Standard Gotchas
Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know

Stuart Sutherland
Sutherland HDL, Inc.
stuart@sutherland-hdl.com

Don Mills
Microchip Technology
don.mills@microchip.com

ABSTRACT

A definition of "*gotcha*" is: "A misfeature of....a programming language...that tends to breed bugs or mistakes because it is both enticingly easy to invoke and completely unexpected and/or unreasonable in its outcome. A classic gotcha in C is the fact that '`if (a=b) {code;}`' is syntactically valid and sometimes even correct. It puts the value of `b` into `a` and then executes code if `a` is non-zero. What the programmer probably meant was '`if (a==b) {code;}`', which executes code if `a` and `b` are equal." (from *http://www.hyperdictionary.com/computing/gotcha*)

The Verilog and SystemVerilog standards define hundreds of subtle rules on how software tools should interpret design and testbench code. These subtle rules are documented in the IEEE Verilog and SystemVerilog Language Reference Manuals...all 1,500 plus pages! The goal of this paper is to reveal many of the mysteries of Verilog and SystemVerilog, and help engineers understand many of the important underlying rules of the Verilog and SystemVerilog languages. Dozens of gotchas in the standards are explained, along with tips on how to avoid these gotchas.

Table of Contents

1.0 Introduction 3
2.0 Declaration gotchas 4
- 2.1 Case sensitivity 4
- 2.2 Implicit net declarations 5
- 2.3 Escaped identifiers in hierarchy paths 6
- 2.4 Verification of dynamic data 7
- 2.5 Variables declared in unnamed blocks 8
- 2.6 Hierarchical references to declarations imported from packages 8
- 2.7 Variables with no hierarchy path are not dumped to VCD files 9
- 2.8 Shared variables in modules 9

2.9 Shared variables in interfaces, packages, and $unit 10
2.10 Shared variables in tasks and functions 10
2.11 Importing enumerated types from packages 11
2.12 Importing from multiple packages 11
3.0 Two-state gotchas 12
3.1 Resetting 2-state models 12
3.2 Locked state machines 13
3.3 Hidden design problems 15
3.4 Out-of-bounds indication lost 16
4.0 Literal number gotchas 16
4.1 Signed versus unsigned literal integers 17
4.2 Default base of literal integers 17
4.3 Size mismatch in literal integers 18
4.4 Literal number size mismatch in assignments 20
4.5 Literal number Z and X extension backward compatibility 21
4.6 Filling vectors 22
4.7 Passing real (floating point) numbers through ports 22
4.8 Port connection rules 23
4.9 Back-driven input ports 26
5.0 Operator gotchas 27
5.1 Self-determined operations versus context-determined operations 27
5.2 Operation size and sign extension in assignment statements 30
5.3 Signed arithmetic 32
5.4 Bit select and part select operations 34
5.5 Increment, decrement and assignment operations 35
5.6 Pre-increment versus post-increment operations 35
5.7 Operations that modify the same variable multiple times in an assignment 36
5.8 Operator evaluation short circuiting 37
6.0 Programming gotchas 38
6.1 Assignments in expressions 38
6.2 Procedural block activation 38
6.3 Combinational logic sensitivity lists 40
6.4 Arrays in sensitivity lists 41
6.5 Vectors in sequential logic sensitivity lists 42
6.6 Operations in sensitivity lists 43
6.7 Sequential blocks with `begin`...`end` groups 44
6.8 Sequential blocks with partial resets 45
6.9 Blocking assignments in sequential procedural blocks 45
6.10 Evaluation of true/false on 4-state values 46
6.11 Mixing up the not operator (!) and invert operator (~) 47
6.12 Nested if...else blocks 47
6.13 Casez/casex masks in case expressions 48
6.14 Incomplete or redundant decisions 49
6.15 Out-of-bounds assignments to enumerated types 50
6.16 Statements that hide design problems 51
6.17 Simulation versus synthesis mismatches 53

7.0 Testbench gotchas 53
7.1 Multiple levels of the same virtual method 53
7.2 Event trigger race conditions 54
7.3 Using semaphores for synchronization 56
7.4 Using mailboxes for synchronization 58
7.5 Coverage reporting 59
7.6 $unit declarations 60
7.7 Compiling $unit 61
8.0 References 62
9.0 About the authors 62

1.0 Introduction

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 (a=b)     /* GOTCHA! assigns b to a, then if a is non-zero sets match */
  match = 1;
else
  match = 0;
```

Most likely, what the programmer intended to code is `if (a==b)` instead of `if (a=b)`. The results are very different!

Why do programming languages allow engineers to make these stupid mistakes? In the case of the Verilog and SystemVerilog languages, the primary reasons are:

- A syntax may be desirable in some contexts, but a gotcha if used incorrectly.
- An underlying philosophy of Verilog and SystemVerilog is that the language should allow proving both what will work correctly in hardware, and what will not work in hardware.
- Verilog and SystemVerilog provide some freedom in how the language is implemented. This is necessary because tools such as simulation, synthesis and formal analysis work differently.
- Verilog and SystemVerilog simulation event scheduling allows tools to optimize order of concurrent events differently, which can lead to race conditions in a poorly written model.
- Some tools are not 100% standards compliant. This is not a gotcha in the standards, but it is still a source of gotchas when simulation or synthesis results are not what was expected.

The C example above, `if (a=b)`, is an example of a syntax that is a gotcha in one context, but is useful in a different context. The following example is similar, in that it contains an assignment operation within a true/false conditional test. In this example, however, there is no gotcha. The `while` loop repeatedly assigns the value of `b` to `a` and does some processing on `a`. The loop continues to run until the value of `a` is 0;

```
while (a=b) {    /* assign b to a; exit loop when a is 0 */
  ...
}
```

Does this same C gotcha exist in Verilog and SystemVerilog? If you don't know the answer, then you need to read this paper! (You will find the answer in Section 6.1.)

2.0 Declaration gotchas

2.1 Case sensitivity

Gotcha: *Verilog is a case-sensitive language, whereas VHDL is a case-insensitive language.*

Verilog is a case sensitive language, meaning that lowercase letters and uppercase letters are perceived as different in identifiers and in keywords.An *identifier* in Verilog and SystemVerilog is the user-specified 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. This case sensitivity is often a gotcha to engineers learning Verilog, especially those migrating from a case insensitive language such as VHDL. Even experienced engineers occasionally get caught making a case sensitivity error.

The gotcha of a case sensitive language is when an identifier is sometimes inadvertently spelled with all lowercase characters, and in another place with some or all uppercase characters. For example:

```
enum logic [1:0] {WAIT, LOAD, READY} State, NextState; // mixed case names

always_comb begin
  case (state)
    WAIT: NextState = L0AD;
    ...
  endcase
end
```

There are three gotchas in the preceding example.

One gotcha is that the enumerated variable `State` is declared using a mix of uppercase and lowercase characters. Later in the code, and, in a large design, possibly hundreds of lines after the declaration, a signal called `state` is referenced. These identifiers read the same in English, but, to a Verilog compiler, they are read as very different names.

A second gotcha is that the enumerated label `LOAD` is in all uppercase letters. But later in the code an identifier called `L0AD` is referenced. Visually, these identifiers appear to be the same, but to a Verilog compiler 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.

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 keyword `wait`. A Verilog compiler won’t get confused between the keyword `wait` and the identifier `WAIT`, but someone who has to read or maintain the code could easily confuse `wait` and `WAIT`. (Several examples in this paper use a state machine with a “`WAITE`” state. The identifier “`WAITE`” is purposely spelled with the letter “`E`” on the end to avoid confusion with the Verilog keyword `wait`.)

How to avoid this gotcha: The best way to avoid this case-sensitive gotcha is to adopt good naming conventions within a company, and then strictly enforce these guidelines. When modules are instantiated, SystemVerilog provides two convenient shortcuts that also help enforce following consistent naming conventions. The shortcuts are the `.<name>` and `.*` implicit port connections. These shortcuts infer netlist connections, but only if the names within a design block and the names within the netlist are consistent.

2.2 Implicit net declarations

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

The examples in Section 2.1, above, contained references to the undeclared identifiers, `state` and `L0AD`. What does a compiler 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, then a compilation or elaboration error occurs.
- If an undeclared identifier is used on the right-hand side of a continuous assignment, then a compilation or elaboration error occurs.
- 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.*
- 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 two rules above are gotchas, as is illustrated in the following example:

```
module bad_adder (input  wire a, b, ci,
                  output wire sum, co);

  wire n1, n2, n3;

  xor g1 (n1, a, b);
  xor g2 (sum, nl, ci);     // GOTCHA!
  and g3 (n2, a, b, c);     // GOTCHA!
  and g4 (n3, n1, ci);
  or  g5 (co, n2, n3);
endmodule
```

One gotcha in this example is the declaration of `n1` (“en-one”) but the usage of `nl` (“en-ell”) in the `g2` primitive instance. Another gotcha is an extra identifier, `c`, in the second `g3` primitive 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!*

This example comes from a debugging lab in Sutherland HDL’s Verilog training course (but without the GOTCHA comments). It is surprising how difficult the two typos in the netlist can be to find. Imagine how much more difficult a simple typo would be to find in a one-million gate netlist with several layers of design hierarchy. Why does Verilog allow this gotcha? Because, like many gotchas, the ability to have implicit data types automatically inferred can be useful, when not abused. 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: There are two ways to avoid this implicit data type gotcha. Verilog provides a `‘default_nettype none` compiler directive. When this directive is set, implicit data types are disabled, which will make any undeclared signal name a syntax error. A limitation of this directive is that the benefits of implicit data types are also lost. Another limitation is that compiler directives are not bound by design blocks, or even by source code files. If the `‘default_nettype none` directive is turned on in one file, it can affect the compilation of other files, which is yet another *GOTCHA!*. To avoid this gotcha, the directive `‘default_nettype wire` should be added at the end of each file where implicit nets have been turned off.

```
‘default_nettype none   // turn off implicit data types
module adder (input  wire a, b, ci,
              output wire sum, co);

  wire n1, n2, n3;

  xor g1 (n1, a, b);
  xor g2 (sum, nl, ci);     // ERROR! nl is not declared
  and g3 (n2, a, b, c);     // ERROR! c is not declared
  and g4 (n3, n1, ci);
  or  g5 (co, n2, n3);
endmodule
‘default_nettype wire  // turn implicit nets on again to avoid side-effects
```

SystemVerilog provides two convenient short cuts for connecting nets to module instances, `.<name>` and `.*`. These shortcuts remove the repetition in Verilog named port connections. By reducing the number of times a signal name must be typed, the possibility of typographical errors is also reduced. The `.<name>` and `.*` shortcuts also require that all nets be explicitly declared. The shortcuts will not infer an implicit data type due to a typo. Another advantage of these SystemVerilog shortcuts is that they are local to the module in which they are used. The shortcuts do not affect other design blocks, the way compiler directives can.

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

module top;
  wire a, b, ci;
  wire s1, s2, s3, c1, c2, c3;

  adder i1 (.a(a), .b(b), .ci(ci), .sum(s1), .co(c1) ); // Verilog style

  adder i2 (.a, .b, .ci, .sum(s2), .co(c2) ); // SystemVerilog .name style

  adder i3 (.*, .sum(s3), .co(c3) );          // SystemVerilog .* style
endmodule
```

2.3 Escaped identifiers in hierarchy paths

Gotcha: *Escaped identifiers in a hierarchy 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 white space character. A white space character is a space, a tab, a return, a form feed, or an end-of-file. Some examples of escaped identifiers are:

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

Note in the above example that a white space character must be used before the commas that separate an escaped identifier from the next item in the list. A white space is also required between the last escaped name, `\reset-`, and the closing parenthesis.

The gotcha is when an escaped identifier is used as part of a hierarchy path. The escaped identifier must be terminated by a white space. That white space looks like it breaks the hierarchy path into two identifiers, but the terminating white space is ignored, which, in effect, concatenates the two names into one name. The following examples illustrate the use of white space after references to escaped identifiers. Module `chip` uses named port connections to escaped port names identifiers. The $display contains a relative hierarchy path that contains an escaped identifier.

```
module chip (output [7:0] q, input [7:0] d, input clk, rstN);

  \d-flop \d-0 (.q(q[0]), .\q~ (), .\d[0] (d[0]), .clk(clk), .\rst- (rstN));
          escaped brackets (part of name)        non-escaped brackets (bit-select)

  initial begin
    $display("d = %b", \d-0.\d[0] );   // GOTCHA! missing white space

    $display("d = %b", \d-0 .\d[0] );  // OK: white space in path
  end                                  //   required; does not split
                                       //   path into two names
endmodule
```

How to avoid this gotcha: One could simply not used escaped names in a design, especially an escaped name with square brackets in the name, as in the example above. Unfortunately, life is not that simple. Not all identifiers are user-defined. Software tools, such as the DC synthesis compiler, 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. The gotcha of having to reference escaped identifiers using hierarchy paths is one that cannot be completely avoided. Engineers need to know that having a space in a hierarchy path involving escaped identifiers is not illegal. It may make the code harder to read, but it is the way Verilog works.

2.4 Verification of dynamic data

Gotcha: *Dynamically allocated variables have no hierarchy paths.*

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 dynamic storage to Verilog, including classes for Object Oriented programming, dynamically sized arrays, queues, automatic variables in static tasks and functions, and local variables in an assertion. These dynamically allocated types are intended for—and are very important in—modeling test programs using modern verification methodologies.

But, there is a gotcha with these new dynamic data types. Unlike other Verilog data types, dynamic types cannot be referenced hierarchically. Hierarchically referencing nets and variables in a design is a common verification technique. It allows the verification code to evaluate, and possibly stimulate, the logic deep down in the hierarchy of a design, without having to pull those internal signals up to the testbench through extra, verification only, module ports.

The reason that dynamic data cannot be referenced hierarchically is that hierarchy paths are static in nature, whereas a dynamic variable or class object comes and goes during simulation. This gotcha places a burden on the verification engineer. Before using a hierarchical to reference a variable, the verification engineer must first examine whether or not the variable is static or dynamic. Since SystemVerilog adds so many types of dynamic storage, it can be very difficult to determine when something can be referenced hierarchically, and when it cannot.

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

2.5 Variables declared in unnamed blocks

Gotcha: *Variables declared in an unnamed scope have no hierarchy paths.*

SystemVerilog allows local variables to be declared in unnamed scopes, including:

- Within the definition of `for` loops.
- Unnamed `begin`...`end` and `fork`...`join` blocks.
- Unnamed loops in generate blocks

These local variables have two primary advantages. First, variables can be declared where they are needed, instead of in the midst of writing code, having to jump back to the beginning of a module to add a new declaration. This allows engineers to keep their train of thought as they are developing code. Second, local variables prevent the inadvertent Verilog gotcha of having multiple `initial` or `always` procedural blocks write to the same variable (this gotcha is discussed in Section 2.8).

There is a gotcha with locally defined variables. Variables in a `for` loop declaration, in an unnamed block, or in an unnamed generate scope, cannot be referenced hierarchically, because there is no named scope to reference in the hierarchy path. Once again, the verification engineer has the burden of determining which data can be referenced hierarchically, and which data cannot.

How to avoid this gotcha: Declare local variables in named `begin`...`end` or `fork`...`join` blocks, so that they can be referenced hierarchically for verification purposes. The use of variables declared as part of a `for` loop definition should not be a problem, as these loop control variables seldom need to be accessed from the verification test programs.

2.6 Hierarchical references to declarations imported from packages

Gotcha: *Imported identifiers cannot be referenced hierarchically.*

Hierarchy paths are a verification construct, used to access the declaration of an object in the scope in which that object is declared. When package items are imported into a module, interface or test program, these items are *not* locally defined within that module, interface or test program. This means these imported items cannot be referenced hierarchically. In the example below, the hierarchy path to `chip.RESET` is an error, because `ERROR` was not declared in module `chip`:

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

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

program test (...);
  ...
  $display ("the value of RESET is %b", top.chip.RESET);  // GOTCHA!
```

How to avoid this gotcha: The correct way to reference package items is using the scope resolution operator (`::`) instead of hierarchy paths. For example:

```
$display ("the value of RESET is %b", chip_types::RESET);
```

2.7 Variables with no hierarchy path are not dumped to VCD files

Gotcha: *Changes on dynamic variables and variables in unnamed scopes are not saved in VCD files.*

Another gotcha with dynamically and locally defined storage is that the IEEE Verilog and SystemVerilog standards explicitly state that this type of storage is not dumped out to a Value Change Dump (VCD) file. VCD files are used as an input to waveform displays and other design analysis tools, in order to analyze what activity has, or has not, occurred during simulation. As a standard, VCD files are portable, and can be used with many different third party and in-house tools. However, changes on dynamically allocated and local data are not dumped to VCD files, which means waveform displays and design analysis tools that read in VCD files do not see all the activity that took place in simulation.

How to avoid this gotcha: There is no work around for this VCD limitation. It should be noted, though, that proprietary dump files that are part of most waveform display tools might not have this limitation.

2.8 Shared variables in modules

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

Syntactically, Verilog 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 block. 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 gotcha, where the variable `i` is shared by two concurrent `always` procedural blocks.

```
module chip (...);
  integer i;    // for-loop control variable

  always @(a or b) begin
    for (i=0; i<15; i=i+1)     // this process uses i
      ...
  end

  always @(c or d) begin
    for (i=0; i<15; i=i+1)     // this process also uses i
      ...
  end
endmodule
```

How to avoid this gotcha: The answer on how to avoid this gotcha depends on whether the code is for a synthesizable RTL model or for verification. For RTL models, the preferred way to avoid this gotcha is to use SystemVerilog's `always_comb`, `always_ff`, `always_comb`, and continuous `assign` to a variable. These processes make it a syntax error if a variable is 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 process synchronization (flags, event triggers, semaphores or mailboxes) so that concurrent processes are not writing to the same variable at the same time.

For the example above, there is another way to avoid this shared variable gotcha. SystemVerilog allows local variables to be declared as part of a `for` loop definition. This solution has its own gotcha, however, as was discussed in Section 2.5.

2.9 Shared variables in interfaces, packages, and $unit

Gotcha: *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 Section 2.8 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 built-in package), and interfaces. These externally declared variables can then be referenced by multiple modules, creating a shared variable. Any `initial` and `always` procedural blocks that write to these 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.

How to avoid this gotcha: For RTL models, SystemVerilog's `always_comb`, `always_ff`, `always_comb`, and continuous `assign` to variables, should be used. These processes make it a syntax error if a variable is written to by multiple processes, even when these processes are in different modules. For verification, or abstract bus functional models, this gotcha can be avoided through the use of process synchronization (flags, event triggers, semaphores or mailboxes).

2.10 Shared variables in tasks and functions

Gotcha: *Variables in tasks and functions contained in interface, package and global variables create shared resource conflicts.*

SystemVerilog allows tasks and functions to be defined in `$unit`, packages, and interfaces. These tasks and functions can then be called from multiple modules. Any static storage in these tasks and functions is shared by all places from which the task or function is called.

How to avoid this gotcha: Declare tasks and functions in `$unit`, packages, or interfaces as `automatic`, with no static storage within the task or function. Each call to an automatic task or function will allocate unique storage for that call, instead of sharing the task/function storage. An exception to this guideline is a task or function within an interface that is only called from within the interface.

2.11 Importing enumerated types from packages

Gotcha: *Importing an enumerated type does not import the 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, WAITE, LOAD, READY} states_t;
endpackage

module chip (...);
  import chip_types::states_t;   // explicit import of states_t type
  states_t state, next_state;

  always_ff @(posedge clock, posedge reset)
    if (reset) state <= RESET;       // GOTCHA: RESET has not been imported
    else       state <= next_state;
  ...
endmodule
```

How to avoid this gotcha: This gotcha can usually be avoided by wildcard importing the package. In this way, both the enumerated type definition and the enumerated labels are imported.

```
module chip (...);
  import chip_types::*;   // wildcard import of package declarations
```

However, wildcard imports have a gotcha if multiple packages are used in a design block, as discussed in Section 2.12, which follows.

2.12 Importing from multiple packages

Gotcha: *Wildcard imports from multiple packages can cause name collisions.*

Large designs, and designs that use IP models, will likely divide declarations into multiple packages. A common modeling style is to wildcard import these packages into a design block, or into the `$unit` global declaration space. Wildcard imports are convenient, as they can save considerable typing over explicitly importing each item from each package. However, wildcard imports of multiple packages can lead to a gotcha, as illustrated in the following example:

```
package bus1_types;
  parameter SIZE = 32;
  ...
endpackage
```

```
package bus2_types;
  parameter SIZE = 64;
  ...
endpackage

module chip (...);
  import bus1_types::*;   // wildcard import of a package
  import bus2_types::*;   // wildcard import of another package
  logic [SIZE-1:0] a;     // GOTCHA! SIZE has more than one definition
  ...
```

How to avoid this gotcha: The gotcha with wildcard package imports occurs when there are some identifiers common to more than one package. In this case, at most only one of the packages with duplicate identifiers can be wildcard imported. Any references to a duplicate identifier in another package must be explicitly referenced, using its package name each place the identifier is referenced. For example:

```
  import bus1_types::*;           // wildcard import of a package
  logic [bus2_types::SIZE-1:0] a; // explicit reference to different package
```

3.0 Two-state gotchas

3.1 Resetting 2-state models

Gotcha: *Reset fails to occur the first time.*

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;
  bit rst_n, clk;     // 2-state types for reset and clock
  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 2 nanoseconds
  end
  chip u1(.rst_n, .clk, ...);    // instance of design under test
endmodule: chip_tb

module chip (input rst_n, clk, ...);
  enum {WAITE, LOAD, STORE} State, NextState;  // 2-state enum. variables
  always_ff @(posedge clk, negedge rst_n)      // active-low async reset
```

```
    if (!rst_n) State <= WAITE;
    else        State <= NextState;
  ...
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 reset to active value using nonblocking assign
  #3ns rst_n = 1;
  ...
```

A second way to avoid this gotcha is to ensure that the reset is held longer then a clock period, thus using the clock to trigger the `always_ff` block. This is not a recommended approach, due to reset acting like a synchronous reset, rather than an asynchronous reset, which means that the RTL simulation does not match the gate-level simulation or the design intent.

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 zero transition, which is a negative edge.

Note: VCS version 2006.06 does not fully adhere to the IEEE SystemVerilog standard for 2-state types. The standard says that 2-state types should begin simulation with a value of zero. In VCS, 2-state variable types automatically transition to zero sometime during simulation time zero. This will cause a negative edge transition at simulation time zero, which *might*, depending on event ordering, trigger the design procedural blocks that are looking for a negative edge transition. This tool-specific, non-standard behavior can hide this 2-state reset gotcha. Engineers should not rely on this tool-specific behavior as a solution to the gotcha. It is dependent on event ordering, and, at some future time, VCS could implement the 2-state behavior as it is specified in the standard.

3.2 Locked state machines

Gotcha: *2-state state machines can lock up in the start-up state.*

By default, enumerated types are 2-state types, and the default value of the first label in the 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  wire    clock, resetN);

  enum {WAITE, LOAD, STORE} State, NextState;  // 2-state enum. variables

  always_ff @(posedge clock, negedge resetN)   // state sequencer
    if (!resetN) State <= WAITE;
    else State <= NextState;

  always @(State) begin                        // next state decoder
    unique case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end
  ...
endmodule
```

In simulation, this example will lock up in the `WAITE` state. Applying reset, whether 2-state or 4-state, will not get the state machine out of this lock up. This is because `State` and `NextState` are 2-state enumerated variables. 2-state types begin simulation with a value of zero, which is the value of `WAITE` in the enumerated list. When the `always_ff` state sequencer is reset, it will assign `State` the value of `WAITE`, which is the same value as the current value of `State`, and thus does not cause a transition on `State`. Since `State` does not change, the `always @(State)` combinational procedural block does not trigger. Since the combinational block is not entered, `NextState` is not updated to a new value, and remains its initial value of `WAITE`. On a positive edge of clock, `State` is assigned the value of `NextState`, but, since the two variables have the same value of `WAITE`, `State` does not change, and, once again, the `always @(State)` combinational block is not triggered and `NextState` 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, `NextState` 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 `WAITE` state.

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

```
enum logic [1:0] {WAITE, LOAD, STORE} State, NextState;  // 4-state
```

By doing this, `State` and `NextState` will begin simulation with the value of X. When `State` is assigned `WAITE` during reset, the `always @(State)` will trigger, setting `NextState` to `LOAD`.

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

```
enum bit [2:0] {WAITE = 3'b001,
                LOAD  = 3'b010,
                STORE = 3'b100} State, NextState;  // 2-state variables
```

In this example, `State` and `NextState` 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` will be assigned `WAITE`. The change on `State` will trigger the `always @(State)` combinational block, which will update `NextState` to `LOAD`, preventing the lock up gotcha.

3.3 Hidden design problems

Gotcha: *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 zero instead of X. It is common for a value of zero 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 resetN)   // data register
  if (!resetN) 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 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!*

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.

3.4 Out-of-bounds indication lost

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

Another type of failure that can be hidden by 2-state types is when an out-of-bounds address is read from a memory device. An example where this can occur is:

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

  bit [D_WIDTH-1:0] mem_array [0:SIZE-1];  // only needs 10 bit index

  assign data_out = read? mem_array[addr] : 'z; // read logic
  ...
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: One way to avoid this gotcha is to use 4-state types for arrays. 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.

4.0 Literal number gotchas

There are several gotchas associated with defining literal integer numbers in Verilog and SystemVerilog. There are actually two different sets of rules involving literal integers: one that defines the generation of an integer number, and another for assigning the number. Many engineers have only a vague understanding of these two sets of rules, and often confuse them with each other. This section will discuss the gotchas on the generation a literal integer. Section 4.4 discusses the issues regarding the assignment of integers.

Review of literal integer syntax. Literal integers 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 value. If not given, the default size is 32 bits (the Verilog standard says "at least" 32 bits, but all implementations the authors are aware of use exactly 32 bits).
- `s` is optional. If given, it specifies that the literal integer should be treated as a 2's complemented signed value. If not given, the default is unsigned.
- `<base>` is required, and specifies whether the value is in binary, octal, decimal, or hex.
- `<value>` is required, and specifies the literal integer value.

4.1 Signed versus unsigned literal integers

Gotcha: *Some literal integers default to signed; other literal integers default to unsigned.*

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`). The `s` specifier was added as part of the Verilog-2001 standard to allow the specification of sized, signed literal values.

The difference in the default signed-ness of a value can catch engineers by surprise. Signed-ness of a value affects several types of operations. Unexpected operation results will occur if an engineer forgets—or is not aware of—the different default signed-ness of a simple literal integer versus a based literal integer number.

Are the following two signed counter statements the same? (Of course not, or we wouldn't have included this example in the paper!)

```
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 (literal 1'b1 is unsigned)

end
```

How to avoid this gotcha: The difference between adding `1` and `1'b1` is that the literal integer `1` is a signed value, whereas the literal integer `1'b1` is an unsigned value. This difference affects how the ADD operation is performed. Signed arithmetic is discussed in more detail in Section 5.3.

4.2 Default base of literal integers

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

A simple literal integer (e.g. `5`) defaults to a decimal base. To use a binary, octal or hex value, a based-literal number must be specified (e.g. `'h5`). The base options are represented using `b`, `o`, `d`, or `h` for binary, octal, decimal and hex, respectively. The base specifier can be either lower case or upper case (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—or is not aware—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 well known 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 example above may look reasonable, and it is syntactically correct. But, since the default base of a simple integer is decimal, the values "`10`" and "`11`" are ten and eleven, respectively. The size of the values is not specified, and so defaults to 32-bits wide. The 2-bit `select` signal will be zero-extended to 32-bits wide, and then compared to zero, one, ten and eleven. The only values of `select` that will ever match are a and b, which extend to 32-bit `00...00` and `00...01`, respectively. The extended value of `select` will never match the decimal "`10`" and "`11`". Therefore, the `c` and `d` inputs of the multiplexer will never be selected. Since 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: An easy way to avoid this gotcha—or more correctly, to detect this gotcha—is to use the SystemVerilog `unique` modifier to the `case` statement, as in:

```
unique case (select)        // intent is for a 4-to-1 MUX behavior
```

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 a simulation error, rather than a functional bug in the code.

Code coverage utilities and/or SystemVerilog functional coverage can also be used to detect that some branches of the case statement are not being executed. Also, some EDA tools, such as LEDA and DC, will warn about a mismatch in the size of select and the literal integer values to which it is compared.

4.3 Size mismatch in literal integers

Gotcha: *Too small a size truncates most-significant bits; too large a size left-extends with...something.*

Size smaller than value gotcha. Verilog rules state that if the bit size specified is fewer bits than the value, then, no matter what, 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. This can significantly affect the result of signed operations! In the following example, a negative `15` (8-bit `F1` hex) is truncated to 2-bits wide, becoming a positive `1`.

```
-2'sd15;   // 11110001 is truncated to 01
```

How to avoid this gotcha: The way to avoid this gotcha is to be sure that the bit size specified is at

least as wide as the size of the value, especially when using signed values. Some tools, such as rule checking programs like LEDA, will detect a size versus value mismatch. Unfortunately, these tools will also report size mismatches when using unsigned values, where a mismatch may not be a problem, and may even be useful.

Size greater than value gotcha. Verilog rules state that when the bit size specified is more bits than the value, then the value will be expanded to the size by left-extending the value. The fill value used to left extend is based on the most significant specified bit of the value, as follows:

- If the most significant bit of the value is a 0 or a 1, then the value is left-extended 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 very 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
```

A surprising gotcha can occur 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 [11:0] a, b, c;

initial begin
  a = 12'h3c;   // unsigned value 00111100 expands to 000000111100
  b = 12'sh3c;  // signed value   00111100 expands to 000000111100
  c = 12'so74;  // signed value     111100 expands to 000000111100
  if (a == b) ...;  // evaluates as true
  if (a == c) ...;  // evaluates as true
end
```

In this example, a hex `3c` is an 8-bit value, which 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 value is signed or unsigned. This is as expected.

The octal value `73` is the same bit pattern as a hex `3c`, but is a 6-bit value with its most-significant bit set. But, the expansion to the 12 bit size still zero-extends, rather than sign-extends. If sign extension was expected, then *GOTCHA!*.

How to avoid this 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.

```
12'h805      // expands to 100000000101, which is 2053 decimal
12'sh805     // expands to 100000000101, which is -2043 decimal
12'shFFB     // expands to 111111111011, which is -5 decimal
-12'sh5      // expands to 111111111011, which is -5 decimal
```

Unspecified size. If the size of a literal integer is not specified, then the bit size of the literal number defaults to 32 bits. This default size can be a gotcha when the `<value>` is not 32 bits, as discussed in the preceding paragraphs.

4.4 Literal number size mismatch in assignments

Gotcha: *A size mismatch in literal numbers follows different rules than a size mismatch in assignment statements.*

Once a literal integer has been created and expanded via the rules discussed in Section 4.3, the operator rules are then applied.

Note: This section discusses assignment of literal value gotchas (e.g. `a = 5;`) Section 5 discusses assignment rules and gotchas when there are operators on the right-hand side of the assignment.

The assignment operation rules for assigning literal values are:

- When the left-hand side expression of an assignment statement is fewer bits than the right-hand side literal value, 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 value, ...
  - If the right-hand side literal value is unsigned, it will be left-extended with zeros.
  - If the right-hand side literal value is signed, it will be left-extended using sign extension.
  - Exceptions: If the right-hand side is an unsized high-impedance literal value (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 Section 4.3, above, 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 value expansion does not sign-extend, but an assignment statement *does* sign-extend.

And now the gotcha. Sign extension of the right-hand side only occurs if the expression on the right-hand side is signed. The signed-ness 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;   // 1100 (right-hand side) is zero-extended to 00001100
s = 4'hC;   // 1100 (right-hand side) is zero-extended to 00001100
               // even though s is a signed variable; GOTCHA

u = 4'shC;  // 1100 (right-hand side) is sign-extended to 11111100
               // even though u is an unsigned variable; GOTCHA
s = 4'shC;  // 1100 (right-hand side) is sign-extended to 11111100

a = 4'hC;
u = a;      // 1100 (right-hand side) is zero-extended to 00001100
s = a;      // 1100 (right-hand side) is zero-extended to 00001100
               // even though s is a signed variable; GOTCHA

b = 4'hC;
u = b;      // 1100 (right-hand side) is sign-extended to 11111100
               // even though u is an unsigned variable; GOTCHA
s = b;      // 1100 (right-hand side) 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 signed-ness 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 Section 5.3.

4.5 Literal number Z and X extension backward compatibility

Gotcha: *Verilog-2001 Z and X extension is not backward compatible with Verilog-1995.*

Verilog-1995 had a gotcha when assigning an unsized high-impedance value (`'bz`) to a bus that is greater than 32 bits. The default size of an unsized number is 32 bits. The literal number expansion rules state that the Z value will extend through those 32 bits (see Section 4.3). But, the assignment extension rules state that this 32-bit value then zero-extends to the size of the left-hand side of the assignment (see Section 4.4). Therefore, with Verilog-1995, only the lower 32 bits would be set to high-impedance, and the upper bits would be set to 0. To set the entire bus to high-impedance requires explicitly specifying the number of high impedance bits. For example:

Verilog-1995:

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

data = 'bz;             // fills with 64'h00000000zzzzzzzz
data = 64'bz;           // fills with 64'hzzzzzzzzzzzzzzzz
```

The fill rules in Verilog-1995 make it difficult to write models that scale to new vector sizes. Redefinable parameters can be used to scale vector widths, but the Verilog source code must still be modified to alter the literal value widths used in assignment statements. (And yes, believe it or not, there are still some companies primarily using Verilog-1995.) The example above applies equally to assigning Xs to a vector.

How to avoid this gotcha: One solution to this gotcha (but not the best) came with Verilog-2001, which changed the rule for assignment expansion of unsized Z and X literal integers. The unsized value of Z or X will automatically expand to fill the full width of the vector on the left-hand side of the assignment.

Verilog-2001:

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

data = 'bz;             // fills with 64'hzzzzzzzzzzzzzzzz
```

The Verilog-2001 enhancement allows writing code that scales when vector sizes are redefined. But, this solution is also a gotcha, because it is not backward compatible with Verilog-1995. If

maintaining backward compatibility with Verilog-1995 is a requirement, then always specify the exact size of literal Z and X values.

How to avoid this gotcha: The best way to avoid these Z and X fill gotchas is to take advantage of a new SystemVerilog literal number, as described in Section 4.6, which follows.

4.6 Filling vectors

Gotcha: *Verilog does not have a literal value that fills all bits of a vector with ones.*

In Verilog-2001, 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.

Verilog-2001:

```
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
```

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. These tricks are not discussed in this paper, because there is a better way to avoid this gotcha...

How to avoid this gotcha: SystemVerilog comes to the rescue to fix this gotcha by defining a 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>`.

SystemVerilog:

```
parameter WIDTH = 64;
reg [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
```

4.7 Passing real (floating point) numbers through ports

Gotcha: *Verilog does not allow real numbers to be passed directly through ports.*

In Verilog, the outputs of a module can be either net or variable types, but the signals receiving values from module instantiations are required to be of type net. Verilog also requires that input ports of a module be net types. Verilog variables include the `real` type, which is a double-precision floating point value, but there is no counterpart to a real variable for the net data types. If outputs of a module can be variable types, but the receiving side must be a net type, how are real values passed through module ports?

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

engineers don't know, however, is that Verilog has a pair of built-in system functions that can be used 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 top;
  wire [63:0] net_real;                  // only net types used in netlist
  real_out ro (.net_real_out(net_real));
  real_in  ri (.net_real_in(net_real));
endmodule

module real_out(output wire [63:0] net_real_out);  // output is net type
  real r;
  assign net_real_out = $realtobits(r);   // must convert real to a
                                          //   bit-vector to pass through
                                          //   module output port
  ...
endmodule

module real_in(input wire [63:0] net_real_in);     // input is net type
  real r;
  always @(net_real_in)
    r = $bitstoreal(net_real_in);         // must use bit-vector on input
                                          //   port, then convert to real
  ...
endmodule
```

How to avoid this gotcha: While Verilog does provide a solution for passing real numbers through ports, the solution is not well known, and is awkward to use. SystemVerilog provides a much more elegant way to avoid this gotcha. In SystemVerilog, real types can be passed directly through ports SystemVerilog also allows both the sending and receiving side of ports to be real variables. The example above can be coded much more simply in SystemVerilog.

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

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

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

4.8 Port connection rules

Gotcha: *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 that one can avoid, 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. 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 type signals, and why the driver (or source) side of ports could be either net or variable. 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 continuous 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 has more bits than the port's receiving signal, then the upper bits of the driving signal are truncated, including any sign bit.
3. If the signal driving the port has fewer bits than the port's receiving signal, then the upper bits are extended, following Verilog's assignment rules:
   - If the driving signal is unsigned, the upper bits are zero-extended to the size of the receiving signal.
   - If the driving signal 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.

Most often, any mismatch in port connection sizes is a design error. But, it is not a syntax error. In Verilog, an incorrect size declaration is very easy design error 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.

Verilog's rules for port connection mismatches are well defined, but the simulation results are a gotcha! 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] address = 8'b11111111;   // decimal 255

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

module block1 (input [7:0] data,
               input [3:0] address,
               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 this code is:

```
data = 00001111
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.

Most simulators generate warnings when code like this is elaborated, but such warnings are not required by the Verilog standard, and engineers are notorious for ignoring these warnings.

How to avoid this gotcha: SystemVerilog provides a great solution to this gotcha: implicit port connections using either `.<name>` or `.*` module instantiations. When using the `.<name>` or the `.*` 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] address;

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

module block1 (input [7:0] data,
               input [3:0] address,
               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 .<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 `address`. Instead of a port connection mismatch (a gotcha), an elaboration error will

occur because the signal at the top level is a different size than the port in `block1`: The `.<name>` will not infer connections that do not match in size.

The `.<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 `.*` shortcut is the best solution. The `.*` shortcut requires explicitly listing all unconnected ports and all the signals with different sizes.

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

  block1 b1 (.*,                      // implicit port connection
             .address(address[3:0]), // explicit connection due to mismatch
             .byte_en() );           // explicit unconnected port
endmodule: top

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

4.9 Back-driven input ports

Gotcha: *Verilog allows input ports to be used as outputs.*

One of the surprising gotchas in Verilog 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 simply treats the input port as if it were a bidirectional inout port. The Verilog Language Reference Manual refers to this a *port coercion*. In this paper, we refer to an input port being used as an output as a *back-driven port*.

The following example illustrates this surprising gotcha:

```
module top;
  wire w1;
  backdrive bd(.a(w1));  // instance of a module with a back-driven port
endmodule

module backdrive(input wire a);   // 'a' will be coerced to be an output
  wire b;
  assign a = b;   // GOTCHA! the intent was to have b = a, not a = b;
endmodule
```

Even though signal `a` is an input to module `backdrive`, it can still be assigned a value within `backdrive`. This has the effect of `a` having two drivers wired together. This is legal because, in Verilog, ports are treated as continuous assignments. The code above is equivalent to:

```
module top;
  wire w1;
  wire a, b;
  assign a = w1;    // same as connecting 'w1' to input port 'a'
  assign a = b;     // a second continuous assignment to 'a'
endmodule
```

How to avoid this gotcha: SystemVerilog provides an easy solution to this gotcha. With SystemVerilog, input ports can be connected to variables, instead of the `wire` type. SystemVerilog restricts variables to only allow a single source. This is different than a `wire` type, where multiple drivers are permitted (whether intentional, or not). In the preceding example, if `a` had been declared as a `logic` type instead of a `wire` type, the assignment of `a = b` would be a syntax error, because it would represent a second source for `a`. The authors recommend that all module inputs and outputs be declared as variable types, unless it is intended to have multiple drivers on the port (e.g. a bidirectional data bus).

5.0 Operator gotchas

5.1 Self-determined operations versus context-determined operations

Gotcha: *Misunderstanding operator rules can lead to unexpected simulation results.*

What should happen if a 4-bit vector is ANDed with a 6-bit vector, and result is assigned to an 8-bit vector? Are 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 than expected. *GOTCHA!* (Self-determined versus context-determined operations also affect the gotchas described in Sections 5.2, 5.3 and 5.4, which follow).

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, as follows:

In context, the operation is:

```
a:      010101 (6-bits)
b:  &    1111 (4-bits)
c:  ???????? (8-bits)
```

After expansion, the operation is:

```
a:      00010101 (8-bits)
b:  &   00001111 (8-bits)
c:      00000101 (8-bits)
```

Why were `a` and `b` left-extended with zeros? That question is answered in Section 5.2, which discusses zero-extension and sign-extension in Verilog.

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

```
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 00000001
```

In this example, the unary AND of `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 operation is part of a compound expression, as in the example above, then the result of the self-determined operation becomes part of the context for the rest of the statement. Thus:

In context, the operation is:	After expansion, the operation is:
a: `010101` (6-bits)	**a:** `00010101` (8-bits)
&b: & `1` (1-bit result of &b)	**&b:** & `00000001` (8-bits)
c: `????????` (8-bits)	**c:** `00000101` (8-bits)

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 00000000, which would be the wrong answer. But this is not a gotcha, because the unary AND operation is self-determined, and therefore gets the right answer.

How to avoid this gotcha: Verilog generally does the right thing. Verilog's rules of self-determined and context-determined operations 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 1, below, should help. This table lists the Verilog and SystemVerilog operators, and whether they are self-determined or context-determined.

Table 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`).
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`)

Table 1: Determination of Operand Size and Sign Extension[1] (Continued)

Operator	Operand Extension Determined By	Notes
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.
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 paper.

[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`.

5.2 Operation size and sign extension in assignment statements

Gotcha: *In an assignment statement, size extension context is dependent on both sides of the assignment, whereas sign extension context is only dependent on one 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. Table 1 in section 5.1 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 involved 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 side 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 side 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 is 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 map to how hardware works. The following examples illustrate cases 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 is when an engineer forgets—or doesn't understand—Verilog's rules for operator expansion rules. The following examples show a few circumstances where an engineer might see different results than expected, if the rules for zero extension versus sign extension are not well understood. These examples use the same declarations as the examples above.

```
s3 = u1 + u2;   // GOTCHA? zero extension, even though S3 is signed type
                // Rule: signed left-hand side does affect sign extension
                // context of operands on right-hand side

u3 = s1 + s2;   // GOTCHA? sign extension, even though U3 is unsigned type
                // Rule: unsigned 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 the
                // 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 (not the actual result) 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 the 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 illustrated in this section 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 signed-ness 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 signed-ness 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);  // cast s1 to 32 bits wide (self-determined)
                            // cast u2 to signed and do sign extension
```

5.3 Signed arithmetic

Gotcha: *Apparent signed arithmetic operations can use unsigned arithmetic, or incorrect sign extension.*

Section 4.1 discussed some of the gotchas with literal number sign extension rules, and Section 5.2 covered gotchas with sign extension in operations. This section covers important gotchas when performing arithmetic operations on signed data. Verilog overloads 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-dependent, meaning 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 signed-ness 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 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 + 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. In synthesis, DC 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-dependent. 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 dependent 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 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 + ci;    // GOTCHA: ci is subtracted
endmodule
```

Now all operands on the right-hand side are signed, and so a signed operation will be performed, right? *GOTCHA!*

The example above does do signed arithmetic, but it does incorrect sign extension—at least it is incorrect for the intended signed adder model. The gotcha again relates to the ADD operator being context-dependent. 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 zero, sign-extending it to 5 bits will be `5'b00000`, which is still zero. However, if `ci` is one, sign-extending it to 5 bits will be `5'b11111`, which is negative 1, instead of positive 1. The result of the ADD operation when carry-in is set is `a + b + -1`. *GOTCHA!*

Using sign casting. Verilog-2001 introduced the `$signed` and `$unsigned` conversion functions, and SystemVerilog adds sign casting. These allow changing the signedness of an operand. The following example uses sign casting to try to fix the signed adder problem.

```
input  logic              ci,           // unsigned 4-bit inputs
...
assign {co,sum} = a + b + signed'(ci);  // GOTCHA: ci is subtracted
```

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. 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.

5.4 Bit select and part select operations

Gotcha: *The result of a bit select or part select operation is always unsigned.*

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 signed-ness can be unexpected, and is another source for signed arithmetic gotchas.

```
parameter SIZE = 31;

logic signed [SIZE:0] a, b;        // signed vectors
logic signed [SIZE:0] sum1, sum2;  // signed vectors
logic signed [   7:0] sum3;        // 8-bit signed vector

  assign sum1 = a + b;                  // signed adder
  assign sum2 = a[SIZE:0] + b[SIZE:0];  // GOTCHA! unsigned adder
  assign sum3 = a[7:0] + b[7:0];        // 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[SIZE:0]) + $signed(b[SIZE:0]);
assign sum3 = signed'(a[7:0]) + signed'(b[7:0]);
```

5.5 Increment, decrement and assignment operations

Gotcha: *Increment, decrement, and assignment operations perform blocking assignments.*

SystemVerilog provides the C-like ++ and -- increment/decrement operators, and the C-like assignment operators such as +=, -=, *= and /=. The usage of the operators is intuitive and useful in C programming, and that intuitive usage carries over to modeling verification testbenches in SystemVerilog. But there is a gotcha when using these operators for modeling hardware. All of these new operators behave as blocking assignments when updating their target variable. 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, posedge reset)
  if (reset)           fifo_write_ptr = 0;
  else if (!fifo_full) fifo_write_ptr++;

always_ff @(posedge clock)
  if (fifo_write_ptr == 15) fifo_full <= 1;
  else                      fifo_full <= 0;
```

The preceding example is not a good design example. It does, however, illustrate the gotcha with using the ++ operator in sequential logic. The first procedural block 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 preforms 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.

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.

5.6 Pre-increment versus post-increment operations

Gotcha: *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 because `++i` and `i++` are stand-alone statements. Nothing is using the value of i in the same statement in which it is incremented. The statement which follows (the `i<=255` test in this example) 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 (or decremented). 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
```

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
```

How to avoid this gotcha: The only 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.

5.7 Operations that modify the same variable multiple times in an assignment

Gotcha: *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;         // increment i, then assign result to j
j = (i += 1);    // 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;
```

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. Some possible scenarios are:

`j = 9 + 10;` — The `i--` operation is executed first and decrements `i` to 9, then `++i` sees the new value of `i` and increments to 10.

`j = 9 + 11;` — The `--i` operation sees the original value of `i` and decrements to 9, and the `i++` operation also sees the original value of `i` and increments to 11.

`j = 10 + 11;` — The `++i` operation is executed first and increments `i` to 11, then `--i` sees the new value of `i` and decrements to 10.

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. The DC synthesis compiler does not permit these types of operations, because of the indeterminate results.

5.8 Operator evaluation short circuiting

Gotcha: *Simulation might not evaluate all operation operands in some circumstances.*

Software simulation does not always evaluate statements exactly the same way as hardware. Consider the following example:

```
always_ff @(posedge clock)
  if (mem_en && write) mem[addr] <= data_in;
```

In this example, the logical-AND operator (`&&`) checks for both `mem_en` and `write` to be true. In hardware, this 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 standard allows, but does not require, software tools to short circuit logical-AND, logical-OR and the `?:` conditional operations. The Verilog standard is not clear as to whether other operators can short circuit. It neither expressly permitted nor expressly prohibited.

Does short circuiting matter? Not in the preceding example. Simulation results of the logical-AND operation will match the behavior of actual hardware. Now consider a slightly different example:

```
always_ff @(posedge clock)
  if ( f(in1, out1) && f(in2, out2) ) ...

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 in this example modifies the value passed into it and passes the value 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 status return is replicated as combinational logic to each input of the gate. As combinational logic, both `out1` and `out2` are continuously updated to reflect their input values. In software, however, the two functions are evaluated from left-to-right. If the return of the first function call is 0, then the operation might short-circuit. If short circuiting does occur, then the function is not called the second time, and `out2` is not updated to reflect the value of `in2`. *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 will correctly match hardware behavior.

6.0 Programming gotchas

6.1 Assignments in expressions

Gotcha: *SystemVerilog allows assignments within expressions, with the same gotchas as C.*

Gotcha: *SystemVerilog's syntax is different than 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 or NULL, is legal. But, to prevent unintentional uses of this capability, such as `if (a=b)`, SystemVerilog requires that the assignment be enclosed in an expression. Thus:

```
if (a=b) ...       // illegal in SystemVerilog
if ( (a=b) ) ...   // legal in SystemVerilog, but 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, to use the C-like syntax, and then wonder why the tool is reporting a syntax error. Is the 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 than 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.

6.2 Procedural block activation

Gotcha: *Initial procedural blocks can activate in any order relative to always procedural blocks.*

A common gotcha for new users to Verilog and SystemVerilog is understanding the scheduling

and/or use of `initial` procedural blocks. Because of the name “initial”, many engineers mistakenly believe that this block is executed before any `always` procedural blocks. Other engineers mistakenly believe just the opposite, thinking that `initial` blocks are guaranteed to execute after all `always` blocks are active.These are incorrect assumptions! The Verilog and SystemVerilog standards state that all procedural blocks, regardless of their type, become active at time zero, and in any order. Verilog `initial` blocks have no precedence over `always` blocks, nor do `always` blocks have any precedence over `initial` blocks. As each procedural block is activated, a simulator can, but is not required to, execute statements in the block until a Verilog timing control is encountered.

The false assumption that `initial` procedural blocks will execute before any `always` procedural blocks, or vice-versa, can lead engineers to create stimulus that does not give the same results on different simulators. A simple, but common, example is of this false assumption is:

```
module test;
  logic reset;
  ... // other declarations
  chip dut (.reset, ...);  // instance of design under test

  initial begin
    reset = 1;  // activate reset at time zero
    #10 reset = 0;
    ...
  end
endmodule

module chip (input reset, ...);

  always @(posedge clock or posedge reset)
    if (reset) q = 0;
    else       q = d;

endmodule
```

In this example, it is false to assume that either the `initial` procedural block in the testbench or the `always` procedural block in the design will activate first. The Verilog standard allows either procedural block to be activated before the other. If a simulator activates the `always` procedural block first, it will encounter the `@` timing control in the sensitivity list and suspend execution while waiting for a positive edge of `clock` or `reset`. Then, when the `initial` procedural block activates and changes `reset` to a 1, the `always` block will sense the change and the flip-flop will reset at time zero. On the other hand, if the `initial` procedural block executes first, `reset` will be set to a 1 before the `always` block is activated Then, when the `always` block is activated, the positive edge of `reset` will already have occurred, and the flip-flop will not reset at time zero. Different simulators can, and usually do, activate multiple procedural blocks in a different order. *GOTCHA!*

How to avoid this gotcha: This gotcha is avoided by proper education and understanding of the Verilog event scheduling of concurrent statements. In the example above, the fix is to make the time zero assignment to `reset` a nonblocking assignment. 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.

6.3 Combinational logic sensitivity lists

Gotcha: *@* might not infer a complete combinational logic sensitivity list.*

Verilog `always` procedural blocks are used to model designs for synthesis. The synthesizable RTL modeling style requires that an `always` block have an edge sensitive timing control (the `@` token) specified immediately following the `always` keyword. This time control is referred to as the procedural block's *sensitivity list*. 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!*

How to avoid this gotcha: The Verilog 2001 standard added an `@*` wildcard sensitivity list that infers a complete sensitivity list for both simulation and synthesis. However, the `@*` has a gotcha that will be discussed later in this section. SystemVerilog introduced two specialized procedural blocks that infer a complete sensitivity list, `always_comb` and `always_latch`. The preferred method to avoid sensitivity list gotchas is to use the SystemVerilog procedural blocks.

Verilog's `@*` has a subtle gotcha that is not widely known in the design community. `@*` will only infer sensitivity to signals directly referenced in the `always` block. It will not infer sensitivity to signals that are externally referenced in a function or a task 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:

```
module chip (input  wire  [ 7:0] a, b,
             input  wire  [15:0] max_prod,
             input  wire  [ 8:0] max_sum,
             input  wire         error,
             output logic [ 8:0] sum_out,
             output logic [15:0] mult_out);

  function [8:0] mult (input [7:0] m, n);
    mult = 0;
    if (!error)               // error is an external signal
      mult = m * n;
    if (mult > max_prod)      // max_prod is an external signal
      mult = max_prod;
  endfunction

 task sum (input [7:0] p, q, output [8:0] sum_out);
  sum_out = 0;
  if (!error)                 // error is an external signal
    sum_out = p + q;
  if (sum_out > max_sum)      // max_sum is an external signal
    sum_out = max_sum;
  endtask

  always @* begin        // @* will only be sensitive to a and b
    sum(a, b, sum_out);     // @* will not be sensitive to max_prod,
                            //   max_sum or error
    mult_out = mult(a, b);
  end
endmodule
```

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: To avoid this gotcha for function calls, the SystemVerilog `always_comb` should be used instead of `always @*`. The `always_comb` procedural block will descend into function calls to infer its sensitivity list. However, `always_comb` does not descend into task calls. If a task style subroutine is needed, a SystemVerilog void function should be used. Void functions are used like a task, but have the syntax and semantic restrictions of a function, and `always_comb` will descend into the function to infer a complete sensitivity list.

6.4 Arrays in sensitivity lists

Gotcha: *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 logic reads a value from an array. For example::

```
logic [31:0] mem_array [0:1023];   // array of 32-bit variables

always @( /* WHAT GOES HERE? */ )  // need combinational sensitivity
  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 changes, `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!
  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 or mem_array[0] or mem_array[1] or 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] )   // Does this work?
  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 @*      // This works correctly
  data = mem_array[addr];

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

The third method 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];
```

6.5 Vectors in sequential logic sensitivity lists

Gotcha: *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 or 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! edge detect on a vector
  ...
```

How to avoid this gotcha: Only single bit items should be used with `posedge` or `negedge` edge events. If a vector is to be used, then specify which bit is to be used for the edge detect trigger. The code above could be written as:

```
always @(posedge address[15])  // trigger to change on MSB of vector
  ...
```

6.6 Operations in sensitivity lists

Gotcha: *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 a separator, not an operator
  sum = a + b;

always @(a | b)       // GOTCHA! "|" is an operator, not a separator
  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. It is strictly a behavioral coding style, and it is one that is rarely used. 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` or `always_latch` procedural blocks. These procedural blocks automatically infer a correct sensitivity list, which removes any possibility of typos or mistakes. The Verilog `@*` can also be used, but this has its own gotcha (see Section 6.3). When modeling sequential logic, engineers need to be careful to avoid using operations within a sensitivity list.

6.7 Sequential blocks with `begin...end` groups

Gotcha: *Resetable sequential procedural blocks with a begin..end block can contain statements that do not function correctly.*

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) begin
  NextState = WAITE;                      // first statement in block
  case (State)                            // second statement in block
    WAITE: if (ready) NextState = LOAD;
    LOAD:  if (done)  NextState = WAITE;
  endcase
end
```

This modeling style has a gotcha when modeling resetable sequential logic, such as flip-flops. A synthesis requirement is that a resetable 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 resetN)
  if (!resetN) State <= RESET;
  else         State <= NextState;
```

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 problems. When modeling sequential logic that has reset functionality, however, adding `begin...end` can lead to modeling errors. This error is that the resetable sequential block should only contain a single if...else statement, but the begin...end allows additional statements without a simulation syntax error. For example:

```
always @(posedge clock or negedge resetN) begin
  if (!resetN) State <= RESET;
  else         State <= NextState;
  fsm_out <= decode_func(NextState);  // second statement in block
end
```

This example will simulate, and, 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 `fsm_out` does not get reset by the reset logic. Even worse, is that when a reset occurs, the `fsm_out` assignment will be executed, asynchronous to the clock, which is not flip-flop behavior.

This gotcha is an example where Verilog allows engineers to prove what won't work in hardware. The Synopsys DC synthesis compiler will not allow statements outside of an `if...else` statement in resetable sequential procedural blocks. The example above can be simulated (and proven to not work correctly), but cannot be synthesized.

How to avoid this gotcha: The modeling style of using `begin...end` in all procedural blocks is not

appropriate for resetable sequential procedural blocks. Indeed, a good modeling style is to mandate that `begin`...`end` *not* be used in sequential procedural blocks that have reset logic.

6.8 Sequential blocks with partial resets

Gotcha: *Resetable sequential procedural blocks can incorrectly only reset some of the outputs.*

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

```
always @(posedge clock or negedge resetN)
  if (!resetN) begin
    q1 <= 0;
    q2 <= 0;
    q3 <= 0;
  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`.

This modeling mistake will simulate and synthesize as working hardware, so technically, it is not a gotcha. However, .To implement this functionality, DC will add additional gates to the synthesized circuit, that is both area and timing inefficient. These unwanted, and probably unexpected, extra gates is a gotcha.

How to avoid this gotcha: The way to avoid this gotcha is careful modeling. Designers need to be sure that the same variables are assigned values in both branches of the `if`...`else` decision. SystemVerilog cross coverage can be useful in verifying that all variables that are assigned values on a clock are also reset.

6.9 Blocking assignments in sequential procedural blocks

Gotcha: *Sequential logic blocks can have combinational logic blocking assignments.*

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;    // load d into q1 without a clock-to-q delay
  q2 = q1;   // load q1 to 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 DC synthesis compiler 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 DC will optimize out the buffer, and `d` will be directly loaded into `q2`.

How to avoid this gotcha: Engineers should adopt a modeling style that requires the use of nonblocking assignments in sequential procedural blocks. Code checking tools such as LEDA can help enforce this coding style. However, engineers also need to fully understand the difference in how blocking and nonblocking assignments work. There will arise the occasional exception to the rule, where a blocking assignment is needed within a sequential procedural block. Only by understanding how these two assignments work, will engineers know when to correctly make an exception to the rule.

6.10 Evaluation of true/false on 4-state values

Gotcha: *A true/false test can result in three answers.*

Is the expression `4'b010Z` true or false? At least one bit in the vector is a logic 1, so does this mean the expression is true?

How Verilog and SystemVerilog evaluate an expression as being true or false is not always well understood. The issue is that the expression can be a vector of any bit width, and can be either a 2-state expression (each bit might have the logic values 0 or 1), or a 4-state expression (each bit might have the logic values 0, 1, Z or X).

The rules for evaluating whether a 2-state expression is true or false are:

- If *all* bits of the expression are zero, then the expression is *false*.
- If *any* bits of the expression is one, then the expression is *true*.

The rules for evaluating whether a 4-state expression is true or false are:

- If *all* bits of the expression are zero, then the expression is *false*.
- If *any* bits of the expression is one, *and* (*GOTCHA!*) no bits are Z or X, then the expression is *true*.
- If any bits of an expression are X or Z, then the expression is *unknown* in a true/false test.

The rules for 2-state true/false evaluations are simple and intuitive. They match the C language (and other languages). The gotcha is not understanding how a vector with 4-state values is evaluated as true or false.

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 expression to be true or false, no bits in the expression can have a Z or X value.

6.11 Mixing up the not operator (!) and invert operator (~)

Gotcha: *The not operator and the inverse operator 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 cases, the results of these operations happen to be the same, but, in other cases, 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 inverts the results of a true/false test. This means the true/false evaluation is done first, and then the 1-bit true/false result is inverted. On the other hand, the bitwise invert operator simply inverts the value of each bit of a vector. If this 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.

6.12 Nested if...else blocks

Gotcha: *Nesting of if statements can lead to incorrect matching of if...else pairs.*

The `else` branch of a Verilog `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!
```

The indentation of the code above implies that the `else` statement goes with the first `if` statement, but that is not how Verilog works (and indentation does not change the language rules). Verilog language rules state that an `else` statement is automatically associated with the nearest previous `if` statement that does not have an `else`. 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”);
```

How to avoid this gotcha: The automatic `if`...`else` association of Verilog 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 good language-aware text editor can also help avoid this gotcha. A good editor can properly indent nested `if`...`else` statements, making it more obvious which `else` goes with which `if`.

6.13 Casez/casex masks in case expressions

Gotcha: *Masked bits can be specified on either side of a case 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. With `casex`, any bits set to X, Z or ? are masked out. These constructs are useful for concisely modeling many types of hardware, as well as in verification code. An example of using the wildcard `casex` statements is:

```
always_comb begin
  casex (instruction)
    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 (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: A partial solution is to use `casez` instead of `casex`. In the example used in this section, 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 cause 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.

SystemVerilog offers two solutions to this gotcha. The first solution is a special one-sided, wildcard comparison operator, `==?` (there is also a `!=?` operator). 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
```

6.14 Incomplete or redundant decisions

Gotcha: *Incomplete case statements or if...else decision statements, and redundant decision select items can result in design errors.*

Verilog's `if...else` and `case` statements (including `casez` and `casex`) have three gotchas that often result in design problems:

- Not all possible branches need to be specified (incomplete decisions)

- Redundant (duplicate) decision branches can be specified
- Software simulation evaluates decisions in the order listed (priority decoding), but the decision might be able to be evaluated in any order in hardware (parallel decoding).

Verilog synthesis compilers attempt to overcome these Verilog gotchas, using synthesis `full_case` and `parallel_case` pragmas. These pragmas, however, are fraught with danger, and often introduce worse gotchas than those that they solve. These gotchas are well documented in a SNUG 1999 paper, *"'full_case parallel_case', the Evil Twins of Verilog Synthesis"*[5].

How to avoid these gotchas: SystemVerilog adds two decision modifiers, `unique` and `priority`. For example:

```
unique case (sel)
  2'b00: y = a;
  2'b01: y = b;
  2'b10: y = c;
  2'b11: y = d;
endcase
```

```
priority if (a == 0) out = 0;
    else if (a < 5)  out = 1;
    else if (a < 10) out = 2;
    else             out = 3;
```

These modifiers eliminate all of the gotchas with incomplete and redundant decision statements, and prevent the gotchas common to synthesis `full_case` and `parallel_case` pragmas. The benefits of the `unique` and `priority` decision modifiers are described in two other SNUG papers, "*SystemVerilog Saves the Day—the Evil Twins are Defeated! 'unique' and 'priority' are the new Heroes*"[6], and "*SystemVerilog's priority & unique—A Solution to Verilog's 'full_case' & 'parallel_case' Evil Twins!*"[7]. These benefits are not repeated in this paper.

There is, however, a Synopsys DC synthesis compiler gotcha with the SystemVerilog `priority` decision modifier. The keyword "*priority*" would seem to indicate that the order of a multi-branch decision statement will be maintained by synthesis. DC does not do this. DC will still optimize `priority case` decision ordering, the same as with a regular `case` decision statement. While gate-level optimization is a good thing, it is a gotcha if the designer is expecting a `priority case` statement to automatically have the identical priority decoding logic after synthesis.

6.15 Out-of-bounds assignments to enumerated types

Gotcha: *Enumerated types can 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] {WAITE = 3'b001,
                        LOAD  = 3'b010,
                        STORE = 3'b100} states_t;

states_t State, NextState;  // two enumerated variables
```

A surprising gotcha is that an enumerated type variable can have values that are outside of the defined set of values. Out-of-bounds values can occur in two ways: uninitialized variables and statically cast values. Each of these is explained in more detail, below.

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` 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 positive one. 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.

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`:

```
NextState = LOAD;         // legal assignment
NextState = State;        // legal assignment
NextState = 5;            // illegal (not in enumerated label list)
NextState = 3'b001;       // illegal (not in enumerated label list)
NextState = State + 1;    // illegal (not in enumerated list)
```

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

```
NextState = states_t'(State + 1);       // legal assignment, but a GOTCHA!
```

When a value is cast to an enumerated type, and assigned to an enumerated variable, the value is forced into the variable, without any type checking. In the example above, if `State` had the value of `WAITE` (`3'b001`), then `State + 1` would result in the value of `3'b010`. This can be forced into the `NextState` variable using casting. As it happens, this value matches the enumerated label `LOAD`. If, however, `State` had the value of `LOAD`, then `State + 1` would result in the value of `3'b011`. When this value is forced into the enumerated variable `NextState`, it does not match any of the enumerated labels. The `NextState` 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 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. In addition, 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:

```
NextState = State.next(1);     // increment by one to next label in list
```

6.16 Statements that hide design problems

Gotcha: *Some programming statements do not propagate design errors.*

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", as it is 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. The design bug has been hidden. *GOTCHA!*

How to avoid this gotcha: The simple example above could be re-coded to trap an X value on sel. However, this extra code must be hidden from synthesis compilers, as it is not really part of the hardware. For example:

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

A better way to avoid this gotcha is to use SystemVerilog assertions. Assertions have several advantages. Assertions are much more concise than using programming statements to trap problems. Assertions do not need to be hidden from synthesis compilers. Assertions are not likely to affect the design code. Assertions can easily be turned on and off using large grain or fine grain controls. Assertions 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
```

For more details on X hiding gotchas and using assertions to detect design problems, refer to the SNUG paper "*Being Assertive With Your X*"[8], and "*SystemVerilog Assertions are for Design Engineers, too*"[9].

6.17 Simulation versus synthesis mismatches

Gotcha: *Some coding styles can simulate correctly, but synthesize to hardware that is not functionally the same.*

A number of programming statements will simulate one way, but have subtle functional differences from the gate-level implementation created by the DC synthesis compiler. Several of the most common gotchas have been covered under other sections in this paper. Another excellent SNUG paper provides additional insights on modeling gotchas that affect synthesis, "*RTL Coding Styles That Yield Simulation and Synthesis Mismatches*"[10].

7.0 Testbench gotchas

7.1 Multiple levels of the same virtual method

Gotcha: *A virtual method can be re-defined multiple times, leading to confusion as to which virtual method definition is actually used.*

A base class can have a virtual method. If the base class itself is not virtual, then its virtual method must be defined so that the base class can be constructed. That is, when the base class is constructed, it will require a stand-alone definition of its virtual method. When that base class is extended, the child class can generate a new definition of the base class virtual method. If the child class designates this method as virtual also, then, when it is extended to yet another child level (grandchild), the grandchild can redefine the virtual method of its parent.

The example below shows a base class, called `base`, with a virtual method called `print_something`. The `base` class is extended by class `ext1`. In turn, `ext1` is extended by class `ext2`. The `print_somthing` method is designated as virtual in all three classes. The gotcha is in understanding which method is actually used when all three levels have different definitions for the same virtual method. The answer is that there will only ever be one definition for a virtual method within a constructed class, and it is always the definition that is the furthest descendent from the base.

```
package tb_classes;

  class base;
    virtual task print_something;
      $display("printing from base");
    endtask: print_something
  endclass: base

  class ext1 extends base;
    virtual task print_something;
      $display("printing from ext1");
    endtask: print_something
  endclass: ext1

  class ext2 extends ext1;
    virtual task print_something;
      $display("printing from ext2");
    endtask: print_something
  endclass: ext2
endpackage: tb_classes
```

```
program tb_base;
  import tb_classes::*;
  base b = new;
  initial begin
    b.print_something;     // message displayed is “print from base”
  end
endprogram: tb_base

program tb_ext1;
  import tb_classes::*;
  base b;
  ext1 e1 = new;
  initial begin
    b = e1;
    b.print_something;     // message displayed is “print from ext1”
    e1.print_something;    // message displayed is “print from ext1”
  end
endprogram: tb_ext1

program tb_ext2;
  import tb_classes::*;
  base b;
  ext1 e1;
  ext2 e2 = new;
  initial begin
    b  = e1;
    e1 = e2;
    b.print_something;     // message displayed is “print from ext2”
    e1.print_something;    // message displayed is “print from ext2”
    e2.print_something;    // message displayed is “print from ext2”
  end
endprogram: tb_ext2
```

How to avoid this gotcha: This gotcha can be avoided by proper training and an understanding of how SystemVerilog virtual methods work, coupled with adopting a proper object-oriented verification usage methodology, such as the Synopsys VMM.

7.2 Event trigger race conditions

Gotcha: *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. There are two gotchas associated with Verilog’s event synchronization. The first gotcha may or may not really be considered a gotcha, but experience has shown this to be an issue over the years. That is, 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 and usage was presented, the engineer asked if this was something new with SystemVerilog. The answer was no, that it has been in the language since the beginning. To this, the veteran Verilog engineer replied, “Why hasn’t anyone told me about this before?”

The second, and more significant, gotcha is that there can easily be simulation race conditions

with Verilog's event triggering. The following code demonstrates this race condition.

```
module event_example1;
  event get_data, send_data;   // handshaking flags
  initial -> get_data;         // trigger get_data event at time zero
  always @(get_data) begin     // 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 @(send_data) begin    // 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
```

In this simple example, the two `always` blocks model simple behavioral handshaking using `event` data type 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: 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 a delay, as shown in the code below.

```
initial #0 -> get_data;  // start handshaking at time 0, but after all
                         // procedural blocks have been activated
```

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.

But, using `#0` is another gotcha! The Verilog `#0` construct is an easily abused construct, and does not truly ensure the delayed statement will execute after all other statements in a given time step. Many Verilog trainers have recommended that `#0` should never be used. There are alternatives based on the nonblocking assignments that have more reliable and predictable event ordering. Not using `#0` is a good guideline, except for event data types. In Verilog, there is no way to defer the event triggering to the nonblocking event queue.

SystemVerilog comes to the rescue with two solutions that will remove the event trigger race condition, and remove the need to use a `#0`.

How to avoid this gotcha, solution 1: SystemVerilog defines a nonblocking event trigger, `->>`, that will schedule the event to trigger in the nonblocking queue. 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` event in the nonblocking queue allows for the `always` procedure blocks to become active before the event is triggered.

```
initial ->> get_data;  // start handshaking at time 0 nonblocking queue,
                       // after all procedural blocks have been activated
```

How to avoid this gotcha, solution 2: SystemVerilog provides a second approach that will provide a solution to many more situations than the simple example shown in this section. This second solution uses a trigger persistence property that will make 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;         // trigger get_data event 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 @(send_data) begin    // wait for a send_data event
    // could have used wait(send_data.triggered) here also, but it is not
    //   needed since there is no race condition between the two always
    //   blocks
    ... // 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 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.

7.3 Using semaphores for synchronization

Gotcha: *Semaphore keys can be added to a bucket without having first obtained those keys.*

Gotcha: *Semaphore keys can be obtained without waiting for prior requests to be serviced.*

The Verilog `event` data types have been used for years as 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 and worked with when being used. This section describes the gotchas involving semaphores. Section 7.4, which follows, 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.

One gotcha is that any number of keys can be placed into the bucket, regardless of how many were retrieved from the bucket. Thus, incorrect code can keep adding more keys to the bucket than were retrieved from the bucket. Indeed, a process can add keys to the bucket without having retrieved any keys at all.

How to avoid this gotcha: This gotcha has to be managed by understanding how the semaphore `get()` and `put()` methods work, and using them properly.

A second gotcha occurs when a process has to wait for keys. Keys can be retrieved from the bucket by using the `get()` method. The `get()` method is blocking. If the number of keys requested is not available, the process suspends execution until that number of keys is available.

The `get()` method has a subtle, non-intuitive gotcha. If the number of keys requested is not available, then the request is put into a FIFO queue and will wait until the number of keys becomes available. If more than one process requests keys that are not available, the requests are queued in the order received. When keys become available, the requests in the queue are serviced in the order in which the requests were received, First In, First Out. The gotcha is that, when `get()` is called (a new request), an attempt will be immediately made to retrieve the requested keys, without first putting the request into the FIFO queue. Thus, a new request for keys can be serviced, even if other requests are waiting in the semaphore request queue. The following example demonstrates this gotcha.

```
module sema4_example ( ... );
  semaphore queue_test = new; // create a semaphore bucket
  initial begin: Block1       // at simulation time zero...
    queue_test.put(5);           // bucket has 5 keys added to it
    queue_test.get(3);           // bucket has 2 keys left
    queue_test.get(4);           // get(4) cannot be serviced because the
                                 //   bucket only has 2 keys; therefore the
                                 //   request is put in the FIFO queue
    $display("Block1 completed at time %0d", $time);
  end: Block1
  initial begin: Block2 #10   // at simulation time 10...
    queue_test.get(2);           // GOTCHA! Even though the get(4) came
                                 //   first, and is waiting in the FIFO
                                 //   queue, get(2) will be serviced first
    queue_test.get(1);           // this request will be put on the fifo
                                 //   queue 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...
    queue_test.put(3);           // nothing is run from the fifo queue
                                 //   since get(4)is first in the queue
```

```
    #10                        // at simulation time 30...
    queue_test.put(2);            // get(4) and get(1) can now be serviced,
                                  //   in the order in which they were
                                  //   placed in the queue
    $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 will be retrieve the requested keys immediately, even if there are previous `get()` requests waiting in the FIFO queue for keys. In the example above, the `Block1` process begins execution at simulation time 0. 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 on the queue. 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 queue waiting to be serviced. The process then executes a `get(1)`. The request cannot be serviced because the bucket is now empty, and therefore is put on the queue.

At simulation time 30, the `Block3` process puts three keys back in the semaphore bucket. The `get(4)` request sitting in the FIFO queue 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 queue, 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: This gotcha of having a `get()` request serviced immediately, even when other `get()` requests are waiting in the FIFO queue, can be avoided, if the `get()` requests are restricted to getting just one key at a time. If a process needs more then one key, then it would need to call `get(1)` multiple times. When the process is done, it could return multiple keys with a single `put()`. It is not necessary to call `put(1)` multiple times.

7.4 Using mailboxes for synchronization

Gotcha: *Run-time errors occur if an attempt is made to read the wrong data type from a mailbox.*

A second inter-process synchronization capability in SystemVerilog is *mailboxes*. Mailboxes provide a mechanism for both 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.

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 is 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 are returned:

- Returns 1 if the message and the receiving variable are type compatible, and the message is retrieved.
- Returns -1 if the message and the receiving variable are type incompatible, in which case the message is not retrieved.
- Returns 0 if there is no message in the mailbox to retrieve.

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 put(), get() and try_get() methods.

```
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: messages of any data type can be put into mbox1
    mbox1.put(i);    // OK: messages of any data type can be put into mbox1
    mbox1.put(d_in); // OK: messages of any data type can be put into mbox1

    mbox1.get(b);    // OK: data type matches first message type in mbox1
    mbox1.get(b);    // ERROR: b is wrong type for next message in mbox1
    s = mbox1.try_get(d_out);  // must check status to see if OK
    case (s)
       1: $display("try_get() succeeded");
      -1: $display("try_get() failed due to type error");
       0: $display("try_get() failed due to no message in mailbox");
    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 or elaborater will give a compilation or elaboration 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 before hand what data type will be in the mailbox.

The next example illustrates declaring a typed mailbox.

```
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 or elaboration error will occur.

7.5 Coverage reporting

Gotcha: *get_coverage() and get_inst_coverage() 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 *crosscover* points. 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, next_state;

covergroup cSM @(posedge clk);
  coverpoint state {
    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 will 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. *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.

7.6 $unit declarations

Gotcha: *$unit declarations can be scattered throughout multiple source code files.*

`$unit` is a declaration space that is visible to all design units that are compiled together. The purpose of `$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:

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

module chip (...);
  ...
  $unit::states_t state, next_state;  // get states_t definition from $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, is that $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, let's say 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 the design files, with their $unit declarations are compiled along with the IP model file, with its $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 serves 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 Section 2.6 for examples of wildcard and explicit imports).

7.7 Compiling $unit

Gotcha: *Separate file compilation may not see the same $unit declarations as multi-file compilation.*

A related, and major gotcha, with $unit is that multi-file compilation and separate file compilation might not see the same $unit definitions. The $unit declaration space is a pseudo-global space. All files that are compiled together share a single $unit space, and thus declarations made in one file are visible, and can be used in another file. This can be a useful feature. A user-defined type definition can be defined in one place, and that definition can be used by any number of modules, interfaces or test programs. If the definition is changed during the design process (as if that ever happens!), then all design blocks that reference that shared definition automatically see the change. On the other hand, a software tool that can compile each file separately, such as DC, will see a separate, and different, $unit each time the compiler is invoked. If some $unit definitions are made in one file, they will not be visible to another file that is compiled separately.

Synopsys VCS, Formality and Magellan are multi-file compilers. DC and LEDA are separate file compilers. If $unit declarations are scattered between multiple files, and the files are not

compiled together, then DC and LEDA will not see the same `$unit` declarations as VCS, formality and Magellan.

How to avoid this gotcha: The gotcha of different tools seeing different `$unit` declarations can easily be avoided by using packages instead of `$unit`. Packages provide the same advantages of shared definitions and declarations, but in a more structured coding style. If `$unit` is used, then a good style is to ensure that all `$unit` definitions and declarations are made in one, and only one file. That file must then always be compiled with any file or files that use those declarations.

8.0 References

[1] *"IEEE P1364-2005 standard for the Verilog Hardware Description Language"*, IEEE, Pascataway, New Jersey, 2001. ISBN 0-7381-4851-2.

[2] *"IEEE 1800-2005 standard for the SystemVerilog Hardware Description and Verification Language"*, IEEE, Pascataway, New Jersey, 2001. ISBN 0- 7381-4811-3.

[3] "*SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling", by* Stuart Sutherland, Simon Davidmann and Peter Flake. Published by Springer, Boston, MA, 2004, ISBN: 0-4020-7530-8.

[4] "*Signed Arithmetic in Verilog 2001—Opportunities and Hazards*", by Dr. Greg Tumbush. Published in the proceedings of DVCon, 2005.

[5] *"'full_case parallel_case', the Evil Twins of Verilog Synthesis"*, by Clifford Cummings. Published in the proceedings of SNUG Boston, 1999.

[6] "*SystemVerilog Saves the Day—the Evil Twins are Defeated! "unique" and "priority" are the new Heroes*", by Stuart Sutherland. Published in the proceedings of SNUG San Jose, 2005.

[7] "*SystemVerilog's priority & unique - A Solution to Verilog's 'full_case' & 'parallel_case' Evil Twins!*", by Clifford Cummings. Published in the proceedings of SNUG Israel, 2005.

[8] "*Being Assertive With Your X*", by Don Mills. Published in the proceedings of SNUG San Jose, 2004.

[9] "*SystemVerilog Assertions are for Design Engineers, Too*", by Don Mills and Stuart Sutherland. Published in the proceedings of SNUG San Jose, 2006.

[10]"*RTL Coding Styles That Yield Simulation and Synthesis Mismatches*", by Don Mills and Clifford Cummings. Published in the proceedings of SNUG San Jose, 1999, and SNUG Europe, 2001.

9.0 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 book *"SystemVerilog for Design"* and is the author of *"The Verilog PLI Handbook"*. He has also authored a number of technical papers on Verilog and SystemVerilog, which are available at www.sutherland-hld.com/papers. You can contact Stuart at stuart@sutherland-hdl.com.

Mr. Don Mills has been involved in ASIC design since 1986. During that time, he has work on more than 30 ASIC projects. Don started using top-down design methodology in 1991 (Synopsys DC 1.2). Don has developed and implemented top-down ASIC design flow 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.