Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SystemVerilog Assertions

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

ABSTRACT

There are engineers (but not you once you read this paper) who say that assertions are for verification that should only be written by verification engineers and bound, rather than embedded, into RTL design code. This misconception short changes the benefits of using assertions! While it is true that verification engineers should use assertions to their full potential, there are many simple SystemVerilog Assertions that designers can—and should—be adding directly into the RTL code as it is written. This paper discusses where embedded RTL assertions can be useful, and the advantages of these assertions. Design engineers, verification engineers and engineering managers need to be aware of the benefits of embedded RTL assertions. Should you be under the misconception that it is too much effort for designers to write assertions — or that this is "old hat" information that everyone knows — then you will want to read this paper!

Target audience: Engineers involved in RTL design and synthesis, targeting ASIC and FPGA implementations.

Note: The information in this paper is based on Synopsys ***VCS*** version 2014.12, Synopsys ***Design Compiler*** (also called ***HDL Compiler***) version 2014.09-SP5, and Synopsys ***Synplify-Pro*** version 2014.09-SP2. These were the most current released versions available to the author at the time this paper was written.

Table of Contents

List of Figures

List of Tables

List of Examples

1.0 Introduction — debunking the SystemVerilog Assertions myth

As a provider of SystemVerilog training and as a design and verification consultant, I have seen how Verilog and SystemVerilog are used at a wide variety of companies in many parts of the world. I have worked with small and large engineering groups, ASIC, FPGA and full-custom projects, and designs that are data-oriented, control-oriented and processor-oriented. I have worked with companies where the design engineers and verification engineers are one and the same, and where design teams and verification teams are completely separate groups of engineers, located on opposite sides of the world.

This broad perspective of how designs are created and verified is what has lead to this paper. I have worked with companies where both design engineers and verification engineers take full advantage of SystemVerilog Assertions, and companies who do not use assertions at all. Most often, I have seen verification teams use a limited number of assertions, but design engineers never write assertions — and don't want the verification team modifying their RTL code by adding assertions to that code. More importantly, I have seen assertions work! Assertions really do find bugs in design code that stimulus-based testing might not reveal, and help to ensure that designs are fully verified.

The intent of this paper is to encourage design teams to rely more on the power of SystemVerilog Assertions. There are definite advantages to having both design engineers and verification engineers writing assertions, but each team needs to focus on the right types of assertions, in order to obtain the full benefit of assertions. In this paper, we will explore the types of SystemVerilog assertions that can, and should be, written by RTL designers as the code is being defined. We will also explore taking advantage of SystemVerilog constructs that have built-in assertion-like checking. Finally, we will look at the use of assertions in conjunction with simulator specific capabilities, such as the VCS Xprop option that reduces the inherent logic X optimism of the SystemVerilog language.

2.0 Types of SystemVerilog Assertions

SystemVerilog provides two types of assertion constructs, *immediate assertions* and *concurrent assertions*. As the names imply, an immediate assertion executes in zero simulation time, whereas a concurrent assertion runs in parallel (concurrently) with other processes.

Both immediate and concurrent assertions perform a test on some aspect of the design. At the completion of the test, pass or fail statements can be executed. The severity level (importance) of a pass or fail message can be set by calling special reporting system tasks:

- **`$fatal`** — run-time fatal
- **`$error`** — run-time error
- **`$warning`** — run-time warning
- **`$info`** — run-time informational message

Each of these severity level system tasks will print a tool-specific message, and may also include user comments using the same syntax as the Verilog `$display` system task. It is optional to specify a pass statement. If not specified, no action is taken when an assertion passes. Specifying a fail statement is also optional. If not explicitly specified, the default for assertion failures is `$error`, with a tool generated message.

2.1 Immediate assertions

The SystemVerilog *immediate assertion* is similar to an **`if`**...**`else`** statement, though there are differences.

```
always @(posedge clock or negedge resetN)
  if (!resetN) q <= 0;
  else begin
    assert (!$isunknown(data)) else $error("data has unknown bits");
    q <= data;
  end
```

Example 1 — Immediate assertion with an optional fail statement

The **assert**...**else** immediate assertion is similar to an **if**...**else**, in that it executes as a programming statement at the moment in simulation time the statement is encountered (every positive edge of `clock` when `resetN` is high, in the example above). This immediate execution is an "*active event*" in the simulation event scheduling, which means processing the statement does not consume any simulation time, including simulation delta cycles. The assert expression (`data`, in this example) is evaluated at the moment the statement is executed, and the **else** branch statement is processed immediately, if the expression evaluates as false.

There are four important characteristics that make an **assert**...**else** different than an **if**...**else** statement:

1. The "*pass statement*", which is executed when the expression evaluates as true, is optional for an **assert**...**else**, whereas a pass statement is required for an **if**...**else**.
2. The **else** clause with its "*fail statement*" is optional for both **assert**...**else** and **if**...**else**, but an **assert** without an **else** clause will generate an automatic error message whenever the expression evaluates as false or unknown, whereas an **if** without an **else** does nothing if the expression is false or unknown.
3. An **assert**...**else** statement can be turned off using assertion controls, whereas an **if**...**else** statement cannot be turned off. For example, assertions can be turned off whenever a chip enters low-power mode, and turned back on when not in that mode. Assertion controls are discussed in section 2.5.
4. Synthesis compilers ignore immediate assertions, whereas synthesis compilers always parse and attempt to implement **if**...**else** statements. To embed an error-check using **if**...**else** decisions in RTL code requires hiding the statement from synthesis using conditional compilation.

2.2 Concurrent assertions

A SystemVerilog *concurrent assertion* runs as a background process throughout simulation. Concurrent assertions are analogous to a continuous assignment statement in that both constructs start running at simulation time 0, and run continuously until simulation ends. Concurrent assertions differ considerably from a continuous assignment, however, because they are not assigning values, and because they are cycle based (evaluations only take place once each clock cycle). Continuous assignments, on the other hand, do assign values, and are event based (evaluations occur whenever an expression on the right-hand side changes).

Concurrent assertions use a keyword pair, **assert property()**, perform a pass/fail test on a "*property specification*", and execute an optional *pass statement* or optional **else** *fail statement*. A property specification consists of a cycle definition, and expressions that are evaluated over a sequence of clock cycles. A special `##` token is used to represent a *cycle delay.* As with immediate assertions, no action is taken if the assertion passes and no pass statement is specified, and an automatic error message is generated if the assertion fails and no fail statement is specified.

The following example, Example 2, illustrates a simple concurrent assertion that verifies each request (`req`) is followed by an acknowledge (`ack`) within 1 to 3 clock cycles, where a cycle is from one positive edge of `d_clock` to the next positive edge of `d_clock`.

```
assert property (@(posedge clk) $rose(req) |-> ##2 $rose(ack))
else $error("acknowledge did not occur in 2 clock cycles");
```

Example 2 — A simple concurrent assertion

Concurrent assertion implication operators. SystemVerilog provides an extensive set of property operators and sequence operators that can be used in concurrent assertions. The purpose and usage of these special operators, and specifying complete property expressions, is beyond the scope of this paper. There is a pair of operators that are critical for concurrent assertions, however, that every engineer should understand, and are, therefore, explained in this paper. These are the **`|->`** and **`|=>`** *implication operators*. Concurrent assertions run as a background process throughout simulation. A new evaluation thread is started every clock cycle. Each thread can run for many clock cycles, continuing until either the entire sequence of expressions completes successfully, or an expression evaluates as false on some cycle. Because a new thread is started each and every clock cycle, the first expression in the property is re-evaluated every clock cycle. Without an implication operator, the assertion would fail every clock cycle in which this first expression is false. This is illustrated in the following simple assertion and waveform.

```
assert property (@(posedge clk) req ##2 ack;);
```

Example 3 — A concurrent assertion without an implication operator

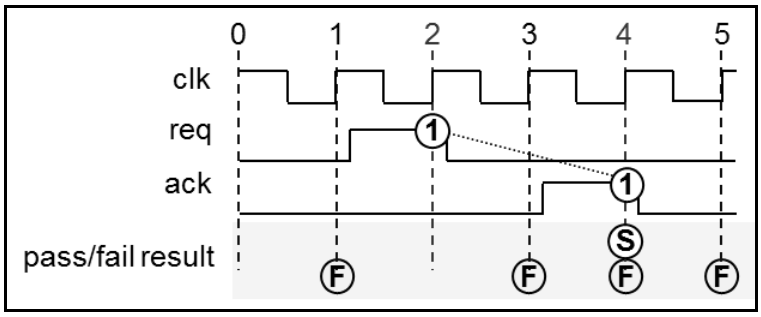

Figure 1. Pass/fail results from a concurrent assertion without an implication operator

In this diagram:

- A circle containing numbers represents the start of a new evaluation thread. The number in the circle is used to distinguish one thread from another.
- A circle containing the letter S represents the successful completion of a thread
- A circle containing the letter F represents the unsuccessful completion of a thread (an assertion Fail).

The concurrent assertion shown in Example 3 above fires on the second clock cycle, and looks to see if `req` is a 1. This evaluation is true, and so that assertion thread, shown as thread 1 in Figure 1, continues, and checks for `ack` two clock cycles later. This evaluation succeeds on the fourth clock cycle, and so thread 1 passes.

However, concurrent assertions start a new evaluation thread every clock cycle. The failure on the first clock cycle is because new thread was started on this clock cycle that checks to see if `req` is 1. Since it was not, this thread fails immediately. This is referred to as a *false failure*, where the design itself is working correctly, but the assertion reports a failure. A similar false failure occurs for the new threads that start on the 3rd, 4th and 5th clock cycles.

The implication operators turn these false failures into don't care conditions. For the design specification, *"every request must be followed by an acknowledge 3 clock cycles later"*, we only need to look for an

acknowledge once a request has occurred. For every clock cycle in which there is no request, we don't need to check for an acknowledge. The expression before an implication operator is referred to as the *antecedent*, and the expression after the operator is the *consequent*. When the antecedent evaluates as true, that assertion thread continues to run in the background, checking that the consequent is also true. When the antecedent evaluates as false, that thread is aborted, without checking the consequent. These aborted threads are reported as a "*vacuous success*", indicating the assertion neither passed nor failed.

The following example uses a **`|->`** implication operator so that each clock cycle in which there was no rising edge of `req`, the assertion aborted as a vacuous success rather than failing (shown with circle containing the letter V).

```
assert property (@(posedge clk) req |-> ##2 ack;);
```

Example 4 — A concurrent assertion with an implication operator

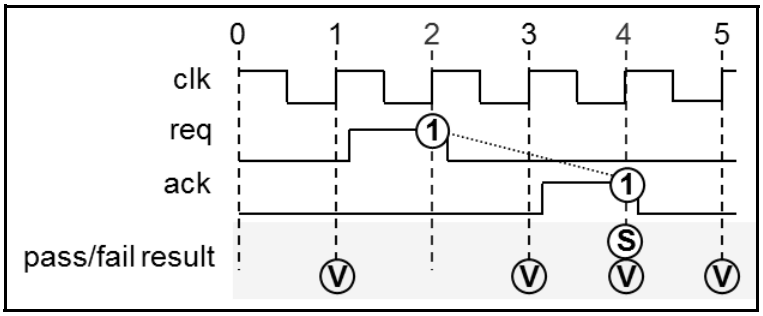

Figure 2. Pass/fail results from a concurrent assertion with an implication operator

There are two forms of the implication operator, **`|->`** and **`|=>`**. The former token is called an "*overlapped implication*". When the antecedent evaluates as true, the evaluation of the consequent begins on the same clock cycle. The latter token is called a "*non-overlapped implication*". When the antecedent evaluates as true, the evaluation of the consequent begins on the next clock cycle. Both implication operators are widely used. Which one to use depends on the requirements of the functionality being verified.

Level sensitive and transition sensitive concurrent assertions. Concurrent assertions are cycle based, meaning they only sample values on a specified clock edge. On that edge, a concurrent assertion can sample for a logic level or logic value. Example 5 illustrates an assertion that tests for a logic value of 1 on the req and ack signals (assuming these signals are 1-bit wide).

```
assert property (@(posedge clk) req |-> ##2 ack;);
```

Example 5 — A concurrent assertion that tests for logic levels

Figure 3 illustrates values for `req` and `ack` that will pass this assertion.

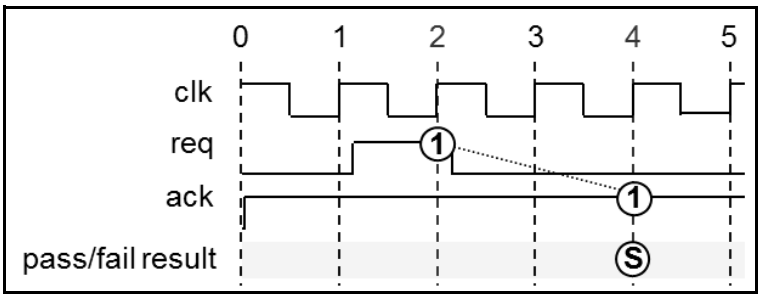

Figure 3. Pass/fail results from a concurrent assertion that tests for logic levels

Testing for logic levels or values can be the correct assertions, but can also lead to an assertion passing when it should not have passed. An assertion that passes when it should not have is referred to as a *false pass*. An important advantage of having design engineers write some types of assertions is that it forces engineers to look closely at the design specification, and to clarify ambiguities in the specification. If the specificated states that ack can be tied high, then the assertion in Example 5 is a good assertion. On the other hand, if the specification states that a rising edge of req must be followed by a rising edge of ack, then Example 5 will generate false passes.

SystemVerilog assertions can also check for transitions on signals using special system functions: `$rose`, `$fell`, and `$changed`. It is important to understand that concurrent assertions only sample values on a specified clock edge. These functions are not triggers that are watching for transitions. The functions compare the value on the previous sample point with the current sample point. For example, `$rose(ack)` returns true if the previous sampled value of `ack` was 0 and the current sample value is 1, which indicates that `ack` had a rising edge sometime within the past clock cycle. Example 6 shows an assertion that requires transitions on `req` and `ack`.

```
assert property (@(posedge clk) $rose(req) |-> ##2 $rose(ack););
```

Example 6 — A concurrent assertion that tests for logic transitions

Concurrent assertion property blocks and sequence blocks. The *property specification* of a concurrent assertion can be directly coded in the **`assert property()`**, as shown in Examples 3 and 4 above, or it can be coded separately in a *property block*, which is encapsulated between the keywords **`proptery`**...**`endproperty`**.

```
property p_a_to_b (clk, a, b, min, max);
  @(posedge clk)
  $rose(a) |-> ##[min:max] $rose(b);
endproperty: p_a_to_b

assert property (p_a_to_b(d_clock, req, ack, 1, 3)) else $error("...");
```

Example 7 — A concurrent assertion using a property block

A complex property expression can be broken into smaller sequence building blocks, specified between **`sequence`** and **`endsequence`**. This is illustrated in the following example.

```
sequence start_bus_xfr (a, b);
  first_match($rose(a) ##1 $rose(b));
endsequence: start_bus_xfr

sequence end_bus_xfr (a, b);
  first_match($fell(a) ##1 $fell(b));
endsequence: end_bus_xfr

property p_bus_xfr (clk, a, b, xfr_size);
  @(posedge clk)
  start_bus_xfr(a, b) |-> ##xfr_size  end_bus_xfr(b, a);
endproperty: p_bus_xfr

assert property (p_bus_xfr(d_clock, req, ack, xfr_size)) else $error("...");
```

Example 8 — A concurrent assertion using sequence blocks

Example 8, as shown here, is not an effective assertion, but the example suffices to show using sequences

as re-usable building blocks in concurrent assertions. There are many more constructs in the SystemVerilog Assertions language which are not discussed in this paper. The purpose of this overview has been to help understand the examples and explanations of assertions in this paper.

Pros and cons of immediate and concurrent assertions.

There are several important differences between immediate assertions and concurrent assertions. These differences are summarized in Table 1.

Immediate Assertions	Concurrent Assertions
Pros:	**Pros:**
Execute as a programming statement	Run as a background task
Can evaluate asynchronously within a clock cycle	Cycle based — not sensitive to glitches
Can be embedded directly within code being verified	Can use assertion binding
Can help document expected code behavior	Works with simulation and formal verification
Cons:	**Cons:**
Can only use SystemVerilog expressions and operators that will evaluate as True or False	More difficult to define (and debug)
Cannot be directly bound into RTL code	Can be far away from code being tested
Difficult to disable during reset or low-power	Cannot detect glitches within a clock cycle
Must follow good RTL coding guidelines to avoid potential read/write race conditions	

Table 1: Immediate and concurrent assertion pros and cons

One of the most important differences between immediate assertions and concurrent assertions is how the handle glitches within a clock cycle. Immediate assertions are programming statements that execute in-line with other programming statements. As such, immediate assertions can evaluate and execute anytime within a clock cycle. Concurrent assertions run as a background process and are cycle based. Concurrent assertions only sample values on their specified clock edge, and cannot evaluate or execute between clock cycles. Figure 4 illustrates an opcode that glitches within a clock cycle.

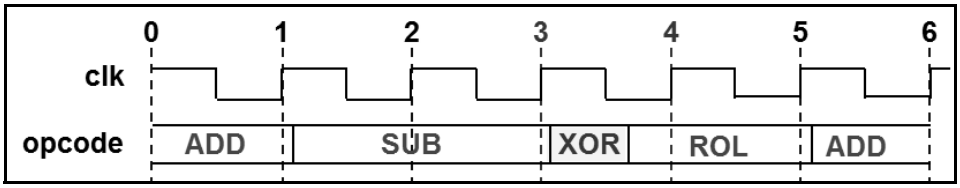

Figure 4. A signal that has a glitch within a clock cycle

The design or verification engineer needs to understand the hardware design, and use the type of assertion that matches the design behavior. If this glitch on opcode could propagate through other logic and cause possible problems in the design, then an immediate assertion should be used to detect the glitch. On the other hand, if the opcode is a registered signal, then only its value at the register's clock edge is important.

A glitch within a clock cycle would not affect the hardware behavior and can be ignored. In this case, a concurrent assertion that only evaluates values on a clock edge should be used.

Another major difference between immediate assertions and concurrent assertions involves SystemVerilog's binding mechanism. This difference is discussed in Sections 2.3 and 2.4.

2.3 Assertions embedded in RTL designs

Both immediate and concurrent assertions can be coded directly within RTL design *modules* and design *interfaces*.

Immediate assertions are programming statements, and can only be placed in procedural code. For synthesizable RTL models, procedural code is limited to *always procedures* (the keywords **`always`**, **`always_ff`**, **`always_comb`** and **`always_latch`**), *functions* and *tasks*. Immediate assertions can be used, and are beneficial, in any of these contexts.

Concurrent assertions can be declared either outside of any procedural blocks (**`always`**, **`always_ff`**, **`always_comb`**, **`always_latch`**, **`task`** and **`function`**), or within an always procedure (but not a task or function). An **`assert property`** statement outside of a procedure is called a *declarative concurrent assertion*. An **`assert property`** statement declared within a sequential logic always procedure is called a *procedural concurrent assertion*. Both styles behave the same — the assertion runs as a background process throughout simulation, starting at simulation time 0. The only real difference is that a declarative concurrent assertion (declared outside of any procedure) must have a cycle definition. A procedural concurrent assertion defined within a sequential logic always procedure can infer its clock cycle from the context of the procedure.

The author's experience has been that both immediate assertions and declarative concurrent assertions can be useful as assertions embedded directly into RTL models. Immediate assertions are the most useful for design engineers, whereas concurrent assertions are the most useful for verification engineers. The reason for this will become apparent in Section 3 of this paper.

2.4 Assertions bound into RTL designs

Some companies have strict file control policies that require engineers to have specific rights to modify project files. A common policy is that only engineers on the design team can modify RTL design files, as these engineers are responsible for synthesizing the RTL models. Synthesis is as much an art form as a science. Subtle changes to RTL code can significantly impact the quality of synthesis results. Should a verification engineer modify the RTL code to add assertions directly in the RTL code, there is a risk of the verification engineer inadvertently affecting the quality of the synthesis results for the model. For reasons such as this, some companies restrict access to RTL design files such that verification engineers can read, but not modify, these files.

This rights restriction, however, limits the ability of verification engineers to use assertions to verify functionality involving signals that are internal to design modules and interfaces. To alleviate this limitation, SystemVerilog provides a binding mechanism that allows modules containing declarative concurrent assertions to be bound to RTL modules and interfaces. The binding mechanism allows the concurrent assertion to access all signals within that module, as if the assertion was embedded within the module. This binding allows verification engineers to add assertions to the RTL code, without having to check out and modify files that are under the control of the design team. This paper does not discuss or give examples of assertion binding. The focus of this paper is on the types of assertions design engineers might embed directly into RTL code as the RTL model is being developed.

It is important to note, however, that only declarative concurrent assertions can be directly bound into RTL

code. Because immediate assertions and procedural concurrent assertions must be coded within procedural code, they cannot be easily defined externally and bound in. (It is possible to duplicate the RTL models always procedure sensitivity in a separate file, place an immediate or procedural concurrent assertion in the duplicate procedure, and bind the procedure to the RTL model. This, however, is not simple and can be subject to errors if the RTL code is changed.) A company that limits the modification rights of RTL design files to just the design team is also limiting the types of assertions that verification engineers can use to be just declarative immediate assertions. Immediate assertions are very useful, however. This is another reason for, and benefit to, having design engineers add assertions to RTL code as the code is being written.

A suggested guideline is that RTL design engineers should embed assertions directly within RTL modules and interfaces. Verification engineers will add many more assertions, but rather than embed these assertions directly within the modules and interfaces, these assertions should be bound into the RTL code.

Some companies might wish to take exception to this general guideline, however. If there are automated processes in place to re-run synthesis whenever the time stamp of RTL files change, then the addition of an assertion could cause synthesis to run, even though the RTL code itself did not change. This impact will be minimized if design engineers are embedding simple assertions in the RTL code at the same time the RTL code is being developed, rather than designers or verifiers adding assertions later on.

2.5 Assertion controls

SystemVerilog provides two primary mechanisms for enabling and disabling assertions at any time during simulation, **`disable iff`** and assertion control system tasks.

The **`disable iff`** construct is used to turn off specific assertions during specific logic conditions in the DUT. A common usage of **`disable iff`** is to disable certain assertions during reset or low-power modes. For example:

```
property p_a_to_b (clk, a, b, min, max);
  @(posedge clk)
  disable iff (!resetN)
  $rose(a) |-> ##[min:max] $rose(b);
endproperty: p_a_to_b
```

Example 9 — Using disable iff to disable a concurrent assertion

The disable condition can be a hard coded signal (as with `resetN` in the example above), or an argument to the property definition, or specified at the module level for all assertions within a module (using **`default disable iff`**).

The **`disable iff`** assertion control is part of the assertion property definition and would, therefore, be used by a design engineer who is embedding an assertion directly within the RTL code. Note, however, that **`disable iff`** can only be used with concurrent assertions. Most assertions written by design engineers will be immediate assertions, which cannot be controlled using **`disable iff`**.

SystemVerilog also provides a set of assertion controls that work with both immediate and concurrent assertions. These controls are system tasks, such as `$asserton`, `$assertoff` and `$assertcontrol`. These controls can be course-grained, affecting assertions throughout the design and verification hierarchy, or fine-grained, affecting only a selected number of assertions. These assertion control system tasks are generally used in verification testbenches. In the author's opinion, these controls are not useful for embedding within RTL models, and are not covered in detail in this paper.

Assertion control system tasks are typically used by RTL designers, but they do make it possible for designers to embed assertions in the RTL code, which verification engineers can then enable or disable as needed when the design is being verified.

3.0 Embedded assertions design engineers should write

As RTL design engineers model the functionality described in a project specification, they must:

- Interpret the intent of the design specification and write the RTL model functionality accordingly.
- Make assumptions regarding the validity of the inputs into each RTL module, and sometimes functional blocks within a module.

Assertions that verify the RTL code matches the intent of the design specification should be written by verification engineers. These verification assertions are outside the scope of this paper. Assertions that validate assumptions made within the RTL code should be written by design engineers and embedded directly within the RTL models.

RTL code makes assumptions regarding the validity of data and control values, in particular the values received on the inputs of a module. For example, the RTL code for a one-hot state machine *assumes* the state values are always one-hot. The RTL code for a bidirectional bus *assumes* the bus control lines are mutually exclusive, so that the bus is not being read and driven at the same time. The RTL code for an arithmetic logic unit *assumes* the opcode and data values are valid. Most register-based logic *assumes* that key registers will come out of reset — or out of low power mode — with valid values.

The RTL code for a complex design will be based on dozens, perhaps hundreds, of such assumptions. What if just one of those assumptions is wrong, and an incorrect data or control value occurs? When that happens, there is a high risk that the design will not function as intended. An incorrect DUT output, however, will often be far removed in the logic flow of the design from where the faulty value first occurred. The incorrect result must be traced back from the DUT outputs through the design logic, possibly crossing multiple module boundaries, to eventually find the RTL code that contained an invalid assumption. Adding to the difficulty of finding where things went wrong is that the faulty results at the DUT outputs might have occurred many clock cycles after the original problem, making it necessary to trace output logic values back through both space and time to find the bug that lead to the incorrect output.

Adding SystemVerilog Assertions at each point where the RTL designer made an assumption can significantly reduce the time it takes to debug a design. Where output verification can only say “something has gone wrong”, assertions embedded in the RTL code can say “something has gone wrong *right here*!” Furthermore, assertions that check any assumptions regarding data and control values can detect a problem that might not be easily detected by only verifying the values visible at the DUT outputs.

3.1 Some examples of embedded assertions to validate assumptions in RTL models

Following are three examples showing how simple one-line assertions can be used to validate assumptions made within RTL code. The benefit of embedding these simple assertions should be readily apparent. Without the assertions, a value that does not meet the condition assumed in the RTL code could result in erroneous DUT outputs that might be very difficult to trace back to the cause of the problem. Adding these simple assertions pinpoints when and where a value does not meet the requirements that were assumed in the RTL code.

Validating assumptions regarding valid 2-state values. RTL sequential logic assumes that reset signals will either be 0 or 1. Signals such as reset will never be tri-stated or unknown. An example of RTL code that makes this assumption is a simple D-type flip-flop with an active-low reset.

```
// assume the resetN control line is never an X or Z
always_ff @(posedge clock or negedge resetN)
  if (!resetN) q <= 0;
  else         q <= d;
```

Example 10 — RTL D flip-flop model that assumes a valid reset value

An interesting—but wrong!—behavior can happen in this example if `resetN` transitions from a 1 to an X or Z. This transition will be seen as a negative edge of `resetN`, which will trigger the sensitivity list of the **`always_ff`**. The **`if`** statement will evaluate `!resetN` as unknown, which means the **`else`** branch will be executed. Thus, the X or Z value of `resetN` is treated as if a clock edge had occurred, changing the state of the flip-flop at an incorrect time.

The effect of this erroneous value on `resetN` might not show up on the DUT outputs until much later. Tracing the problem back to the point where the problem occurred can be difficult. A simple one-line immediate assertion can trap this erroneous value on `resetN` at the exact point in time and space where the error occurred.

```
// assume the resetN control line is never an X or Z
always_ff @(posedge clock or negedge resetN) begin
  assert (!$isunknown(resetN)) else $error("unknown value on resetN");
  if (!resetN) q <= 0;
  else         q <= d;
end
```

Example 11 — RTL D flip-flop model with an assertion that validates the reset value

Validating assumptions that values fall within a specified range. Immediate assertions can also be used to verify that data values fall into an expected range. The following example is an excerpt of code from an Arithmetic Logic Unit (ALU). One of the ALU operations is to shift the `a` input right by 1, 2 or 3 (divide by 2, 4 or 8). The number of times to shift is controlled by the value of the `b` input. This operation assumes that the value of `b` is between 1 and 3, inclusive. Since `b` is a 16-bit vector, it could potentially have a value much greater than 3. A simple immediate assertion can detect an out-of-range value of `b` at the very moment the shift operation is performed. Rather than having to examine the DUT output values and tracing an incorrect output back to the shift operation, the assertion effectively says, *"something went wrong right here, right now!"*

```
logic [15:0] a, b, result;

// assume value of b is within the range of 1 to 3, inclusive
always_ff @(posedge clock)
  case (opcode)
    ADD_A_TO_B : result <= a + b;
    ... // other operations
    SHIFT_BY_B : begin
      assert (b inside {[1:3]}) else $error("b out of range for shift");
      result <= a >> b;
    end
  endcase
endmodule
```

Example 12 — RTL variable shifter with an assertion to validate a value is within a valid range

Validating assumptions that a value remains stable multiple clock cycles. Concurrent assertions can be used to validate assumptions that span one or more clock cycles. On simple example is an assumption that once reset becomes active, it must remain active for several clock cycles in order for all sequential logic to stabilize in a reset state. The following example illustrates a 4-bit Johnson Counter modeled using simple resets that do not have a reset input. When the counter is reset (using an active-low reset value), the first flip-flop in the counter is loaded with a 0 on the next positive edge of clock. This 0 will ripple through the subsequent flip-flops in the chain over the next three clock cycles. The RTL code assumes the reset input will remain a 0 for at least 4 clock cycles, in order for the reset to completely propagate through the four

flip-flops. If the reset input goes high sooner than 4 clock cycles, the counter might not correctly reset to all zeros (depending on the last value that was stored in the last flip-flop prior to the reset).

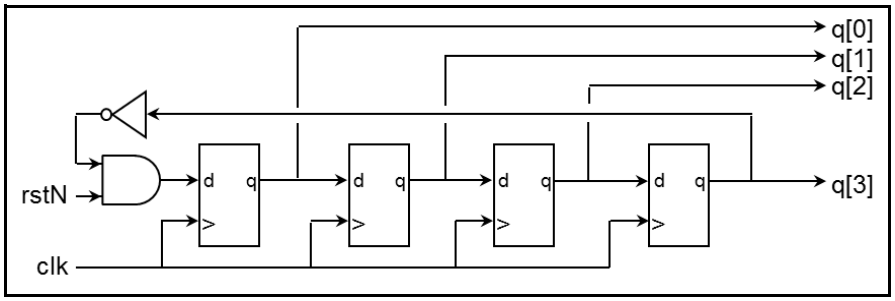

Figure 5. A 4-bit Johnson counter where reset must remain active at least 4 clock cycles

Example 13 shows the RTL code for this Johnson counter, along with an embedded declarative concurrent assertion to validate that, whenever reset goes low, it remains low for at least 4 clock cycles. The `[*4]` used in this example is called a *consecutive repetition operator*, and checks that the expression `!rstN` evaluates as true on 4 consecutive clock cycles.

```
module jcounter (output logic [3:0] q, input logic clk, rstN);

  assert property (@(posedge clk) $fell(rstN) |-> !rstN[*4])
  else $error("rstN did not remain low for at least 4 clock cycles");

  always_ff @(posedge clk) begin
    q[0] <= ~q[3] & rstN;
    q[1] <= q[0];
    q[2] <= q[1];
    q[3] <= q[2];
  end
endmodule: jcounter
```

Example 13 — 4-bit Johnson counter with check that reset is low for at least 4 clock cycles

3.2 Avoiding glitches when using immediate assertions in combinational logic

Care needs to be taken when embedding an immediate assertion in combinational logic. A combinational RTL procedure can trigger multiple times within a moment of simulation time before settling on the final value for that moment. Consider the following simple combinational logic code, which is an excerpt from a finite state machine controller:

```
logic S1, S2, S3, S4; // 1-bit variables to represent different states

always_comb begin // triggers on changes to each state bit
  assert ($onehot({S1,S2,S3,S4}) else $error("state bits not one-hot");
  case (1'b1)
    S1: ... // do stuff for state 1
    S2: ... // do stuff for state 2
    S3: ... // do stuff for state 3
    S4: ... // do stuff for state 4
  endcase
end
```

Example 14 — An immediate assertion that could have potential glitches

Example 14 only shows the combinational logic block that decodes these state variables, and not the code that assigns values to `S1` through `S4`. It is possible that two or more of these state bits could change at the same moment of simulation time. Each change could trigger the **`always_comb`** procedural block, causing the immediate assertion to be evaluated more than once within the same moment of simulation time. At the end of that moment in time, `S1` through `S4` should be stable, and presumably have a one-hot value (one, and only one, of the four state bits is set). The assertion(s) that fired before the state bits had stabilized for that moment in time, however, might have seen more than one bit set, and reported a false assertion failure.

SystemVerilog has a special form of immediate assertions called a *"deferred immediate assertion"* to handle this potential combinational logic glitch. A deferred immediate assertion is specified using the keyword pair **`assert final`**. The assertion in Example 14 would be rewritten as:

```
assert final ($onehot({S1,S2,S3,S4}) else $error("state bits not one-hot");
```

Example 15 — An deferred immediate assertion that prevents potential glitches

Deferred immediate assertions evaluate the true/false expression when the **`assert`** statement is executed, but defer executing the pass or fail statement until the activity in the current simulation time is stable. If the procedure triggers multiple times in the same moment of simulation time, any pending pass/fail statements are deleted from the execution queue and replaced with the most recent action. Only the pass/fail action of the last evaluation of the assertion is executed.

The pass and fail action blocks of an **`assert final`** deferred immediate assertion are restricted, and can only be used to print messages. SystemVerilog also has a second form of deferred immediate assertion, which is specified using **`assert #0`**. This form schedules the execution of deferred pass or fail statements differently than **`assert final`**. An **assert #0** allows the pass or fail statement to do more than just print a message, but this flexibility can still result in an assertion glitches. The **assert final** construct is the preferred construct to use as an embedded immediate assertion in RTL combinational logic blocks.

3.3 Assertion templates for use in RTL models

Very often, the same general assertion will be used in many places, just with different signals. To avoid having to duplicate the code for similar assertions, a library of assertion templates can be defined. The templates can be defined using traditional Verilog **`‘define`** text substitution macros, or the more powerful assertion **`let`** statement. Let statements can often be used in place of **`‘define`** macros, but have advantages over macros that are beyond the scope of this paper. The library of templates can be place in a SystemVerilog package, making them available to all design and verification engineers involved in a project. SystemVerilog also provides a **`checker`** construct for defining more complex assertion templates.

```
package assertion_templates;
   `define isknown(a) assert (!$isunknown(a)) else $error("unknown value")
   let mutex(a,b) = $onehot0({a,b});
endpackage

module my_chip (...);
   import assertion_templates::*;

   always_ff @(posedge clock) begin
      `isknown({rd,wr});
      assert (mutex(rd,wr)) else $error("rd and wr both set");
      if (wr) storage <= data;
   end
endmodule
```

Example 16 — Definition and usage of an assertion templates using a ‘define and let statements

3.4 Checking parameters for legal values

RTL models often use parameter constants to make the model configurable. Vector widths, for example, are often based on one or more parameters. These parameter constants can be redefined for each instance of a parameterized module or interface. This redefinition takes place prior to simulation or synthesis actually starting, while the RTL code is being compiled and elaborated.

One problem with parameterized modules is the risk of a parameter being redefined to a value that is not valid for the code within the module. This invalid value can become a run-time error that can only be detected by effective verification. Run-time errors such as this might not become evident until after running extensive, lengthy simulations.

RTL code is often written with the assumption that the final values of parameters, after redefinition, are valid values. An *"elaboration time assertion statement"* (officially called *"elaboration system tasks"* can be used to check the final value of parameters before simulation starts running. Elaboration time assertions are conditional generate **`if`**…**`else`** or **`case`** statements that print a failure message using `$info`, `$warning`, `$error`, or `$fatal`. Generate statements are executed at the end of elaboration, before simulation is run. Only `$fatal` will cause elaboration to abort and prevent simulation from running.

The following example uses an elaboration time assertion to validate that a multiplexor has been properly configured to be either a 2:1 mux (the `S` parameter is 1) or a 4:1 mux (`S` is 2).

```
module muxN  // 2:1 MUX (S == 1) or 4:1 MUX (S == 2)
 #(parameter N=8, S=1)
  (output logic [N-1:0] y,
   input  logic [N-1:0] a, b,     // a and b should always be connected
   input  tri0  [N-1:0] c, d,     // c and d pull down if unconnected
   input  logic [S-1:0] sel);

   generate
     if (!(S inside {[1:2]})) $fatal(0,"In %m S=%0d; must be 1 or 2", S);
   endgenerate

   always_comb begin
     case (sel)
       2'b00: y = a;
     2'b01: y = b;
     2'b10: y = c;
     2'b11: y = d;
     endcase
   end
endmodule: muxN
```

Example 17 — An elaboration-time assertion to check final parameter values

Note that the **`assert`** keyword is not used for elaboration time assertions. A SystemVerilog generate block can only use **`if`**…**`else`** or **`case`** decisions, and not **`assert`**.

Also note that the **`generate`**/**`endgenerate`** keywords shown in the preceding example are not required, and are often omitted. In SystemVerilog, a generate block is inferred if an **`if`**...**`else`**, **`case`**, or **`for`** statement is not enclosed in some type of procedural code (such as an **`initial`** or **`always`** block).

4.0 SystemVerilog constructs with built-in assertions

In addition to the **`assert`** construct, SystemVerilog provides several important RTL modeling constructs

that have built-in assertion behavior. Three of these constructs are briefly covered in this section. By using these constructs, design engineers gain the advantages of assertion-like checking in a design, without having to actually write the assertions.

4.1 RTL always_comb, always_comb and always_ff procedures

always_comb — This specialized procedural block provides several capabilities that the Verilog **always** procedural block does not have. In brief, this specialized block enforces a synthesizable modeling style. In addition, the **always_comb** block allows software tools to check that the code within the block functions as combinational logic. For example, there are a number of coding mistakes in intended combinational logic that can result in latches. By using **always_comb**, these coding mistakes can be detected early in the design stage. Writing assertions to detect inadvertent latched behavior is not necessary. The checks are built into the **always_comb** construct.

The following example illustrates an **always_comb** procedure that does not model combinational logic:

```
always_comb begin
   if (!mode)
      o1 = a + b;
   else
      o2 = a - b;
end
```

Example 18 — An always_comb procedure with non-combinational logic code

DC Compiler generates the following error message when this code is compiled:

Warning: test.sv:5: Netlist for always_comb block contains a latch.

Note that the IEEE SystemVerilog standard does not require a tool to detect and report latched logic functionality in an **always_comb** procedural block. At the time this paper was written, VCS does not report such warnings (nor does any other major commercial simulator), but DC Compiler and Synplify-Pro do report these warnings (as do other commercial synthesis compilers and many RTL lint programs).

The **always_latch** procedural block performs similar checking as **always_comb**, with the obvious difference that a warning is issued if the code in the procedure does not represent latched functionality. The checking for non-synthesizable code and latched behavior would be difficult to write as assertions, and there is no need to! The **always_latch** procedure has this difficult checking built in.

An **always_ff** specialized procedure has different rules. This procedure generates an error if the procedure contains an **@** token anywhere other than as the procedure's sensitivity list. It is also an error for the procedure to have any statements that block simulation time, such as **#**, **wait**, and **task** calls. More details on **always_comb**, **always_latch** and **always_ff** SystemVerilog specialized procedural blocks can be found the book "*SystemVerilog for Design*" [3].

4.2 Unique, unique0 and priority decision modifiers

Synthesis compilers can perform a number of gate-level optimizations on **if**...**else** and **case** decision statements. These optimizations are controlled by two synthesis pragmas (embedded commands): `synthesis full_case` and `synthesis parallel_case`. These synthesis commands are hidden inside comments in the RTL source code. There is a major hazard with these optimizations! Simulation does not know about these optimizations. Therefore, it is possible for an RTL model to pass exhaustive verification testing, only to fail in the post-synthesis gate-level design because of the optimizations performed by synthesis.

SystemVerilog provides three special constructs that add run-time simulation checking to help verify that the `full_case` and `parallel_case` synthesis optimizations will work as intended. **`unique`** is a decision modifier that requires tools to check that a decision sequence (a **`case`** statement or an **`if`**...**`else`**...**`if`** statement) is coded in a way that would be safe for the synthesis `full_case` and `priority_case` optimizations. If, during simulation, the decision statement is entered and no branch is taken, a run-time *violation report* is issued, indicating the case statement is not complete and might not function correctly after synthesis if `full_case` synthesis optimization is performed.

In addition, the **`unique`** modifier also requires that simulation report a warning any time two or more decision branches are true at the same time, indicating the case items are not mutually exclusive, and might not work if synthesis `parallel_case` optimizations are performed. These two built-in checks are another built-in assertion that designers should take advantage of.

`unique0` is similar to **`unique`**, except that it only checks that synthesis `full_case` optimization would be safe. A run-time violation report is issued during simulation anytime the decision statement is entered and no branch is taken.

`priority` modifies a decision statement by checking that synthesis `parallel_case` optimization would be safe. A run-time warning is issued during simulation anytime two or more decision branches are true at the same time.

It is important to note that **`unique`**, **`unique0`** and **`priority`** turn on synthesis optimizations that are sometimes useful, but are not always desirable. RTL designers must still understand the effects of these synthesis optimizations in the gate-level implementation of the design, and use **`unique`**, **`unique0`** and **`priority`** appropriately. For a more detailed description on these decision modifiers, refer to the papers *"SystemVerilog Saves the Day—the Evil Twins are Defeated! "unique" and "priority" are the new Heroes"* [4], and *"SystemVerilog's priority & unique - A Solution to Verilog's 'full_case' & 'parallel_case' Evil Twins!"*[5].

4.3 Enumerated types

Traditional Verilog nets and variables are loosely typed. Loosely typed nets and variables have built-in conversion rules that allow a value that is out-of-range, or of a different type, to be assigned. There is no checking of the value being assigned; the value is simply converted to the data type of the net or variable.

The following example shows code for a small state machine modeled with traditional Verilog. The code is syntactically correct, and will simulate and synthesize. Due to Verilog's loosely typed behavior, however, there are six functional bugs that could end up in the gate-level implementation of the state-machine. (Synthesis might issue a warning for some of these bugs, but the warnings are not fatal, and synthesis will implement the bugs in the gate-level design.)

```
module controller
(input  wire      clock, resetN,
 output reg [2:0] control);

  // Names for state machine states (one-hot)
  parameter [2:0] WAIT = 3'b001,
                  LOAD = 3'b010,
                  DONE = 3'b001;      // BUG 1: 2 constants with same value

  // Names for control output values
  parameter [1:0] READY = 3'b101,     // BUG 2: parameter width and value
                  SET   = 3'b010,     //   width are different sizes
                  GO    = 3'b110;
```

```
  // State and next state variables
  reg [2:0] state, n_state;

  // State Sequencer
  always @(posedge clock or negedge resetN)
    if (!resetN) state <= 0;          // BUG 3: resetting to an undefined
    else         state <= n_state;    //   state value

  // Next State Decoder (sequentially cycle
  // through the three states)
  always @(state)
    case (state)
      WAIT: n_state = state + 1;
      LOAD: n_state = state + 1;      // BUG 4: results in undefined state
      DONE: n_state = state + 1;      // BUG 5: results in undefined state
    endcase

  // Output Decoder
  always @(state)
    case (state)
      WAIT: control = READY;
      LOAD: control = SET;
      DONE: control = DONE;           // BUG 6: assigning wrong constant
    endcase
endmodule //controller
```

Example 19 — RTL state machine model with functional bugs, using Verilog parameters

SystemVerilog adds *enumerated type* nets and variables that are more strongly typed. Enumerated types allow nets and variables to be defined with a specific set of named values, referred to as *labels*. Designers can specify explicit values for each label.

Enumerated types have stronger rule checking than built-in variables and nets. These rules include:

- The value of each label in the enumerated list must be unique.
- The variable size and the size of the label values must be the same.
- An enumerated variable can only be assigned:
  - A label from its enumerated list.
  - The value of another enumerated variable from the same enumerated definition.

The stronger rules of enumerated types provide significant advantages over traditional Verilog. Example 20 is the same state machine as the one previously shown in Example 19, but this time modeled using SystemVerilog enumerated types. This next example has the same six bugs as above, but, all six bugs become syntax errors when using enumerated types. The built-in checking of enumerated types makes it so the bugs must be fixed before the design can be simulated. Synthesis compilers will allow this faulty state machine to be implemented in gates until these bugs are fixed.

```
module controller
(input  logic       clock, resetN,
 output logic [2:0] control_out
);

// Enumerated variables for fsm states
// One-hot encoding for the labels
```

```
enum logic [2:0] {WAIT  = 3'b001,
                  LOAD  = 3'b010,
                  DONE  = 3'b001} // SYNTAX ERROR: duplicate value
                  state, n_state;

// Enumerated variables for control output values
enum logic [1:0] {READY = 3'b101, // SYNTAX ERROR: size mismatch
                  SET   = 3'b010,
                  GO    = 3'b110}
                  control;

  assign control_out = control;

  // State Sequencer
  always_ff @(posedge clock or negedge resetN)
    if (!resetN) state <= 0;      // SYNTAX ERROR: undefined state value
    else         state <= n_state;

  // Next State Decoder (sequentially cycle
  // through the three states)
  always_comb
    case (state)
      WAIT: n_state = state + 1;  // SYNTAX ERROR: undefined n_state value
      LOAD: n_state = state + 1;  // SYNTAX ERROR: undefined n_state value
      DONE: n_state = state + 1;  // SYNTAX ERROR: undefined N-state value
    endcase

  // Output Decoder
  always_comb
    case (state)
      WAIT: control = READY;
      LOAD: control = SET;
      DONE: control = DONE;       // SYNTAX ERROR: label not in definition
    endcase
endmodule: controller
```

Example 20 — Same RTL state machine model but using SystemVerilog enumerated types

More details on the rules and advantages of enumerated types can be found in the book "*SystemVerilog for Design*" [3].

5.0 X-propagation versus X-trapping

SystemVerilog simulators use four primary values to represent digital hardware behavior: 0, 1, Z and X (there are actually more than 120 values when logic levels and logic strengths are taken into account, but 0, 1, Z and X are the values used the most). The values 0, 1 and Z represent values that also exist in digital logic. The value X is unique to simulation, and, in essence, is the simulator's way of saying "*I don't know what the actual hardware will do*". Logic X is often referred to as "*unknown*".

Obviously, design and verification engineers want to know when functionality in the design results in X values. A problem can arise — and frequently does arise — however, where an X value does not propagate all the way through the design functionality to the point where verification is observing design outputs. This means unknown conditions within the design can go undetected. Conversely, it also a problem if

simulation propagates an X value, but actual silicon would work as intended. Verification time can be lost chasing down seeming problems that are not real.

The terms used to describe how logic X values propagate through a design are *X-optimism* and *X-pessimism*. *X-optimism* is defined as any time simulation converts an X value on an expression or logic gate input into a 0 or a 1 on the result. *X-pessimism* is defined as any time simulation passes an X on an input to an expression or logic gate through to the result. A recent conference paper, *"I'm Still In Love With My X!"* [6], explains in detail several ways simulation might result in a logic X value. The paper also itemizes the X-optimistic and X-pessimistic constructs in SystemVerilog, and the hazards of propagating logic X values using optimistic or pessimistic rules.

5.1 SystemVerilog's X pessimism hazard

X-optimism is often considered more serious than X-pessimism for thorough design verification. If something goes wrong within a design that results in a logic X (unknown) value, but an X-optimistic construct does not propagate that X, a potentially serious design bug can go undetected. Example 21 illustrates a common, situation where X-optimism can hide a design problem (refer to the paper *"I'm Still In Love With My X! (but, do I want my X to be an optimist, a pessimist, or eliminated?)"* [6] for other examples).

```
always_comb begin
  if (sel) y = a;   // if sel is 1
  else     y = b;   // if sel is 0, X or Z
end
```

Example 21 — if...else statement X-optimism

SystemVerilog has an optimistic behavior when the *control condition* of an **`if`**...**`else`** statement is unknown. The simulation rule is simple: should the *control condition* evaluate to unknown, the **`else`** branch is executed. This optimistic behavior can hide a problem with `sel`, the *control condition*. In actual silicon, the ambiguous value of `sel` will be 0 or 1, and **`y`** will be set to a known result. How accurately does SystemVerilog's X-optimistic behavior match the behavior in actual silicon? The answer depends in part on how the **`if`**...**`else`** is implemented in silicon.

The behavior of this simple **`if`**...**`else`** statement might be implemented a number of ways in silicon. Figures 6 and 7 illustrate two possibilities, using a Multiplexor or NAND gates, respectively.

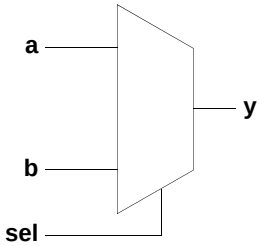

Figure 6. 2-to-1 selector, implemented using a MUX gate

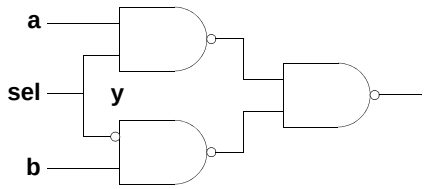

Figure 7. 2-to-1 selector, implemented using NAND gates

Table 2 shows the simulation results for an X-optimistic **`if`**...**`else`** when the control expression (`sel`) is unknown, compared to the simulation behavior of the MUX and NAND implementations, and to actual silicon behavior.

inputs			output (y)			
			simulation behavior			actual silicon behavior
sel	a	b	if...else RTL	MUX gate	NAND gates	
X	0	0	0	0	0	0
X	0	1	1	X	X	0 or 1
X	1	0	0	X	X	0 or 1
X	1	1	1	1	X	1

Table 2: if...else versus gate-level X propagation, versus actual silicon

Some important things to note from this table include:

- For all rows, the **`if`**...**`else`** statement propagates a known value instead of the X value of `sel`. This X-optimistic behavior could hide error conditions in the design.
- For rows 2 and 3, the X-optimistic **`if`**...**`else`** behavior only matches one of the possible values that could occur in actual silicon. *The other possible value is not propagated, and therefore the design is not verified with that other possible value.*
- The MUX implementation of an **`if`**...**`else`** is the most accurate, and propagates an X when there is a potential of actual silicon having either a 0 or a 1.
- The NAND-gate implementation is overly pessimistic for when `a` and `b` are both 1 (row 4), and propagates an X value, even though the actual silicon would have a known value of 1.

All of these simulation behaviors pose a problem for design verification. The overly optimistic **`if`**...**`else`** model might completely hide a design bug, and could leave a design only partially verified before tape-out. The overly pessimistic NAND gate model propagates X values even when there is no ambiguity in how actual silicon would behave. The MUX example propagates the X value appropriately, but depends on that X continuing to propagate to an observed verification point, possibly many clock cycles into the future from when the problem occurred. None of the modeling styles report that a problem has occurred when and where something went wrong. Section 5.3 will show how a simple embedded assertion can eliminate all of these shortcomings.

5.2 VCS prop X propagation

the Synopsys VCS simulator provides a way to reduce X-optimism in RTL simulation by using a more pessimistic, non-standard algorithm. The **Xprop** simulation option causes VCS to use simulator-specific X propagation rules for **`if`**...**`else`** and **`case`** decision statements and **`posedge`** or **`negedge`** edge sensitivity. This non-standard approach tries to find a balance between X-optimism and X-pessimism.

In brief (and perhaps overly simplified), when simulation is run with the Xprop option, VCS evaluates the effect of both a logic 0 and a logic 1 in place of the X, and merges the results of both evaluations. If the results of using either a 0 or 1 are the same, that result is propagated. If the results of using a 0 or 1 in place of an X are different, then an X is propagated. This is different than normal SystemVerilog X-optimistic RTL behavior, which always propagates a known value (as can be seen in the **`if`**...**`else`** column of Table 2. Instead, Xprop makes RTL simulation behave more like the MUX gate behavior shown in the Table. The papers and tutorials [7], [8] and [9] provide more information on—and experience with—using Xprop

The Xprop algorithm effectively resolves the concern of X-optimism and helps to ensure that all possible conditions in a design are verified. Using Xprop is not a perfect solution, however. Xprop's purpose is to propagate a bug downstream from the cause of the problem, so that the bug will be detected at some observed point. This then requires tracing the faulty output value from that observed point back through many lines of code and clock cycles to find the original cause of the problem. This debugging process can be tedious and time consuming.

Xprop can still hide design bugs, in a similar way that X-optimism can hide bugs. When the result of Xprop merging the evaluation of replacing an X with both a 0 and a 1 is the same, Xprop propagates that known-value result. The logic X was an indication that something had gone wrong, but, because the X did not propagate, the problem that cause the X can go undetected. While it can be argued that the X did not matter and therefore was not a real problem, whatever caused the X has been left unresolved.

5.3 Using assertions to trap X conditions

Sections 5.1 and 5.2 have shown that both SystemVerilog's X-optimism rules and VCS's Xprop option can hide problems in a design. Both approaches can propagate a known value when a logic X occurs deep inside design functionality. While Xprop's algorithm propagate a more accurate value than SystemVerilog's X-optimism semantics, the underlying cause of the logic X might go undetected and unfixed. This can be risky! Trapping a functional error at the point and time the error occurs can eliminate that risk.

Example 22 shows how a simple one-line assertion will detect an X value on `sel`, eliminating the concerns regarding whether the design problem will propagate to an observed output.

```
always_comb begin
  assert final (!$isunknown(sel)) else $error("select is X or Z.");
  if (sel) y = a;   // if sel is 1
  else     y = b;   // if sel is 0, X or Z
end
```

Example 22 — if...else statement with an X detection assertion

A simple X-trap using a one-line assertion can be used for all inputs to a module, and can detect problems with clocks, resets, control signals, coming out of low-power mode, and much more. For more examples and details on using immediate assertions to trap logic X and Z problems, refer to the paper *"Being Assertive With Your X"* [2], published in the proceedings of SNUG 2004.

Using assertions to trap a logic X value at its source does not negate the use of the VCS Xprop option. Rather, embedded assertions can supplement Xprop by localizing the cause of an error, while Xprop

propagates values that accurately represent silicon behavior. Using Xprop helps ensure more accurate RTL simulations, and using assertions helps detect and isolate the occurrence of logic X values within RTL models.

6.0 Simulation, formal verification and synthesis assertion support

Assertions are not just for simulation. Assertions also play a vital role in formal verification. Formal verification is beyond the scope of this paper. What is important, however, is that formal tools utilize an additional type of assertion that use the keyword **`assume`** instead of **`assert`**. The **`assume`** and **`assert`** assertions do the same thing in simulation, but have a different meaning for formal tools.

A design that will be formally verified would benefit from having **`assume`** statements embedded in the RTL code, as well **`assert`** statements. VCS correctly supports **`assume`** statements, which should allow this to be done. Synplify-Pro ignores **`assume`** statements, just like it ignores **`assert`** statements. The problem, however, is that the version of the DC synthesis compiler available when this paper was written only ignores **`assert`** statements. DC errors out when an **`assume`** statement is encountered.

One work-around for this shortcoming of DC is to use conditional compilation to hide any embedded **`assume`** statements from synthesis. This solution is not ideal. It places an added burden on the RTL designer, and loses some of the advantages of embedded assertions that have been discussed. A second work-around is for design engineers to only embed **`assert`** statements into RTL code, and for verification engineers to bind in **`assume`** statements that are coded outside of the RTL code. Only concurrent **`assume`** statements can be bound in, however. Immediate **`assume`** statements cannot be bound into another module.

A similar situation can arise with the assertion **`cover`** statement. The keyword is supported by VCS, but is not ignored by DC. This lack of support is not as serious as the **`assume`** construct, however, because there is no real advantage for RTL designers to embed **`cover`** statements within RTL code. These statements will most likely be written by verification engineers, and can be written externally to the synthesizable RTL models.

6.1 Recommendations to Synopsys R&D

Synopsys R&D has done a great job with implementing SystemVerilog assertions in VCS. Synopsys has also done a great job of properly ignoring the **`assert`** statement in both the DC and Synplify-Pro synthesis compilers. This support allows both verification engineers and design engineers to take advantage of using assertions, including having designers embed assertions into RTL code.

There are times, however, where an **`assume`** statement should be embedded into RTL code, rather than an **`assert`** statement. VCS supports the assume statements for simulation, and Synplify-Pro ignores **`assume`** statements in the same way it ignores **`assert`** statements. DC, however, generates a fatal error on **`assume`** statements (and **`cover`** statements). The author encourages Synopsys R&D to also ignore **`assume`** and **`cover`** assertions in DC, the same way **`assert`** statements are ignored.

Elaboration-time assertions, as discussed in Section 3.4, can help ensure that models are correctly parameterized for simulation and synthesis. Currently, however, neither VCS nor DC nor Synplify-Pro support elaboration-time checking. This is an important construct to add to these tools.

The following table summarizes how VCS, DC and Synplify-Pro handle the primary assertion constructs that engineers might embed in RTL models. This support is based on the versions of these products available to the author at the time this paper was written.

Assertion Construct	VCS 2014.12	DC 2014.09-SP5	Synplify-Pro 2014.09-SP2
`assert` immediate assertion	supported	ignored	ignored
`assert property` declarative concurrent assertion	supported	ignored	ignored
`property`/**`endproperty`** property definition	supported	ignored	ignored
`sequence`/**`endsequence`** sequence definition	supported	ignored	ignored
`disable iff` assertion control	supported	ignored	ignored
`assert final` deferred immediate assertion	supported	**not supported**	**not supported**
`assert #0` deferred immediate assertion	supported	**not supported**	**not supported**
`let` assertion templates	supported	ignored	**not supported**
`checker` assertion templates	supported	**not supported**	**not supported**
elaboration time assertion (using **`generate if`**...**`else`**)	**not supported**	**not supported**	**not supported**
`assume` and **`assume property`** assertions	supported	ignored	ignored
`cover` and **`cover property`** assertions	supported	ignored	ignored

Table 3: Synopsys tool support for primary assertion constructs

Note: The information in this paper is based on Synopsys ***VCS*** version 2014.12, Synopsys ***Design Compiler*** (also called ***HDL Compiler***) version 2014.09-SP5 and Synopsys ***Synplify-Pro*** version 2014.09-SP2. These were the most current released versions available to the author at the time this paper was written.

7.0 Summary

SystemVerilog Assertions can play an important role in verifying that a design meets its intended functionality. Assertions are a verification construct, but there are significant advantages to having RTL designers embed certain types of assertions into RTL code as the code is being written. These advantages include:

- Documenting assumptions made by the designer regarding the RTL code.
- Validating values used within the RTL code, in particular values used by decision statements.
- Reducing the risks of SystemVerilog's X-optimism, X-pessimism.
- Supplementing the VCS Xprop simulation option.
- Localizing exactly when and where functional error conditions in the RTL code occur.

Another important advantage of having RTL designers embed assertions into the RTL code is that writing an assertion requires closely examining the design specification. This examination will often reveal ambiguities in the spec. The paper *"Adding Last-Minute Assertions: Lessons Learned (a little late) about Designing for Verification"*[10] provides a real-life case study on how the mere act of writing assertions found serious design errors due to ambiguities in the project specification.

The types of assertions that RTL designers might embed into their RTL code should focus on checking that the values used in the RTL are legal values for that code. These are usually simple, one-line assertions, and

can most often be written as immediate assertions. Occasionally, a design engineer might want to use a concurrent assertion to validate an assumption that spans multiple clock cycles. Verification engineers, on the other hand, will write assertions based on the design specification, in order to ensure that the design meets the intent of the spec. These assertions will most often be concurrent assertions, and can sometimes be quite complex. These specification-related assertions do not need to be embedded in the RTL code. They can be written in a separate module or interface, and bound into the RTL modules.

In addition to the **`assert`** statement, SystemVerilog provides a number of RTL design constructs that have built-in assertion-like behavior. These include the **`always_comb`**, **`always_latch`** and **`always_ff`** procedural blocks, **`enum`** enumerated types, and the **`unique`**, **`unique0`** and **`priority`** decision statements.

As with all aspects of design and verification, formal training on the SystemVerilog Assertions language will help ensure efficient and proper usage of SVA.

8.0 Acknowledgements

The author expresses appreciation to Cliff Cummings, Dan Lefrancois, and Karim Ameziane of the SNUG Technical Committee members for reviewing drafts of this paper for technical content and correctness.

9.0 References

[1] *1800-2012 IEEE Standard for System Verilog: Unified Hardware Design, Specification and Verification Language"*, IEEE, Pascataway, New Jersey. Copyright 2013. ISBN: 978-0-7381-8110-3 (PDF), 978-0-7381-8111-0 (print).

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

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

[4] "*SystemVerilog Saves the Day—the Evil Twins are Defeated! "unique" and "priority" are the new Heroes*", by Stuart Sutherland. Published in the proceedings of SNUG San Jose, 2005 (also available at sutherland-hdl.com/papers.php).

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

[6] *."I'm Still In Love With My X! (but, do I want my X to be an optimist, a pessimist, or eliminated?)"*, by Stuart Sutherland. Published in the proceedings of DVCon San Jose, 2013 (also available at sutherland-hdl.com/papers.php).

[7] *"Getting X Propagation Under Control"*, Greene, a tutorial presented at SNUG San Jose, 2012.

[8] *"X-Propagation: An Alternative to Gate Level Simulation"*, Evans, Yam and Forward, Published in the proceedings of SNUG San Jose, 2012.

[9] *"X-Optimism Elimination during RTL Verification"*, Greene, Salz and Booth, Published in the proceedings of SNUG San Jose, 2012.

[10] "*Adding Last-Minute Assertions: Lessons Learned (a little late) about Designing for Verification"*, by Stuart Sutherland. Published in the proceedings of DVCon San Jose, 2013 (also available at sutherland-hdl.com/papers.php).