Gotcha Again
More Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know

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

Don Mills
LCDM Engineering
mills@lcdm-eng.com

Chris Spear
Synopsys, Inc.
chris.spear@synopsys.com

ABSTRACT

The 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." (*http://www.hyperdictionary.com/computing/gotcha*).

This paper documents 38 gotchas when using the Verilog and SystemVerilog languages. Some of these gotchas are obvious, and some are very subtle. The goal of this paper is to reveal many of the mysteries of Verilog and SystemVerilog, and help engineers understand the important underlying rules of the Verilog and SystemVerilog languages. The paper is a continuation of a paper entitled "*Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know*" that was presented at the Boston 2006 SNUG conference [1].

Table of Contents

1.0 Introduction 3
2.0 Design modeling gotchas 4
    2.1 Overlapped decision statements 4
    2.2 RTL and synthesized gate-level simulation of full_case or unique case do not match 5
    2.3 Simulation versus synthesis mismatch in intended combinational logic 6
    2.4 Nonblocking assignments in combinational logic 8
    2.5 Loading memory models modeled with always_latch 9
    2.6 Default of 1-bit internal nets 11
    2.7 Port direction coercion 12
3.0 General programming gotchas 13
    3.1 Compile errors with clocking blocks 13
    3.2 Misplaced semicolons after end or join statement groups 14
    3.3 Misplaced semicolons after decision statements 15
    3.4 Misplaced semicolons in for loops 16
    3.5 Infinite for loops 17
    3.6 Locked simulation due to concurrent for loops 18
    3.7 Referencing for loop control variables outside of the loop 19
    3.8 Summing a subset of value in an array returns an incorrect value 20
    3.9 Task/function arguments with default values 20
    3.10 Static tasks and functions are not re-entrant 21
    3.11 Compile error from a local variable declaration 23
4.0 Object Oriented Programming (OOP) gotchas 23
    4.1 Programming statements in a class get compilation errors 23
    4.2 Compile errors when using interfaces with classes 25
    4.3 Objects in mailbox have the same values 26
    4.4 Passing object handles to methods using input versus ref arguments 26
    4.5 Creating an array of objects 27
5.0 Constrained random verification gotchas 28
    5.1 Some object variables are not getting randomized 28
    5.2 Boolean constraints on more than two random variables 29
    5.3 Undetected randomization failures 30
    5.4 Unwanted negative numbers in random values 31
6.0 SystemVerilog coverage gotchas 32
    6.1 Coverage is always reported as 0% for a cover group 32
    6.2 The coverage report lumps all instances together 33
    6.3 Covergroup arguments directions are sticky 34
7.0 SVA gotchas 35
    7.1 Assertion pass statements execute with a vacuous success 35
    7.2 Concurrent assertions in procedural blocks 36
    7.3 Mismatch in assert...else statements 37
8.0 Tool compatibility gotchas 38
    8.1 Default simulation time units and precision 38
    8.2 Package chaining 39
    8.3 Non-standard keywords 40
    8.4 Array literals versus concatenations 41

8.5 Module ports that pass floating point values (real types) 42
9.0 Corrections to the first gotchas paper 43
10.0 References and resources 44
11.0 About the authors 44

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!

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

- Inheritance of C and C++ gotchas

  Verilog and SystemVerilog leverage the general syntax and semantics of the C and C++ languages. Many of the gotchas of C and C++ carry over into Verilog and SystemVerilog. (As an aside, the common C gotcha shown at the beginning of this introduction cannot occur in Verilog and SystemVerilog; it is an illegal syntax.)

- Loosely typed operations

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

- Allowance to model good and bad designs

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

- Not all tools implement the Verilog and SystemVerilog standards in the same way.

  Software tools do not always execute Verilog and SystemVerilog code in the same way. This is not a problem with the definition of the Verilog and SystemVerilog languages; it is a problem with software tools. Nevertheless, these differences can result in unexpected simulation and synthesis differences.

- Ambiguities in the IEEE standards

  The IEEE Verilog Language Reference Manuals (LRM)[2] and SystemVerilog Language Reference Manual[3] are complex documents, numbering over 1500 pages combined. Two types of ambiguities occasionally occur in these complex documents: the rule for a corner case usage of the language is not covered, or different sections of the LRMs describe conflicting rules. These ambiguities in the standards can lead to differences in tool behavior.

This paper is a continuation of a paper presented six months earlier at the Synopsys Users Group (SNUG) Conference held in Boston, Massachusetts, in October, 2006). The first paper was titled *"Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know"* [1]. This first paper presented 57 gotchas. The gotchas listed in that paper are not repeated in this paper. It is intended that the first paper and this paper be used together.

2.0 Design modeling gotchas

2.1 Overlapped decision statements

Gotcha: *One of my decision branches never gets executed.*

Synopsis: *Cut and paste errors in decision statements can go undetected in simulation.*

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

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

The coding error in the example above is not a syntax error. The code will compile and simulate, but the third branch will never execute. Since it is not a syntax error, the modeling error can go undetected in simulation.

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

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

Both of the above examples are very easy errors to make. Often, engineers will write the code for the first decision, then cut-and-paste that line for the rest of the decision, modifying the pasted lines as needed. If, after pasting, a decision values is not modified, then a difficult to detect coding error has occurred. *Gotcha!*

Synthesis compilers will warn about the overlap in decisions in the preceding examples. However, since it is only a warning message, it may go unnoticed. Both examples will synthesize to latches because the combinational logic is not fully defined for synthesis.

How to avoid this gotcha: SystemVerilog adds a **`unique`** modifier that can be used with both **`if`**...**`else`** and **`case`** decision statements.

```
always_comb begin
  unique if (sel == 2'b00) y = a;
    else if (sel == 2'b01) y = b;  // SIMULATION WARNING DUE TO OVERLAP
    else if (sel == 2'b01) y = c;  // SIMULATION WARNING DUE TO OVERLAP
    else    (sel == 2'b11) y = d;
end
```

```
always_comb begin
  unique case (sel)
    2'b00: y = a;
    2'b01: y = b;  // SIMULATION WARNING DUE TO OVERLAP
    2'b01: y = c;  // SIMULATION WARNING DUE TO OVERLAP
    2'b11: y = d;
  endcase
end
```

The **`unique`** modifier requires that simulators report a warning if multiple branches of a decision are true at the same time. The **`unique`** modifier also requires that simulation generate a warning message if no decision branch is taken.

Warning! In synthesis, DC treats **`unique case`** the same as if both the synthesis pragmas of `full_case` and `parallel_case` are set. With the `full_case` pragma set, DC will not generate latches in the two examples above. This will result in differences in the RTL behavior and the synthesized gate-level behavior. *Do not ignore the simulation warnings generated by using unique—the warnings indicate there is a coding problem!*

2.2 RTL and synthesized gate-level simulation of full_case or unique case do not match

Gotcha: *My default assignment before a case statement disappears after synthesis.*

Synopsis: *Synthesis "optimizes" full_case and unique case decisions as self-contained logic.*

When a case statement must assign to several variables, a common modeling style is to assign a default value to all variables before the case statement, and within the case statement only assign values that are different than the defaults. This style can substantially reduce the code within a case statement, and document the values that are different for each branch of a case statement.

The DC synthesis compiler treats a **`case`** statement that has a **`full_case`** pragma as a complete, and therefore self-contained, decision. This will cause the following example to synthesize differently than the RTL simulation.

```
always_comb begin
  load_s   = 1'b0;      // default values for combinational outputs
  load_b   = 1'b0;
  load_f   = 1'b0;
  load_pc  = 1'b0;
  inc_pc   = 1'b0;
```

```
    set_br    = 1'b0;
    rslt_oeN = 1'b0;
    dout_oeN = 1'b0;
    dmem_rdN = 1'b0;
    dmem_wrN = 1'b0;
    case (state) // synopsys full_case -- Gotcha!
      state1 : begin               // only values different than default
                 load_s   = 1'b1;  //   are listed in case statement
                 inc_pc   = 1'b1;
               end
      state2 : set_br     = 1'b1;
      state3 : begin
                 dmem_wrN = 1'b1;
                 rslt_oeN = 1'b1;
               end
      ...
    endcase
  end
```

In simulation, the combinational logic outputs are first assigned a default value. Then, if specific conditions are true, specific outputs are modified to a different value. Simulation results are correct for this example.

In synthesis, the **`full_case`** pragma instructs DC to treat the case statement as complete and self-contained. Therefore, DC will consider the default assignments before the case statement as redundant, and optimize away these assignments. The result is that the gate-level implementation of the example above will not function the same way as what was simulated. *Gotcha!*

The Synopsys DC synthesis compiler treats the SystemVerilog **`unique case`** the same as if both the synthesis pragmas of **`full_case`** and **`parallel_case`** were set. Thus, using **`unique case`** in the example above will have the same synthesis gotcha.

How to avoid this gotcha: This gotcha is not a Verilog or SystemVerilog language problem. It is a problem with how DC optimizes **`full_case`** pragmas. To avoid this gotcha, neither **`full_case`** nor **`unique case`** should be used if a case statement is not completely self-contained.

2.3 Simulation versus synthesis mismatch in intended combinational logic

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

Synopsis: *Synthesis may optimize away inferred storage in combinational logic.*

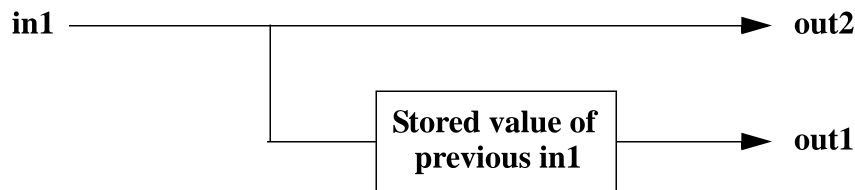
Verilog and SystemVerilog require that the left-hand side of procedural assignments be variable types. In simulation, variables have storage, and preserve values between assignments. In hardware, combinational logic devices do not have storage. If the designer’s intent is to model combinational logic,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

Generally, the DC synthesis compiler is very good at detecting if a combinational logic procedural block is relying on simulation storage. When variable storage is used, DC will add latches to the

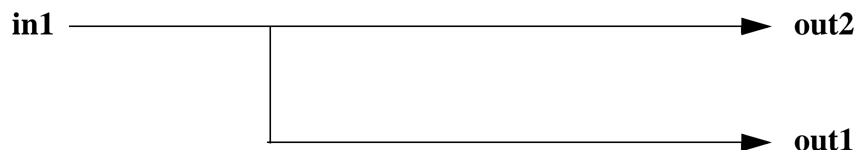
gate-level implementation to preserve that storage. In the following example, however, DC does not detect that the RTL model is using the storage of the variables.

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

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



When this example is synthesized, Design Compiler generates the following gate-level functionality:



Simulation of the post-synthesis functionality will not match the RTL simulation functionality. DC failed to detect that out1 is reflecting the stored value of out2 (which is the previous value of in1), and does not implement the RTL functionality. DC does not generate any warnings or errors for this example. *Gotcha!*

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

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

A style check tool, such as LEDA, can also be used to detect the incorrectly modeled combinational logic. LEDA generated the following warnings on the incorrect example at the beginning of this section:

```
W446: Reading from an output port out2
W502: A variable in the sensitivity list is modified inside the block
```

2.4 Nonblocking assignments in combinational logic

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

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

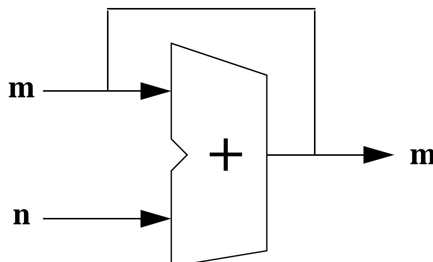
Verilog's nonblocking assignment is intended to model the behavior of sequential logic clock-to-q delay. A nonblocking assignment evaluates the right-hand side expression immediately, and schedules a change on the left-hand variable after a clock-to-q delay. Any statements following the nonblocking assignment statement are "not blocked" in the simulation execution flow. This delta between evaluation and change behaves as a clock-to-q delay, even in a zero-delay RTL model.

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

```
always @(m or n) // combinational logic sensitivity list (no clock edge)
  m <= m + n;    // scheduled change to m after zero-delay clock-to-q delta
```

In the example above, the `always` block triggers when either `m` or `n` changes value. The left-hand side, which is `m`, is scheduled to be updated later in the same simulation time. During this delta in time, the nonblocking assignment does not block the execution flow of the procedural block, and so the block returns to the sensitivity list to wait for the next change on `m` or `n`. After the zero-delay clock-to-q delta, the simulator will update the value of `m`. This change will trigger the sensitivity list. As long as the result of `m + n` results in a new value of `m`, simulation will be locked up in the current simulation time, continually scheduling changes to `m`, and then triggering on that change. *Gotcha!*

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



How to avoid these gotchas: The simulation lock-up problem can be fixed by changing the

assignment statement from nonblocking to blocking. A blocking assignment immediately updates the variable on the left-hand side. The value of m will have a new, stable value before the procedural block returns to the sensitivity list, and thus will not re-trigger the procedural block.

```
always @(m or n) // combinational logic sensitivity list (no clock edge)
  m = m + n;     // immediate update to m with no clock-to-q delay
```

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

SystemVerilog comes to the rescue! The SystemVerilog **always_comb** and **always_ff** constructs can be used to help avoid this coding error gotcha. These constructs document what type of logic is intended, and allow tools to check that the functionality matches the designer's intent.

The **always_comb** procedural block infers a proper combinational logic sensitivity list. In addition, **always_comb** enforces some coding rules that help ensure proper combinational logic is modeled. One of these rules is that only one source can write to a variable. In the code `m <= m + n;`, m is being used as both an input and an output of the adder. If any other part of the design also writes a value to m (as an input to the adder), it is a syntax error. In the context of a full design, the following code causes a syntax error, instead of causing simulation to lock up.

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

VCS generates the following error message when **always_comb** is used, and some other source also generates values for the adder inputs:

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

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

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

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

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

2.5 Loading memory models modeled with always_latch

Gotcha: *When I use SystemVerilog, I cannot load my memory models using $readmemb.*

Synopsis: *The* **$readmemb()** *and* **$readmemh()** *system tasks cannot be used to load a RAM model*

that uses **`always_latch`**.

Typically, a bus-functional model of a RAM models are either synchronous (clock based) or asynchronous (enable based). Synchronous RAMs behave at the abstract level as flip-flops. Asynchronous RAMs behave at the abstract level as latches. However, there is a gotcha if these devices are modeled using the SystemVerilog's **`always_ff`** or **`always_latch`** procedural blocks.

```
module RAM
(inout wire [63:0] data,
 input wire [ 7:0] address,
 input wire        write_enable, read_enable
);

  logic [63:0] mem [0:255];

  always_latch   // asynchronous write (latch behavior)
    if (write_enable) mem[address] <= data;  // write to RAM storage

  assign data = read_enable? mem[address] : 64'bz;
endmodule

module test;
  wire  [63:0] data;
  logic [ 7:0] address;
  logic        write_enable, read_enable;

  RAM ram1 (.*); // instance or RAM model

  initial begin
   $readmemh("ram_data.dat", ram1.mem);  // GOTCHA!
   ...
```

In this example, the `RAM` model is correct—at least functionally. The problem is that the **`always_latch`** procedural block enforces a synthesis rule that multiple procedural blocks cannot write to the same variable. In this example, the testbench is attempting to load the RAM model using the Verilog $readmemh task, which is a common way to load Verilog memory models. This is a second procedural block writing to the RAM storage (`mem`), which is illegal. VCS generates the following error:

```
Error-[ICPD] Invalid combination of procedural drivers
   Variable "mem" is driven by an invalid combination of procedural
     drivers. Variables written on left-hand of "always_latch" cannot
     be written to by any other processes, including other
     "always_latch" processes.
```

How to avoid this gotcha: The fix for this coding problem is to use Verilog's general purpose **`always`** procedural block for this abstract RAM model. SystemVerilog's **`always_latch`** procedural block is intended to model synthesizable RTL models, and is the wrong construct for abstract models.

```
module RAM
  ...
  always @*    // asynchronous write (latch behavior)
    if (write_enable) mem[address] <= data;  // write to RAM
```

```
...
```

2.6 Default of 1-bit internal nets

Gotcha: *My netlist only connects up bit zero of my vector ports.*

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

Verilog has a convenient shortcut when modeling netlists, in that it is not necessary to declare all of the interconnecting nets. Undeclared connections default to a **`wire`** net type. In a netlist with hundreds or thousands of connections, implicit wires can significantly simplify the Verilog source code.

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

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

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

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

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

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

How to avoid this gotcha: VCS, DC, LEDA and other Verilog tools will generate elaboration

warning messages for this gotcha, reporting size mismatches in port connections. *Engineers should not ignore these warnings!* Almost always, warnings about size mismatches in port connections indicate unintentional errors in a netlist.

To fix this gotcha, all internal vectors of a netlist must be explicitly declared. Verilog does allow turning off implicit net types in some or all modules, using the compiler directive **`default_nettype none`**. This directive makes it a requirement that all nets be declared, including 1-bit nets.

Another way to avoid this gotchas is to use the SystemVerilog **`.name`** or **`.*`** port connection shortcuts. These shortcuts will not infer undeclared nets. Further, these shortcuts will not infer connections that do not match in size.

2.7 Port direction coercion

Gotcha: *I declared my port as an input, and synthesis changed it to an output port, or vice-versa.*

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

A little known feature of Verilog is that tools can change a port that was declared as either **`input`** or **`output`** to be an **`inout`** port. This is referred to as "port coercion". Port coercion can occur in two circumstances:

- If a module *writes* values to a port that is defined as **`input`**
- If a module *reads* values of a port that is defined as **`output`**

Port coercion can be useful. It allows the output values of modules to also be read within a module. However, port coercion can also cause unexpected design behavior (gotchas), as illustrated in the following example.

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

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

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

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

Note that port coercion can only occur when net data types (such as **`wire`**) are used on both sides of a port. The reason for this is because net types allow, and resolve, multi-driver functionality.

Since ports are coerced to **inout** ports, they become multi-driver ports, which require net types.

How to avoid this gotcha: Port coercion cannot occur if a variable type (e.g. **reg** or **logic**) is used as a port. Verilog allows output ports to be declared as a variable type, but input ports must be a net type. SystemVerilog relaxes the Verilog data type rules, and allows variables to be used on both input and output ports (bidirectional inout ports must still be a net type, as in Verilog). SystemVerilog also allows continuous assignments to assign to variables. Therefore, module buffer8, can be coded as follows.

```
module buffer8
(output logic [7:0] y,  // variable data type
 input  logic [7:0] a   // variable data type
);
  assign a = y;         // ELABORATION ERROR! this should have been y = a;
endmodule
```

By using variables, port coercion cannot occur, and the coding mistake becomes an error that is detected at elaboration time.

Another way port coercion can be prevented is to use the new Verilog-2005 **uwire** (unresolved wire) net data type. The **uwire** type only allows a single driver on a net. Thus, when buffer8 is connected within module top, an elaboration error occurs because the buffer8 input port (a) has multiple drivers.

```
module buffer8
(output uwire [7:0] y,  // variable data type
 input  uwire [7:0] a   // variable data type
);
  assign a = y;         // ELABORATION ERROR! this should have been y = a;
endmodule
```

3.0 General programming gotchas

3.1 Compile errors with clocking blocks

Gotcha: *I get a syntax error when my test program waits for a clocking block edge.*

Synopsis: *When a test waits for a clocking block edge to occur, the posedge or negedge keyword should not be used.*

Test code that uses the @ event control to delay until a clocking block clock occurs should not specify **posedge** or **negedge** of the clocking block name. The following example causes a compilation error:

For example:

```
program test (input  bit clk,
              input  bit grant,
              output bit request
             );
  clocking cb @(posedge clk);
    output request;
    input  grant;
  endclocking
```

```
  initial begin
    @(posedge cb)   // ERROR: cannot select edge of a clocking block
      $display("At %0d: clocking block triggered", $time);
    ...
  end
endprogram
```

How to avoid this gotcha: When test code needs to delay for a clocking block clock using the @ event control, only the clocking block name should be used. This is because clocking block definitions already specify which edge of the clock is being used. For example:

```
  initial begin
    @(cb)           // OK: delay until clocking block event occurs
      $display("At %0d: clocking block triggered", $time);
    ...
  end
endprogram
```

Using just the clocking block name for an event control can make test code more robust and easier to maintain, especially when the clocking block is defined in an interface. The test program does not need to know if the interface uses a positive edge, negative edge, or both edges (double data rate) of the clock. All the test program needs to reference is the clocking block name.

3.2 Misplaced semicolons after end or join statement groups

Gotcha: *Sometimes I get compilation errors after end or join statements, but not other times.*

Synopsis: *A semicolon after* **`end`** *or* **`join`** *is not legal syntax, but, depending on context, might not be an error.*

Multiple programming statements are grouped using **`begin`**...**`end`** or **`fork`**...**`join`**, **`fork`**...**`join_any`**, or **`fork`**...**`join_none`**. These statement grouping constructs are not followed by a semicolon. However, it is not a syntax error to place a semicolon after **`begin`** or **`fork`**. The semicolon is merely an additional statement within the statement group. A semicolon by itself is a complete programming statement, representing a null-operation statement.

A semicolon after **`end`**, **`join`**, **`join_any`** or **`join_none`** might, or might not, be a syntax error, depending on context. If the keyword is nested within another statement group, then a semicolon is not an error; it is simply another statement in the outer statement group. If the **`end`**, **`join`**, **`join_any`** or **`join_none`** keyword is not enclosed in an outer statement group, then the semicolon is a syntax error, because it is a null statement that is not part of procedural block.

```
module foo;
  ...
  initial begin;              // semicolon is NOT an error
    if (enable) begin;        // semicolon is NOT an error
      ...
    end;                      // semicolon is NOT an error
   end;                       // semicolon IS an error
endmodule
```

How to avoid this gotcha: This is not really a language gotcha, in that there are not any

unexpected or undesirable run-time results. However, the misplaced semicolons can be confusing, because sometimes they are legal, and sometimes they are illegal. To avoid this gotcha, never put a semicolon after the keywords **`begin`**, **`end`**, **`fork`**, **`join`**, **`join_any`** or **`join_none`**.

3.3 Misplaced semicolons after decision statements

Gotcha: *Statements in my* **`if()`** *decision execute even when the condition is not true.*

Synopsis: *A semicolon after the closing parenthesis of a decision statement is legal, and causes the statements that should be within the* **`if()`** *to be outside the* **`if()`**.

A semicolon (**`;`**) by itself is a complete programming statement, representing a null-operation statement. A misplaced semicolon after **`if ()`** is legal. However, the misplaced semicolon can cause the statement or **`begin`**...**`end`** group after the misplaced semicolon to execute at times that were not intended.

```
module foo;
  reg a;
  initial begin
    a = 1;
    if (a);                     // semicolon is wrong, but NOT syntax error
      $display("'a' is true"); // GOTCHA! also prints when 'a' is false
  end
endmodule
```

In the example above, there is no syntax error. The semicolon is a legal statement, and is the only statement associated if the **`if`** condition. The **`$display`** statement, though nicely indented, is not part of the **`if`** statement. The **`$display`** message prints every time, regardless of whether the variable `a` is true or false. *Gotcha!*

The next example illustrates how a misplaced semicolon can lead to a syntax error on a subsequent line of code.

```
module bar;
  reg a;
  initial begin
    a = 1;
    if (a);                     // semicolon is NOT an error
      $display("'a' is true");
    else                        // SYNTAX ERROR! 'else' does not follow 'if'
      $display("'a' is false");
  end
endmodule
```

The **`else`** line in the example above appears to be paired with the **`if`** statement. However, the only statement in the **`if`** branch the misplaced semicolon, which is a null statement. Therefore, the **`$display`** statement that follows is not part of the **`if`** statement, which means the **`else`** statement is not paired with the **`if`** statement. The compiler will report an error on the line with **`else`**, which is actually two lines after the real problem. *Gotcha!*

How to avoid this gotcha: This is an example of a gotcha that is inherited from the C language, from which Verilog and SystemVerilog have their syntax and semantic roots. The same coding mistakes illustrated above can be made in C. The way to prevent this coding gotcha is to know

Verilog syntax, and to correctly use semicolons.

A language-aware text editor can help to avoid this gotcha. A good language-aware editor for Verilog can add auto indentation. The examples above would have obvious indentation errors with such an editor. For example, the first example, above, might be indented as follows:

```
initial begin
  a = 1;
  if (a);
  $display("'a' is true");   // statement is not auto-indented
end
```

3.4 Misplaced semicolons in for loops

Gotcha: *My* **for** *loop only executes one time.*

Synopsis: *A semicolon at the end of a* **for** *loop declaration effectively makes the loop always execute just one time.*

A semicolon (**;**) by itself is a complete programming statement, representing a null-operation statement. A misplaced semicolon after **for()** is syntactically legal. However, the misplaced semicolon has the effect of making loop appear to only execute one time.

```
module foo;
  integer i;
  initial begin
    for (i=0; i<=15; i=i+1);              // semicolon is NOT an error
      begin
        $display("Loop pass executing");  // GOTCHA! only executes once
    end
  end
endmodule
```

In the example above, there is no syntax error. The semicolon is a legal statement, and is the only statement within the **for** loop. The **begin**...**end** group with the **$display** statement is not part of the **for** loop. The loop will execute 16 times, executing a null statement. After the loop has completed, the group of statements that appear to be inside the loop—but which are not—will execute one time. *Gotcha!*

Note that this gotcha can also occur with **while**, **repeat**, **forever** and **foreach** loops.

Looping multiple times executing a null statement is not necessarily a coding error. A common verification coding style is to use an empty **repeat** loop to skip multiple clock cycles. For example:

```
initial begin
  resetN <= 0;
  repeat (8) @(posedge clock) ;  // loop for 8 clock cycles doing nothing
  resetN = 1;
  ...
end
```

How to avoid this gotcha: This gotcha is inherited from the C programming language, where the same coding error is syntactically legal. A language-aware editor with auto-indenting can help to

avoid this gotcha. A good Verilog editor will show the indentation to be wrong for this code, which will indicate a misplaced semicolon.

There is another gotcha with the **for** loop example above. Even though a null statement in a **for** loop is legal code, some tools, including VCS, make it a syntax error. The intent in making this an error is to help engineers avoid a common C programming gotcha. Unfortunately, it also means that if the engineer actually wanted an empty **for** loop, these tools do not allow what should be legal code. The workaround, if an empty loop is actually intended, is to replace the null statement with an empty **begin**...**end** statement group.

3.5 Infinite for loops

Gotcha: *My* **for** *loop never exits.*

Synopsis: *Declaring too small of a* **for** *loop control variable can result in loops that never exits.*

A **for** loop executes its statements until the loop control expression evaluates as false. As in most programming languages, it is possible to write a **for** loop where the control expression is always true, creating an infinite loop that never exits. This general programming gotcha is more likely to occur in Verilog, because Verilog allows engineers to define small vector sizes.

The intent in the following example is to loop 32 times, with the loop control variable having a value from 0 to 31.

```
reg [3:0] i;                 // 4-bit loop control variable
for (i=0; i<=31; i=i+1)      // GOTCHA! i<=31 will always be true
  begin  /* loop body */  end
```

In this example, the loop will run until `i` is incremented to a value greater than 31. But, as a 4-bit variable, when `i` has a value of 31 and is incremented, the result is 0, which is less than or equal to 31. The loop control test will be true, and the loop will continue to execute, starting over with i equal to 0.

How to avoid this gotcha: A simple way to avoid this gotcha is to increase the size of the loop control variable so that it can hold a larger value. Typically, either **integer** (a Verilog type) or **int** (a SystemVerilog type) should be used as loop control variables, both of which are 32-bit signed variables.

SystemVerilog allows the loop control variable to be declared as part of the **for** loop. This makes it easier to avoid the gotcha of too small of a loop control variable, because variable declaration and usage of the variable are in the same line of code.

```
reg [3:0] result;               // 4-bit design or test variable
for (int i=0; i<=31; i=i+1)     // OK: i can have a value greater than 31
  @(posedge clk) result = i;    // OK, but mismatch in assignment sizes
```

Synthesis results are not affected the extra bits of the loop control variable. In synthesis, the name of the loop control variable is primarily used just as a label in the flattened gate-level netlist generated by synthesis. If the value of the larger loop control variable is assigned to a smaller vector, as in the example above, synthesis optimizes out any unused bits of the larger variable.

3.6 Locked simulation due to concurrent for loops

Gotcha: *When I run simulation it either locks up or my combinational logic gets incorrect results.*

Synopsis: *Using* **`always @*`** *for multiple combinational logic procedural blocks that use to the same for loop control variable can lock up simulation.*

The Verilog **`always @*`** construct infers a combinational logic sensitivity list. This list includes all external signals that are read within the procedural block.

The following example illustrates a simple testbench that forks off two tests to run in parallel. Each test contains a **`for`** loop that uses a variable called `i` as the loop control. One loop increments `i`, and the other loop decrements `i`.

```
module test;
  logic [7:0] a, b, c, sum, dif;
  int i;

  adder_subtractor dut (.*);

  initial begin
    fork
      begin: add_test
        for (i = 0; i < 10; i++) begin   // increment i
          a = i;
          b = i + 10;
          #10 $display("At %0d, in scope %m: i=%0d sum=%0d", $time, i, sum);
        end
      end

      begin: dif_test
        for (i = 8; i > 2; i--) begin    // decrement i
          c = i;
          #10 $display("At %0d, in scope %m: i=%0d dif=%0d", $time, i, dif);
        end
      end
    join
    $display("\nTests finished at time %0d\n", $time);
    $finish;
  end
endmodule
```

The intent is that after both loops complete, $finish is called and simulation exits. One loop should execute 10 times, and the other 8 times. Instead of completing, however, simulation looks up, and never exits. The reason is because each loop is changing the same control variable, preventing either loop from ever reaching a value that will cause the loop to exit. *Gotcha!*

How to avoid this gotcha: The way to correct the problem above is to use different variables for each loop. The simplest and most eloquent way to do this is to use the SystemVerilog feature of declaring a local variable within each **`for`** loop. As a local variable instead of an external variable, `i` is not be in the sensitivity list of each procedural block.

```
initial begin
  fork
    begin: add_test
      for (int i = 0; i < 10; i++) begin   // i is a local variable
        ...
      end
    end

    begin: dif_test
      for (int i = 8; i > 2; i--) begin    // i is a local variable
        ...
      end
    end
  join
```

3.7 Referencing for loop control variables outside of the loop

Gotcha: *My Verilog code no longer compiles after I convert my Verilog-style* **`for`** *loops to a SystemVerilog style.*

Synopsis: *Loop control variables declared as part of a* **`for`** *loop declaration cannot be referenced outside of the loop.*

Verilog requires that loop control variables be declared before the variable is used in a **`for`** loop. Since the variable is declared outside the **`for`** loop, it is a static variable that can also be used outside the **`for`** loop.

```
reg [31:0] a, b, c;
integer i;                           // static loop control variable
initial begin
  for (i=0; i<=31; i=i+1) begin
    c[i] = a[i] + b[i];              // OK to reference i inside of loop
  end
  $display("i is %0d", i);           // OK to reference i outside of loop
end
```

SystemVerilog allows declaring **`for`** loop control variables within the declaration of the loop. These are dynamic variables that are local to the loop. The variable is dynamically created when the loop starts, and disappears when the loop exits. Because the variable is dynamic, it is illegal to reference the variable outside of the scope in which it exists. The following code causes a syntax error:

```
reg [31:0] a, b, c;
initial begin
  for (int i=0; i<=31; i=i+1) begin  // dynamic variable declaration
    c[i] = a[i] + b[i];              // OK: i is used inside the loop
  end
  $display("i is %0d", i);           // ILLEGAL: i is used outside of loop
end
```

How to avoid this gotcha: Technically speaking, this is not a gotcha, because it is a syntax error, rather than an unexpected run-time behavior. However, there are times when it is useful to reference loop control variables outside of the loop. In those situations, the loop variable should

be declared outside of the loop, using the Verilog coding style shown in the first example in this section.

3.8 Summing a subset of value in an array returns an incorrect value

Gotcha: *When I try to sum of all array elements greater than 7, I get the wrong answer.*

Synopsis: *Using* **`sum with()`**, *returns a sum of the* **`with`** *expressions, not a sum of a subset of array element values.*

The **`.sum`** method returns the sum of the values stored in all elements of an array. An optional **`with()`** clause can be used to filter out some array values. But, when using **`.sum with()`**, there is a subtle gotcha. In the following example, the intent is to sum up all values in the array that are greater than 7:

```
program automatic test;
  initial begin
    int count, a[] = '{9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
    count = a.sum with (item > 7);  // GOTCHA: expect 17, get 2
    $display("\"a.sum with(item > 7)\" returns %0d", count);
  end
endprogram
```

When the optional **`with()`** clause is used, the **`.sum`** method adds up the return values of the expression inside the **`with()`** clause, instead of summing the values of the array elements. In the example above, the `(item > 7)` is a true/false expression, which is represented with the values 1 or 0. If the array contains the values `{9,8,7,3,2,1}`, then the true/false test for each array element returns the set of values `{1,1,0,0,0,0}`. The sum of these true/false values is 2. *Gotcha!*

How to avoid this gotcha: The true/false result of the relational expression can be used to select just the element values where the test is true. Two simple ways to do this are:

```
count = a.sum with( (item > 7) ? item : 0 );

count = a.sum with( (item > 7) * item );
```

3.9 Task/function arguments with default values

Gotcha: *I get a syntax error when I try to assign my task/function input arguments a default value.*

Synopsis: *Only task function input and inout arguments can be assigned a default value.*

The formal arguments of a task or function can be **`input`**, **`output`**, **`inout`** or **`ref`**. In SystemVerilog, task/function arguments default to **`input`** if no direction has been specified. Once a direction has been explicitly specified, however, that direction is sticky; it affects all subsequent arguments until a new direction is specified.

SystemVerilog allows **`input`** and **`inout`** arguments of a task or function to be specified with a default value. When the task or function is called, a value does not need to be passed to formal arguments that have a default value.

The following function header gets a compile error because the second argument, `start`, has a default value specified.

```
function automatic int array_sum(ref int a[], int start=0);
  for (int i=start; i<a.size(); i++)
    array_sum += a[i]);
endfunction
```

Only **input** and **inout** arguments can have a default value. The problem with this code is that the `start` argument does not have a direction explicitly specified. If no directions at all had been specified, `start` would default to an **input** argument, which can have a default value. In this example, however, the first formal argument of the function, `a[]`, has been defined with a direction of **ref**. This direction is sticky; it also apples to `start`. Assigning a default value to a **ref** argument is illegal.

How to avoid this gotcha: To avoid a direction gotcha, specify a direction for all task/function arguments.

```
function int array_sum(ref int a[], input int start=0);
```

Note: The example shown in this section only causes a compilation error because `start` has a default assignment, which is illegal for **ref** arguments. Sticky argument directions can cause other subtle programming gotchas that are not compilation errors.

3.10 Static tasks and functions are not re-entrant

Gotcha: *My task seems to work fine sometimes, but gets bogus results other times.*

Synopsis: *Invoking a static task or function while a previous call is still executing may cause undesirable results.*

In Verilog, tasks and functions are static by default. This is different than C, where functions are dynamic by default. The difference between static and dynamic is important. In static tasks and functions, any local storage within the task or function, including input arguments, are shared by every call to the task or function. In a dynamic task or function, new storage is created for each call, which is unique to just that call.

This default of static tasks and functions makes sense when modeling hardware, because storage within hardware is static. A testbench, on the other hand, is more of a software program rather than hardware. This different default behavior between C and Verilog can cause unexpected behavior if a verification engineer is expecting C-like programming behavior. Static storage is particularly evident when Verilog tasks are used in a testbench. Tasks can take simulation time to execute. Therefore, it is possible for a task to be invoked while a previous call to the task is still executing, as is illustrated in the following example.

In the following example a task called watchdog is called when the test issues an interrupt request. The task delays for some number of clock cycles, and then prints out a time out error if the interrupt is not acknowledged. The interrupt number and number of cycles to count are passed in as input arguments. The test code calls this task twice, in parallel, for two different interrupt requests.

```
program test (input  bit clock,
              input  bit [1:0] ack,
              output bit [1:0] irq
             );
```

```
  initial begin: irq_test
    $display("Forking off two interrupt requests...");
    fork
      watchdog (0, 20);  // must receive ack[0] within 20 cycles
      watchdog (1, 50);  // must receive ack[1] within 50 cycles
      begin
        irq[0] = 1'b1;
        irq[1] = 1'b1;
        wait (ack)
        $display("Received ack at %0d, disabling watchdog", $time);
        disable watchdog;  // ack received; kill both watchdog tasks
      end
    join_any
    $display("\At %0d, test completed or timed out", $time);
    $finish;   // abort simulation
  end: irq_test

  task watchdog (input int irq_num,      // GOTCHA: static storage
                 input int max_cycles    // GOTCHA: static storage
                );
    $display("At %0d: Watchdog timer started for IRQ[%0d] for %0d cycles",
             $time, irq_num, max_cycles);
    repeat(max_cycles) @(posedge clock) ; // delay until max_cycles reached
    $display("Error at time %0d: IRQ[%0d] not received after %0d cycles",
             $time, irq_num, max_cycles);
  endtask: watchdog
endprogram: test
```

The example above will not cause a syntax error, but it will result in undesired behavior. The second call to the watchdog task will overwrite the `irq_num` and `max_count` values being used by the first call. The first call is still running, but now has incorrect values. *Gotcha!*

How to avoid this gotcha: This gotcha is easy to avoid. The Verilog-2001 standard adds automatic tasks that resolve this gotcha. All that is required is to add the keyword **`automatic`** to the task (or function) declaration.

```
  task automatic watchdog (input int irq_num,      // automatic storage
                           input int max_cycles    // automatic storage
                          );
   ...
endtask: watchdog
```

An automatic task or function is also referred to as a re-entrant task or function. The task or function can be invoked (or entered) while previous calls are still executing. Each call to the re-entrant task or function creates new storage that is local to just that call.

SystemVerilog adds two important enhancements. First, SystemVerilog allows a mix of static and dynamic storage in a task or function. That is, dynamic variables can be declared in static tasks and functions, or static variables can be declared in automatic tasks and functions. Second, SystemVerilog allows the **`automatic`** keyword to be specified as part of the declaration of a module, interface, or program.

```
module automatic chip ( ... );
```

```
program automatic test ( ... );

interface automatic bus_if( ... );
```

All variables, tasks and functions within an automatic module, program or interface is automatic by default. This behavior is like C, where all storage is automatic by default.

3.11 Compile error from a local variable declaration

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

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

Verilog and SystemVerilog allow local variables to be declared within a **begin**...**end** or **fork**...**join** statement group. Local variables must be declared before any programming statements.

```
initial begin
  transaction tr = new;    // this is a declaration, not procedural
  bit status;              // this is a declaration
  status = tr.randomize;   // this is a procedural statement
  extended_trans etr;      // ILLEGAL: this is a declaration
  ...
end
```

How to avoid this gotcha: Two modeling styles can be used to fix this local variable gotcha: declare all variables at the top of the block, or create a new block to localize the scope of a variable. Each style is shown below:

```
initial begin  // move all declarations to the top of the block
  extended_trans etr;      // LEGAL: this is a declaration
  transaction tr = new;    // this is a declaration, not procedural
  bit status;              // this is a declaration
  status = tr.randomize;   // this is a procedural statement
  ...
end

initial begin
  transaction tr = new;    // this is a declaration, not procedural
  bit status;              // this is a declaration
  status = tr.randomize;   // this is a procedural statement
  begin  // create a new block to localize the scope of the variable
    extended_trans etr;    // LEGAL: this is a declaration
    ...                    // use etr variable in local scope
  end
  ...
end
```

4.0 Object Oriented Programming (OOP) gotchas

4.1 Programming statements in a class get compilation errors

Gotcha: *Programming code in an initial procedure compiles OK, but when I move the code to a*

class definition, it won't compile

Synopsis: *Class definitions can only have properties (variables) and methods (tasks and functions); they cannot have procedural programming statements.*

The `Bar` class definition below constructs a `Foo` object, and attempts to initialize the variable `i` within `Foo`:

```
class Foo;
  int data;                 // property

  function int get (...);   // method
    ...
  endfunction
  task put (...);
    ...
  endtask
endclass

class Bar;
  Foo f = new;          // create object f
  f.data = 3;           // ILLEGAL -- initialize a in object f
endclass
```

This example causes a compilation error, because any executable code in a class must be in a task or function. The assignment `f.data = 3;` in the example above is an executable statement, and therefore not allowed. A class is a definition, and cannot contain assignment statements, programming statements, **`initial`** blocks or **`always`** blocks.

How to avoid this gotcha: All programming statements within a class definition must be within tasks or functions. If the intent is to initialize an object's values, as in the preceding example, then the initialization assignments can be in the **`new`** method of the object:

```
class Foo;
  int data;
  function new (int d);
    this.data = d;
  endfunction
endclass

class Bar;
  Foo f = new(3);   // pass initial values to new method of f
endclass
```

If the intent is for one object to change a child object's values at any time after the child object is constructed, then the parent object must contain a function or task to make the change:

```
class Foo;
  int data;
  function new (int d);
    this.data = d;
  endfunction
endclass

class Bar;
  Foo f = new(3);        // pass initial values to new method of f
```

```
    function change_i(m);  // use a method to assign to f.i
      f.data = m;
    endfunction
  endclass
```

Guideline: It is legal to call a **new** constructor from within a class, as shown above. However, this is discouraged as the object will be constructed before any of the test program procedural statements have been executed. This can cause problems if you need to initialize objects in a certain order.

4.2 Compile errors when using interfaces with classes

Gotcha: *I get a compilation error when I try to use interfaces in a class.*

Synopsis: *Static structural components cannot be directly driven from dynamic code.*

In the following example, the `driver` class, which is a dynamic object, needs to drive data into the `arb_ifc` interface, which is a static design object. Since a dynamic object cannot directly drive static objects such as a module or an interface port, this code is illegal.

In the following example, the driver class needs to

```
  interface arb_ifc(input bit clk);
    ...
  endinterface

  program automatic test (arb_ifc.TEST arb);

    class driver;
      arb_ifc arb;               // ERROR: class cannot instantiate interface

      function new(arb_ifc arb); // ERROR: task/func arg cannot be interface
        this.arb = arb;
      endfunction
    endclass

    initial begin
      driver d;
      d = new(arb);
    end
  endprogram
```

How to avoid this gotcha: An interface is a structural component that represents hardware. It can contain signals, code, and assertions. Structural components cannot be passed around for use by dynamic code. Instead, a pointer to the interface is used in the dynamic class object. A pointer to an interface is called a *virtual interface*. The purpose of a virtual interface is to allow dynamic objects to have a handle to a statically elaborated object, and to move data between a dynamic class object and a static object.

The correct way to model the example above is to make the `arb_ifc` instance in the driver class virtual:

```
    class driver;
      virtual arb_ifc arb;                 // pointer to interface
```

```
    function new(virtual arb_ifc arb);   // pointer to interface
      this.arb = arb;
    endfunction
  endclass
```

Virtual interfaces are the bridge or link between the class-based testbench and the DUT.

4.3 Objects in mailbox have the same values

Gotcha: *My code creates random object values and puts them into a mailbox, but all the objects coming out of the mailbox have the same value.*

Synopsis: *The class constructor creates a handle to an object. In order to have multiple objects, the class constructor must be called multiple times.*

The following example puts 10 random object values into a mailbox:

```
class My_class;
  rand int data;
  rand logic [47:0] address;
  ...
endclass

My_class h = new;      // construct a My_class object
repeat(10) begin
  h.randomize();       // randomize properties in the object
  mbx.put(h);          // store handle to object in a mailbox
end
```

The thread that retrieves the objects from the mailbox will find that all the objects contain the same values, the ones generated by the last call to randomize. The gotcha is caused because the code only constructs one object, and then randomizes it over and over. The mailbox is full of handles, but they all refer to a single object.

How to avoid this gotcha: Put the call to the **`new`** constructor inside the loop:

```
my_class h;
repeat(10) begin
  h = new;             // construct a My_class object
  h.randomize();       // randomize properties in the object
  mbx.put(h);          // store handle to object in a mailbox
end
```

4.4 Passing object handles to methods using input versus ref arguments

Gotcha: *My method constructs and initializes an object, but I can never see the object's value.*

Synopsis: *Method input arguments create local copies of variables that are not visible in the calling scope.*

The default direction of methods (tasks and functions) is **`input`**. Inputs create local variables for use within the method. When a method is called, the values of the actual arguments are copied into the local storage. Any changes to this local storage that are made within the method are not passed back to the calling scope.

The following function constructs an object and sets the value of certain properties.

```
function void build_env(Consumer c, Producer p);  // c and p are inputs
  mailbox mbx;
  mbx = new;
  c = new(mbx);   // construct object and store handle in c
  p = new(mbx);   // construct object and store handle in p
endfunction
```

The code that calls the `build_env` function will not be able to see the constructed objects because the function argument directions are not specified, and therefore default to **`input`**. Within the function, `c` and `p` are local variables. The new handles that are stored in the local `c` and `p` variables are not passed back to code that called the `build_env` function.

How to avoid this gotcha: In a method that constructs objects, declare the handle arguments as **`ref`**: A **`ref`** argument is a reference to storage in the calling scope of the method. In the declaration below, when the method constructs an object and stores the handle `c` and `p`, the code that calls `build_env` will see the new handles, because `build_env` is changing the storage in the calling scope.

```
function void build_env(ref Consumer c, ref Producer p);
```

4.5 Creating an array of objects

Gotcha: *I declared an array of objects, but get a syntax error when I try to construct the array.*

Synopsis: *An "array of objects" is actually an array of object handles; each handle must be constructed separately.*

It is often useful to declare an array of object handles in order to store handles to multiple objects. Such an array is often called an "array of objects", but in actuality it is an array of handles, not an array of actual objects.

The following example attempts to create an array to hold 8 objects, but the code does not work:

```
class Transaction;
  ...
endclass

initial begin
  Transaction trans[8];   // An array of 8 Transaction objects

  trans = new;            // ERROR: cannot call new on an object array
  trans = new[8];         // ERROR: cannot call new on an array element
end
```

How to avoid this gotcha: There is no such thing as an array of objects, only an array of handles. Each handle in the array points to an individual object. Each object must be constructed individually, and its handle assigned to an element of the array. The correct way to code the example above is:

```
initial begin
  Transaction trans[8];  // An array of 8 Transaction objects

  foreach (trans[i])
    trans[i] = new();    // Construct an object and store handle in array
```

```
end
```

5.0 Constrained random verification gotchas

5.1 Some object variables are not getting randomized

Gotcha: *Some of my class variables are not getting randomized, even though they were tagged as rand variables.*

Synopsis: *In order for a property to be randomized, it must have a rand or randc tag.*

In order for object variable values to be randomized, each variable in the object must be declared with a **rand** or **randc** tag. Random values are generated when the object's **.randomize** method is called.

The example below has a Payload class, which has a property called data that is tagged to be randomized. A Header class contains an addr property which is tagged to be randomized, and a handle to a Payload object. When a Header object is randomized, however, only addr gets a random value. The payload data is not randomized, even though it has a **rand** tag.

```
program test;
  class Payload;
    rand int data[8];          // data is tagged to be randomized
  endclass

  class Header;
    rand int addr;             // addr is tagged to be randomized
    Payload p;

    function new;
      this.p = new;
    endfunction
  endclass

  initial begin
    header h = new;
    assert(h.randomize());     // randomize address and payload data
    $display(h.addr);          // addr shows random value
    foreach (h.p.data[i])
      $display(h.p.data[i]);   // GOTCHA! payload data is still 0
  end
endprogram
```

The **.randomize** method only randomizes properties in the scope of the object being randomized if the property is declared with a **rand** or **randc** tag. If the property is a handle to another object, the tag must be specified for both the handle and on the properties in the child object. In the example above, header::addr has been tagged, so it gets updated with random values. The payload object, header::p, however, is missing the **rand** modifier, so none of its variables are randomized, even though Payload::data has the **rand** tag.

How to avoid this gotcha: All object variables that are to have random values generated, *including handles*, must have the **rand** modifier.

```
  class Header;
    rand int addr;             // addr is tagged to be randomized
```

```
    rand Payload p;           // payload is tagged to be randomized
    ...
  endclass
```

5.2 Boolean constraints on more than two random variables

Gotcha: *When I specify constraints on more than two random variables, I don't get what I expect.*

Synopsis: *In a series of two Boolean relational operators, the second operation is compared to the true/false result of the previous operation.*

The intent of the constraint in the class below is to randomize `lo`, `med` and `hi` such that `lo` is less then `med` and `med` is less than `hi` by using the expression (`lo < med < hi;`).

```
class bad1;
  rand bit [7:0] lo, med, hi;
  constraint increasing { lo < med < hi; }
endclass
```

A sample output from running the code above looks like this:

```
lo =  20, med = 224, hi = 164
lo = 114, med =  39, hi = 189
lo = 186, med = 148, hi = 161
lo = 214, med = 223, hi = 201
```

This constraint does not cause the solver to fail, but the randomized values are not as expected; `lo` is sometimes greater than `med`, and `med` is sometimes greater than `hi`. The reason that the constraint does not work is because the Boolean less-than expressions are evaluated from left to right. This means that the operation is not comparing `med` to `hi`, it is comparing the true/false result of (`lo < med`) to `hi`. The constraint above is evaluated as:

```
constraint increasing { (lo < med) < hi; }
```

To resolve this constraint, random values for `lo` and `med` are first chosen. The relational less-than operation (**<**) returns either 0 or 1 (representing false or true) based on the values of `lo` and `med`. Note that `lo` has not been constrained to be less than `med`, only to be evaluated in a Boolean expression. The value of `hi` is then constrained to be greater than the result of the test of (`lo < med`). Therefore, the constraint is actually that `hi` has a value greater than 1, and lo and med are unconstrained. *Gotcha!*

The following example illustrates the same type of problem. This constraint is intended to create a values `a`, `b` and `c` such that the three properties have the same value.

```
class bad2;
  rand bit [7:0] a, b, c;
  constraint equal {a == b == c; }
endclass
```

A sample output from running the code above gave the following output:

```
a =  25, b = 173, c =  0
a =  65, b = 151, c =  0
a = 190, b =  33, c =  0
a =  65, b =  32, c =  0
```

A different simulator gives this output:

```
a =  61, b =   1, c =  0
a =   9, b =   9, c =  1
a = 115, b = 222, c =  0
a = 212, b = 212, c =  1
```

In this constraint, `a`, `b` and `c` will only be equal if both `a` and `b` are randomly selected to be the value of 1. The probability of this occurring is $1/2^{16}$, which is nearly 0. This is because the **==** equality operator returns 0 or 1 (for false or true), and the compound expression is evaluated from left to right. The constraint is equivalent to: `(a == b) == c`. Therefore, if the randomly selected values for `a` and `b` are equal, `c` is constrained to be 1, otherwise `c` is constrained to be 0. This is very different from what may have been expected. *Gotcha!*

In should be noted that in both of the examples above, the constraint will not fail. It is always possible to find a set of values that satisfies the constraint. The effect of the incorrectly coded constraints will likely be low verification coverage. Determining why the coverage is low could be very difficult. *Gotcha, again!*

How to avoid this gotcha: Constraints involving compound Boolean operations should be broken down to separate statements. The above constraints should be modeled as:

```
constraint increasing {
  lo < med;       // lo is constrained to be less than med
  med < hi;       // med is constrained to be less than hi
}

constraint equal {
  a == b;         // a is constrained to be equal to b
  b == c;         // b is constrained to be equal to c
}
```

5.3 Undetected randomization failures

Gotcha: *My class variables do not get random values assigned to them, and no errors or warnings are generated.*

Synopsis: *The .randomize returns an error status if a constraint cannot be solved; the error status must be checked.*

It is possible to write constraints that cannot be solved under all conditions. If a constraint can not be met, then the variables are not randomized. The **`.randomize`** method returns a 1 when the constraint solver succeeds in randomizing the class variables, and a 0 if it does not succeed.

The following example erroneously specifies a constraint such that `a` must be less than `b` and `b` must be less than `a`. These randomization failures could go undetected.

```
program test;
  class Bad;
    rand bit [7:0] a, b;
    constraint ab {a < b;
                   b < a;}  // GOTCHA: coding error could go undetected
  endclass
```

```
    initial begin
      Bad b = new;
      b.randomize();  // return value from randomize method ignored
    end
  endprogram
```

If the success flag is not checked, the only symptom when a constraint cannot be solved is that one or more class variables were not randomized. The failure to randomize some class variables could go undetected. *Gotcha!*

How to avoid this gotcha: Use SystemVerilog assertions to check the return status of .randomize. In the example below, an assertion failure is fatal, which will abort simulation or formal verification.

```
  program test;
    class Bad;
      rand bit [7:0] a, b;
      constraint ab {a < b;
                     b < a;}  // GOTCHA: coding error could go undetected
    endclass

    initial begin
      Bad b = new;
      assert(b.randomize()) else $fatal;  // abort if randomize fails
    end
  endprogram
```

5.4 Unwanted negative numbers in random values

Gotcha: *I am getting negative numbers in my random values, where I only wanted positive values.*

Synopsis: *Unconstrained randomization considers all possible 2-state values within a given data type.*

In the following class, both `i` and `b` are signed variables which can store negative values.

```
    class Negs;
      rand int  i;
      rand byte b;
    endclass
```

The **`int`** and **`b`** types are signed types. Therefore, the **`.randomize`** method will generate both positive and negative values for these variables. If either of these variables is used in a context where a positive number is required, the outcome could be unexpected or erroneous.

How to avoid this gotcha: When the randomized test variables are to be passed to hardware as stimulus, it is generally best to use signed types such as **`logic`**. This ensures that randomized values will always be positive values.

There are times when 2-state types need to be used, but only positive numbers are desired. An example of needing the C-like **`int`**, **`byte`**, **`shortint`** and **`longint`** 2-state types is when the variables are passed to C functions using the SystemVerilog DPI. When 2-state types need to be used, randomization can be constrained to non-negative numbers.

```
class Negs;
  rand int  i;
  rand byte addr;
  constraint pos
    { i >= 0;
      b >= 0;}
endclass
```

6.0 SystemVerilog coverage gotchas

6.1 Coverage is always reported as 0% for a cover group

Gotcha: *You defined a cover group, but in the cover report, the group always has 0% coverage.*

Synopsis: *Covergroups are specialized classes and must be constructed before they can be used.*

The following example defines a covergroup as part of a class definition. The intent is to provide coverage of the properties within the class. When the class object is constructed, however, the covergroup does not keep track of information intended.

```
program test;
  event cg_sample;

  class Abc;
    rand bit [7:0] a, b, c;
    covergroup CG @(cg_sample);    // covergroup definition
      coverpoint a;
      coverpoint b;
      coverpoint c;
    endgroup
  endclass

  initial begin
    Abc a1 = new;                  // instance of Abc object
    ...
```

A covergroup is a special type of class definition. In order to generate coverage reports, the covergroup object must first be constructed using the covergroups **`new`** method, in the same way as constructing a class object. The example above constructs an instance of the `Abc` object, but does not construct an instance of the `CG` covergroup. Hence, no coverage information is collected for the `CG` group. No errors or warnings are reported for this coding error. The only indication that there is a problem is an erroneous or incomplete coverage report. *Gotcha!*

How to avoid this gotcha: An instance of a covergroup must always be constructed in order to collect coverage information about that group. When the group is defined in a class, as in the example above, the covergroup instance can be constructed as part of the class's new function. In that way, each time a class object is constructed, the covergroup instance for that object will automatically be constructed.

```
program test;
  event cg_sample;

  class Abc;
    rand bit [7:0] a, b, c;
```

```
    covergroup CG @(cg_sample);    // covergroup definition
      coverpoint a;
      coverpoint b;
      coverpoint c;
    endgroup

    function new;
      CG = new;                    // instance of covergroup CG
    endfunction
  endclass

  initial begin
    Abc a1 = new;                  // instance of Abc object
    ...
```

Another reason why coverage could be reported as 0% is that the cover group was never triggered. This could be because its trigger never fired, or the **.sample** method for the covergroup instance was never called.

6.2 The coverage report lumps all instances together

Gotcha: *I have several instances of a covergroup, but the coverage report lumps them all together.*

Synopsis: *By default, the cover report for a covergroup combines all the instances of a covergroup together.*

The intent in the example below is to measure the coverage on each of two pixel x:y pairs. Two covergroup objects are constructed, px and py. Instead of seeing separate coverage for each covergroup, however, the coverage report combines the counts for both groups into a single coverage total.

```
covergroup pixelProximity(ref bit signed [12:0] pixel1,
                          ref bit signed [12:0] pixel2)
  @(newPixel);
  cp1: coverpoint (pixel2 - pixel1) {
         bins lt = {[1:$]};      // is pixel1's coordinate less than pixel2
         bins eq = {0};          // did the pixels coincide?
         bins gt = {[-4095:-1]}; // is pixel1's coord. greater then pixel2
       }
endgroup

pixelProximity px, py;

initial begin
  bit signed [12:0] x1, y1, x2, y2;
  px = new(x1, y1);      // construct first covergroup
  py = new(x2, y2);      // construct second covergroup
  ...
end
```

How to avoid this gotcha: Use the **.per_instance** option. Put the following statement at the top of the cover group:

```
covergroup pixelProximity(ref bit signed [12:0] pixel1,
```

```
                          ref bit signed [12:0] pixel2)
    @(newPixel);
    option.per_instance = 1;  // report each covergroup instance separately
    cp1: coverpoint (pixel2 - pixel1) {
           bins lt = {[1:$]};       // is pixel1's coordinate less than pixel2
           bins eq = {0};           // did the pixels coincide?
           bins gt = {[-4095:-1]}; // is pixel1's coord. greater then pixel2
         }
  endgroup
```

6.3 Covergroup arguments directions are sticky

Gotcha: *Sometimes my covergroup compiles OK when I construct a covergroup instance, and sometimes it won't compile.*

Synopsis: *A covergroup ref covergroup argument cannot be passed a constant value.*

A generic covergroup has arguments so you can pass in values and variables. The default direction is **`input`**, for passing in fixed values, and **`ref`** for passing in variables for coverpoints. The direction is specified is sticky. It remains in effect until a new direction is specified.

In the following example, `t`he call to the covergroup **`new`** constructor passes in the variable `va` and the constants 0 and 50 for `low` and `high`. The code looks like it should do what is expected, but instead gets a compilation error.

```
covergroup cg (ref int ra, int low, int high )
  @(posedge clk);
  coverpoint ra // sample variable passed by reference
    {bins good = { [low : high] };
    bins bad[] = default;
  }
endgroup

initial begin
  int va, vb;
  int min=0, max=50;
  cg c1 = new(va, min, max);  // OK
  cg c2 = new(vb, 0, 50);     // ERROR: cannot pass constants to ref args
end
```

In the covergroup definition above, `ra` is a **`ref`** argument. This direction is sticky; it affects all arguments that follow until a different direction is specified. Since no direction is given for `low` and `high`, they also default to **`ref`** arguments. The call to the constructor fails because the actual values passed to **`ref`** arguments must be variables; it is not allowed to pass a constant into a **`ref`** argument. The sticky direction behavior of covergroup arguments is similar to task/function arguments, as described in gotcha 3.9.

How to avoid this gotcha: It is best to always specify the direction for every argument to a covergroup.This documents the code intent, and prevents the gotcha of an argument inheriting the direction of a previous argument. In the above example, the first line should be:

```
covergroup cg (ref int ra, input int low, input int high )
  ...
endgroup
```

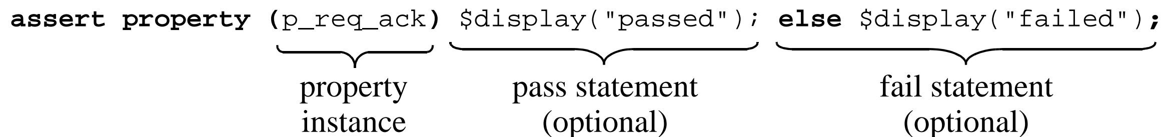
7.0 SVA gotchas

7.1 Assertion pass statements execute with a vacuous success

Gotcha: *My assertion pass statement executed even though I thought the property was not active.*

Synopsis: *A vacuous success will execute the assertion pass statement.*

The **assert property** construct can be followed by optional pass and fail statements.



The optional pass statement is executed if the property succeeds, and the fail statement is executed if the assertion fails. The pass fail statements can be any executable statement. Multiple statements can be executed by grouping them between **begin** and **end**.

Most property specifications contain an *implication operator*, represented by either |-> or =>, which qualifies when the assertion should be run. The sequence expression before the implication operator is called the *antecedent*. The sequence expression after the operator is called the *consequent*. A property specification that uses an implication operator has three possible results: *success*, *failure* and *vacuous success*. If the implication antecedent is true, the consequent is evaluated, and the property will pass or fail based on the results of testing the consequent. If the implication antecedent is false, the consequent is a “don’t care”, and the property returns a vacuous success.

The gotcha is that the **assert property** statement does not distinguish between a real success and a vacuous success. Consider the following code which increments a test counter on each successful assertion. The assertion checks to see if a `req` is followed by `ack` 1 clock cycle later.

```
property p_req_ack;
  @(posedge clk) req |-> ##1 ack;  // if req, check for ack on next cycle
endproperty

assert property (p_req_ack) req_ack_count++;  // GOTCHA
```

The intention of this code is to count of how many times `req` was successfully followed by `ack`. Instead, the assertion counts both how many times `req` was followed by `ack` (successes) *and* how many clock cycles in which there was no `req` (vacuous successes). *Gotcha!*

How to avoid this gotcha: There are no easy solutions to avoiding this gotcha using the SystemVerilog-2005 standard. Generally speaking, the best solution is to not use pass statements with assertions. In the example above, counting how many times `req` was followed by `ack` could, and probably should, be done using a SystemVerilog coverage definitions instead of the assertion pass statement.

This gotcha has been addressed in the next version of the SystemVerilog standard, planned for late 2008. The IEEE 1800 SystemVerilog standards committee has proposed new system tasks to control the execution of assertion pass statements: **$assertvacuousoff** and **$assertvacuouson** These system tasks will allow a designer to disable or enable the assertion pass statements on

vacuous successes.

7.2 Concurrent assertions in procedural blocks

Gotcha: *My assertion pass statements are executing even when the procedural code does not execute the assertion.*

Synopsis: Concurrent assertions in procedural code actually fire every clock cycle, not just when the procedural code executes.

A concurrent assertion can be placed inside an **initial** or **always** block and guarded by procedural code, such as an **if** statement.

```
property p_req_ack;
  @(posedge clk) req |-> ##[1:3] ack;
endproperty

always_ff @(posedge clk) begin
  if (state == FETCH)
    assert property (p_req_ack) req_ack_count++;
  ...
end
```

This example illustrates two gotchas: the assertion fires every clock cycle instead of just when the procedural code executes the assert property statement, and a vacuous success can occur even when the property sequence should fail.

Concurrent assertions in procedural code are still concurrent assertions, which mean they run as concurrent threads, in parallel with the procedural block. Because the assertion is a concurrent assertion, it executes on every positive edge of `clk`, even when the **if** condition is false. *Gotcha!*

The **if** condition that guards the assertion in the procedural code becomes an implicit implication operation in the property. When the **if** condition is false on a clock edge, the property is a vacuous success, and the assertion will execute its pass statement, regardless of the state or condition of the property. The table below summarizes the different conditions for the assertion above.

```
state  req  ack  ap1
INIT    0    0   vacuous success
INIT    0    1   vacuous success
INIT    1    0   vacuous success  // GOTCHA - the property is false here
INIT    1    1   vacuous success

FETCH   0    0   vacuous success
FETCH   0    1   vacuous success
FETCH   1    0   assert failure
FETCH   1    1   assert pass (real success)
```

The implication operator in the property will cause a vacuous success on each cycle when `req` is 0. But, in line 3 of the table above, `req` is 1 but is not followed by `ack`. This should be an assertion failure, but is reported as a vacuous success. *Gotcha, again!*

The reason is that the procedural **if** statement implies implication operation. When the **if** condition is false, the property is a vacuous success, even if `req` is not followed by an `ack`.

How to avoid this gotcha: Do not have pass statements with concurrent assertions that are called from procedural code that is guarded by a conditional statement. The **$assertvacuouson** (see gotcha 7.1) might also resolve this case, depending on final implementation.

7.3 Mismatch in assert...else statements

Gotcha: *My assertion fail statement executes when the assertion succeeds instead of fails.*

Synopsis: *An "if" without an "else" in an assert pass statement causes the assert "else" (fail) statement to be paired with the "if" in the pass statement.*

The **assert** and **assert property** constructs can be followed by optional pass and fail statements.

```
assert property (p_req_ack) $display("passed"); else $display("failed");
```

property instance — pass statement (optional) — fail statement (optional)

The optional pass statement is executed if the property succeeds, and the optional fail statement is executed if the assertion fails. The pass fail statements can be any executable statement.

The pass statement can be any executable statement, including a conditional **if** or **if...else** statement. The following example has a gotcha:

```
assert property (p_req_ack)
  if (cnt_en) req_ack_count++; // assertion pass statement
else $fatal;                   // GOTCHA — not the assertion fail statement
```

Assertions follow the same syntax as nested **if...else** statements, in that the **else** is associated with the nearest **if**. In the example above, the **else** statement is associated with the **if** condition in the assertion pass statement. Syntactically, there is no assertion fail statement in this example. This is not a syntax error, since the fail statement is optional. Instead the **else** branch executes whenever the assertion succeeds or vacuously succeeds, and `cnt_en` is false. *Gotcha!*

How to avoid this gotcha:

How to avoid this gotcha: When an **if** condition is used in an assertion pass statement, then the code must follow the coding rules of nested **if** statements. Either an **else** must be paired with the **if,** or the **if** condition must be encapsulated within a **begin**...**end** statement group, as shown below.

```
assert property (p_req_ack)
  begin
    if (cnt_en) req_ack_count++;  // assertion pass statement
  end
else $fatal;                      // assertion fail statement
```

8.0 Tool compatibility gotchas

8.1 Default simulation time units and precision

Gotcha: *My design outputs do not change at the same time in different simulators.*

Synopsis: *Simulators have different defaults for delay time units (the 'timescale directive).*

Time in Verilog is a 64-bit unsigned integer. Delays are specified using a hash mark (#) followed by as a number. A delay does not have any indication of what increment of time is being represented.

```
#2 sum = a + b;     // delayed execution of a programming statement

and #3 (y, a, b);  // 2-input AND gate with propagation delay
```

In Verilog, the time unit represented by delays is specified as a characteristic of a module, using a **`'timescale`** compiler directive. The directive contains two parts, the module's *time units* and the module's *time precision*. Each are specified in increments of 1, 10 or 100, in units ranging from seconds down to femtoseconds. The time precision allows a module to represent non-whole delays. The precision is relative to the time unit. Within simulation, all delays are scaled to the smallest precision used by the design. An example of using **`'timescale`** is:

```
'timescale 1ns/100ps
module A (...);
  #2.3 ...         // delay represents 2.3 nanoseconds
endmodule

module B (...);
  #5.5 ...         // GOTCHA: delay represents 5.5 what???
endmodule

'timescale 1ps/1ps
module C (...);
  #7 ...         // delay represents 7 picoseconds
endmodule
```

There are two common gotchas with Verilog **`'timescale`** directive, file order dependencies and no standard default.

The **`'timescale`** directive is not bound to modules or files. Once specified, the directive affects all modules and files that follow the directive, until a new **`'timescale`** is encountered by the compiler. This means if some design and/or test files contain time scale directives, and other files do not, then changing the order the in which files are compiled will change the what a delay represents in the files that do not have a time scale directive. This can cause radically different simulation results, even with the same simulator. *Gotcha!*

The IEEE Verilog standard specifically states that if a file is read in when no **`'timescale`** has been specified at all, then a compiler might, or might not, apply a default time unit. This can cause radically different simulation results when simulating the same design on different simulators. *Gotcha!*

How to avoid this gotcha: To avoid this gotcha when using just Verilog, strong coding rules must be adhered to. One recommendation is to make sure a **`'timescale`** directive is specified at the beginning of each and every module, in each and every design or testbench file.

SystemVerilog has two very important enhancements that help avoid the gotchas inherent with the **`'timescale`** directive. The time unit and time precision specification have been made keywords that can be specified within a module, and local to just the module. They can also be specified within interfaces, programs and packages. This eliminates file order dependency problems.

Second, SystemVerilog allows an explicit time unit to be specified with a delay value. this both documents the intended time unit, and prevents dependency on what order **`'timescale`** directives were encountered by the compiler.

```
module B (...);
  timeunit 1ns;
  timeprecision 1ps;
  #5.5 ...        // delay represents 5.5 nanoseconds
  #1ms ...        // delay represents 1 millisecond
endmodule
```

8.2 Package chaining

Gotcha: *My packages compile fine on all simulators, but my design that uses the packages will only compile on some simulators.*

Synopsis: *When one package imports another package, and a design or testbench file imports the second package, some simulators make declarations from both packages available, and some do not.*

SystemVerilog packages provide a declarations space for definitions that are to be shared. A design block or testbench block can import specific package items, or do a wildcard import, making all items in the package visible. A package can also import items from other packages, as illustrated below.

```
package foo;
  typedef int unsigned uint_t;
  function int func_a (int a);
    return ~a;
  endfunction
endpackage

package bar;
  import foo::func_a;               // specific import
  import foo::*;                    // wildcard import
  function int func_b (uint_t b);
    return ~func_a(b);
  endfunction
endpackage

module test;
  import bar::*;                    // wildcard import bar, chain foo
  uint_t c;                         // reference definition in foo
  initial begin
    $display("func_a(5)=%0d", func_a(5)); // from package foo
    $display("func_b(5)=%0d", func_b(5)); // from package bar
    $finish;
  end
endmodule
```

The example above illustrates a gotcha. The `test` module does a wildcard import of package `bar`, and then references the `uint_t` and `func_a` definitions. These definitions were not defined in package bar, however. They were imported into package `bar` from package `foo`. Some software

tools permit package chaining in this form, and some simulators do not.

This example compiles with VCS, but will not compile on some other simulators. *Gotcha!*

How to avoid this gotcha: The gotcha in the example above is a result of an ambiguity in the SystemVerilog-2005 standard. The standard does not say whether package chaining is, or is not, allowed. To avoid this gotcha and ensure that design and verification code will work on all software tools, package chaining should not be used. Instead, a design or verification block should explicitly import each package that contains definitions used in the module. Either specific object imports or wildcard imports can be used, so long as each package that is used is explicitly referenced.

```
module test;
import foo::*;                  // wildcard import foo
import bar::*;                  // wildcard import bar
...
```

The IEEE SystemVerilog standard working group has addressed this ambiguity in the LRM, and has proposed a change for the next version of the SystemVerilog standard. The change is to make implicit package chaining illegal, and to provide a mechanism for explicit package chaining. When tools implement this proposed change, the example illustrated at the beginning of this section, which uses implicit package chaining, will be illegal (which means VCS will be wrong). However, package `bar` can enable chaining by importing definitions from package `foo`, and then exporting some or all of those definitions, thus making them visible to any design or testbench blocks, or other packages, that import bar.

8.3 Non-standard keywords

Gotcha: *My SystemVerilog code only run on one vendor's tools.*

Synopsis: *Some tools add proprietary keywords to the IEEE standard's official reserved keywords.*

Synopsys allows an optional keyword **`hard`** to be used with the **`solve`**...**`before`** constraint operator. Without this additional keyword, VCS (and presumably other Synopsys tools) do not enforce the constraint solution order that is specified by **`solve`**...**`before`**.

```
constraint ab {
  `ifdef SNPS
    solve a before b hard;  // 'hard' enforces solve before
  `else
    solve a before b;
  `endif
  if (a inside {32, 64, 128, 256})
    a == b ;
  else
  a > b;
}
```

The keyword **`hard`** is *not* a SystemVerilog keyword, and is not in the IEEE 1800-2005 SystemVerilog standard. If **`hard`** is used with any vendor's tool other than Synopsys tools, a syntax error will result. *Gotcha!*

Not to be outdone, another EDA vendor allows an illegal keyword pair, **`pure virtual`**, to be used in the declaration of class methods. This vendor supplies verification libraries with this illegal construct. Testbenches written with this illegal keyword pair might not compile in other tools.

How to avoid this gotcha: Using non-standard keywords or syntax might be necessary to get the desired results in a specific product. In VCS, for example, the **`hard`** keyword is needed to get a desired constraint solution order. However, specifying this keyword will prevent the same verification code from working with other software tools. To avoid this gotcha, conditional compilation must be used in order to control whether or not the **`hard`** keyword (or some other non-standard construct) is compiled.

```
constraint ab {
  `ifdef VCS
    solve a before b hard;  // add Synopsys 'hard' specification
  `else
    solve a before b;
  `endif
  if (a inside {32, 64, 128, 256})
    a == b ;
  else
  a > b;
}
```

In the example above, the macro name `VCS` is predefined in the VCS simulator, and does not need to be set by the user. If a user-defined macro name is used, then it must be set somewhere in the design or testbench source code, or on the tool command line.

It should be noted that the **`pure virtual`** construct is illegal in the SystemVerilog-2005 syntax, but a proposal has been approved by the IEEE SystemVerilog standards group to add this to the next version of the IEEE SystemVerilog standard, along with its syntax and semantics. At the time this paper was written, there was no proposal to add the **`hard`** keyword.

8.4 Array literals versus concatenations

Gotcha: *Some tools require one syntax for array literals; other tools require a different syntax.*

Synopsis: *Array literals and structure literals are enclosed between the tokens* **`'{`** *and* **`}`***, but an early draft of the SystemVerilog standard used* **`{`** *and* **`}`** *(without the apostrophe).*

The Verilog concatenation operator joins one more values and signals into a single vector. The array and structure literals are a list of one or more individual values. To make the difference between these constructs obvious (to both engineers and software tools), the syntax for an array or structural literal is different than a Verilog concatenation. That difference is that the array or structure literal list of separate values is preceded by an apostrophe.

```
reg [7:0] data_bus = {4'hF, bus};   // concatenate values into a vector

int data [4] = '{0, 1, 2, 3};       // list of separate values

struct {
  int a, b;
  logic [3:0] opcode;
} instruction_word = '{7, 5, 3};    // list of separate values
```

The literal values can be assigned at the time of the array or structure declaration (as shown below) or any time during simulation as an assignment statement. Multidimensional arrays use nested sets of array literals, as in:

```
int twoD[2][3] = '{ '{0, 1, 2}, '{3, 4, 5} };
```

A nested array literal can also be replicated, similar to Verilog's replicated concatenation.

```
int twoD[2][3] = '{ '{0, 1, 2}, '{3{4}} };
```

There are three gotchas with array and structure literals:

- The difference in syntax between an array literal and a concatenation is subtle and easy to inadvertently use the wrong construct.
- A concatenation must have fixed size values in its list; an array or structure literal can have both fixed size and unsized values in its list.
- An early, non-IEEE preliminary draft of the proposed SystemVerilog standard, known as SystemVerilog 3.1a [6], used curly braces for both concatenations and array/structure literals.

At the time this paper was written, some software tools supported the preliminary syntax, and some tools required the final syntax. As of the 2006.06-6 release of VCS, the official IEEE 1800 construct is supported. As of early 2007, however, DC, LEDA, Magellan and Formality required the SystemVerilog 3.1a preliminary syntax for array literals, and reported a fatal syntax error for the correct IEEE 1800 SystemVerilog syntax. Tools from other EDA vendors, at the time this paper was wriiten, also have mixed support; some tools require the official IEEE syntax, and some require the preliminary SystemVerilog 3.1a syntax.

How to avoid this gotcha: Engineers must learn the difference in syntax between concatenations and array literals. The different tokens help indicate when an list values is intended to represent a list of separate values (an array or structure literal) and when a list of values is intended to represent a single value (a concatenation).

The gotcha of some tools requiring an old, non-IEEE syntax cannot be avoided. A work around is to use conditional compilation around statements containing array or structure literals, to allow the model to be compiled with either the SystemVerilog 3.1a syntax or the IEEE 1800 syntax.

8.5 Module ports that pass floating point values (real types)

Gotcha: *Some tools allow me to declare my input ports as real (floating point), but other tools do not.*

Synopsis: *Module output ports that pass floating values are declared as* **`real`***, but module input ports that pass floating point values must be declared as* **`var real`**.

SystemVerilog allows floating point values to be passed through ports. However, the official IEEE syntax is not intuitive. An **`output`** port of a module can be declared as a **`real`** (double precision) or **`shortreal`** (single precision) type, but **`input`** ports must be declared with a keyword pair, **`var real`** or **`var shortreal`**.

```
module fp_adder
(output real       result,
 input  var real   a, b
```

```
);
  always @(a, b)
    result = a + b;
endmodule
```

An early, non-IEEE preliminary draft of the proposed SystemVerilog standard, known as SystemVerilog 3.1a [6], did not require the **`var`** keyword be used on **`input`** floating point ports. At the time this paper was written, some SystemVerilog tools require the official IEEE syntax, as shown above, and get an error if the **`var`** keyword is omitted. Other tools, however, require the preliminary SystemVerilog 3.1a syntax, and get an error if the **`var`** keyword is used. The Synopsys VCS 2006.06 is one of the tools that does not support the official IEEE syntax. Designers are forced to write two versions of any models that have floating point input ports. *Gotcha!*

How to avoid this gotcha: There are two gotchas to be avoided:

- The official IEEE syntax in not intuitive

  Floating point **`input`** ports are declared differently than floating point **`output`** ports. This inconsistency can be confusing to engineers who are not familiar with the syntax. Unfortunately, this gotcha cannot be avoided. Engineers need to know the correct declaration syntax.

- Not all tools support the official IEEE syntax

  This gotcha also cannot be avoided. The only work around is to use conditional compilation around the module port declarations, to allow the same model to be compiled with either the obsolete, unofficial declaration style or with the official IEEE 1800 declaration style.

9.0 Corrections to the first gotchas paper

Part One of the paper was presented at the SNUG 2006 conference held in Boston, Massachusetts. The paper title is *"Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know"*[1]. Two erratum have been pointed out in the published version of this paper.

Gotcha 2.12, "Importing from multiple packages"

Published text:

> *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.

Correct text:

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

Gotcha 5.1, "Self-determined operations versus context-determined operations"

Published text in Table 1, Row 3, Column 1:

> Assignment operations
> `**=  <<=  >>=  <<<=  >>>=`

Correct text in Table 1, Row 3, Column 1:

Assignment operations

`<<=    >>=    <<<=    >>>=`

10.0 References and resources

[1] *"Standard Gotchas: Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know"*, by Stuart Sutherland and Don Mills. Published in the proceedings of SNUG Boston, 2006. Available at *www.sung-universal.org* or *www.sutherland-hdl.com/papers*.

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

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

[4] "*SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling, Second Edition", by* Stuart Sutherland, Simon Davidmann and Peter Flake. Published by Springer, Boston, MA, 2006, ISBN: 0-387-33399-1.

[5] "*SystemVerilog for Verification", by* Chris Spear. Published by Springer, Boston, MA, 2006, ISBN: 0-387-27036-1.

[6] "*SystemVerilog 3.1a Language Reference Manual Accellera's Extensions to Verilog*", Copyright 2004 by Accellera Organization, Inc., Napa, CA, http://www.eda.org/sv/SystemVerilog_3.1a.pdf.

11.0 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-hdl.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.

Mr. Chris Spear is a verification consultant for Synopsys, Inc., and has advised companies around the world on testbench methodology. He is the author of the EDA best seller, *"SystemVerilog for Verification"*, and the File I/O package for Verilog. You can contact Chris at Chris@Spear.net.