotcha. Unfortunately, it also means that if the engineer actually wanted an empty **for** loop, these tools do not allow what should be legal code. The workaround, if an empty loop is actually intended, is to replace the null statement with an empty **begin**...**end** statement group.

Gotcha 65: Infinite for loops

Gotcha: My for loop never exits. When the loop variable reaches the exit value, the loop just starts over again.

Synopsis: Declaring too small a for loop control variable can result in loops that never exits.

A **for** loop executes its statements until the loop control expression evaluates as false. As in most programming languages, it is possible to write a **for** loop where the control expression is always true, creating an infinite loop that never exits. This general programming gotcha is more likely to occur in Verilog, because Verilog allows engineers to define small vector sizes.

The intent in the following example is to loop 16 times, with the loop control variable having a value from 0 to 15.

```
reg [3:0] i;                    // 4-bit loop control variable
for (i=0; i<=15; i=i+1)         // GOTCHA! i<=15 is always true
  begin  /* loop body */  end
```

In this example, the loop will run until `i` is incremented to a value greater than 15. But, as a 4-bit variable, when `i` has a value of 15 and is incremented, the result is 0, which is less than or equal to 15. The loop control test will be true, and the loop will continue to execute, starting over with i equal to 0.

How to avoid this Gotcha

A simple way to avoid this gotcha is to increase the size of the loop control variable so that it can hold a larger value. Typically, either **integer** (a Verilog type) or **int** (a SystemVerilog type) should be used as loop control variables, both of which are 32-bit signed variables.

With SystemVerilog, the loop control variable can be declared as part of the **for** loop. This puts the loop control variable declaration and usage of the variable in the same line of code, making type or size of the variable more obvious.

```
reg [3:0] result;               // 4-bit design or test variable
for (int i=0; i<=15; i=i+1)     // OK, i can have a value greater
                                // than 15
  @(posedge clk) result = i;    // OK, but mismatch in assignment
                                // sizes
```

The difference in size of the loop control variable and the value to which it is assigned can result in a warning. The assignment is still correct. The lower 4 bits of `i` are assigned to `result`, and the remaining bits are truncated. To prevent the warning message, explicitly select the lower bits of i, as in `result = i[3:0]`..

Gotcha 66: Locked simulation due to concurrent for loops

Gotcha: When I run simulation, my for loops lock up or do strange things.

Synopsis: Parallel for loops that use the same control variable can interfere with each other.

Verilog has two programming constructs that can invoke parallel execution threads within the same scope: multiple combinational **always** procedural blocks that trigger at the same time, and **fork**...**join** statement groups. Parallel threads running in the same name space can interfere with each other if they share any of the same storage variables. The following example illustrates a simple testbench that forks off two tests to run in parallel. Each test contains a **for** loop that uses a variable called `i` as the loop control. One loop increments `i`, and the other loop decrements `i`.

```
program automatic test;
  logic [7:0] a, b, c, sum, dif;
  int i;                              // GOTCHA! shared loop variable
  adder_subtractor dut (.*);
  initial begin
    fork
      begin: add_test
        for (i = 0; i < 10; i++) begin      // increment i
          a = i;
          b = i + 10;
          #10 $display("At %0d, in scope %m: i=%0d sum=%0d",
                       $time, i, sum);
        end
      end

      begin: dif_test
        for (i = 8; i > 2; i--) begin       // decrement i
          c = i;
          #10 $display("At %0d, in scope %m: i=%0d dif=%0d",
                       $time, i, dif);
        end
      end
    join
    $display("\nTests finished at time %0d\n", $time);
    $finish;
  end
endmodule
```

The intent of this example is that, after both loops complete, `$finish` is called and simulation exits. One loop should execute 10 times, and the other 6 times. Instead of completing, however, simulation locks up, and never exits. The reason

is that each loop is changing the same control variable, preventing either loop from ever reaching a value that will cause the loop to exit. *Gotcha!*

How to avoid this Gotcha

The way to correct this problem is to use different variables for each loop. The simplest way to do this is to use the SystemVerilog feature of declaring a local variable as part of each **for** loop.

```
initial begin
  fork
    begin: add_test
      for (int i = 0; i < 10; i++) begin  // i is local variable
        ...
      end
    end

    begin: dif_test
      for (int i = 8; i > 2; i--) begin   // i is local variable
        ...
      end
    end
  join
```

Gotcha 67: Referencing for loop control variables

Gotcha: My Verilog code no longer compiles after I convert my Verilog-style for loops to a SystemVerilog style.

Synopsis: Loop control variables declared as part of a for loop declaration cannot be referenced outside of the loop.

Verilog requires that loop control variables be declared before the variable is used in a **for** loop. Since the variable is declared outside the **for** loop, it is a static variable that can also be used outside the **for** loop.

```
reg [31:0] a, b, c;
integer a[0:31], b[0:31], c[0:31]; // arrays of 32 elements
integer i;                         // static loop control variable
initial begin
  for (i=0; i<=31; i=i+1) begin
    c[i] = a[i] + b[i];          // OK to reference i inside of loop
  end
  $display("i is %0d", i);       // OK to reference i outside of loop
end
```

SystemVerilog allows declaring **for** loop control variables within the declaration of the loop. These are automatic variables that are local to the loop. The variable is dynamically created when the loop starts, and disappears when the loop exits. Because the variable is automatic, it is illegal to reference the variable outside of the scope in which it exists. The following code causes a syntax error:

```
reg [31:0] a, b, c;
initial begin
  for (int i=0; i<=31; i=i+1) begin  // i is automatic variable
    c[i] = a[i] + b[i];          // OK, i is used inside the loop
  end
  $display("i is %0d", i);       // GOTCHA! i is used outside of loop
end
```

How to avoid this Gotcha

Technically speaking, this is not a gotcha, because it is a syntax error, rather than an unexpected run-time behavior. However, there are times when it is useful to reference loop control variables outside of the loop. In those situations, the loop variable should be declared outside of the loop, using the Verilog coding style shown in the first example above.

Gotcha 68: Default function return size

Gotcha: My function only returns the least significant bit of the return value.

Synopsis: Verilog functions have a default return type of 1-bit logic, if no return type is specified.

The function listed below adds two integer values together and returns the result. However, this example will add 3 and 4, and return a result of 1 instead of 7.

```
module test;
  int result;                  // 32-bit variables

  function sum (int a, b);     // no return size specified
    return (a + b);
  endfunction

  initial begin
    result = sum(3,4);   // GOTCHA! 32-bit result is always 0 or 1
    $display("sum(3,4) return = %h (hex)", result);
  end
endmodule: test
```

This test case will print:

```
sum(3,4) return = 00000001 (hex)
```

Non-void functions return a sized, typed value. If the type or size is not specified, the default return type is **logic** and the default return size is scalar (1-bit). If within the function, a multi-bit vector is specified as the return from an unsized, typed function, then all of the upper bits of the vector are truncated without error or warning, and only the least significant bit of the vector is actually returned. *Gotcha!*

If a function return value is assigned to a variable, then the Verilog/SystemVerilog assignment rules come into play. These rules are discussed in Gotcha 46 on page 105. In brief, if the 1-bit function return is assigned to a variable that is more than one bit wide, the return value will be zero-extended to the variable size, again with no error or warning. This zero extension can hide the first gotcha of the function return value having been truncated. *Gotcha, again!*

How to avoid this Gotcha

To avoid this gotcha, an explicit return size or type needs to be specified. A good coding guideline would be to always specify the function return type, even when a 1-bit return is desired. Some example function declarations with a return type and/or return size specified are:

```
function int sum1 (int a, b);          // return int type (32-bit)
function [15:0] sum1 (int a, b);       // return 16-bit logic type
function logic sum1 (int a, b);        // return 1-bit logic type
function bit [31:0] sum1 (int a, b); // return 32-bit bit type
function real sum1 (int a, b);         // return real type
```

Since this gotcha of not specifying a function return type and size is legal code, it is not easy to detect this gotcha without debugging simulation results. Lint tools (coding style checkers) might be able to check that functions always have a return type defined. If the function return is being assigned to a variable, lint checkers can also check for assignment size mismatches. Most synthesis tools will generate a warning message when there is an assignment mismatch. In the gotcha example above, a warning such as this might help make it more obvious that the function return is a different size than what was expected.

An additional way this gotcha might be detected is by using a language-aware text editor. The editor[1] used for testing the example above changed the color of the function identifier when the function had a return type and size specified. As a designer becomes familiar with such editors, gotchas (or bugs) such as this can be identified while writing the code.

1. The editor used was Emacs with the Verilog mode from www.verilog.com

Gotcha 69: Task/function arguments with default values

Gotcha: I get a syntax error when I try to assign my task/function input arguments a default value.

Synopsis: Task/function argument directions are inherited from the previous argument, and only input and inout arguments can have a default value.

The formal arguments of a task or function can be **input**, **output**, **inout**, or **ref**. In SystemVerilog, task/function arguments default to **input** if no direction has been specified. However, the direction is sticky, so that, once it has been specified, it affects all subsequent arguments until a new direction is specified.

SystemVerilog allows **input** and **inout** arguments of a task or function to be specified with a default value. When the task or function is called, a value does not need to be passed to formal arguments that have a default value.

The following function header gets a compilation error because the second argument, `start`, has a default value specified.

```
function automatic int array_sum(ref int a[], int start=0);
  for (int i=start; i<a.size(); i++)
    array_sum += a[i]);
endfunction
```

Only **input** and **inout** arguments can have a default value. The problem with this code is that the `start` argument does not have a direction explicitly specified. If no directions at all had been specified, `start` would default to an **input** argument, which can have a default value. In this example, however, the first formal argument of the function, `a[]`, has been defined with a direction of **ref**. This direction is sticky. It also apples to `start`. Assigning a default value to a **ref** argument is illegal.

How to avoid this Gotcha

To avoid a direction gotcha, specify a direction for all task/function arguments.

```
function int array_sum(ref int a[], input int start=0);
```

The example shown in this section only causes a compilation error because `start` has a default assignment, which is illegal for **ref** arguments. Sticky argument directions can cause other subtle programming gotchas that are not compilation errors.

Gotcha 70: Continuous assignments with delays cancel glitches

Gotcha: Some delayed outputs show up with continuous assignments and others do not.

Synopsis: Continuous assignments with delays will cancel input glitches.

Continuous assignment statements are continuously running processes that transfer an expression from the right-hand side to a net or variable on the left-hand side. Designers frequently use continuous assignments to model combinational logic behavior. In a synthesizable RTL model, continuous assignments are typically written to use zero delay. When the right-hand expression changes, the left-hand net or variable is immediately updated. In a testbench or bus-functional model, it is sometimes desirable to add a propagation delay between the right-hand expression change and when the left-hand side net or variable is updated.

The intent in the following example is to generate two delayed version of a clock .

```
module clock_gen;
  timeunit 1ns; timeprecision 1ns;

  logic clock0, clock3, clock6;

  initial begin
    clock0 <= 0;
    forever #5 clock0 = ~clock0;
  end

  assign #3 clock3 = clock0 ;     // OK, clock3 works as expected

  assign #6 clock6 = clock0 ;     // GOTCHA! clock6 flat lines

  initial begin
    $timeformat(-9, 0, "ns", 7);
    $monitor("%t: clock0 = %b    clock3 = %b    clock6 = %b",
             $time, clock0, clock3, clock6);
    #30 $finish;
  end
endmodule: clock_gen
```

The output of the example above is:

```
    0ns: clock0 = 0    clock3 = x    clock6 = x
    3ns: clock0 = 0    clock3 = 0    clock6 = x
    5ns: clock0 = 1    clock3 = 0    clock6 = x
    8ns: clock0 = 1    clock3 = 1    clock6 = x
   10ns: clock0 = 0    clock3 = 1    clock6 = x
   13ns: clock0 = 0    clock3 = 0    clock6 = x
   15ns: clock0 = 1    clock3 = 0    clock6 = x
   18ns: clock0 = 1    clock3 = 1    clock6 = x
```

The outputs show that `clock0` toggles every 5 nanoseconds, as expected. The delayed `clock3` changes 3 nanoseconds after `clock0` just as it should. The delayed `clock6`, which is functionally generated in exactly the same manner as `clock3`, but with a 6 nanosecond delay, never changes value. *Gotcha!*

How to avoid this Gotcha

The reason `clock3` changes, but `clock6` does not, is that continuous assign statements use an *inertial delay mechanism* to propagate value changes. This means that if two or more changes are scheduled on the left-hand side net or variable (in essence, the output) of the continuous assignment, then only the *last* scheduled change actually occurs. In other words, each scheduled change cancels any earlier scheduled changes that have not yet occurred.

The 3 nanosecond delay for `clock3` is less than the 5 nanosecond half-cycle of `clock0`, and so all changes on `clock0` propagate through to `clock3`. But, the 6 nanosecond delay for `clock6` is greater than the 5 nanosecond half-cycle of `clock0`, so all changes on `clock0` do not propagate through to `clock6`.

To avoid this gotcha, clock6 needs to be modeled using a *transport delay mechanism*, instead of inertial delay mechanism. The simplest way to do this is to use an **`always`** procedural block with a nonblocking intra-assignment delay, as follows:

```
always @(clock0)
  clock6 <= #6 clock0;      // OK, 6ns intra-assignment delay
```

Primitive delays and net delays also use inertial delay propagation, and will have the same gotcha as continuous assignments. Verilog's specify block pin-to-pin path delay construct can be defined to use either inertial delay or transport delay. This construct would be overly complex for the intended logic of this example, however.

For more details on how to model inertial and transport delays using Verilog procedural blocks, refer to a paper from one of the authors, "*Understanding Verilog Blocking and Nonblocking Assignments*"[1].

1. *Understanding Verilog Blocking and Nonblocking Assignments*, by Stuart Sutherland. Published in the proceedings of International Cadence Users Group, San Jose, 1996. Also available at from the author's web site, http://www.sutherland.com/papers.html

Chapter 6

Object Oriented and Multi-Threaded Programming Gotchas

Gotcha 71: Programming statements in a class

Gotcha: Some programming code in an initial procedure compiles OK, but when I move the code to a class definition, I get compilation errors.

Synopsis: Class definitions can only have properties (variables) and methods (tasks and functions). They cannot have procedural programming statements.

The `Bar` class definition below constructs a `Foo` object, and attempts to initialize the variable `i` within `Foo`:

```
class Foo;
  int data;                  // property

  function int get (...);    // method
    ...
  endfunction
  task put (...);
    ...
  endtask
endclass

class Bar;
  Foo f = new;          // create object f
  f.data = 3;           // GOTCHA! illegal assignment statement
endclass
```

Note: the code examples in this chapter are contrived in order to illustrate each gotcha using small examples. In real design and verification code, these gotchas might not be as obvious or easy to debug.

This example causes a compilation error, because any executable code in a class must be in a task or function. A class is a definition, and cannot contain assignment statements, programming statements, **initial** blocks or **always** blocks. The assignment `f.data = 3;` in the example above is an executable statement that is not in a task or function, and therefore not allowed.

How to avoid this Gotcha

The fix for this gotcha depends on when class `Bar` needs to assign a value to the data in a `Foo` object. If the assignment only needs to occur once when a `Foo` object is constructed, a simple fix is to initialize `data`, using `Foo`'s constructor function, as follows:

```
class Foo;
  int data;
  function new (int d);
    this.data = d;          // assign to data at time of construction
  endfunction
  ...
endclass

class Bar;
  Foo f = new(3);           // pass initial value to new method of Foo
endclass
```

If `Bar` needs to assign to the data variable at any time, the fix is to add a method in `Bar` that contains the assignment statement. All programming statements within a class definition must be within tasks or functions:

```
class Foo;
  int data;
  ...
endclass

class Bar;
  Foo f = new;

  function change_data(d);  // assign to data after construction
    f.data = d;
  endfunction
endclass
```

Guideline: It is legal to call a constructor as part of the declaration of a class handle variable within another class (e.g. `Foo f = new;` in the example above). However, this is discouraged, as the object will be constructed before any code in the enclosing class has been executed. This can cause problems if there is a need to create or initialize objects in a specific order. It is usually preferable to call such constructors in the constructor of the enclosing class, where there is more control over the initialization.

Gotcha 72: Using interfaces with object-oriented testbenches

Gotcha: I get a compilation error when I try to use a class object to create test values when the testbench connects to the design using an interface.

Synopsis: Static structural components, such as interfaces, cannot be directly driven from dynamic code.

In the following example, the `Driver` class, which is a dynamic object, needs to drive data into the `arb_ifc` interface, which is a static design object. Since a dynamic object cannot directly drive static objects such as a module or an interface port, this code is illegal.

```
interface arb_ifc(input logic clk);
  ...
endinterface

program automatic test (arb_ifc.TEST arb);

  class Driver;
    arb_ifc arb;   // GTOCHA! class cannot instantiate interface

    function new(arb_ifc arb);   // GOTCHA! task/func arg cannot
                                 // be an interface
      this.arb = arb;
    endfunction
  endclass

  initial begin
    Driver d;
    d = new(arb);
  end
endprogram
```

How to avoid this Gotcha

An interface is a structural component that represents hardware. It can contain signals, code, and assertions. Structural components cannot be passed around for use by dynamic code. Instead, a pointer to the interface is used in the dynamic class object. A pointer to an interface is called a *virtual interface*. The purpose of a virtual interface is to allow dynamic objects to have a handle to a statically instantiated object, and to move data between a dynamic class object and a static object.

The correct way to model the example above is to make the `arb_ifc` instance in the driver class virtual.

```
class driver;
  virtual arb_ifc arb;                  // pointer to interface

  function new(virtual arb_ifc arb);  // pointer to interface
    this.arb = arb;
  endfunction
endclass
```

Virtual interfaces are the bridge or link between the class-based testbench and the Device Under Test (DUT).

Gotcha 73: All objects in mailbox come out with the same values

Gotcha: My code creates random object values and puts them into a mailbox, but all the objects coming out of the mailbox have the same value.

Synopsis: The class constructor creates a handle to an object. In order to have multiple objects, the class constructor must be called multiple times.

Mailboxes are used to synchronize activity between parallel processes, and to pass information between the processes during the synchronization.

The intent in the following example is to put 10 random object values into a mailbox:

```
class My_class;
  rand int data;
  rand logic [47:0] address;
  ...
endclass

My_class h = new;      // Gotcha! construct one My_class object

repeat(10) begin
  h.randomize();       // randomize properties in the object
  mbx.put(h);          // store handle to object in a mailbox
end
```

The thread that retrieves the objects from the mailbox will find that all the objects contain the same values, the ones generated by the last call to randomize. *Gotcha!*

The gotcha happens because the code only constructs one object, and then randomizes it over and over. The mailbox is full of handles, but they all refer to a single object.

How to avoid this Gotcha

In order to randomize multiple objects, they must first be constructed. In the example above, the call to the constructor should be inside the loop:

```
my_class h;
repeat(10) begin
  h = new;             // construct a My_class object
  h.randomize();       // randomize properties in the object
  mbx.put(h);          // store handle to object in a mailbox
end
```

Gotcha 74: Passing handles to methods using input versus ref arguments

Gotcha: My method constructs and initializes an object, but I can never see the object's value.

Synopsis: Method input arguments create local copies of variables that are not visible in the calling scope.

The default direction of method (task and function) arguments is **input**. Inputs create local variables for use within the method. When a method is called, the values of the actual arguments are copied into the local storage. Any changes to this local storage that are made within the method are not passed back to the calling scope.

The intent of the following function is to construct two objects and assign the object handles to the handle variables passed into the function.

```
function void build_env(Consumer c, Producer p);  // GOTCHA!
  c = new();   // construct object and store handle in c
  p = new();   // construct object and store handle in p
  ... // set up rest of environment
endfunction

initial begin
  Consumer c;
  Producer p;
  build_env(c, p);  // construct and set up objects
  c.randomize;      // ERROR: c does not contain an object handle
end
```

The code that calls the `build_env` function will not be able to see the constructed objects because the function argument directions are not specified, and therefore default to **input**. Within the function, `c` and `p` are local variables. The new handles that are stored in the local `c` and `p` variables are not passed back to the code that called the `build_env` function.

How to avoid this Gotcha

In a method that constructs objects, declare the handle arguments as **ref**. A **ref** argument is a reference to storage in the calling scope of the method. In the declaration below, when the method constructs an object and stores the handles `c` and `p`, the code that calls `build_env` will see the new handles, because `build_env` is changing the storage in the calling scope.

```
function void build_env(ref Consumer c, ref Producer p);
```

Gotcha 75: Constructing an array of objects

Gotcha: I declared an array of objects, but get a syntax error when I try to construct the array.

Synopsis: An "array of objects" is actually an array of object handles. Each handle must be constructed separately.

It is often useful to declare an array of object handles, in order to store handles to multiple objects. Such an array is often called an "array of objects". In reality, it is an array of handles, not an array of actual objects.

The following example attempts to create an array to hold 8 objects, but the code does not work.

```
class Transaction;
  ...
endclass

initial begin
  Transaction trans[8];  // An array of 8 Transaction objects

  trans = new;           // ERROR: cannot call new on object array
  trans = new[8];        // ERROR: cannot call new on array element
end
```

This example will get compilation errors. Both calls to the constructor for `trans` are syntactically incorrect. The reason is that `trans` is the name of an array, not the name of a handle variable.

How to avoid this Gotcha

Each object in the array of handle variables must be constructed individually, and its handle assigned to an element of the array. The correct way to code the example above is:

```
initial begin
  Transaction trans[8];  // An array of 8 Transaction objects

  foreach (trans[i])
    trans[i] = new(); // Construct object and store handle in array
end
```

Gotcha 76: Static tasks and functions are not re-entrant

Gotcha: My task works OK sometimes, but gets bogus results other times.

Synopsis: Invoking a static task or function while a previous call is still executing may cause unexpected results.

In Verilog and SystemVerilog, tasks and functions are static by default, which is different from C, where functions are automatic by default. *This difference is important!* In static tasks and functions, any local storage, including input arguments, are shared by every call to the task or function. In an automatic task or function, new storage is created for each call, which is unique to just that call. A default of static tasks and functions generally works well when modeling hardware, because storage within hardware is static. A testbench, on the other hand, is more of a software program rather than hardware.

Verilog/SystemVerilog's default static storage can cause unexpected behavior if a verification engineer is expecting C-like programming behavior. Static storage is particularly evident in tasks that are used for verification. Tasks can take simulation time to execute. Therefore, it is possible for a task to be invoked while a previous call to the task is still executing, as is illustrated below.

In the following example, a task called `watchdog` is called when the test issues an interrupt request. The task delays for some number of clock cycles, and then prints out a time out error if the interrupt is not acknowledged. The interrupt number and number of cycles to count are passed in as input arguments. The test code calls this task twice, in parallel, for two different interrupt requests.

```
program test (input  logic clock,
              input  logic [1:0] ack,
              output logic [1:0] irq);
  initial begin: irs_test
    $display("Forking off two interrupt requests...");
    fork
      watchdog (0, 20);  // must receive ack[0] within 20 cycles
      watchdog (1, 50);  // must receive ack[1] within 50 cycles
      begin
        irq[0] = 1'b1;
        irq[1] = 1'b1;
        wait (ack)
        $display("Received ack at %0d, disabling watchdog", $time);
        disable watchdog;  // got ack; kill both watchdog tasks
      end
    join_any
    $display("\At %0d, test completed or timed out", $time);
    $finish;   // abort simulation
  end: irs_test
```

```
  task watchdog (input int irq_num,     // GOTCHA! static storage
                 input int max_cycles   // GOTCHA! static storage
                );
    $display("At %0d: Watchdog started for IRQ[%0d] for %0d cycles",
             $time, irq_num, max_cycles);
    repeat(max_cycles) @(posedge clock) ; // delay until max_cycles
                                          // reached
    $display("Error at %0d: IRQ[%0d] no after %0d cycles",
             $time, irq_num, max_cycles);
  endtask: watchdog
endprogram: test
```

This example will run, but not as desired. The second call to the `watchdog` task will overwrite the `irq_num` and `max_count` values being used by the first call. The first call is still running, but now has incorrect values. *Gotcha!*

How to avoid this Gotcha using Verilog

In Verilog, all storage in a task or function can be made automatic by adding the keyword **`automatic`** to the task or function declaration.

```
task automatic watchdog ( ...  // automatic storage
```

An automatic task or function is also referred to as a re-entrant task or function. The task or function can be invoked (or entered) while previous calls are still executing. Each call creates new storage that is local to just that call. The example above illustrated a re-entrant task. Another example is recursive function calls, which must also be declared as automatic, so that the function can be re-entered without affecting the storage of already active calls to the same function.

How to avoid this Gotcha using SystemVerilog

SystemVerilog allows the **`automatic`** keyword to be specified as part of the declaration of a module, interface, or program. The gotcha above can be fixed by changing the program declaration to:

```
program automatic test ( ... );
  ...
  task watchdog (...    // OK, automatic storage
```

By adding the **`automatic`** keyword to the program declaration, all tasks and functions within the program will be automatic by default. This makes the default behavior like C, where all functions automatic by default. Note that tasks and functions declared in class definitions are automatic by default, unless explicitly declared as static.

See Gotcha 7 on page 22 for coding guidelines on declaring programs, packages and interfaces as automatic.

Gotcha 77: Static versus automatic variable initialization

Gotcha: The variables in my testbench do not initialize correctly.

Synopsis: By default, variables in tasks and functions contained in interfaces, packages, and statement groups are static, unlike C and C++. Static variables only initialize one time.

Many verification engineers have a strong programming background, and may expect verification programs to use automatic storage. That is, the variables will be stored on a stack and initialized when the block is entered.

In the following example, a local variable called `addr` is declared within an **`if`** decision. This local `address` variable is initialized to the value of the `address` bus that is in an interface port called `bus`.

```
interface bus_ifc;
  ...
  logic [31:0] address;
  ...
endinterface

program monitor(bus_ifc.MONITOR bus);
  initial begin
    @(posedge bus.cb.grant);
    if (bus.cb.command == READ) begin
      logic [31:0] addr = bus.cb.address; // GOTCHA! addr will be X
      $display("Bus addr = %h", addr);
      ...
    end
  end
endprogram
```

In this example, no matter what the current value of `address` in the interface is when the **`if`** statement executes, the local `addr` variable will always have a value of X. *Gotcha!*

The intent of the function in the next example is to print the largest value of the array.

```
program sums;
  function void maxx(ref int a[]);
    int max = a[0];                    // local variable
    foreach (a[i])
      if (a[i] > max) max = a[i];
    $display("Max value is %0d", max); // GOTCHA!
  endfunction
  ...
endprogram
```

In this example, only the first call produces the right result. Later calls print either the largest value or the result from the previous call. *Gotcha!*

The problem in both of these examples is that tasks, functions, and **begin...end** blocks use static storage by default. Static variables are allocated and initialized once, at compilation time. In the first example above, `addr` is initialized to the value of `address`, but, at the time the static storage is created, `address` has a value of X. When the **if** statement is executed during simulation, the static `addr` variable is not re-initialized, and hence always has its original initial value of X. *Gotcha!*

In the second example above, the local variable `max` is statically allocated and initialized to the value of `a[0]` one time when the storage is created. Every subsequent call to the function does not re-initialize `max`. Since `max` is not re-initialized, it contains the value of the previous call to the function. When the array is searched, if a larger value is found, `max` will be correctly updated to this new largest value. If a larger value is not found, however, the value of `max` will not be the largest current value in the array. It will be the largest value found the previous time the function was called. *Gotcha!, again*

How to avoid this Gotcha

To avoid this gotcha, variables that need to be re-initialized each time the procedural block, task or function is entered need to be automatic variables. Three ways to do this are:

- Explicitly declare specific variable as **automatic**.
- Tasks and functions can be declared as **automatic**, making all variables within the task or function automatic by default.
- Program, module, interface and package definitions can be declared as **automatic**, making all variables within declared within tasks, functions, within procedural blocks automatic by default

For the two examples above, the static storage gotcha can easily be avoided by declaring the programs as automatic.

```
program automatic monitor;
```

```
program automatic sums;
```

See Gotcha 7 on page 22 for coding guidelines on declaring programs, packages and interfaces as automatic.

Gotcha 78: Forked programming threads need automatic variables

Gotcha: When I fork off multiple tests, I get incorrect results, but each test runs OK by itself.

Synopsis: Concurrent threads can have conflicts with shared variables, such as indices.

When a test program spawns multiple concurrent test threads, it is important that each thread have its own storage. Otherwise, one thread could interfere with the storage being used by another thread.

The following example spawns three concurrent threads using a **for** loop that contains a **fork**...**join_none** block. The **join_none** is important; it is what allows three concurrent threads to be spawned. If **join** had been used, each thread would run to completion before the next thread could start. It is a common verification requirement to spawn concurrent threads, as in this example, rather than sequential threads.

Each concurrent thread in this example uses the loop control variable as a `thread_id` number. the `thread_id` is used to index into an array of data.

```
program automatic test (input logic clock);
  int d_array[3] = '{10,11,12};
  initial begin
    for (int thread_id=0; thread_id<3; thread_id++)
      fork
        $write("  thread_id=%0d  ", thread_id);          // GOTCHA!
        $display("d_array[thread_id]=%0d",
                 d_array[thread_id]);                    // GOTCHA!
      join_none  // don't wait for each fork to complete
    #10 $finish;
  end
endprogram
```

The expected results from this example are

```
  thread_id=0 d_array[thread_id]=10
  thread_id=1 d_array[thread_id]=11
  thread_id=2 d_array[thread_id]=12
```

When simulation runs, however, the messages printed are not as expected.

```
  thread_id=3  d_array[thread_id]=0
  thread_id=3  d_array[thread_id]=0
  thread_id=3  d_array[thread_id]=0
```

Gotcha!

The gotcha in this example that `thread_id` variable is declared outside of the **`fork`**...**`join_none`** block, which means that every forked thread shares the same `thread_id` variable, instead of having a unique variable for each thread. Since the threads are invoked from within a **`fork`**...**`join_none`**, all three threads are scheduled to start concurrently. The threads do not actually start running, however, until the for loop completes. When the **`for`** loop completes, `thread_id` is 3, and that is the value used by each thread. *Gotcha!*

Where did the value of 0 for what is read out of d_array come from? The d_array is storing int values, which are 2-state data types. When an out-of-bounds access occurs with 2-state arrays, a value of 0 is returned. This can hide functional errors, which is a gotcha described in Gotcha 40 on page 90. Had this example only been printing, or just using, the value read from d_array and not printing the index value, the code problem would have been very obscure. *Gotcha, again!*

How to avoid this Gotcha

When storage is required *within* a concurrent thread, each thread needs to define its own local storage. This is done by defining automatic variables within each thread. The following example works correctly.

```
program test (input logic clock);
  int d_array[3] = '{10,11,12};
  initial begin
    for (int i=0; i<3; i++)
      fork
        automatic int thread_id = i;  // local var for each thread
        $write("thread_id=%0d  ", thread_id);
        $display("d_array[thread_id]=%0d",
                 d_array[thread_id]);                    // OK
      join_none  // don't wait for each fork to complete
    #10 $finish;
  end
endprogram
```

What if thread_id had not been declared as automatic? The default lifetime of local variables is static. A static `thread_id` would have been initialized once at the very beginning of simulation, and not re-initialized for each pass of the **`for`** loop. This common coding error is described in Gotcha 77 on page 162. To avoid that gotcha, the default lifetime of local variables in test programs should be changed to automatic, as follows:

```
program automatic test (input logic clock);     // default storage
                                                // is automatic
```

See Gotcha 7 on page 22 for coding guidelines on declaring programs, packages and interfaces as automatic.

Gotcha 79: Disable fork kills too many threads

Gotcha: When I execute a disable fork statement, sometimes it kills threads that are outside the scope containing the disable fork statement.

Synopsis: The disable fork statement kills all threads started from the current thread.

The **disable fork** statement stops all active threads that were spawned from the current thread. The problem is that this may accidentally stop threads outside the scope of the code that contains the disable.

The following example calls the do_action task, and then calls the start_a_thread task twice, to spawn two threads with delays of 10 and 30. It then waits a short time and does a **disable fork**, which stops the two threads. However, this also unintentionally stops the start_a_thread thread that was started from the do_action task. *Gotcha!*

```
program automatic test;

  task start_a_thread (int delay);
    fork
      begin
        $display("@%0d start_a_thread(%0d) - started",
                 $time, delay);
        #(delay);
        $display("@%0d start_a_thread(%0d) - complete",
                  $time, delay);
      end
    join_none
  endtask

  initial begin
    do_action();
    start_a_thread(10);
    start_a_thread(30);
    #15 disable fork;                                  // GOTCHA!
    #100;   // wait for all threads to complete
  end

  task do_action;
    start_a_thread(20);
  endtask

endprogram
```

The simulation results from this test are:

```
@0 start_a_thread(20) - started
@0 start_a_thread(10) - started
@0 start_a_thread(30) - started
@10 start_a_thread(10) - complete
```

Note that, in this output, thread 20 never completes, even though it was not explicitly disabled. This is the *Gotcha!*

How to avoid this Gotcha

Always put a **fork...join** block around code that uses a **disable fork** to create a firewall. This creates a thread and limits the scope of the **disable fork** statement.

The example below changes the **initial** block by adding a **fork...join** to insulate the thread started from `do_action()` from the effect of the **disable fork**.

```
initial begin
  do_action();
  fork     // isolate following statements as a separate thread
    begin
      start_a_thread(10);
      start_a_thread(30);
      #15 disable fork;          OK, only affects the fork...join
    end
  join
  #100;   // wait for all threads to complete
end
```

The simulation results from this modified test are:

```
@0 start_a_thread(20) - started
@0 start_a_thread(10) - started
@0 start_a_thread(30) - started
@10 start_a_thread(10) - complete
@20 start_a_thread(20) - complete
```

Observe that thread 20 now completes execution.

Gotcha 80: Disabling a statement block stops more than intended

Gotcha: When I try to disable a statement block in one thread, it stops the block in all threads.

Synopsis: A disable block_name statement stops the execution of all blocks with that name in all threads.

In the following example, the task `start_test` spawns a thread containing a block named `terminator`. The task is called three times inside a module that is instantiated three times in the top module. Thus, there are nine `terminator` blocks running concurrently as nine separate threads.

The intent of this code is that the `terminator` block disables itself if the block is in the second call of the second instance.

```
module test (input int instance_id);
  initial begin
    #1;
    start_test(1, instance_id);   // three calls to task, with
    start_test(2, instance_id);   // different thread numbers
    start_test(3, instance_id);
  end

  task automatic start_test (int thread, int inst);
    fork : monitor
      begin
        $display("@%0d: %m inst: %0d, thread: %0d, before disable",
           $time, inst, thread);
        #10;
        if ((thread==2) && (inst==2))
           disable monitor;     // GOTCHA! affects multiple threads
        #1;
        $display("@%0d: %m inst: %0d, thread: %0d, after disable",
           $time, inst, thread);
      end
    join_none  // don't wait for thread to complete
  endtask: start_test
endmodule: test


module top;
  test t1 (.instance_id(1));    // three instances of test module
  test t2 (.instance_id(2));
  test t3 (.instance_id(3));
endmodule: top
```

The simulation output for this code is:

```
@ 1: top.t1.start_test.monitor inst: 1, thread: 3, before disable
@ 1: top.t1.start_test.monitor inst: 1, thread: 2, before disable
@ 1: top.t1.start_test.monitor inst: 1, thread: 1, before disable
@ 1: top.t2.start_test.monitor inst: 2, thread: 3, before disable
@ 1: top.t2.start_test.monitor inst: 2, thread: 2, before disable
@ 1: top.t2.start_test.monitor inst: 2, thread: 1, before disable
@ 1: top.t3.start_test.monitor inst: 3, thread: 3, before disable
@ 1: top.t3.start_test.monitor inst: 3, thread: 2, before disable
@ 1: top.t3.start_test.monitor inst: 3, thread: 1, before disable

@12: top.t1.start_test.monitor inst: 1, thread: 3, after disable
@12: top.t1.start_test.monitor inst: 1, thread: 2, after disable
@12: top.t1.start_test.monitor inst: 1, thread: 1, after disable
@12: top.t3.start_test.monitor inst: 3, thread: 3, after disable
@12: top.t3.start_test.monitor inst: 3, thread: 2, after disable
@12: top.t3.start_test.monitor inst: 3, thread: 1, after disable
```

This output shows that the `monitor` block stops all threads in instance 2, not just thread 2 of instance 2. *Gotcha!*

This gotcha happens because the label `monitor` is the name of the block, not the name of a specific thread. Disabling the block name stops all active threads of the block name for the module instance containing the block.

How to avoid this gotcha

The **`disable`** statement takes a static hierarchical name as its argument, which can be a relative hierarchical name, as in the example above, or a full hierarchical name. What is needed to avoid this gotcha and cancel just one thread is to reference a specific call to the `start_test` task. However, the `start_test` task above, is an automatic task. Automatic task calls do not have a static hierarchical name. Therefore, specific instances of that task cannot be referenced to disable a specific thread.

In order to run all nine threads concurrently and yet have the ability to disable a single thread, the concurrency must occur at the module instance level, and the task must be static. This is shown in the code below:

```
module test (input int instance_id, input int thread_id);
  initial begin
    #1;
    start_test(thread_id, instance_id);   // one call to task
  end
```

```
  task start_test (int thread, int inst);   // static task
    begin : monitor   // named block (not forked threads)
        $display("@%0d: %m inst: %0d, thread: %0d, before disable",
           $time, inst, thread);
        #10;
        if ((thread==2) && (inst==2))
           disable monitor;          // OK, only affects this thread
        #1;
        $display("@%0d: %m inst: %0d, thread: %0d, after disable",
           $time, inst, thread);
    end: monitor
  endtask: start_test
endmodule: test


module top;
  test t1 (.instance_id(1), .thread_id(1));  // nine instances of
  test t2 (.instance_id(1), .thread_id(2));  // test module
  test t3 (.instance_id(1), .thread_id(3));
  test t4 (.instance_id(2), .thread_id(1));
  test t5 (.instance_id(2), .thread_id(2));
  test t6 (.instance_id(2), .thread_id(3));
  test t7 (.instance_id(3), .thread_id(1));
  test t8 (.instance_id(3), .thread_id(2));
  test t9 (.instance_id(3), .thread_id(3));
endmodule: top
```

The simulation output below shows all nine threads starting at time 1, and then shows that only instance 2 thread 2 is removed after the disable. The remaining threads for instance 2 are not disabled, and continue running.

```
@ 1: top.t1.start_test.monitor inst: 1, thread: 1, before disable
@ 1: top.t2.start_test.monitor inst: 1, thread: 2, before disable
@ 1: top.t3.start_test.monitor inst: 1, thread: 3, before disable
@ 1: top.t4.start_test.monitor inst: 2, thread: 1, before disable
@ 1: top.t5.start_test.monitor inst: 2, thread: 2, before disable
@ 1: top.t6.start_test.monitor inst: 2, thread: 3, before disable
@ 1: top.t7.start_test.monitor inst: 3, thread: 1, before disable
@ 1: top.t8.start_test.monitor inst: 3, thread: 2, before disable
@ 1: top.t9.start_test.monitor inst: 3, thread: 3, before disable

@12: top.t1.start_test.monitor inst: 1, thread: 1, after disable
@12: top.t2.start_test.monitor inst: 1, thread: 2, after disable
@12: top.t3.start_test.monitor inst: 1, thread: 3, after disable
@12: top.t4.start_test.monitor inst: 2, thread: 1, after disable
@12: top.t6.start_test.monitor inst: 2, thread: 3, after disable
@12: top.t7.start_test.monitor inst: 3, thread: 1, after disable
@12: top.t8.start_test.monitor inst: 3, thread: 2, after disable
@12: top.t9.start_test.monitor inst: 3, thread: 3, after disable
```

Gotcha 81: Simulation exits prematurely, before tests complete

Gotcha: My simulation exits prematurely, before I call $finish, and while some tests are still running.

Synopsis: When all program blocks complete, $finish is implicitly called, even if there is simulation activity still running.

Code that models hardware design needs to run continuously, such as an **always** block that triggers on every clock cycle. Verification code, on the other hand, needs to finish execution after testing is complete. In Verilog, modules that are used for testing must explicitly state when testing is finished, using the **$finish** system task.

SystemVerilog adds a construct to encapsulate verification code, called a program block. Instead of using Verilog modules, program blocks are declared between the keywords **program**...**endprogram**.

```
program automatic test                          // verification program
(output logic [63:0] test_data,
 output logic        reset_n,
 input  logic [63:0] results,
 input  logic        test_clk
);
  initial begin
    reset_n <= 0;
    @(posedge test_clk) reset_n = 1;
    fork
      test1_task (...);
      test2_task (...);
    join
  end                        // NOTE: not necessary to call $finish
endprogram
```

Unlike modules, program blocks do not need to run continuously. When the end of a program block is reached, simulation will automatically finish, without the need to automatically call **$finish**. If verification code is divided into multiple program blocks, simulation automatically exits when the end of all program blocks has been reached.

In general, it makes sense to have simulation automatically finish when testing is complete. In fact, having an implicit automatic finish solves a common gotcha in Verilog of forgetting to explicitly call **$finish** at the end of test code. However, there are some gotchas with this automatic termination. One gotcha occurs if some test code is in program blocks, and other test code is in Verilog modules (perhaps some legacy test code that was written in Verilog). Simulation will automatically exit when the program blocks complete, even if other verification

code that is not in program blocks is still running. Another possible gotcha can occur when the program block spawns parallel test threads, but does not wait for those threads to complete. When the program block reaches its end, simulation will automatically finish, even if the spawned threads are still running. Using **fork...join_any** and **fork...join_none**, a test program can spawn off threads, and not wait for some or all of the threads to complete.

```
program automatic test (...);
  initial begin
    fork
      test1_task (...);
      test2_task (...);
    join_none
  end       // GOTCHA! simulation might exit before tasks finish
endprogram
```

How to avoid this Gotcha: The gotcha of an unexpected, automatic exit from simulation can be avoided by suspending execution of the test program until all tests have completed. If some tests are running in a Verilog module, the program block can contain a **wait** statement or some other delay that waits or suspends the test program until the test module has reached its end. The test module can set a flag, trigger and event, or use SystemVerilog semaphores to indicate that it has completed. If the test program forks off verification threads, it can suspend until all threads have completed, using a **wait fork** statement. For example:

```
program automatic test (...);
  initial begin
    fork
      test1_task (...);
      test2_task (...);
    join_none
    wait fork;        // waits for all spawned processes to complete
  end                 // OK, can't get here until tasks finish
endprogram
```

Chapter 7

Randomization, Coverage and Assertion Gotchas

Gotcha 82: Variables declared with rand are not getting randomized

Gotcha: Some of my class variables are not getting randomized, even though they were tagged as rand variables.

Synopsis: Properties must have a rand or randc tag, in order to be randomized. This includes handles to other objects.

In order for object variable values to be randomized, each variable in the object must be declared with a **`rand`** or **`randc`** tag. Random values are generated when the object's **`.randomize`** method is called.

The following example has a `Payload` class, which has a property called `data` that is tagged to be randomized. A `Header` class contains an `addr` property which is tagged to be randomized, and a handle to a `Payload` object. When a `Header` object is randomized, however, only `addr` gets a random value. The payload `data` is not randomized, even though it has a **`rand`** tag.

Note: the code examples in this chapter are contrived in order to illustrate each gotcha using small examples. In real design and verification code, these gotchas might not be as obvious or easy to debug.

```
program automatic test;
  class Payload;
    rand int data[8];          // data is tagged to be randomized
  endclass

  class Header;
    rand int addr;             // addr is tagged to be randomized
    Payload p;                 // handle to Payload object -- GOTCHA!

    function new;
      this.p = new;
    endfunction
  endclass

  initial begin
    Header h = new;
    assert(h.randomize());     // randomize address and payload data
    $display(h.addr);          // addr shows random value
    foreach (h.p.data[i])
      $display(h.p.data[i]); // GOTCHA! data was not randomized
  end
endprogram
```

The **.randomize** method only randomizes properties in the scope of the object being randomized if the property is declared with a **rand** or **randc** tag. If the property is a handle to another object, the tag must be specified for both the handle and the properties in the child object. In the example above, `Header::addr` has been tagged with **rand**, so it gets updated with random values. The payload object, `Header::p`, however, is missing the **rand** modifier, so none of its variables are randomized, even though `Payload::data` has the **rand** tag.

How to avoid this Gotcha

All object variables that are to have random values generated, *including handles*, must have the **rand** modifier.

```
  class Header;
    rand int addr;          // addr is tagged to be randomized
    rand Payload p;         // OK, Payload is tagged to be randomized
    ...
  endclass
```

Gotcha 83: Undetected randomization failures

Gotcha: My class variables do not get random values, even though I called the randomize function.

Synopsis: The .randomize method returns an error status, and does not randomize variables if a constraint cannot be met.

It is possible to write constraints that cannot be solved under all conditions. If a constraint cannot be met, then the variables are not randomized. The **`.randomize`** method returns a 1 when the constraint solver succeeds in randomizing the class variables, and a 0 if it does not succeed.

The following example erroneously specifies a constraint, such that `a` must be less than `b`, and `b` must be less than `a`. These randomization failures could go undetected.

```
program automatic test;
  class Bad;
    rand bit [7:0] a, b;
    constraint ab {a < b;
                   b < a;}  // this constraint cannot be solved
  endclass

  initial begin
    Bad b = new;
    void'(b.randomize());   // GOTCHA! return from method ignored
  end
endprogram
```

If the success flag is not checked, the only symptom when a constraint cannot be solved is that one or more class variables were not randomized. The failure to randomize some class variables could go undetected. *Gotcha!*

How to avoid this Gotcha

Use SystemVerilog assertions to check the return status of `.randomize`. The method will return:

- 1 if successful in generating the random values
- 0 if unsuccessful in generating random values that met the constraints

In the example below, an assertion is used to test the return value if the **`.randomize`** succeeded. The assertion fail statement defines an assertion failure as fatal, which will abort simulation or formal verification.

```
program automatic test;
  class Bad;
    rand bit [7:0] a, b;
    constraint ab {a < b;
                   b < a;}  // this constraint cannot be solved
  endclass

  initial begin
    Bad b = new;
    assert(b.randomize()) else $fatal;  // OK, checking if
                                        // randomize fails
  end
endprogram
```

Gotcha 84: $assertoff could disable randomization

Gotcha: I used an assertion to detect randomization failures, and now nothing gets randomized during reset.

Synopsis: The $assertoff assertion control will disable any statements executed within the assertion.

When the **.randomize** method is called, it returns a value indicating the pass or fail of the randomization. A common approach to monitor the return value from the .randomization method is to use an immediate assert, as shown in the code below.

```
program test;
  ...
  Bustrans tr;

  initial begin
    tr = new;
    $assertoff();    // GOTCHA! disable all assertions during reset
    rst_n <= 0;

    assert (tr.randomize) else $display("randomization failed");

    #10 rst_n <= 0;
    $asserton();
    ...
  end
endprogram
```

The intent of the example above is to turn off assertions during reset, to prevent false assertion failure messages. During reset, the object `tr` is randomized. But, the properties of `tr` will not get randomized, during reset because the assert statement calling `tr.randomize` is disabled. *Gotcha!*

How to avoid this gotcha

There are a few ways to get around this gotcha. One is to use an **if...else**, instead of an **assert** statement, to check the return of the **.randomize** method. Using this approach, however, means the **.randomize** call will not be included in any assertion monitoring or assertion reports provided by the simulator at the end of simulation.

A second way to avoid this gotcha is to not call **$assertoff** on the scope containing the calls to randomization. The **$assertoff** task can be passed hierarchical scope names, such as the name of the top-level of the design or a specific module containing the assertions t be turned off.

A third approach is to add a label to the assertion statement containing the call to **.randomize**, and then turn that assertion back on immediately following the **$assertoff**. For example:

```
program test;
  ...
  Bustrans tr;

  initial begin
    tr = new;
    $assertoff();        // disable all assertions during reset
    $asserton(tr_rand);  // OK, tr randomization turned back on
    rst_n <= 0;

    tr_rand: assert (tr.randomize)
             else $display("randomization failed");

    #10 rst_n <= 0;
    $asserton();
    ...
  end
endprogram
```

A fourth way to avoid this gotcha is to apply the **.randomize** call before or after the reset is applied.

```
program test;
  ...
  Bustrans tr;

  initial begin
    tr = new;

    tr_rand: assert (tr.randomize)
             else $display("randomization failed");

    $assertoff();        // OK, tr randomization already executed
    rst_n <= 0;
    #10 rst_n <= 0;
    $asserton();
    ...
  end
endprogram
```

With this solution, if the call to the **.randomize** method is in a different process than the call to **$assertoff**, it may be necessary to use some form of process synchronization, to ensure that the randomization occurs before assertions are turned off.

Gotcha 85: Boolean constraints on more than two random variables

Gotcha: When I specify constraints on more than two random variables, I don't get what I expect.

Synopsis: In a series of two Boolean relational operators, the second operation is compared to the true/false result of the previous operation.

The intent of the constraint in the class below is to randomize `lo`, `med` and `hi`, such that `lo` is less than `med` and `med` is less than `hi`, by using the expression `(lo < med < hi;)`.

```
class bad1;
  rand bit [7:0] lo, med, hi;
  constraint increasing { lo < med < hi; }  // GOTCHA!
endclass
```

A sample output from running the code above looks like this:

```
lo =  20, med = 224, hi = 164
lo = 114, med =  39, hi = 189
lo = 186, med = 148, hi = 161
lo = 214, med = 223, hi = 201
```

This constraint does not cause the solver to fail, but the randomized values are not as expected. In line one, above, `med` is greater than `hi`. In lines two and three, `lo` is greater than `med`. In line four, both `lo` and `med` are greater than `hi`.

The reason that the constraint does not work is that the Boolean less-than expressions are evaluated from left to right. This means that the operation is not comparing `med` to `hi`. It is comparing the true/false result of `(lo < med)` to `hi`. The constraint above is evaluated as:

```
constraint increasing { (lo < med) < hi; }
```

The constraint is actually only constraining `hi`, such that `hi` has a value greater than 0 or 1 (depending on the result of the test `(lo < med)`. The variables `lo` and `med` are randomized, but are not constrained. *Gotcha!*

The following example illustrates a similar problem. This constraint is intended to create values `a`, `b` and `c`, such that the three properties have the same value.

```
class bad2;
  rand bit [7:0] a, b, c;
  constraint equal {a == b == c; }
endclass
```

A sample output from running the code above gave the following output:

```
a =  25, b = 173, c =  0
a =  65, b = 151, c =  0
a = 190, b =  33, c =  0
a =  65, b =  32, c =  0
```

A different simulator gives this output:

```
a =  61, b =   1, c =  0
a =   9, b =   9, c =  1
a = 115, b = 222, c =  0
a = 212, b = 212, c =  1
```

The constraint is equivalent to: `(a == b) == c`. Random values are chosen for `a` and `b`, and then those values are tested to see if they are equal. Variable `c` is then constrained to be equal to the true/false result of `(a == b)`, which is 0 or 1. *Gotcha!*

How to avoid these Gotchas

Constraints involving compound Boolean operations should be broken down to separate statements. The above constraints should be modeled as:

```
constraint increasing {
  lo < med;       // lo is constrained to be less than med
  med < hi;       // med is constrained to be less than hi
}

constraint equal {
  a == b;         // a is constrained to be equal to b
  b == c;         // b is constrained to be equal to c
}
```

Gotcha 86: Unwanted negative values in random values

Gotcha: I am getting negative values in my random values, where I only wanted positive values.

Synopsis: Unconstrained randomization considers all possible 2-state values within a given data type.

In the following class, both `i` and `b` are signed variables which can store negative values.

```
class Negs;
  rand int  data;
  rand byte address;   // GOTCHA! address can be negative
endclass
```

The **`int`** and **`byte`** types are signed types. Therefore, the **`.randomize`** method will generate both positive and negative values for these variables. If either of these variables is used in a context where a positive number is required, the outcome could be unexpected or erroneous, such as generating a negative value for an address bus.

How to avoid this Gotcha

When the randomized test variables are to be passed to hardware as stimulus, it is generally best to use unsigned types such as **`bit`** or **`logic`**. This ensures that randomized values will always be positive values.

There are times when signed types need to be used, but only positive numbers are desired. For example, it may be preferred to use the C-like **`int`**, **`byte`**, **`shortint`** and **`longint`** types when the variables are to be passed to C functions using the SystemVerilog Direct Programming Interface (DPI). When signed types need to be used, but only non-negative values are desired, randomization can be constrained to non-negative numbers. For example:

```
class Negs;
  rand int  data;
  rand byte address;   // OK, address constrained to non-negative
  constraint pos
    { data >= 0;
      address >= 0;}
endclass
```

Gotcha 87: Coverage reports default to groups, not bins

Gotcha: I've defined specific coverage bins inside my covergroup to track coverage of specific values, but the report only shows the coverage of the entire covergroup.

Synopsis: The get_coverage() and get_inst_coverage() methods do not break down coverage to individual bins.

SystemVerilog provides powerful functional coverage for design verification. As part of functional coverage, verification engineers define *covergroups*. A covergroup encapsulates one or more definitions of *coverpoints* and *cross coverage*. A coverpoint is used to divide the covergroup into one or more *bins*, where each bin includes specific expressions within the design, and specific ranges of values for those expressions. Cross coverage specifies coverage of combinations of cover bins. An example covergroup definition is:

```
enum {s1,s2,s3,s4,s5} state_e, nstate_e;

covergroup cSM @(posedge clk);
  coverpoint state_e {
    bins state1  = (s1);
    bins state2  = (s2);
    bins state3  = (s3);
    bins state4  = (s4);
    bins state5  = (s5);
    bins st1_3_5 = (s1=>s3=>s5);
    bins st5_1   = (s5=>s1);
  }
endgroup
```

These covergroup bins count the number of times each state of a state machine was entered, as well as the number of times certain state transition sequences occurred.

SystemVerilog also provides built-in methods for reporting coverage. It seems intuitive for coverage reports to list coverage by the individual bins within a covergroup. However, this is not the default for how coverage is reported. *Gotcha!*

When the SystemVerilog `get_inst_coverage()` method is called to compute coverage for an instance of a covergroup, the coverage value returned is based on all the coverpoints and crosspoints of the instance of that covergroup.

When the SystemVerilog `get_coverage()` method is called, the computed coverage is based on data from all the instances of the given covergroup.

The gotcha with coverage reporting is that coverage is based on crosspoints or coverpoints. There are no built in methods to report details of individual bins of a crosspoint. If the coverage is not 100%, the designer has no way to tell which bins are empty.

How to avoid this Gotcha

If the coverage details for each bin are needed, then each covergroup should have just one coverpoint, and that coverpoint should have just one bin. Then, when the coverage is reported for that cover group, it represents the coverage for the coverpoint bin.

Gotcha 88: Coverage is always reported as 0%

Gotcha: I defined a covergroup, but the group always has 0% coverage in the cover report.

Synopsis: Covergroups are specialized classes and must be constructed before they can be used.

The following example defines a covergroup as part of a class definition. The intent is to provide coverage of the properties within the class. When the class object is constructed, however, the covergroup does not keep track of the information intended.

```
program automatic test;
  event cg_sample;

    covergroup CG_xyz @(cg_sample);  // covergroup definition
      coverpoint x;
      coverpoint y;
      coverpoint z;
    endgroup

  class Abc;
    rand bit [7:0] a, b, c;
    covergroup CG_abc @(cg_sample);  // covergroup definition
      coverpoint a;
      coverpoint b;
      coverpoint c;
    endgroup
  endclass

  initial begin
    Abc a1 = new;                    // instance of Abc object
    ... // generate stimulus
    $get_coverage();                 // GOTCHA! reports 0 coverage
  end
endprogram
```

The reason no coverage is reported is that a covergroup is a special type of class definition. In order to generate coverage reports, the covergroup object must first be constructed using the covergroup's **new** method, in the same way as when constructing a class object. The example above never constructs a `CG_xyz` coverage object. The example constructs an instance of the `Abc` object, but constructing the class object does not construct an instance of the `CG_abc` covergroup within the class. Hence, no coverage information is collected for either the `CG_xyz` or `CG_abc` cover groups. No errors or warnings are reported for this coding error. The only indication that there is a problem is an erroneous or incomplete coverage report. *Gotcha!*

How to avoid this Gotcha

An instance of a covergroup must always be constructed in order to collect coverage information about that group. When the group is defined in a class, as in the example above, the covergroup instance should be constructed as part of the class constructor. In that way, each time a class object is constructed, the covergroup instance for that object will automatically be constructed.

```
program automatic test;
  event cg_sample;

    covergroup CG_xyz @(cg_sample);  // covergroup definition
      coverpoint x;
      coverpoint y;
      coverpoint z;
    endgroup

  class Abc;
    rand bit [7:0] a, b, c;
    covergroup CG_abc @(cg_sample);  // covergroup definition
      coverpoint a;
      coverpoint b;
      coverpoint c;
    endgroup

    function new;
      CG_abc = new;                  // instance of covergroup
    endfunction
  endclass

  initial begin
    Abc a1 = new;                    // instance of Abc object
    CG_xyz = new;                    // instance of covergroup
    ... // generate stimulus
    $get_coverage();                 // OK, reports coverage
  end
endprogram
```

Another reason why coverage could be reported as 0% is that the cover group was never triggered. This could be because its trigger never fired, or the **.sample** method for the covergroup instance was never called.

Gotcha 89: The coverage report lumps all instances together

Gotcha: I have several instances of a covergroup, but the coverage report lumps them all together.

Synopsis: By default, the coverage report combines all the instances of a covergroup together.

The intent in the example below is to measure the coverage on each of two pixel `x:y` pairs.

```
covergroup pixelProximity(ref bit signed [12:0] pixel1,
                          ref bit signed [12:0] pixel2)
  @(newPixel);
  cp1: coverpoint (pixel2 - pixel1) {
         bins lt = {[1:$]};       // pixel1's coord less than pixel2
         bins eq = {0};           // did the pixels coincide?
         bins gt = {[-4096:-1]}; // pixel1's coord greater than
                                  // pixel2
       }
endgroup

pixelProximity px, py;

initial begin
  bit signed [12:0] x1, y1, x2, y2;
  px = new(x1, y1);                  // construct first covergroup
  py = new(x2, y2);                  // construct second covergroup
  ... // generate stimulus
  $get_coverage();                   // GOTCHA! report lumps
                                     // px and py together
end
```

In this example, two covergroup objects are constructed, `px` and `py`. Instead of seeing separate coverage for each covergroup, however, the coverage report combines the counts for both groups into a single coverage total. *Gotcha!*

How to avoid this Gotcha

The covergroup needs to set the **`.per_instance`** coverage option, as shown below:

```
covergroup pixelProximity(ref bit signed [12:0] pixel1,
                          ref bit signed [12:0] pixel2)
  @(newPixel);
  option.per_instance = 1;  // report for each covergroup instance
  cp1: coverpoint (pixel2 - pixel1) {
         ...
       }
endgroup
```

Gotcha 90: Covergroup argument directions are sticky

Gotcha: Sometimes the call to my covergroup constructor does not compile.

Synopsis: A covergroup ref argument cannot be passed a constant value.

A generic covergroup has arguments that pass in values and variables. The default direction is **input**, for passing in fixed values, and **ref**, for passing in variables for coverpoints. The direction is sticky, and remains in effect until a new direction is specified.

In the following example, the call to the covergroup constructor passes in the variable va, and the constants 0 for low and 50 and high. The code looks like it should do what is expected, but instead gets a compilation error.

```
covergroup cg (ref int ra, int low, int high )
  @(posedge clk);
  coverpoint ra // sample variable passed by reference
    {bins good = { [low : high] };
    bins bad[] = default;
  }
endgroup

initial begin
  int va, vb;
  int min=0, max=50;
  cg c1 = new(va, min, max);  // OK
  cg c2 = new(vb, 0, 50);     // GOTCHA! cannot pass constants
                              // to ref args
end
```

In the covergroup definition above, ra is a **ref** argument. This direction is sticky, and affects all arguments that follow, until a different direction is specified. Since no direction is given for low and high, they also default to **ref** arguments. The call to the constructor fails, because the actual values passed to **ref** arguments must be variables. It is not allowed to pass a constant into a **ref** argument. The sticky direction behavior of covergroup arguments is similar to task/function arguments, as described in Gotcha 69 on page 150.

How to avoid this Gotcha

It is best to specify the direction for each covergroup argument, especially when ref arguments are used. This documents the code intent, and prevents the gotcha of an argument inheriting the direction of a previous argument. For example:

```
covergroup cg (ref int ra, input int low, input int high )
  ...
endgroup
```

Gotcha 91: Assertion pass statements execute with a vacuous success

Gotcha: My assertion pass statement executed, even though I thought the property was not active.

Synopsis: A vacuous success will execute the assertion pass statement.

The **assert property** construct can be followed by optional pass and fail statements.

```
assert property (p_req_ack) $display("passed");
else $display("failed");
```

The optional pass statement is executed if the property succeeds, and the fail statement is executed if the assertion fails. The pass/fail statements can be any executable statement. Multiple statements can be executed by grouping them between **begin** and **end**.

Most property specifications contain an *implication operator*, represented by either `|->` or `|=>`, which qualifies when the assertion should be run. The sequence expression before the implication operator is called the *antecedent*. The sequence expression after the operator is called the *consequent*. A property specification that uses an implication operator has three possible results: *success*, *failure*, and *vacuous success*. If the implication antecedent is true, the consequent is evaluated, and the property will pass or fail, based on the results of testing the consequent. If the implication antecedent is false, the consequent is a "don't care", and the property returns a vacuous success.

The intent of the following assertion is to increment a counter on each successful assertion. The assertion checks to see if a `req` is followed by `ack` 1 clock cycle later.

```
assert property (p_req_ack) req_ack_count++; else $error; // GOTCHA

property p_req_ack;
  @(posedge clk) req |-> ##1 ack;  // if req, check for ack
                                   // on next cycle
endproperty
```

The gotcha is that the **assert property** statement does not distinguish between a real success and a vacuous success. Either one will cause the pass statement to be executed. As a result, this example counts both how many times `req` was followed by `ack` (successes) *and* how many clock cycles in which there was no `req` (vacuous successes). *Gotcha!*

How to avoid this Gotcha

This gotcha can be avoided by executing the desired statement(s) from within the assertion property, instead of as a pass statement. SystemVerilog assertions can have executable statements associated with the evaluation of an expression. The following example places the code to increment the counter in a function, which is then called when the *property* (not the assertion) successfully sees `req` followed by `ack`.

```
assert property (p_req_ack) else $error;  // OK. no pass statement

property p_req_ack;
  @(posedge clk) $rose(req) |-> ##1 ($rose(ack), inc_cnt);
endproperty

function void inc_cnt;
  req_ack_count++;            // OK, not executed on vacuous success
endfunction
```

This gotcha has been addressed in the next version of the SystemVerilog standard, planned for ratification 2008. The IEEE 1800 SystemVerilog standards committee has proposed new system tasks to control the execution of assertion pass statements: **`$assertvacuousoff`** and **`$assertvacuouson`** These system tasks will allow a designer to disable or enable the assertion pass statements on vacuous successes.

Specific to the previous example, there is another solution. SystemVerilog coverage can be used to count how many times `req` was successfully followed by `ack`, instead of the assertion pass statement.

Gotcha 92: Concurrent assertions in procedural blocks

Gotcha: My assertion pass statements are executing, even when the procedural code does not execute the assertion.

Synopsis: Concurrent assertions in procedural code actually fire every clock cycle, not just when the procedural code executes.

A concurrent assertion can be placed inside an **`initial`** or **`always`** block, and the assertion guarded by procedural code, such as an **`if`** statement.

```
always_ff @(posedge clk) begin
  if (state_e == FETCH)
    assert property (p_req_ack)          // GOTCHA!
      $display("passed")                 // pass statement
    else $display("failed");             // fail statement
  ...
end

property p_req_ack;
  @(posedge clk)
      req |-> ##1 ack;     // a req should get an ack 1 cycle later
endproperty
```

The intent of this assertion is that `req` followed by `ack` is only checked when `state_e` is `FETCH`, and that each `req` should be followed one clock cycle later by an `ack`.

In the simulation results, below, simulation was run for 9 clock cycles, four with the `state_e` variable equal to `INIT`, and four with the variable equal to `FETCH`. The table shows the value of `state_e` and `req` on one clock cycle, and the value of `ack` on the next clock cycle.

```
  1st cycle   2nd cycle   output message and notes
------------  ---------   ---------------------------------------
state_e  req     ack
------------  ---------
INIT      0       0       passed  // vacuous success  -- GOTCHA!
INIT      1       0       passed  // vacuous success  -- GOTCHA!
INIT      1       1       passed  // vacuous success  -- GOTCHA!

FETCH     0       0       passed  // vacuous success
FETCH     1       0       failed  // true failure
FETCH     1       1       passed  // true success
```

This example illustrates two gotchas. The first gotcha is shown on line one of the output. The assertion ran, even when `state_e` was not equal to `FETCH`. The reason the assertion ran, even when the **`if`** condition was false, is that concurrent assertions in procedural code are still concurrent assertions. As such, the assertions run as concurrent threads, in parallel with the procedural block. Because the assertion is a concurrent assertion, it executes on every positive edge of `clk`, even when the **`if`** condition is false. *Gotcha!*

The second gotcha in this example is shown on line 2 of the output. There is a `req` on the first cycle, but it is not followed by an `ack` on the next cycle. This should be an assertion failure, but the assertion reported a success. *Gotcha, again!*

This second gotcha occurs because the procedural **`if`** statement is treated as an implication operation in the assertion property. When the **`if`** condition is false, the property is a vacuous success, regardless of the values of `req` and `ack`.

How to avoid these Gotchas

In reality, there are no gotchas to avoid. The assertion in the previous example worked exactly as it should. A vacuous success occurs when the antecedent of an implication operator is false. While it is not obvious in the example above, there are actually two implications: the `if(state_e == FETCH)` in the procedural code, and the `req |->` in the assertion property. If either implication is false, a vacuous success will occur, which is exactly what happened in this example.

What can be misleading is that the pass statement is executed on a vacuous success (see Gotcha 91 on page 188). The message printed from the pass statement can make it appear that the assertion passed when it should not have.

To avoid being mislead by the assertion pass statement, do not use pass statements with concurrent assertions that are guarded by a conditional statement in procedural code . The **`$assertvacuousoff`** referenced in Gotcha 91 on page 188 can also resolve the pass statement being executed on a vacuous success.

Gotcha 93: Mismatch in assert...else statements

Gotcha: My assertion fail statement executes when the assertion succeeds instead of fails.

Synopsis: An "if" without an "else" in an assert pass statement causes the assert "else" (fail) statement to be paired with the "if" in the pass statement.

The **assert** and **assert property** constructs can be followed by optional pass and fail statements.

```
assert property (p_req_ack) $display("passed");
else $display("failed");
```

The optional pass statement is executed if the property succeeds, and the optional fail statement is executed if the assertion fails.

The pass statement can be any executable statement, including a conditional **if** or **if...else** statement. The following example has a gotcha:

```
assert property (p_req_ack)
  if (cnt_en) req_ack_count++;   // assertion pass statement
else $fatal;                     // GOTCHA! this is not the
                                 // assertion fail statement
```

Assertions follow the same syntax as nested **if...else** statements, in that the **else** is associated with the nearest **if**. In the example above, the **else** statement is associated with the **if** condition in the assertion pass statement. Syntactically, there is no assertion fail statement in this example. This is not a syntax error, since the fail statement is optional. Instead, the **else** branch executes whenever the assertion succeeds or vacuously succeeds, and the pass statement **if** condition is false. *Gotcha!*

How to avoid this Gotcha

This gotcha is similar to the nested **if...else** gotcha described in Gotcha 57 on page 128, and is avoided in the same way. Either an **else** must be paired with the **if**, or the **if** condition must be encapsulated within a **begin...end** statement group, as shown below.

```
assert property (p_req_ack)
  begin
    if (cnt_en) req_ack_count++;  // OK, assertion pass statement
  end
else $fatal;                      // OK, assertion fail statement
```

Gotcha 94: Assertions that cannot fail

Gotcha: I have an assertion property with an open-ended delay in the consequent, and doesn't fail when it should.

Synopsis: Once the consequent of a property comes to an open-ended delay, the consequent will wait forever for the remaining conditions to return true.

The behavior of assertion sequences with an open-ended range is not intuitive, and can lead to unexpected assertion results.

The intent of the following assertion property is that a `req` should eventually be followed by an `ack`, which should be followed one cycle later by `done`.

```
property p_req_ack2;
  @(posedge clk)
  $rose(req) |-> ##[1:$] $rose(ack) ##1 $rose(done); // GOTCHA!
endproperty
```

In the example above, the assertion antecedent will wait for a rise on `req`. Until `req` rises, the assertion will be a vacuous success on each clock cycle. After `req` goes high, the consequent will wait forever for `ack` to go high. Once `ack` goes high, `done` should transition high on the next clock. If it does, the property passes. If `done` does not follow `ack`, *the assertion does not fail!* Instead, the consequent will just continue to wait for a rise on `ack`, followed by `done` going high one clock later. *Gotcha!*

The next property model is similar to the previous example, except that the open-ended range is between the evaluations of `ack` and `done`.

```
property p_req_ack2;
  @(posedge clk)
  $rose(req) |-> ##[1:5] $rose(ack) ##[1:$] $rose(done); // GOTCHA!
endproperty
```

In this example, if the rise on `ack` does not occur within five clocks of the rise of `req`, the consequence will fail. If `ack` does go high within the five clocks of `req`, the consequent will then wait forever for `done` to go high. This property can fail up to the point of starting the open-ended range. Once the open-ended range is started, the consequence cannot fail. It will wait forever for a passing condition of the remaining sequence. *Gotcha!*

The property shown next also illustrates that, once an open-ended range is reached, the consequent will not fail, but will wait until a passing sequence follows the open-ended delay.

```
property p_req_ack2;
  @(posedge clk)
  $rose(req) |-> ##[1:5] $rose(ack)
                 ##[1:$] $rose(done) ##1 $fell(bus_en); // GOTCHA!
endproperty
```

In this example, the property can fail if `req` is not followed by `ack`. But, once the open-ended range is reached, the property will wait forever for a rise on `done`, followed one clock later by `bus_en` having gone low. If `done`, goes high and `bus_en` does not go low within the next clock cycle, the consequent will just continue waiting until the passing sequence occurs. *Gotcha!*

How to avoid this gotcha

The cause of the gotchas in the preceding examples is having an open-ended range in the consequent that tests for more than one condition after the range. One solution is to break up the consequent into separate sequences and properties. The first property below checks for `req`, followed eventually by a rise on `ack`. The second property checks for `ack`, followed one cycle later by `done`.

```
property p_req_ack;
  @(posedge clk) $rose(req) |-> ##[1:$] $rose(ack); // OK
endproperty

property p_ack_done;
  @(posedge clk) $rose(ack) |-> ##1 $rose(done); // OK
endproperty
```

Another solution is to use a go-to repetition operator (`[->n]`). This sequence operator will provide the same "wait forever" for `ack`, but will fail if `ack` is not followed by `done`, instead of continuing to wait for an `ack` followed by `done`.

```
property p_req_ack2;
  @(posedge clk)
  $rose(req) |-> ##1 ($rose(ack)[->1]) ##1 $rose(done); // OK
endproperty
```

Chapter 8

Tool Compatibility Gotchas

Gotcha 95: Default simulation time units and precision

Gotcha: My design outputs do not change at the same time in different simulators.

Synopsis: Simulators have different defaults for delay time units (the 'timescale directive).

Time in Verilog is a 64-bit unsigned integer. Delays are specified by using a hash mark (#) followed by a number. A delay does not have any indication of what unit of time is being represented.

```
#2 sum = a + b;     // delayed execution of a programming statement
and #3 (y, a, b);  // 2-input AND gate with propagation delay
```

In Verilog, the time unit represented by delays is specified as a characteristic of a module, using a **`timescale** compiler directive. The directive contains two parts, the module's *time units* and the module's *time precision*. Each are specified in increments of 1, 10 or 100, in units ranging from seconds down to femtoseconds. The time precision allows a module to represent non-whole delays. The precision is relative to the time unit. Within simulation, all delays are scaled to the smallest precision used by the design.

An example of using **`timescale** is:

Note: the code examples in this chapter are contrived in order to illustrate each gotcha using small examples. In real design and verification code, these gotchas might not be as obvious or easy to debug.

```
`timescale 1ns/100ps          // 1 nanosecond units, 100 ps precision
module A (...);
  #2.3 ...                    // delay represents 2.3 nanoseconds
endmodule

module B (...);
  #5.5 ...                    // GOTCHA! delay represents 5.5 what?
endmodule

`timescale 1ps/1ps            // 1 picosecond units, 1 ps precision
module C (...);
  #7 ...                      // delay represents 7 picoseconds
endmodule
```

There are two common gotchas with Verilog **`timescale** directive: file order dependencies and no standard default.

The **`timescale** directive is not bound to modules or files. Once specified, the directive affects all modules and files that follow the directive, until a new **`timescale** is encountered by the compiler. This means that, if some design and/or test files contain time scale directives, and other files do not, then changing the order in which files are compiled will change how much time a delay represents in the files that do not have a time scale directive. This can cause radically different simulation results, even with the same simulator. *Gotcha!*

If a file is read in when no **`timescale** has been specified at all, then a compiler might, or might not, apply a default time unit. This, too, can cause radically different simulation results when simulating the same design on different simulators. *Gotcha!*

Some Verilog/SystemVerilog tools require that all files be compiled together. Timescale directives in one file can impact other files, as noted in the paragraphs above. Other Verilog/SystemVerilog tools support separate file compilation. In this case, a timescale directive in one file will not affect other files, but files with no directive depend on the tool's default timescale.

How to avoid this Gotcha using Verilog

To avoid this gotcha when using just Verilog, company-imposed or self-imposed coding rules must be strictly adhered to. There are three different coding styles for where to specify the **`timescale** directive:

- One style is to not use **`timescale** directives anywhere, and instead use the default time units of the software tool. This avoids the gotcha, but depends on being able to control the code of all models that make up a design. This may not be possible if models are obtained from multiple sources, such as IP models or ASIC cell library models. Commercial Verilog models are likely to contain **`timescale** directives.

- Another style is to make sure a **`timescale** directive is specified at the beginning of each and every module, in each and every design or testbench file. This eliminates both file order dependencies in multi-file compilation, and dependencies on the tool's default timescale in single-file compilation.
- A third style is to only specify **`timescale** in one file, and then include that file at the beginning of every other file. This approach also eliminates both file order dependencies in multi-file compilation, and dependencies on the tool's default timescale in single-file compilation.

How to avoid this Gotcha using SystemVerilog

SystemVerilog has two very important enhancements that help avoid the gotchas inherent with the **`timescale** directive. First, the time unit and time precision specifications are keywords that can be specified within a module, and made local to just the module. The keywords are **timeunit** and **timeprecision**. These keywords can also be specified within interfaces, programs and packages.

By making the time unit and precision part of the module definition, file order dependency problems and multi-file versus single-file compilation issues are eliminated.

The second SystemVerilog enhancement is allowing an explicit time unit to be specified with a delay value. This both documents the intended time unit, and eliminates dependency on what order **`timescale** directives are encountered by the compiler.

The following example shows both of these enhancements:

```
module B (...);
  timeunit 1ns;
  timeprecision 1ps;
  #5.5 ...          // delay represents 5.5 nanoseconds
  #1ms ...          // delay represents 1 millisecond
endmodule
```

Gotcha 96: Package chaining

Gotcha: My packages compile fine on all simulators, but my design that uses the packages will only compile on some simulators.

Synopsis: When one package imports a second package, and a design or testbench imports the first package, some simulators make declarations from both packages available, and some do not.

SystemVerilog packages provide a declarations space for definitions that are to be shared. A module, interface, or program can import specific package items, or use a wildcard import to make all items in a package visible. A package can also import items from other packages, as illustrated below.

```
package foo;
  typedef int unsigned uint_t;

  function int func_a (int a);
    return ~a;
  endfunction
endpackage

package bar;

  import foo::*;                    // wildcard import package foo

  function int func_b (uint_t b);
    return ~func_a(b);
  endfunction
endpackage

module test;
  import bar::*;                    // wildcard import bar

  uint_t c;                         // GOTCHA! reference definition
                                    // that is in package foo

  ...
endmodule
```

In this example, the `test` module does a wildcard import of package `bar`, and then references the `uint_t` definition. This definition is not defined in package `bar`. But, package `bar` imported this definition from package `foo`. This is referred to as *package chaining*.

Some software tools permit package chaining, and some simulators do not. *Gotcha!*

The gotcha in the example above is a result of an ambiguity in the SystemVerilog-2005 standard. The standard does not say whether package chaining is, or is not, allowed.

How to avoid this Gotcha

To ensure that design and verification code will work on all software tools, package chaining should not be used. Instead, a design or verification block should explicitly import each package that contains definitions used in the module. Either specific object imports or wildcard imports can be used, so long as each package that is used is explicitly referenced.

```
module test;
  import foo::*;                    // wildcard import bar
  import bar::*;                    // wildcard import bar

  uint_t c;                         // OK, reference definition
                                    // that is in package foo
  ...
endmodule
```

The IEEE SystemVerilog standard working group has addressed this ambiguity in the standard, and has proposed a change for the next version of the SystemVerilog standard. The change is to make implicit package chaining illegal, and to provide a mechanism for explicit package chaining. When tools implement this proposed change, the example illustrated at the beginning of this section, which uses implicit package chaining, will be illegal. However, package `bar` can enable chaining by importing definitions from package `foo`, and then exporting some or all of those definitions, thus making them visible to blocks that import `bar`.

Gotcha 97: Random number generator is not consistent across tools

Gotcha: I cannot repeat my constrained random tests on different tools.

Synopsis: The random number generator (RNG) used for constrained random generation in SystemVerilog is not defined in the IEEE specification.

The IEEE 1800-2005 SystemVerilog specification outlines and specifies the requirements for the constrained random number generator (RNG). However, the standard does not specify the algorithm to be used for random number generation or for solving constraints.

The following example shows a gotcha that is a result of not having a standard RNG and constraint solver. A test using the constraint shown below was run on simulators from two different tool vendors. This same test was also run on two different revisions of the same simulator from one of the two vendors.

```
class bad_constraint;
  rand bit [7:0] a, b, c;
  constraint equal { a == b == c; }
endclass
```

Random values generated by vendor 1:

```
a =  25, b = 173, c = 0
a =  65, b = 151, c = 0
a = 190, b =  33, c = 0
a =  65, b =  32, c = 0
```

Random values generated by vendor 2:

```
a =  61, b =   1, c = 0
a =   9, b =   9, c = 1
a = 115, b = 222, c = 0
a = 212, b = 212, c = 1
```

Random values generated by vendor 2, with a different version of the same tool:

```
a =   9, b =   9, c = 1
a = 212, b = 212, c = 1
a =  17, b =  17, c = 1
a = 150, b = 184, c = 0
```

The random values generated by each tool are not repeatable between tools, or even between different versions of the same tool. This means the constrained random tests run with tools from one vendor will not match results when the same test is run on another vendor's tool. Additionally, as shown in the test results above, there is no guarantee that tests from one revision to the next revision within the same tool vendor will give the same results. *Gotcha!*

How to avoid this gotcha

A standardized RNG and constraint solver would ensure that constrained random test generation would be consistent between different tools. At the time this book was written, however, the IEEE SystemVerilog standard working group had no plans for standardizing these important algorithms.

The next best solution is for the verification team to keep track of tools and revisions used for simulation. The team needs to make sure that results are checked as tools are upgraded from revision to revision, to ensure the test results are consistent. Additionally, if a verification team has access to simulators from multiple tool vendors, the team must keep track of which test results belong with which tool and revision.

Note: the constraint definition shown above has another gotcha, which is discussed in Gotcha 85 on page 179):

Gotcha 98: Loading memories modeled with always_latch/always_ff

Gotcha: When I use SystemVerilog, some simulators will not let me load my memory models using $readmemb.

Synopsis: The $readmemb() and $readmemh() system tasks cannot be used to load a RAM model that uses always_latch or always_ff.

Typically, a bus-functional model of a RAM is either synchronous (clock based) or asynchronous (enable based). Synchronous RAMs behave at the abstract level like flip-flops. Asynchronous RAMs behave at the abstract level like latches. However, there is a gotcha if these devices are modeled using SystemVerilog's **always_ff** or **always_latch** procedural blocks.

```
module RAM
(inout wire  [63:0] data,
 input logic [ 7:0] address,
 input logic        write_enable, read_enable
);

  logic [63:0] mem [0:255];

  always_latch   // asynchronous write (latch behavior)
    if (write_enable) mem[address] <= data;  // write to RAM

  assign data = read_enable? mem[address] : 64'bz;
endmodule

module test;
  wire  [63:0] data;
  logic [ 7:0] address;
  logic        write_enable, read_enable;

  RAM ram1 (.*); // instance or RAM model

  initial begin
   $readmemh("ram_data.dat", ram1.mem);  // GOTCHA!
   ...
```

In this example, the RAM model is correct—at least functionally. The problem is that the **always_latch** procedural block enforces a synthesis rule that multiple procedural blocks cannot write to the same variable. The testbench is attempting to load the RAM model using the Verilog `$readmemh` task, which is a common way to load Verilog memory models. This is a second procedural block writing to the RAM storage (mem), which is illegal. One simulator generates the following error:

```
Error-[ICPD] Invalid combination of procedural drivers
 Variable "mem" is driven by an invalid combination of procedural
  drivers. Variables written on left-hand of "always_latch" cannot
  be written to by any other processes, including other
  "always_latch" processes.
```

Some simulators, however, execute the example above without any errors or warnings. These products do not treat the `$readmemh` and `$readmemb` commands as an assignment.

How to avoid this Gotcha

The fix for this coding problem is to use Verilog's general purpose **always** procedural block for this abstract RAM model. SystemVerilog's **always_latch** and **always_ff** procedural blocks are intended to model synthesizable RTL models. These constructs are not intended for abstract models that do not need to adhere to synthesis coding rules.

The asynchronous, latch-like RAM model in the previous example should be coded as:

```
module RAM
  ...
  always @*     // asynchronous write (latch behavior)
    if (write_enable) mem[address] <= data;  // write to RAM
  ...
```

Gotcha 99: Non-standard language extensions

Gotcha: My SystemVerilog code only works on one vendor's tools.

Synopsis: Some tools add proprietary extensions to the IEEE Verilog and SystemVerilog standards.

Some Verilog/SystemVerilog tool vendors extend the IEEE standard by adding special, vendor-specific features to their product. These extensions can be useful for that vendor's tools. However, using these extensions also means the Verilog or SystemVerilog code will not work with tools from other vendors.

One SystemVerilog tool company allows an optional keyword **hard** to be used with the **solve...before** constraint operator. Without this additional keyword, that company's tools do not enforce the constraint solution order that is specified by **solve...before**. An example of using this vendor-specific keyword is:

```
constraint ab {
  solve a before b hard;     // 'hard' enforces solve before
  if (a inside {32, 64, 128, 256})
    a == b ;
  else
  a > b;
}
```

The keyword **hard** is *not* a SystemVerilog keyword, and is not in the IEEE 1800-2005 SystemVerilog standard. If **hard** is used with any tool other than that vendor's tool, a syntax error will result. *Gotcha!*

Another SystemVerilog tool vendor allows the keyword pair **pure virtual** to be used in the declaration of class methods. This keyword pair is not permitted in the official IEEE 1800-2005 SystemVerilog standard. Testbenches written with this keyword pair might not compile in other tools. This vendor also supplies verification libraries that contain this keyword. These libraries might not work with tools from other vendors.

How to avoid this Gotcha

Using non-standard keywords or syntax might be necessary to get the desired results in a specific product. However, specifying this keyword will prevent the same verification code from working with other software tools. To avoid this gotcha, conditional compilation can be used to control whether or not the vendor-specific construct is compiled. For example:

```
constraint ab {
  `ifdef VENDOR_A
    solve a before b hard; // add proprietary 'hard' specification
  `else
    solve a before b;
  `endif
  if (a inside {32, 64, 128, 256})
    a == b ;
  else
  a > b;
}
```

In the example above, the macro name VENDOR_A must be set before the code is compiled. It can be specified in the source code, or on the tool's command line.

It should be noted that the **pure virtual** keyword pair is illegal in the SystemVerilog-2005 standard, but a proposal has been approved by the IEEE SystemVerilog standard group to add this to the next version of the IEEE standard. At the time this book was written, there was no proposal to add the **hard** keyword.

The two non-standard extensions shown above illustrate this type of gotcha. They are not the only non-standard extensions that exist in Verilog and SystemVerilog tools.

Gotcha 100: Array literals versus concatenations

Gotcha: Some tools require one syntax for array literals. Other tools require a different syntax.

Synopsis: Array literals and structure literals are enclosed between the tokens '{ and }, but an early draft of the SystemVerilog standard used the tokens { and }, without the apostrophe.

The Verilog concatenation operator joins one or more values and signals into a single vector. SystemVerilog array and structure literals (also known as an *assignment patterns*) are lists of one or more individual values. To make the difference between these constructs obvious to both engineers and software tools, the syntax for an array or structure literal ('{ }) is different from the syntax for a Verilog concatenation({ }) . The difference is that the array or structure literal list of separate values is preceded by an apostrophe

```
logic [7:0] data;        // 8-bit vector
data = {4'hF, bus};      // concatenate values into a vector


int data [4];              // array of 4 integers
data = '{0, 1, 2, 3};        // list of separate values


typedef struct {
  int a, b;
  logic [3:0] opcode;
} instruction_word;
instruction_word = '{7, 5, 3};     // list of separate values
```

The similarity of these two constructs can be a gotcha. It is easy to forget to add the apostrophe before the array or structure literal, turning the list of values into a concatenation. In most contexts, this mistake will be a syntax error, and will not lead to a functional gotcha. There is one exception, though, which is described in Gotcha 17 on page 38.

Another gotcha is that an unofficial preliminary draft of the SystemVerilog standard, known as SystemVerilog 3.1a[1], used the tokens { } for both concatenations and array/structure literals.

1. *SystemVerilog 3.1a Language Reference Manual: Accellera's Extensions to Verilog*, Copyright 2004 by Accellera Organization, Inc., Napa, CA, http://www.eda.org/sv/SystemVerilog_3.1a.pdf.

At the time this book was written, some software tools required the preliminary SystemVerilog 3.1a syntax, some tools required the official IEEE 1800 syntax, and some tools allowed either syntax. *Gotcha!*

How to avoid this Gotcha

The gotcha of some tools requiring a non-standard syntax cannot be avoided. A workaround is to use conditional compilation around statements containing array or structure literals, to allow the model to be compiled with either the preliminary SystemVerilog 3.1a syntax or the official IEEE 1800 syntax. For example:

```
int data [4];              // array of 4 integers

initial begin
  `ifdef VENDOR_A
    data = '{0, 1, 2, 3};   // IEEE 1800 list of values
  `else
    data = {0, 1, 2, 3};    // old SV 3.1a list of values
  `endif
  ...
end
```

Gotcha 101: Module ports that pass floating point values (real types)

Gotcha: Some SystemVerilog tools allow me to declare my input ports as real (floating point), but other tools do not.

Synopsis: Module output ports that pass floating point values are declared as real, but module input ports that pass floating point values are declared as var real.

SystemVerilog allows floating point values to be passed through ports. However, the official IEEE syntax is not intuitive. An **output** port of a module can be declared as a **real** (double precision) or **shortreal** (single precision) type, but **input** ports must be declared with the keyword pair **var real** or **var shortreal**. For example:

```
module fp_adder (output real      result,
                 input  var real  a, b
                );
  ...
endmodule
```

An unofficial preliminary draft of the proposed SystemVerilog standard, known as SystemVerilog 3.1a[1], did not require the **var** keyword be used on **input** floating point ports. At the time this book was written, some SystemVerilog tools require the official IEEE syntax, as shown above, and get an error if the **var** keyword is omitted. Other tools, however, require the preliminary SystemVerilog 3.1a syntax, and get an error if the **var** keyword is used. Designers are forced to write two versions of any models that have floating point input ports. *Gotcha!*

How to avoid this Gotcha

This gotcha cannot be avoided. The only workaround is to use conditional compilation around the module port declarations, to allow the same model to be compiled with either the unofficial SystemVerilog 3.1a declaration style or with the official IEEE 1800 declaration style.

1. *SystemVerilog 3.1a Language Reference Manual: Accellera's Extensions to Verilog*, Copyright 2004 by Accellera Organization, Inc., Napa, CA, http://www.eda.org/sv/SystemVerilog_3.1a.pdf.

Index

H

I

Z